A Semantics-Based Approach to Malware Detection

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

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

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

Abstract

Malware detection is a crucial aspect of software security. Current malware detectors work by checking for “signatures,” which attempt to capture (syntactic) characteristics of the machine-level byte sequence of the malware. This reliance on a syntactic approach makes such detectors vulnerable to code obfuscations, increasingly used by malware writers, that alter syntactic properties of the malware byte sequence without significantly affecting their execution behavior.

This paper takes the position that the key to malware identification lies in their semantics. It proposes a semantics-based framework for reasoning about malware detectors and proving properties such as soundness and completeness of these detectors. Our approach uses a trace semantics to characterize the behaviors of malware as well as the program being checked for infection, and uses abstract interpretation to “hide” irrelevant aspects of these behaviors. As a concrete application of our approach, we show that the semantics-aware malware detector proposed by Christodorescu et al. is complete with respect to a number of common obfuscations used by malware writers.

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

General Terms Security, Languages, Theory, Verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL’07 January 17–19, 2007, Nice, France.
Copyright © 2007 ACM 1-59593-575-4/07/0001...$5.00

Keywords malware detection, obfuscation, trace semantics, abstract interpretation.

1. Introduction

Malware is a program with malicious intent that has the potential to harm the machine on which it executes or the network over which it communicates. A malware detector identifies malware. A misuse malware detector (or, alternately, a signature-based malware detector) uses a list of signatures (traditionally known as a signature database [22]). For example, if part of a program matches a signature in the database, the program is labeled as malware [26]. Misuse malware detectors’ low false-positive rate and ease of use have led to their widespread deployment. Other approaches for identifying malware have not proved practical as they suffer from high false positive rates (e.g., anomaly detection using statistical methods [19, 20]) or can only provide a post-infection forensic capability (e.g., correlation of network events to detect propagation after infection [15]).

Malware writers continuously test the limits of malware detectors in an attempt to discover ways to evade detection. This leads to an ongoing game of one-upmanship [23], where malware writers find new ways to create undetected malware, and where researchers design new signature-based techniques for detecting such evasive malware. This co-evolution is a result of the theoretical undecidability of malware detection [2, 5]. This means that, in the currently accepted model of computation, no ideal malware detector exists. The only achievable goal in this scenario is to design better detection techniques that jump ahead of evasion techniques and make the malware writer’s task harder.

Attackers have resorted to program obfuscation for evading malware detectors. Of course, attackers have the choice of creating new malware from scratch, but that does not appear to be a favored tactic [25]. Program obfuscation transforms a program, either manually or automatically, by inserting new code or modifying existing code to make understanding and detection harder, at the same time preserving the malicious behavior. Obfuscation transformations can easily defeat signature-based detection mechanisms. If a signature describes a certain sequence of instructions [26], then those instructions can be reordered or replaced with equivalent instructions [29, 30]. Such obfuscations are especially applicable on CISC architectures, such as the Intel IA-32 [16], 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 [17, 27, 30] 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 the program, packing or encryption with varying keys [13, 24] can effectively hide the relevant data. Therefore, an important require-
mation of a robust malware detection technique is to handle obfuscation transformations.

Program semantics provides a formal model of program behavior. Therefore addressing the malware-detection problem from a semantic point of view could lead to a more robust detection system. Preliminary work by Christodorescu et al. [4] and Kinder et al. [18] on a formal approach to malware detection confirms the potential benefits of a semantics-based approach to malware detection. The goal of this paper is to provide a formal semantics-based framework that can be used by security researchers to reason about and evaluate the resilience of malware detectors to various kinds of obfuscation transformations. This paper makes the following specific contributions:

- We present a formal definition of what it means for a detector to be sound and complete with respect to a class of obfuscations. We also provide a framework which can be used by malware-detection researchers to prove that their detector is complete with respect to a class of obfuscations. As an integral part of the formal framework, we provide a trace semantics to characterize the program and malware behaviors, using abstract interpretation to “hide” irrelevant aspects of these behaviors.
- We show our formal framework in action by proving that the semantic-aware malware detector $A_{MD}$ proposed by Christodorescu et al. [4] is complete with respect to some common obfuscations used by malware writers. The soundness of $A_{MD}$ was proved in [4].

2. Preliminaries

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

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

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

Note that this definition of soundness and completeness can be applied to a deobfuscator as well. In other words, our definitions are not tied to the concept of malware detection. Most malware detectors are built on top of other static-analysis techniques for problems that are hard or undecidable. For example, most malware detectors [4, 18] that are based on static analysis assume that the control-flow graph for an executable can be extracted. As shown by researchers [21], 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** (Oracle). An oracle is an algorithm over programs. For example, a CFG oracle is an algorithm that takes a program as its input and produces its control-flow graph.

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

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

2.1 A Framework for Proving Soundness and Completeness of Malware Detectors

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

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

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

1. **[Step 1] Relating the two worlds.** Let $D^{OR}$ be a detector that uses a set of oracles $OR$. Assume that we are given a program $P$ and malware $M$. Let $\mathcal{S}[P]$ and $\mathcal{S}[M]$ be the set of traces corresponding to $P$ and $M$, respectively. In Section 3 we describe a detector $D_T$ which works in the semantic world of traces. We then prove that if the oracles in $OR$ are perfect, the two detectors are equivalent, i.e., for all $P$ and $M$ in $P$, $D^{OR}(P, M) = 1$ if $D_T(\mathcal{S}[P], \mathcal{S}[M]) = 1$. 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.**

After step 1, we prove the desired property (e.g., completeness) about the trace-based detector $D_T$, with respect to the chosen class of obfuscations. In this step, the detector’s effects on the trace semantics are compared to the effects of obfuscation on the trace semantics. This also allows us to evaluate the detector against whole classes of obfuscations, as long as the obfuscations have similar effects on the trace semantics.

