Published in Proceedings of the 34^{th} ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2007), pages ??-??, January 17–19, 2007, Nice, France.

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 and Reasoning about Programs. Mechanical verification. [Malware Detection]

General Terms Security, Languages, Theory, Verification

* The work of M. Dalla Preda was partially supported by the MUR project "InterAbstract" and by the FIRB project "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.

POPL'07 January 17–19, 2007, Nice, France.

Reprinted from POPL'07, Proceedings of the 34^{th} ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, January 17–19, 2007, Nice, France., pp. 1–12.

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 a program, packing or encryption with varying keys [13, 24] can effectively hide the relevant data. *Therefore, an important require-*

ment 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 $\mathcal{A}_{MD}$ proposed by Christodorescu *et al.* [4] is complete with respect to some common obfuscations used by malware writers. The soundness of $\mathcal{A}_{MD}$ was proved in [4].

2. Preliminaries

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

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

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

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

DEFINITION 3 (Soundness and completeness relative to oracles). *A malware detector $D^{\mathcal{OR}}$ is complete with respect to an obfuscation $\mathcal{O}$, if $D^{\mathcal{OR}}$ is complete for that obfuscation $\mathcal{O}$ when all oracles in the set $\mathcal{OR}$ are perfect. Soundness of a detector $D^{\mathcal{OR}}$ 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 $D^{\mathcal{OR}}$ be a detector that uses a set of oracles $\mathcal{OR}$. Assume that we are given a program P and malware M. Let $\mathfrak{S}[\![P]\!]$ and $\mathfrak{S}[\![M]\!]$ be the set of traces corresponding to P and M, respectively. In Section 3 we describe a detector D_{Tr} which works in the semantic world of traces. We then prove that if the oracles in $\mathcal{OR}$ are perfect, the two detectors are equivalent, i.e, for all P and M in $\mathbf{P}$, $D^{\mathcal{OR}}(P, M) = 1$ 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.
   After step 1, we prove the desired property (e.g., completeness) about the trace-based detector D_{Tr}, with respect to the chosen class of obfuscations. In this step, the detector'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 $D^{\mathcal{OR}}$, 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 $D^{\mathcal{OR}}$.

[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.

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:

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

Semantics:

ARITHMETIC EXPRESSIONS

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

Value Domains:

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

BOOLEAN EXPRESSIONS

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

ACTIONS

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

COMMANDS

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

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

Figure 1. A simple programming language.

Labels:

$$
\begin{array}{l}
\mathit{lab}[\![L : A \to L']\!] = L \\
\mathit{lab}[\![L : B \to \{L_T, L_F\}]\!] = L \\
\mathit{lab}[\![P]\!] = \{\mathit{lab}[\![C]\!] \mid C \in P\}
\end{array}
$$

Successors of a command:

$$
\begin{array}{l}
\mathit{suc}[\![L : A \to L']\!] = L' \\
\mathit{suc}[\![L : B \to \{L_T, L_F\}]\!] = \{L_T, L_F\}
\end{array}
$$

Action of a command:

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

Variables:

$$
\begin{array}{l}
\mathit{var}[\![L_1 : A \to L_2]\!] = \mathit{var}[\![A]\!] \\
\mathit{var}[\![P]\!] = \bigcup_{C \in P} \mathit{var}[\![C]\!] \\
\mathit{var}[\![A]\!] = \{\text{variables occuring in } A\}
\end{array}
$$

Memory locations used by a program:

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

Commands in sequences of program states:

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

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

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 $\langle C, \leq_C\rangle$, i.e., a complete lattice which models the values computed by programs. The partial ordering $\leq_C$ models relative precision: $c_1 \leq_C c_2$ 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 [8, 9]. 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 $\alpha : C \to A$ and $\gamma : A \to C$ form an adjunction, namely $\forall a \in A, \forall c \in C : \ \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$ where $\alpha(\gamma)$ is the left(right) adjoint of $\gamma(\alpha)$. α and γ are called, respectively, abstraction and concretization maps. A tuple (C, α, γ, A) is a 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 surjective iff γ is injective. Abstract domains can be related to each other w.r.t. their relative degree of precision. We say that an abstraction $\alpha_1 : C \rightarrow A_1$ is more concrete then $\alpha_2 : C \rightarrow A_2$, i.e., A_2 is more abstract than A_1, if $\forall c \in C : \gamma_1(\alpha_1(c)) \leq_C \gamma_2(\alpha_2(c))$.