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

1 We assume that detector $D$ can query an oracle from the set $OR$, and the query is answered perfectly and in $O(1)$ time. These types of relative completeness and soundness results are common in cryptography.
Syntactic Categories:

- **n ∈ N** (integers)
- **X ∈ X** (variable names)
- **L ∈ L** (labels)
- **E ∈ E** (integer expressions)
- **B ∈ B** (Booleans)
- **A ∈ A** (actions)
- **D ∈ E ∪ (A × ϕ(L))** (assignment r-values)
- **C ∈ C** (commands)
- **P ∈ P** (programs)

Semantics:

**Arithmetic Expressions**

- \( \delta : A × X → Z \) (integer expressions)
- \( \delta [n] = n \)
- \( \delta [X] = m(\rho(X)) \) for \( \xi = (\rho, m) \)
- \( \delta [E_1 \text{ op } E_2] = \) if \( \delta [E_1] \xi \in \mathbb{Z} \) and \( \delta [E_2] \xi \in \mathbb{Z} \)
- Then \( \delta [E_1] \xi \text{ op } \delta [E_2] \xi \) else \( \perp \)

**Boolean Expressions**

- \( \mathcal{B} : B × X → \mathbb{B} \)
- \( \mathcal{B} [true] = true \)
- \( \mathcal{B} [false] = false \)
- \( \mathcal{B} [E_1 < E_2] = \) if \( \delta [E_1] \xi \in \mathbb{Z} \) and \( \delta [E_2] \xi \in \mathbb{Z} \) then \( \delta [E_1] \xi < \delta [E_2] \xi \) else \( \perp \)
- \( \mathcal{B} [true] = \) if \( \mathcal{B} [B] \xi \in \mathbb{Z} \) then \( \mathcal{B} [B] \xi \) else \( \perp \)
- \( \mathcal{B} [B_1 ∪ B_2] = \) if \( \mathcal{B} [B_1] \xi \in \mathbb{Z} \) and \( \mathcal{B} [B_2] \xi \in \mathbb{Z} \) then \( \mathcal{B} [B_1] \xi \) else \( \mathcal{B} [B_2] \xi \)

**Actions**

- \( α : A × X × X \)
- \( α [skip] = \xi \)
- \( α [X := D] = (\rho, m') \) where \( \xi = (\rho, m) \) and \( m' = \mu(\rho(X) \leftarrow \delta) \), and \( \delta = \begin{cases} D & \text{if } D \in A \times ϕ(L) \\ \perp & \text{if } D \in E \end{cases} \)
- \( α [assign(L', X)] = (\rho', m) \) where \( \xi = (\rho, m) \) and \( \rho' = \rho[X → L'] \)

Figure 1. A simple programming language.

**Labels:**

- \( \text{lab}[L : A → L'] = L \)
- \( \text{lab}[L : B → (L_T, L_F)] = L \)
- \( \text{lab}[P] = \{ \text{lab}[C] | C ∈ P \} \)

**Successors of a command:**

- \( \text{succ}[L : A → L'] = L' \)
- \( \text{succ}[L : B → (L_T, L_F)] = \{ L_T, L_F \} \)

**Action of a command:**

- \( \text{act}[L : A → L'] = A \)

**Variables:**

- \( \text{var}[L_1 : A → L_2] = \text{var}[A] \)
- \( \text{var}[P] = \bigcup_{C ∈ P} \text{var}[C] \)
- \( \text{var}[A] = \{ \text{variables occurring in } A \} \)

**Memory locations used by a program:**

- \( \text{Luse}[L : A → L'] = \text{Luse}[A] \)
- \( \text{Luse}[P] = \bigcup_{C ∈ P} \text{Luse}[C] \)
- \( \text{Luse}[A] = \{ \text{locations occurring in } A \} \cup ρ(\text{var}[A]) \)

**Commands in sequences of program states:**

- \( \text{cmd} \{ (C_1, ξ_1), ..., (C_k, ξ_k) \} = \{ C_1, ..., C_k \} \)

2.2 Abstract Interpretation

The basic idea of abstract interpretation is that program behavior at different levels of abstraction is an approximation of its formal semantics [8, 9]. The (concrete) semantics of a program is computed on the (concrete) domain \((C, ≤_C)\), i.e., a complete lattice which models the values computed by programs. The partial ordering \(≤_C\) models relative precision: \(c_1 ≤_C c_2\) means that \(c_1\) is more precise (concrete) than \(c_2\). Approximation is encoded by an abstract domain \((A, ≤_A)\), i.e., a complete lattice, that represents some approximation properties on concrete objects. Also in the abstract domain the ordering relation \(≤_A\) denotes relative precision.

As usual abstract domains are specified by Galois connections [8, 9]. Two complete lattices \(C\) and \(A\) form a Galois connection \((C, α, γ, A)\), also denoted \(C \xrightarrow{α, γ, A} A\), when the functions \(α : C → A\) and \(γ : A → C\) form an adjunction, namely ∀\(a ∈ A\), ∀\(c ∈ C\) : \(α(c) ≤_A a \Leftrightarrow c ≤_C γ(a)\) where \(α(γ)\) is the left(right) adjoint of \(γ(α)\), α and γ are called, respectively, abstraction and concretization maps. A tuple \((C, α, γ, A)\) is a Galois connection if 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. $(C, \alpha, \gamma, A)$ specifies a Galois insertion, denoted $C \sqcup \sqcup \sqcup \sqcup A$, if each element of $A$ is an abstraction of a concrete element in $C$, namely $(C, \alpha, \gamma, A)$ is a Galois insertion iff $\alpha$ is surjective iff $\gamma$ is injective. Abstract domains can be related to each other w.r.t. their relative degree of precision. We say that an abstraction $\alpha_1 : C \to A_1$ is more concrete than $\alpha_2 : C \to A_2$, i.e., $A_2$ is more abstract than $A_1$, if $\forall c \in C : \gamma_1(\alpha_1(e)) \subseteq \gamma_2(\alpha_2(e))$.

### 2.3 Programming Language

The language we consider is a simple extension of the one introduced by Cousot and Cousot [10], the main difference being the ability of programs to generate code dynamically (this facility is added to accommodate certain kinds of malware obfuscations where the payload is unpacked and decrypted at runtime). The syntax and semantics of our language are given in Figure 1. Given a set $S$, we use $S_{\bot}$ to denote the set $S \cup \{\bot\}$, where $\bot$ denotes an undefined value.\footnote{We abuse notation and use $\bot$ to denote undefined values of different types, since the type of an undefined value is usually clear from the context.} Commands can be either conditional or unconditional. A conditional command at a label $L$ has the form '$L : B \to \{L_r, L_f\}$', where $B$ is a Boolean expression and $L_r$ (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 (\bot), or it can store either an integer or (appropriately encoded) pair $(A, S) \in A \times S$. 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 $P_{\text{init}}$ denotes the initial set of commands, then $P = \text{cmd} \left( \bigcup_{C \in P_{\text{init}}} \left( \bigcup_{\xi, \varsigma} \mathcal{E}^{\text{ext}}(C, \xi, \varsigma) \right) \right)$, where we extend $\mathcal{E}$ to a set of program states, $\mathcal{E}(S) = \bigcup_{\sigma \in S} \mathcal{E}(\sigma)$. Since each command explicitly mentions its successors, the program need not to maintain an explicit sequence of commands. This definition allows us to represent programs that generate code dynamically.

An environment $\rho \in E$ maps variables in $\text{dom}(\rho) \subseteq X$ to memory locations $L \subseteq X$. Given a program $P$ we denote with $\mathcal{E}(P)$ its environments, i.e. if $\rho \in \mathcal{E}(P)$ then $\text{dom}(\rho) = \text{var}(P)$. Let $\rho[X \rightarrow I]$ denote environment $\rho$ where label $L$ is assigned to variable $X$. The memory is represented as a function $m : L \to Z \cup (A \times S)$. 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 $\text{dom}(m) = \text{Luse}[P]$. 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 $X = X \times M$. A program state is a pair $(C, \xi)$ where $C$ is the next command that has to be executed in the execution context $\xi \in C \times X$ denotes the set of all possible states. Given a state $s \in \Sigma$, the semantic function $\mathcal{E}(s)$ gives the set of possible successor states of $s$; in other words, $\mathcal{E} : \Sigma \to \mathcal{S}$ defines the transition relation between states. Let $\Sigma(P) = P \times X(P)$ be the set of states of a program $P$, then we can specify the transition relation $\mathcal{E} : \Sigma(P) \to \mathcal{S}(P)$ on program $P$ as:

$$\mathcal{E}(P, C, \xi) = \left\{ (C', \xi') : (C', \xi') \in \mathcal{E}(C, \xi) \land C' \in P, \text{ and } \xi, \xi' \in X(P) \right\}.$$ Let $A^\ast$ denote the Kleene closure of a set $A$, i.e., the set of finite sequences over $A$. A trace $\sigma \in \Sigma^\ast$ is a sequence of states $s_1...s_n$ of length $|\sigma| \geq 0$ such that for all $i \in [1, n]$, $s_i \in \mathcal{E}(s_{i-1})$. The finite partial traces semantics $\mathcal{E}(P) \subseteq \Sigma^\ast$ of program $P$ is the least fixpoint of the function $F$:

$$F[P](T) = \Sigma(P) \cup \{ ss' | s \in \mathcal{E}[P] \ (s), \ s' \in T \}.$$

where $T$ is a set of traces, namely $\mathcal{E}[P] = \text{lfp}_S F[P]$. 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 $f : S \to$ to denote the restriction of function $f$ to elements not in $S$, namely $A \setminus S$.

### 3. Semantics-Based Malware Detection

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

The process of considering only a portion of program semantics can be seen as an abstraction. A subset of a program $P$’s labels (i.e., commands) $\text{lab}_r[P] \subseteq \text{lab}[P]$ characterizes a restriction of program $P$. In particular, let $\text{var}_r[P]$ and $\text{Luse}_r[P]$ denote, respectively, the set of variables occurring in the restriction and the set of memory locations used:

$$\text{var}_r[P] = \bigcup \{ \text{var}[C] : \text{lab}[C] \in \text{lab}_r[P] \}$$

$$\text{Luse}_r[P] = \bigcup \{ \text{Luse}[C] : \text{lab}[C] \in \text{lab}_r[P] \}.$$

The set of labels $\text{lab}_r[P]$ induces a restriction on environment and memory maps. Given $\rho \in \mathcal{E}(P)$ and $m \in \mathcal{M}(P)$, let $\rho^r = \rho_{\text{var}_r[P]}$ and $m^r = m_{\text{Luse}_r[P]}$ denote the restricted set of environments and memories induced by the restricted set of labels $\text{lab}_r[P]$. Let $\Sigma_r = \{ (C, (\rho^r, m^r)) : \text{lab}[C] \in \text{lab}_r[P] \}$ be the set of restricted program states. Define $\alpha_r : \Sigma_r \to \Sigma_r$ that propagates restriction $\text{lab}_r[P]$ on a given a trace $\sigma = (C_1, (\rho_1, m_1))\sigma'$:

$$\alpha_r(\sigma) = \left\{ \begin{array}{ll} \epsilon & \text{if } \sigma = \epsilon \\ \alpha_r(\sigma') & \text{if } \sigma = (C_1, \xi), m_1 \end{array} \right..$$

Given a function $f : A \to B$ we denote, by a slight abuse of notation, its pointwise extension on powerset as $f : \mathcal{E}(A) \to \mathcal{E}(B)$, where $f(X) = \{ f(x) | x \in X \}$. Note that the pointwise extension is additive. Therefore, the function $\alpha_r : \mathcal{E}(\Sigma_r) \to \mathcal{E}(\Sigma_r)$ is an abstraction that discards information outside the restriction $\text{lab}_r[P]$. Moreover $\alpha_r$ is surjective and defines a Galois insertion $\langle \mathcal{E}(\Sigma_r), \subseteq \rangle \supseteq \langle \mathcal{E}(\Sigma_r), \subseteq \rangle$. Let $\alpha_r(\mathcal{E}[P])$ be the restricted semantics of program $P$.

Observe that program behavior is expressed by the effects that program execution has on environment and memory. Consider a transformation $\alpha_r : \Sigma_r \to \Sigma_r^\ast$ that, given a trace $\sigma$, discards from $\sigma$ all information about the commands that are executed, retaining only information about changes to the environment and effects on memory during execution:

$$\alpha_r(\sigma) = \left\{ \begin{array}{ll} \epsilon & \text{if } \sigma = \epsilon \\ \xi_1 \alpha_r(\sigma') & \text{if } \sigma = (C_1, \xi_1)\sigma' \end{array} \right..$$
Given an abstraction $\alpha$, let us consider the fragment of program $P$ that computes the factorial of variable $X$ and its obfuscation $O(P)$ obtained inserting commands that do not affect the execution context (at labels $L_2$ and $L_{F+1}$ in the example).