2.3 Programming Language

The language we consider is a simple extension of the one introduced by Cousot and Cousot [10], the main difference being the ability of programs to generate code dynamically (this facility is added to accommodate certain kinds of malware obfuscations where the payload is unpacked and decrypted at runtime). The syntax and semantics of our language are given in Figure 1. Given a set S, we use $S_\perp$ to denote the set $S \cup \{\perp\}$, where $\perp$ denotes an undefined value.[2] Commands can be either conditional or unconditional. A conditional command at a label L has the form '$L : B \rightarrow \{L_T, L_F\}$,' 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 '$L : A \rightarrow L_1$,' 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 $(A, S) \in \mathbf{A} \times \wp(\mathbf{L})$. 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} denotes the initial set of commands, then $P = cmd\left[\!\left[\bigcup_{C \in P_{init}} \left(\bigcup_{\xi \in \mathcal{X}} \mathscr{C}^*(C, \xi)\right)\right]\!\right]$, where we extend $\mathscr{C}$ to a set of program states, $\mathscr{C}(S) = \bigcup_{\sigma \in S} \mathscr{C}(\sigma)$. 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* $\rho \in \mathcal{E}$ 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 $m : \mathbf{L} \rightarrow \mathbb{Z}_\perp \cup (\mathbf{A} \times \wp(\mathbf{L}))$. 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, $\mathscr{C} : \Sigma \rightarrow \wp(\Sigma)$ 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 $\mathscr{C}[\![P]\!] : \Sigma(P) \rightarrow \wp(\Sigma(P))$ on program P as:

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

[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.

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

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

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

Finally, we use the following notation. Given a function $f : A \rightarrow B$ 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 $A \smallsetminus S$.

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 $\mathfrak{S}[\![P]\!]$ that "matches" (in a sense that will be made precise) $\mathfrak{S}[\![M]\!]$. In the following we show how program restriction as well as semantic matching are actually appropriate abstractions of program semantics, in the abstract interpretation sense.

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

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

The set of labels $lab_r[\![P]\!]$ induces a restriction on environment and memory maps. Given $\rho \in \mathcal{E}(P)$ and $m \in \mathcal{M}(P)$, let $\rho^r = \rho_{|var_r[\![P]\!]}$ and $m^r = m_{|Luse_r[\![P]\!]}$ denote the restricted set of environments and memories induced by the restricted set of labels $lab_r[\![P]\!]$. Let $\Sigma_r = \big\{(C, (\rho_r, m_r)) \,\big|\, lab[\![C]\!] \in lab_r[\![P]\!]\big\}$ be the set of restrected program states. Define $\alpha_r : \Sigma^* \rightarrow \Sigma^*$ that propagates restriction $lab_r[\![P]\!]$ on a given a trace $\sigma = (C_1, (\rho_1, m_1))\sigma'$:

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

Given a function $f : A \rightarrow B$ we denote, by a slight abuse of notation, its pointwise extension on powerset as $f : \wp(A) \rightarrow \wp(B)$, where $f(X) = \{f(x) | x \in X\}$. Note that the pointwise extension is additive. Therefore, the function $\alpha_r : \wp(\Sigma^*) \rightarrow \wp(\Sigma_r^*)$ is an abstraction that discards information outside the restriction $lab_r[\![P]\!]$. Moreover α_r is surjective and defines a Galois insertion: $\langle\wp(\Sigma^*), \subseteq\rangle \underset{\alpha_r}{\overset{\gamma_r}{\leftrightarrows}} \langle\wp(\Sigma_r^*), \subseteq\rangle$. Let $\alpha_r(\mathfrak{S}[\![P]\!])$ be the *restricted semantics* of program P.

Observe that program behavior is expressed by the effects that program execution has on environment and memory. Consider a transformation $\alpha_e : \Sigma^* \rightarrow \mathcal{X}^*$ that, given a trace σ, discards from σ all information about the commands that are executed, retaining only information about changes to the environment and effects on memory during execution:

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

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

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

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

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

A *semantic malware detector* is a system that verifies the presence of a malware in a program by checking the truth of the inclusion relation of the above definition. In this definition, the program exhibits behaviors that, under the restricted semantics, match all of the behaviors of the vanilla malware. We will later consider a weaker notion of malware infection, where only some (not all) behaviors of the malware are present in the program (Section 5).