4. Obfuscated Malware

To prevent detection malware writers usually obfuscate the malicious code. Thus, a robust malware detector needs to handle possibly obfuscated versions of a malware. While obfuscation may modify the original code, the obfuscated code has to be equivalent (up to some notion of equivalence) to the original one. Given an obfuscating transformation $O : P \rightarrow P$ on programs and a suitable abstract domain $A$, we define an abstraction $\alpha : \varphi(X^*) \rightarrow A$ that discards the details changed by the obfuscation while preserving the maliciousness of the program. This, 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 abstraction restriction that matches the behavior up to $\alpha$, formally:

$$\exists \text{lab}_1 [P] \in \varphi(\text{lab} [P]) : \alpha(\alpha_e(\mathbb{S} [M])) \subseteq \alpha(\alpha_e(\mathbb{S} [P])).$$

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

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

$$P \quad O(P)$$

$\begin{align*}
L_1 & : F := 1 \rightarrow L_2 \\
L_2 & : (X = 1) \rightarrow \{L_T, L_F\} \\
L_F & : X := X - 1 \rightarrow L_{F+1} \\
L_{F+1} & : F := F \times X \rightarrow L_2 \\
L_T & : \ldots
\end{align*}$

$\begin{align*}
L_1 & : F := 1 \rightarrow L_2 \\
L_2 & : F := F \times X \rightarrow L_3 \\
L_3 & : (X = 1) \rightarrow \{L_T, L_F\} \\
L_F & : X := X - 1 \rightarrow L_{F+1} \\
L_{F+1} & : F := F \times X \rightarrow L_3 \\
L_T & : \ldots
\end{align*}$

A suitable abstraction here is the one that observes modifications in the execution context, namely $\alpha_e((p_1, m_1)(p_2, m_2)\ldots(p_n, m_n))$ if $(p_1 = p_2$ and $(m_1 = m_2) and \ldots)$ otherwise.

4.1 Soundness vs Completeness

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

**Definition 5.** A semantic malware detector on $\alpha$ is sound for a set of transformations if and only if:

$$\forall O \in \mathbb{O} : \alpha(\mathbb{S} [O]) \subseteq \alpha(\mathbb{S} [P]).$$

**Theorem 1.** If $\alpha$ is preserved by the transformation $O$ then the semantic malware detector on $\alpha$ is complete for $O$.

However, the preservation condition of Theorem 1 is too weak to imply soundness of the semantic malware detector. As an example, let us consider the abstraction $\alpha_T = \lambda X . T$ that loses all information. It is clear that $\alpha_T$ is preserved by every obfuscating transformation, and the semantic malware detector on $\alpha_T$ classifies every program as infected by every malware. Unfortunately, we do not have a result analogous to Theorem 1 that provides a property of $\alpha$ that characterizes soundness of the semantic malware detector. However, given an abstraction $\alpha$, we characterize the set of transformations for which $\alpha$ is sound.

**Theorem 2.** Given an abstraction $\alpha$, consider the set of transformations such that: $\forall P \in \mathbb{P}$,

$$\alpha(\mathbb{S} [T]) \subseteq \alpha(\mathbb{S} [P]).$$

Then, a semantic malware detector on $\alpha$ is sound for $O$.

4.2 A Semantic Classification of Obfuscations

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

4.2.1 Conservative Obfuscations

An obfuscation $O : P \rightarrow P$ is a conservative obfuscation if there is a malware trace $\sigma \in \mathbb{S} [P]$ such that: $\alpha_c(\sigma) \leq \alpha_c(\delta)$. Let $\mathbb{O}_c$ denote the set of conservative obfuscating transformations.

When dealing with conservative obfuscations we have that a trace $\delta$ of a program $P$ presents a malicious behavior $M$, if there is a malware trace $\sigma \in \mathbb{S} [M]$ whose environment-memory evolution is contained in the environment-memory evolution of $\delta$, namely if $\alpha_e(\sigma) \leq \alpha_e(\delta)$. Let us define the abstraction $\alpha_e : \varphi(X^*) \rightarrow \varphi(X^*)$ 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 substraces of \( s \):

\[
\alpha_t[S](s) = S \cap \text{SubSeq}(s)
\]

where \( \text{SubSeq}(s) = \{t \mid t \preceq s \} \) denotes the set of all subsequences of \( s \). For any \( S \in \wp(\mathcal{X}^*) \), the additive function \( \alpha_t[S] \) defines a Galois connection: \( \langle \wp(\mathcal{X}^*) \rangle \rightarrow \langle \wp(\mathcal{X}^*) \rangle \). The abstraction \( \alpha_t \) turns out to be a suitable approximation when dealing with conservative obfuscations. In fact the semantic malware detector on \( \alpha_t[\wp(\mathcal{M})] \) is complete and sound for the class of conservative obfuscations \( \mathcal{O}_c \).

**Theorem 3.** Considering a vanilla malware \( M \) we have that \( \exists \bar{O}_c \in \mathcal{O}_c \) such that \( \bar{O}_c(M) \rightarrow P \) iff \( \exists \bar{lab} \cdot [P] \in \wp(\bar{lab}[P]) \) such that:

\[
\alpha_t[\wp(\mathcal{M})](\alpha_t[\wp(\mathcal{M})]) \subseteq \alpha_t[\wp(\mathcal{M})](\alpha_t[\wp(\mathcal{M})]).
\]

Many obfuscating transformations commonly used by malware writers are conservative; a partial list of such conservative obfuscations is given below. 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.

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

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

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

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

**Lemma 1.** Given \( \bar{O}_1, \bar{O}_2 \in \mathcal{O}_c \) then \( \bar{O}_1 \circ \bar{O}_2 \in \mathcal{O}_c \).

**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 \( L \), the decryption of memory locations starting at location \( A \) and then executes the decrypted instructions. Let \( \bar{O}_c(M) \) be an abstraction of the malware:

\[
M
\]

<table>
<thead>
<tr>
<th>( \bar{O}_c(M) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{O}_1 )</td>
</tr>
<tr>
<td>( \bar{L}_1 : \text{assign}(L_B, B) \rightarrow L_2 )</td>
</tr>
<tr>
<td>( \bar{L}_2 : \text{assign}(L_A, A) \rightarrow L_3 )</td>
</tr>
<tr>
<td>( \bar{L}_3 : \text{cond}(A) \rightarrow {L_4, L_F} )</td>
</tr>
<tr>
<td>( \bar{L}_4 : \text{assign}(L_A, A) \rightarrow L_5 )</td>
</tr>
<tr>
<td>( \bar{L}_5 : \text{assign}(\bar{x}_2, B), B \rightarrow L_T )</td>
</tr>
<tr>
<td>( \bar{L}_T : \text{assign}(\bar{x}_2, A), A \rightarrow L_C )</td>
</tr>
<tr>
<td>( \bar{L}_F : \text{skip} \rightarrow L_B )</td>
</tr>
</tbody>
</table>

Given a variable \( X \), the semantics of \( \bar{x}_2(X) \) is the label expression by \( \bar{x}_2(m_1(\rho(X))) \), in particular \( \bar{x}_2(n) = \bar{n} \), while \( \bar{x}_2(A, S) = S \).

Given a variable \( X \), let \( \text{Dec}(X) \) denote the execution of a set of commands that decrypts the value stored in the memory location \( \rho(X) \). The obfuscations are as follows:

- \( L_2 : \text{skip} \rightarrow L_3 \) and \( L_3 : \text{skip} \rightarrow L_4 \) inserted by code reordering;
- \( L_N : X := X + 3 \rightarrow L_N \) and \( L_{N \_} : X := X - 3 \rightarrow L_T \) represent semantic NOP insertion, and \( L_O : P^T \rightarrow \{L_N, L_{N \_}\} \) true opaque predicate insertion.

It can be shown that \( \alpha_t[\wp(\mathcal{M})](\alpha_t[\wp(\mathcal{M})]) = \alpha_t[\wp(\mathcal{M})](\alpha_t[\wp(\mathcal{M})]), \) i.e., our semantics-based approach is able to see through the obfuscations and identify \( \bar{O}_c(M) \) as matching the malware \( M \).

**4.2.2 Non-Conservative Obfuscations**

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

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

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

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

**Variable Renaming** Variable renaming is a simple obfuscating transformation, often used to prevent signature matching, that replaces the names of variables with some different names. Assuming that every environment function associates variable \( V_e \) to memory location \( L \), allows us to reason on variable renaming also in the case of compiled code, where variable names have disappeared. Let \( \bar{O}_c : \mathcal{P} \times \Pi \rightarrow \mathcal{P} \) denote the obfuscating transformation that, given a program \( P \), renames its variables according to a mapping \( \pi \in \Pi \), where \( \pi : \text{var} \rightarrow \text{Names} \) is a bijective function that relates the name of each program variable to its new name.

\[
\bar{O}_c(P, \pi) = \left\{ \begin{array}{l}
\bar{C} \mid \exists C^* \in \mathcal{P} : \text{lab}[C] = \text{lab}[C^*] \\
\text{suc}[C] = \text{suc}[C^*] \\
\text{act}[C] = \text{act}[C^*] \end{array} \right\}
\]

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 \( \alpha_t \) of traces, thus it is interesting to observe that:

\[
\alpha_t[\wp(\bar{O}_c(P, \pi))] = \alpha_t[\pi(\alpha_t[\wp(P)]]
\]

where \( \alpha_t : \Pi \rightarrow (\mathcal{X}^* \rightarrow \mathcal{X}^*) \) is defined as:

\[
\alpha_t[\pi](\rho_1, m_1) \ldots (\rho_n, m_n) = (\pi(\rho_1), m_1) \ldots (\pi(\rho_n), m_n).
\]

In order to deal with variable renaming obfuscation we introduce the notion of **canonical variable renaming** \( \bar{\pi} \). The idea of canonical mappings is that there exists a renaming \( \pi : \text{var} \rightarrow \text{var} \) that transforms program \( P \) into program \( Q \), namely such that \( \bar{O}_c(P, \pi) = Q \), if \( \alpha_t[\bar{\pi}(\alpha_t[\wp(Q)]) = \alpha_t[\bar{\pi}(\alpha_t[\wp(P)])] \). This means that a program \( Q \) is a renamed version of program \( P \).
if $Q$ and $P$ are indistinguishable after canonical renaming. In the following we define a possible canonical renaming for the variables of a given program.

Let $\{V_i\}_{i \in \mathbb{N}}$ be a set of canonical variable names. The set $L$ 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 $\leq_S$ over traces $\Sigma^*$ where, given $\sigma, \delta \in \Sigma^*$, $\sigma \leq_S \delta$ if $|\sigma| \leq |\delta|$ or $|\sigma| = |\delta|$ and $\text{lab}(\sigma_1) \text{lab}(\sigma_2) \ldots \text{lab}(\sigma_n) \leq_L \text{lab}(\delta_1) \text{lab}(\delta_2) \ldots \text{lab}(\delta_n)$, where $\text{lab}(\rho, C) = \text{lab}(C)$. It is clear that, given a program $P$, the ordering $\leq_S$ on its traces induces an order on the set $Z = \alpha_e(\mathcal{S}[P])$ of its environment-memory traces, i.e., given $\sigma, \delta \in \mathcal{S}[P]$, $\sigma \leq_S \delta \iff \alpha_e(\sigma) \leq \alpha_e(\delta)$. By definition, the set of variables assigned in $Z$ is exactly $\text{var}[P]$. Therefore, a canonical renaming $\pi : \text{var}[P] \to \{V_i\}_{i \in \mathbb{N}}$ is such that $\alpha_e(\mathcal{S}[\pi(P)]) = \alpha_e(\mathcal{S}[Z])$. Let $Z$ denote the list of environment-memory traces of $Z = \alpha_e(\mathcal{S}[P])$ ordered following the order defined above. Let $B$ be a list, then $\text{hd}(B)$ returns the first element of the list, $\text{tl}(B)$ returns list 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 uninitialized environment $\rho_{\text{initial}} = \lambda X.\bot$, and that each command assigns at most one variable. Let $\text{def}(\rho)$ denote the set of variables that have defined (i.e., non-$\bot$) values in an environment $\rho$. This means that considering $s \in X^*$ we have that $\text{def}(\rho_{s-1}) \subseteq \text{def}(\rho_s)$, and if $\text{def}(\rho_{s-1}) \subseteq \text{def}(\rho_s)$ then $\text{def}(\rho_s) = \text{def}(\rho_{s-1}) \cup \{X\}$ where $X \in X$ is the new variable assigned to memory location $\rho_{s-1}$. Given $s \in X^*$, let us define $\text{List}(s)$ as the list of variables in $s$ ordered according to their assignment time. Formally, let $s = (\rho_1, m_1)(\rho_2, m_2) \ldots (\rho_n, m_n) = (\rho_1, m_1) \ldots (\rho_{n-1}, m_{n-1})c(\rho_n, m_n)$, and $c \in \text{List}(s)$.