4. Obfuscated Malware

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

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

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

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

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	$: ...$

$\mathcal{O}(P)$

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

A suitable abstraction here is the one that observes modifications in the execution context, namely $\alpha((\rho_1, m_1)(\rho_2, m_2)...(\rho_n, m_n))$ returns $\alpha((\rho_2, m_2)...(\rho_n, m_n))$ if $(\rho_1 = \rho_2) \wedge (m_1 = m_2)$ and $(\rho_1, m_1)\alpha((\rho_2, m_2)...(\rho_n, m_n))$ 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 α. 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 α is complete for a set $\mathbb{O}$ of transformations if and only if $\forall \mathcal{O} \in \mathbb{O}$:*

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

A semantic malware detector on α is sound for a set $\mathbb{O}$ of transformations if and only if:

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

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 $\forall P \in \mathbf{P} : \alpha(\alpha_e(\mathfrak{S} [\![P]\!])) = \alpha(\alpha_e(\mathfrak{S} [\![\mathcal{O}(P)]\!]))$.

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

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

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

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

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

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 $s \preceq t$ denote that s is a subsequence of t, i.e., if $s = s_1 s_2 \ldots s_n$ then t is of the form $\ldots s_1 \ldots s_2 \ldots s_n \ldots$.

4.2.1 Conservative Obfuscations

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

When dealing with conservative obfuscations we have that a trace δ of a program P presents a malicious behavior M, if there is a malware trace $\sigma \in \mathfrak{S} [\![M]\!]$ 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 $\alpha_c : \wp(\mathcal{X}^*) \rightarrow (\mathcal{X}^* \rightarrow \wp(\mathcal{X}^*))$ 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 subtraces of s:

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

where $\mathit{SubSeq}(s) = \{t | t \preceq s\}$ denotes the set of all subsequences of s. For any $S \in \wp(\mathcal{X}^*)$, the additive function $\alpha_c[S]$ 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 turns out to be a suitable approximation when dealing with conservative obfuscations. In fact the semantic malware detector on $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])]$ is complete and sound for the class of conservative obfuscations $\mathbb{O}_c$.

THEOREM 3. *Considering a vanilla malware M we have that $\exists \mathcal{O}_c \in \mathbb{O}_c$ such that $\mathcal{O}_c(M) \hookrightarrow P$ iff $\exists\ lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that:*

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

Many obfuscating transformations commonly used by malware writers are conservative; a partial list of such conservative obfuscations is given below. It follows that Theorem 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 $\mathcal{O}_1, \mathcal{O}_2 \in \mathbb{O}_c$ then $\mathcal{O}_1 \circ \mathcal{O}_2 \in \mathbb{O}_c$.*

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 B, the decryption of memory locations starting at location A and then executes the decrypted instructions. Let $\mathcal{O}_c(M)$ be a conservative obfuscation of M:*

M

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

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

4.2.2 Non-Conservative Obfuscations

A non-conservative transformation modifies the program semantics in such a way that the original environment-memory traces are not present any more. A possible way to 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 α is preserved by $\mathcal{O}$ then the semantic malware detector on α is complete w.r.t. $\mathcal{O}$. Recall that, given a program transformation $\mathcal{O} : \mathbf{P} \to \mathbf{P}$, it is possible to systematically derive the most concrete abstraction preserved by $\mathcal{O}$ [12]. This systematic methodology can be used in presence of non-conservative obfuscations in order to derive a complete semantic malware detector when it is not easy to identify a canonical abstraction.

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

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

Variable Renaming Variable renaming is a simple obfuscating transformation, often used to prevent signature matching, that replaces the names of variables with some different new names. Assuming that every environment function associates variable V_L to memory location L, allows us to reason on variable renaming also in the case of compiled code, where variable names have disappeared. Let $\mathcal{O}_v : \mathbf{P} \times \Pi \to \mathbf{P}$ denote the obfuscating transformation that, given a program P, renames its variables according to a mapping $\pi \in \Pi$, where $\pi : var[\![P]\!] \to \mathit{Names}$ 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 $\alpha_v : \Pi \to (\mathcal{X}^* \to \mathcal{X}^*)$ is defined as:

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

In order to deal with variable renaming obfuscation we introduce the notion of *canonical variable renaming* $\widehat{\pi}$. The idea of canonical mappings is that there exists a renaming $\pi : var[\![P]\!] \to var[\![Q]\!]$ 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

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 $(\bar{\mathcal{Z}} \neq \varnothing)$ **do**
  $trace = List(hd(\bar{\mathcal{Z}}))$
  while $(trace \neq \varnothing)$ **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.