$\text{List}(s) = \{\epsilon\}$ if $s = \epsilon$
$\{X : \text{List}(s')\}$ if $\text{def}(s_2) \setminus \text{def}(s_1) = \{X\}$
$\{\text{List}(s')\}$ if $\text{def}(s_2) \setminus \text{def}(s_1) = \emptyset$

Given $Z = \alpha_e(\mathcal{S}[P])$ we rename its variables following the canonical renaming $\pi_P : \text{var}[P] \to \{V_i\}_{i \in \mathbb{N}}$ that associates the new canonical name $V_i$ to the variable of $P$ in the $i$-th position in the list $\text{Name}[Z]$ defined in Algorithm 1. Thus, the canonical renaming $\pi_P : \text{var}[P] \to \{V_i\}_{i \in \mathbb{N}}$ is defined as follows:

$\pi_P(X) = \text{rename}(Z[i]) = X$

The following result is necessary to prove that the mapping $\pi_P$ defined in Equation (1) is a canonical renaming.

Algorithm 1: Canonical renaming of variables.

**Lemma 2.** Given two programs $P, Q \in P$ let $Z = \alpha_e(\mathcal{S}[P])$ and $Y = \alpha_e(\mathcal{S}[Q])$. The following holds:

- $\alpha_e(\pi_P[Z]) = \alpha_e(\pi_Q[Y]) \iff \exists \pi : \text{var}[P] \to \text{var}[Q] : \alpha_e(\pi[Z]) = Y$
- $(\exists \pi : \text{var}[P] \to \text{var}[Q]) \iff (\alpha_e(\pi[Z]) = Y) \text{ and } (\alpha_e(\pi[Z]) = Y) \iff \alpha_e(\pi_P[Z]) = \alpha_e(\pi_Q[Y])$

Let $\Pi$ denote a set of canonical variable renaming, the additive function $\alpha_e : \Pi \to (\rho_X^*, \pi_P(\rho_X^*))$, where $\rho_X$ 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:

$\langle \rho(X^*), \subseteq \rangle_{\alpha_e(\Pi)} \approx \langle \rho(\rho_X^*), \subseteq \rangle_{\alpha_e(\mathcal{S}[P])}$.

The following result, where $\pi_P$ and $\pi_Q$ denote respectively the canonical rename of the malware variables and of restricted program variables, shows that the semantic malware detector on $\alpha_e(\Pi)$ is complete and sound for variable renaming.

**Theorem 4.** $\exists \pi : \alpha_e(\Pi) \wedge P$ iff $\exists \pi : \alpha_e(\mathcal{S}[P]) : \alpha_e(\pi_M)[(\alpha_e(\mathcal{S}[M]) \subseteq \alpha_e(\pi_{\mathcal{P}})[(\alpha_e(\mathcal{S}[\mathcal{P}])])$.

**4.3 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 $\alpha_1$ and $\alpha_2$, that are complete (sound) respectively for obfuscations $O_1$ and $O_2$, and the abstraction that is complete (sound) for their compositions, i.e. for $O_1 \circ O_2$, $O_2 \circ O_1$, we have obtained the following result.

**Theorem 5.** Given two abstractions $\alpha_1$ and $\alpha_2$ and two obfuscations $O_1$ and $O_2$ then:

1. if the semantic malware detector on $\alpha_1$ is complete for $O_1$, the semantic malware detector on $\alpha_2$ is complete for $O_2$, 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 $O_1 \circ O_2$.
2. if the semantic malware detector on $\alpha_1$ is sound for $O_1$, the semantic malware detector on $\alpha_2$ is sound for $O_2$, and $\alpha_1 \circ \alpha_2$ is sound for $O_1 \circ O_2$.

Thus, in order to propagate completeness through composition $O_1 \circ O_2$ and $O_2 \circ O_1$, the corresponding abstractions have to be independent. On the other side, in order to propagate soundness through composition $O_1 \circ O_2$, the abstraction $\alpha_1$, corresponding to the last applied obfuscation, has to be an order-embedding, namely $\alpha_1$ has to be both order-preserving and order-reflecting, i.e., $\alpha_1(X) \subseteq \alpha_1(Y) \iff X \subseteq Y$. Observe that, when composing a non-conservative obfuscation $O_1$, for which the semantic malware detector on $\alpha_0$ is complete, with a conservative obfuscation $O_c$, the commutation condition of point 1 is satisfied if and only if $(\alpha_e(\sigma) \leq \alpha_c(\bar{\sigma})) \iff \alpha_0(\alpha_c(\bar{\sigma})) \leq \alpha_0(\alpha_c(\bar{\sigma}))$.

**Example 3.** Let us consider $\alpha_e(\mathcal{S}[\mathcal{P}(\alpha_1)])$ obtained by obfuscating the portion of malware $M$ in Example 2 through variable renaming and some conservative obfuscations:
A program

Int

mantic malware detector on

∀

O

α

that

M

5. Further Malware Abstractions

Definition 4 characterizes the presence of malware M in a program P as the existence of a restriction lab. P ∈ ϕ(lab[P]) such that αe(Θ[M]) ⊆ αe(Θ[P])). This means that program P is infected by malware M if P matches all malware behaviors. This notion of malware infection can be weakened in two different ways. First, we can abstract the malware traces eliminating the states that are not relevant to determine maliciousness, and then check if program P matches this simplified behavior. Second, we can require program P to match a proper subset of malicious behaviors. Furthermore these two notions of malware infection can be combined by requiring program P to match the interesting states of the interesting behaviors of the malware. It is clear that a deeper understanding of the malware behavior is necessary in order to identify both the set of interesting states and the set of interesting behaviors.