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 $\sigma, \delta \in \Sigma^*$, $\sigma \leq_\Sigma \delta$ if $|\sigma| \leq |\delta|$ or $|\sigma| = |\delta|$ and $lab(\sigma_1)lab(\sigma_2)...lab(\sigma_n) \leq_L lab(\delta_1)lab(\delta_2)...lab(\delta_n)$, where $lab(\langle\rho, C\rangle) = lab[\![C]\!]$. 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 $\widehat{\pi}_P : var[\![P]\!] \to \{V_i\}_{i\in\mathbb{N}}$, is such that $\alpha_e(S[\![\mathcal{O}_v[\![P, \widehat{\pi}_P]\!]]\!]) = \alpha_v[\widehat{\pi}_P](\mathcal{Z})$. 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 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 $s = (\rho_1, m_1)(\rho_2, m_2)...(\rho_n, m_n) = (\rho_1, m_1)s'$:

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

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 $Rename[\mathcal{Z}]$ defined in Algorithm 1. Thus, the canonical renaming $\widehat{\pi}_P : var[\![P]\!] \to \{V_i\}_{i\in\mathbb{N}}$ is defined as follows:

$$\widehat{\pi}_P(X) = V_i \Leftrightarrow Rename[\mathcal{Z}][i] = X \tag{1}$$

The following result is necessary to prove that the mapping $\widehat{\pi}_P$ defined in Equation (1) 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 followings hold:*

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

Let $\widehat{\Pi}$ denote a set of canonical variable renaming, the additive function $\alpha_v : \widehat{\Pi} \to (\wp(\mathcal{X}^*) \to \wp(\mathcal{X}_c^*))$, 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*

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

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 α_1 and α_2, that are complete(sound) respectively for obfuscations $\mathcal{O}_1$ and $\mathcal{O}_2$, and the abstraction that is complete(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 5. *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 $\mathcal{O}_1 \circ \mathcal{O}_2$.*

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. 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 of point 1 is satisfied if and only if $(\alpha_e(\sigma) \preceq \alpha_e(\delta)) \Leftrightarrow \alpha_{\mathcal{O}}(\alpha_e(\sigma)) \preceq \alpha_{\mathcal{O}}(\alpha_e(\delta))$.

EXAMPLE 3. *Let us consider $\mathcal{O}_v(\mathcal{O}_c(M), \pi)$ obtained by obfuscating the portion of malware M in Example 2 through variable renaming and some conservative obfuscations:*

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

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

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

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 $\mathcal{O}_v \circ \mathcal{O}_c$.

5. Further Malware Abstractions

Definition 4 characterizes the presence of malware M in a program P as the existence of a restriction $lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$ such that $\alpha_e(\mathfrak{S}\,[\![M]\!]) \subseteq \alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))$. This means that program P is infected by malware M if P matches all malware 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 $\alpha_{Int(M)} : \Sigma^* \rightarrow \Sigma^*$ that considers only the interesting states in a given trace $\sigma = \sigma_1\sigma'$:

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

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

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

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

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

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

THEOREM 6. *Let $Int(M)$ be the set of interesting states of a vanilla malware M, then there exists an obfuscation $\mathcal{O}$ conservative w.r.t. $Int(M)$ such that $\mathcal{O}(M) \hookrightarrow_{Int(M)} P$ iff $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$ such that:*

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

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

EXAMPLE 4. *Let us consider the malware M and its obfuscation $\mathcal{O}(M)$ obtained by reordering independent instructions.*

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

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

Interesting Behaviors. Assume we have an oracle that given a malware M returns the set $T \subseteq \mathfrak{S}\,[\![M]\!]$ 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 $T \subseteq \mathfrak{S}\,[\![M]\!]$, i.e., $M \hookrightarrow_T P$ if:*

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

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

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

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.

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

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

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

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

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

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

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 $\mathcal{A}_{MD}$

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

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

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

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

- For each command $C \in \mathbf{C}$, create a CFG node annotated with that command, $v_{lab[\![C]\!]}$. Correspondingly, we write $C\,[\![v]\!]$ to denote the command at CFG node v.
- For each command $C = L_1 : A \to S$, 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:

$$mem\,[\![(C, (\rho, m))]\!] = m$$
$$env\,[\![(C, \rho, m))]\!] = \rho$$

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

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

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

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

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

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

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

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

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

The semantics-aware malware detector $\mathcal{A}_{MD}$ makes use of two oracles, OR_{CFG} and OR_{SNop}, described in Table 2. Thus $\mathcal{A}_{MD} = D^{\mathcal{OR}}$, for the set of oracles $\mathcal{OR} = \{OR_{CFG}, OR_{SNop}\}$. Our goal is then to verify whether $\mathcal{A}_{MD}$ is $\mathcal{OR}$-complete with respect to

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

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

the obfuscations from Table 1. Since three of those obfuscations (code reordering, semantic-nop insertion, and substitution of equivalent commands) are conservative, we only need to 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 2.1. First, in step 1 below, we develop a trace-based detector D_{Tr} based on an abstraction α, and show that $D^{\mathcal{OR}} = \mathcal{A}_{MD}$ and D_{Tr} are equivalent. This equivalence of detectors holds only if the oracles in $\mathcal{OR}$ are perfect. Then, in step 2, we show that D_{Tr} is complete w.r.t. the obfuscations of interest.