Interesting States. Assume that we have an oracle that, given a malware M, returns the set of its interesting states. These states could be selected based on a security policy, for example, the states could represent the result of network operations. This means that, in order to verify if P is infected by M, we have to check whether the malicious sequences of interesting states are present in P. Let us define the trace transformation αInt(M): ν* → ν* that considers only the interesting states in a given trace σ = σ1σ′:

αInt(M)(σ) = \{ ε if σ = ε

σ′ if σ ∈ Int(M)

otherwise

The following definition characterizes the presence of malware M in terms of its interesting states, i.e., through abstraction αInt(M).

Definition 6. A program P is infected by a vanilla malware M with interesting states Int(M), i.e., M →Int(M) P, if ϕ(lab[P]) ∈ ϕ(lab[P]) such that:

αInt(M)(Θ[M]) ⊆ αInt(M)(Θ(Θ[P])).

Thus we can weaken the standard notion of conservative transformation by saying that O : P → P is conservative w.r.t. Int(M) if ∀α ∈ Θ[M], ∀β ∈ Θ[O(P)] such that αInt(M)(α) = αInt(M)(β).

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

Theorem 6. Let Int(M) be the set of interesting states of a vanilla malware M, then there exists an obfuscation O conservative w.r.t. Int(M) such that O(M) ←Int(M) P iff ∃lab. [P] ∈ ϕ(Θ[P]) such that:

αInt(M)(Θ[P]) ⊆ αInt(M)(Θ(Θ[P])).

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

Example 4. Let us consider the malware M and its obfuscation O(M) obtained by reordering independent instructions.

M

O(M)

L1 : A1 → L2

L2 : A2 → L3

L3 : A3 → L4

L4 : A4 → L5

L5 : A5 → L6

In the above example A2 and A5 are independent, meaning that αe[A1] (αe[A3] (ρ, m)) = αe[A1] (αe[A2] (ρ, m))). Considering malware M, we have the trace σ = σ1σ2σ3σ4σ5 where:

- σ1 = (L1 : A1 → L2, m),
- σ2 = (L2 : A2 → L3, m),
- σ3 = (L3 : A3 → L4, m),
- σ4 = (L4 : A4 → L5, m),
- σ5 = (L5 : A5 → L6, m).

Let Int(M) = {σ1, σ2, σ3, σ4, σ5}. Then αInt(M)(σ) = σ1σ2σ3σ4σ5 as well as αInt(M)(σ) = δ1δ2δ3δ4δ5, which concludes the example. It is obvious that δ1 = σ1, moreover δ5 = σ5 follows from the independence of A2 and A5.

Interesting Behaviors. Assume we have an oracle that given a malware M returns the set T ⊆ Θ[M] of its behaviors that characterize the maliciousness of M. Thus, in order to verify if P is infected by M, we check whether program P matches the malicious behaviors T. The following definition characterizes the presence of malware M in terms of its interesting behaviors T.

Definition 7. A program P is infected by a vanilla malware M with interesting behaviors T ⊆ Θ[M], i.e., M →P if:

∃lab. [P] ∈ ϕ(lab[P]) : αe(T) ⊆ αe(Θ[P]).

It is interesting to observe that, when program infection is characterized by Definition 7, all the results obtained in Section 4 still hold if we replace Θ[M] 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 interesting sequences of states obtained abstracting the interesting behaviors of the malware, i.e., ∃lab. [P] ∈ ϕ(lab[P]) : αe(αInt(M)(T)) ⊆ αe(αInt(M)(Θ[P])).

To conclude, we present a matching relation based on (interesting) program actions rather than environment-memory evolutions. In this case we consider the syntactic information contained in program states. The main difference with purely syntactic approaches is the ability of observing actions in their execution order and not in the order in which they appear in the code.
A program $\alpha$ identifies a one-to-one map from template nodes in the path $\sigma$ with bad actions $\tau$. Assume we have an oracle that given a malware $M$ characterized in terms of the execution of a sequence of bad actions $\tau$, returns the sequence of malicious actions executed by each malware infected by malware $M$. Sometimes a malicious behavior can be translated into Interesting Actions.

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

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>Completeness of $A_{MD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code reordering</td>
<td>Yes</td>
</tr>
<tr>
<td>Semantic-nop insertion</td>
<td>Yes</td>
</tr>
<tr>
<td>Substitution of equivalent commands</td>
<td>No</td>
</tr>
<tr>
<td>Variable renaming</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Interesting Actions.

Sometimes a malicious behavior can be characterized in terms of the execution of a sequence of bad actions. Assume we have an oracle that given a malware $M$ returns the set $Bad \subseteq act[M]$ 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 sequence of bad actions of the malware matches the ones of the program.

**Definition 8.** A program $P$ is infected by a vanilla malware $M$ with bad actions $Bad$, i.e., $M \leftrightarrow_{Bad} P$ if:

$$\exists lab \subseteq P \subseteq \varphi(lab[P]): a_\alpha(\varphi([M])) \subseteq a_\alpha(\varphi([P]))$$

Where, given the set $Bad \subseteq act[M]$ of bad actions, the abstraction $a_\alpha$ keeps track of the sequence of malicious actions executed in each trace. Formally, given a trace $\sigma = \sigma_1 \sigma_2$:

$$a_\alpha(\sigma) = \begin{cases} 
\epsilon & \text{if } \epsilon = \epsilon \\
A_1 a_\alpha(\sigma_1) & \text{if } A_1 \in \text{Bad} \\
\alpha_\sigma(\sigma_1) & \text{otherwise}
\end{cases}$$

Even if this abstraction considers syntactic information (program actions), it is able to deal with some sort of obfuscations. In fact considering the sequence of malicious actions in a trace it observes actions in their execution order, and not in the order in which they are written in the code. Thus, ignoring for example unconditional jumps, it is able to deal with code reordering.

### 6. Case Study: Completeness of Semantics-Aware Malware Detector $A_{MD}$

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

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

### Description of the Algorithm

The semantics-aware malware detection algorithm $A_{MD}$ 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 $A_{MD}$.

The algorithm $A_{MD}$ attempts to find a subset of the program $P$ that matches the commands in the malware $M$, possibly after renaming of variables and locations used in the subset of $P$. Furthermore, $A_{MD}$ 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 $C \in G$, 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 $C = L_1 : A \rightarrow S$, where $S \in \varphi(L)$, and for each label $L_2 \in S$, create a CFG edge $(v_{L_1}, v_{L_2})$.