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

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

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

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

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

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

DEFINITION 9 (α-Semantic Malware Detector). *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*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The cause for this incompleteness is the fact that the abstraction applied by D_{Tr} still preserves some of the actions from the program. 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.

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

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 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 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

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology (CRYPTO'01)*, volume 2139 of *Lecture Notes in Computer Science*, pages 1 – 18, Santa Barbara, CA, USA, Aug. 19–23, 2001. Springer Berlin / Heidelberg.

[2] D. Chess and S. White. An undetectable computer virus. In *Proceedings of the 2000 Virus Bulletin Conference (VB2000)*, Orlando, FL, USA, Sept. 27–29, 2000. Virus Bulletin.

[3] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In G. Davida and Y. Frankel, editors, *Proceedings of the 4th International Information Security Conference (ISC'01)*, volume 2200 of *Lecture Notes in Computer Science*, pages 144–155, Malaga, Spain, Oct. 1–3, 2001. Springer Berlin / Heidelberg.

[4] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, Oakland, CA, USA, May 8–11, 2005. IEEE Computer Society.

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

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

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 184–196, San Diego, CA, USA, Jan. 19–21, 1998. ACM Press.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, CA, USA, Jan. 17–19, 1977. ACM Press.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282, San Antonio, TX, USA, Jan. 29–31, 1979. ACM Press.

[10] P. Cousot and R. Cousot. 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)*, pages 178–190, Portland, OR, USA, Jan. 16–18, 2002. ACM Press.

[11] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Proceedings of the 3rd IEEE International Conference on Software Engineeering and Formal Methods (SEFM'05)*, pages 301–310, Koblenz, Germany, Sept. 5–9, 2005. IEEE Computer Society.

[12] M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1325–1336, Lisboa, Portugal, July 11–15, 2005. Springer Berlin / Heidelberg.

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

[14] S. Goldwasser and Y. T. Kalai. On the impossibility of obfuscation with auxiliary input. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 553–562, Washington, DC, USA, Oct. 22–25, 2005. IEEE Computer Society.

[15] A. Gupta and R. Sekar. An approach for detecting self-propagating email using anomaly detection. In G. Vigna, E. Jonsson, and C. Kruegel, editors, *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID'03)*, volume 2820 of *Lecture Notes in Computer Science*, pages 55–72, Pittsburgh, PA, USA, Sept. 8–10, 2003. Springer Berlin / Heidelberg.

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

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

[18] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In K. Julisch and C. Krügel, editors, *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05)*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187, Vienna, Austria, July 7–8, 2005. Springer Berlin / Heidelberg.

[19] J. Z. Kolter and M. A. Maloof. 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)*, pages 470–478, Seattle, WA, USA, Aug. 22–25, 2004. ACM Press.

[20] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog. 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)*, pages 64–71, West Point, NY, June 15–17, 2005. United States Military Academy.

[21] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 290–299, Washington, DC, USA, Oct. 27–30, 2003. ACM Press.

[22] P. Morley. Processing virus collections. In *Proceedings of the 2001 Virus Bulletin Conference (VB2001)*, pages 129–134, Prague, Czech Republic, Sept. 27–28, 2001. Virus Bulletin.

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

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

[25] Symantec Corporation. *Symantec Internet Security Threat Report: Trends for January 06–June 06*, volume X. Symantec Corporation, Sept. 25, 2006.

[26] P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[27] P. Ször and P. Ferrie. Hunting for metamorphic. In *Proceedings of the 2001 Virus Bulletin Conference (VB2001)*, pages 123 – 144, Prague, Czech Republic, Sept. 27–28, 2001. Virus Bulletin.

[28] H. Wee. On obfuscating point functions. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05)*, pages 523–532, Baltimore, MD, USA, May 21–24, 2005. ACM Press.

[29] z0mbie. Automated reverse engineering: Mistfall engine. Published online at `http://www.madchat.org//vxdevl/papers/vxers/Z0mbie/autorev.txt` (last accessed on Sep. 29, 2006).

[30] z0mbie. Real permutating engine. Published online at `http://vx.netlux.org/vx.php?id=er05` (last accessed on Sep. 29, 2006).