Consider a path $\theta$ through the CFG from node $v_{l_1}$ to node $v_{v_k}$, $\theta = v_{l_1} \rightarrow \ldots \rightarrow v_{v_k}$. 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 $\varphi^*$ $P[\theta]$ $\varphi(C_1, (\rho, m))$, by extending the transition relation $\varphi$ to a set of states, such that $\varphi: \varphi(\Sigma) \rightarrow \varphi(\Sigma)$. Let us define the following basic functions:

$$\text{mem} \left(\{(C,(\rho,m))\}\right) = m$$
$$\text{env} \left(\{(C,(\rho,m))\}\right) = \rho$$

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

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

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 $\nu_\theta: X^T \times V^T \rightarrow X^P$ variables at a template node to variables at the corresponding program node, such that when renaming the variables in the template command $C[n^T]$ according to the map $\nu_\theta$, we obtain the program command $C[n^P] = C[n^T][X/\nu_\theta(X,n^T)]$.

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

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

For each pair of template nodes $m^T, n^T$ on the path $\theta$, and for each template variable $x^T$ defined in action $C_n^T[n^T]$, let $\lambda$ be a program path $\mu_{v_1^T} \rightarrow \ldots \rightarrow \mu_{v_k^T}$, where $m^T \rightarrow v_1^T \rightarrow \ldots \rightarrow v_k^T \rightarrow n^T$ is part of the path $\theta$ in the template CFG. $\lambda$ is therefore a program path connecting the program CFG node corresponding to $m^T$ with the program CFG node corresponding to $n^T$. We denote by $T[\theta] = \{C[m^T], C_1^T, \ldots, C_k^T, C[n^T]\}$ the sequence of commands corresponding to the template path $\theta$.

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

$$\forall \rho \in \mathcal{E}, \forall m \in M, \forall s \in \varphi^* P[\lambda] \left( \mu_\theta (v_{C_n^T}), (\rho, m) \right) :$$

$$\left[ \nu_\theta \,\left[ x^T, v_{C_n^T} \right] \right] (\rho, m) =$$

$$\left[ \nu_\theta \,\left[ x^T, v_{C_n^T} \right] \right] (\text{env}[s], \text{mem}[s]).$$

This check is implemented in $A_{MD}$ 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 $\psi$.

The semantics-aware malware detector $A_{MD}$ makes use of two oracles, $OR_{CFG}$ and $OR_{SNop}$, described in Table 2. Thus $A_{MD} = D^{OR}$, for the set of oracles $\mathcal{O} = \{OR_{CFG}, OR_{SNop}\}$. Our goal is then to verify whether $A_{MD}$ is $OR$-complete with respect to
A semantics-aware malware detector is complete if the oracles used by the semantics-aware malware detection algorithm $A_{MD}$ are the semantics-aware malware detector in terms of its effect on the trace semantics of a program $P$.

**Definition 10 (Semantics-Aware Malware Detection).** A program $P$ is infected by a vanilla malware, i.e., $M \leftarrow P$, if:

$$\exists \text{lab}_r \in \varphi(\text{lab}(P)), \gamma_1 \geq 1 : \alpha_{\text{Act}}(\alpha_{\text{SAMD}}(\mathcal{S}[M], \{\pi_1, \gamma_1\})) \subseteq \alpha_{\text{Act}}(\alpha_{\text{SAMD}}(\alpha_r(\mathcal{S}[P]), \{\pi_1, \gamma_1\})).$$

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

**Proposition 2.** A semantics-aware malware detector is complete on $\alpha_{\text{SAMD}}$ w.r.t. the code-reordering obfuscation $O_{J}$:

$$O_{J}(M) \rightarrow P \Rightarrow \exists \text{lab}_r \in \varphi(\text{lab}(P)), \gamma_1 \geq 1 : \alpha_{\text{Act}}(\alpha_{\text{SAMD}}(\mathcal{S}[M], \{\pi_1, \gamma_1\})) \subseteq \alpha_{\text{Act}}(\alpha_{\text{SAMD}}(\alpha_r(\mathcal{S}[P]), \{\pi_1, \gamma_1\})).$$

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

**Proposition 3.** A semantics-aware malware detector is complete on $\alpha_{\text{SAMD}}$ w.r.t. the variable-renaming obfuscation $O_{N}$.

**Proposition 4.** A semantics-aware malware detector is complete on $\alpha_{\text{SAMD}}$ w.r.t. the semantic-nop insertion obfuscation $O_{N}$.

Additionally, $D_{Tr}$ is $\mathcal{O}$-complete on $\alpha_{\text{SAMD}}$ 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 $\mathcal{O}$-complete w.r.t. all conservative obfuscations, as the following result illustrates.

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

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 $O_{T}$ that substitutes the action of at least one command for each path through the malware (i.e., $\mathcal{S}[M] \cap \mathcal{S}[O_{T}(M)] = \emptyset$). 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 $P = O(M)$, will defeat the detector.

7. Related Work

There is a considerable body of literature on existing techniques for malware detection: Szor gives an excellent summary [26].

Code obfuscation has been extensively studied in the context of protecting intellectual property. The goal of these techniques is to make reverse engineering of code harder [3, 6, 7, 11, 12, 21]. Cryptographers are also pursuing research on the question of possibility of obfuscation [1, 14, 28]. To our knowledge, we are not aware of existing research on formal approaches to obfuscation in the context of malware detection.

8. Conclusions and Future Work

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

Given an obfuscating transformation $O$, we assumed that the abstraction $O_\alpha$ is provided by the malware detector designer. We are currently investigating how to design a systematic (ideally automatic) methodology for deriving an abstraction $O_\alpha$ that leads to a sound and complete semantic malware detector. We observed that if the abstraction $O_\alpha$ is preserved by the obfuscation $O$ 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.

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—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 [18]. Observe that the abstraction $\alpha$, actually defines a set of program traces that are equivalent up to $\mathfrak{O}$. In model checking, sets of program traces are represented by formulae of some linear/branching temporal logic. Hence, we aim at defining a temporal logic whose formulae are able to express normal forms of obfuscations together with operators for composing them. This would allow to use standard model checking algorithms to detect 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.

Acknowledgments

We would like to thank Roberto Giacobazzi and the anonymous referees for their constructive comments and suggestions that helped us in improving this work.

References


