ISBN 978-952-5726-07-7 (Print), 978-952-5726-08-4 (CD-ROM)
Proceedings of the Second Symposium International Computer Science and Computational Technology(ISCSCT '09)
Huangshan, P. R. China, 26-28,Dec. 2009, pp. 139-142

Formal Description and Analysis of Malware Detection Algorithm $\mathcal{A}_{MOM}$

Ying Zeng, Fenlin Liu, Xiangyang Luo, and Chunfang Yang
Information Science and Technology Institute, Zhengzhou, China
Email: zengying510@yahoo.com.cn

***Abstract*—Code obfuscation can alter the syntactic properties of malware byte sequences without significantly affecting their execution behaviors. Thus it can easily foil signature-based detection. In this paper, the ability of handling obfuscation transformations of the semantics-based malware detection algorithm $\mathcal{A}_{MOM}$ proposed by Gao et al. is discussed using abstract interpretation theory from a semantic point of view. First, a formal description of the algorithm $\mathcal{A}_{MOM}$ is proposed. Then an equivalent trace-based detector is developed. Finally, the oracle soundness and oracle completeness of the trace-based detector for a restricted class of obfuscation transformations which preserve the variation relation are shown.**

***Index Terms*—malware detection, code obfuscation, trace semantics, abstract interpretation**

I. INTRODUCTION

As the complexity of modern computing systems is growing, various bugs are unavoidable in software systems. This increases the possibility of the malware attack that usually exploits such vulnerabilities in order to damage the systems. Thus, the malware attack has become a serious threat in computer security, and therefore it is crucial to detect the presence of malicious code in software systems.

Nowadays, one of the most popular approaches to malware detection is signature-based detection [1]. In order to foil this detection, malware writers often use code obfuscation such as instructions reordering, semantics NOP insertion, and substitution of equivalent instructions to automatically generate metamorphic malware [2]. In fact, the majority of malware that appears today is a simple repacked version of old malware [3].

Different obfuscated versions of the same malware have to share (at least) the malicious intent, namely the maliciousness of their semantics, even if they might express it through different syntactic forms. Therefore, addressing the malware detection problem from a semantic point of view can lead a more robust detection system [4]. For example, Christodorescu et al. [5] put forward a semantics-aware malware detector $\mathcal{A}_{MD}$ that is able to handle some commonly used obfuscations, such as semantics NOP insertion, instructions reordering and so on. While Gao et al. [6] introduce another semantics-based malware detection algorithm $\mathcal{A}_{MOM}$ which is able to handle not only the obfuscations that $\mathcal{A}_{MD}$ can handle, but also some other obfuscations like the flattening obfuscation proposed by Wang et al. [7]. And this detection scheme can largely reduce the updating of virus definition databases. However, the authors did not give the specific obfuscations that $\mathcal{A}_{MOM}$ could handle, and discussed the ability of $\mathcal{A}_{MOM}$ handling obfuscation transformations using only the experiment results.

In this paper, a formal description of the semantics-based malware detection algorithm $\mathcal{A}_{MOM}$ proposed by Gao et al. is given from a semantic point of view. Then an equivalent trace-based detector D_{Tr} is constructed using abstract interpretation theory. Finally, the oracle soundness and oracle completeness of D_{Tr} have been shown for a restricted class of obfuscation transformations which preserve the variation relation.

II. ABSTRACT INTERPRETATION AND PROGRAMMING LANGUAGE

A. Abstract Interpretation

Abstract interpretation [8] was originally developed by P. Cousot and R. Cousot as a general theory for designing and approximating the fixpoint semantics of programs. The basic idea is to approximate semantics obtained from computation on the concrete domain by substituting the concrete domains of computation and concrete semantic operations with abstract domains and corresponding abstract semantic operations. The concrete semantics of a program is computed on the concrete domain $\langle C, \leq_C \rangle$ which is a complete lattice, modeling the values computed by the program. The partial ordering $\leq_C$ models relative precision: $c_1 \leq_C c_2$ means that c_1 is more precise than c_2. Approximation is encoded by an abstract domain $\langle A, \leq_A \rangle$ which is also a complete lattice representing some approximation properties on concrete objects. The abstract semantics is computed on an abstract domain. Usually abstract domains are specified by Galois connections.

B. Programming Language

The language we consider is the simple imperative language introduced in Ref. [9]. The syntax of the language is given in Table I. The auxiliary functions in Table II are useful in defining the semantics of the considered programming language, which is described in Table III.

An environment $\rho \in \mathfrak{E}$ maps variables $X \in dom(\rho)$ to their values $\rho(X)$, so $\mathfrak{E} \triangleq \cup_{\mathcal{X} \subseteq \mathbb{X}} \mathfrak{E}[\![\mathcal{X}]\!]$, where $\mathfrak{E}[\![\mathcal{X}]\!] \triangleq \mathcal{X} \to \mathfrak{D}_{\perp}$ is the subset of environments ρ with domain $dom(\rho) \triangleq \mathcal{X}$.

AP-PROC-CS-09CN005

$\mathfrak{E}[\![P]\!]$ is the set of environments of a program P whose domain is the set of program variables: $\mathfrak{E}[\![P]\!] \triangleq \mathfrak{E}[\![var[\![P]\!]]\!]$. $\rho|_{\mathcal{X}}$, where $\mathcal{X} \subseteq \mathbb{X}$, is the restriction of environment ρ to the domain $dom(\rho) \cap \mathcal{X}$. Let $\rho[X := n]$ be the environment ρ where value n is assigned to variable X.

TABLE I.
THE SYNTAX OF THE SIMPLE IMPERATIVE LANGUAGE

Syntactic Categories:	Syntax:
$n \in \mathbb{Z}$ (integers) $X \in \mathbb{X}$ (variable names) $L \in \mathbb{L}$ (labels) $E \in \mathbb{E}$ (integer expressions) $B \in \mathbb{B}$ (Boolean expressions) $A \in \mathbb{A}$ (actions) $C \in \mathbb{C}$ (commands) $P \in \mathbb{P}$ (programs)	$E ::= n \mid X \mid E_1 - E_2$ $B ::= true \mid false \mid$ $E_1 < E_2 \mid \neg B_1 \mid B_1 \vee B_2$ $A ::= X := E \mid X := ? \mid skip \mid B$ $C ::= L_1 : A \to L_2; \mid$ $L_1 : B \to \{L_T, L_F\};$ $\mathbb{P} ::= \wp(\mathbb{C})$

Let $\mathfrak{F}[\![P]\!]$ denote the set of final states of program P, the set of finite maximal execution traces $\mathbf{S}^n[\![P]\!]$ can be defined as: $\mathbf{S}^n[\![P]\!] \triangleq \{\sigma \in \Sigma^n \mid n>0 \wedge \forall i \in [0,n-1]: \sigma_i \in \mathbf{C}(\sigma_{i-1}) \wedge \sigma_{n-1} \in \mathfrak{F}[\![P]\!]\}$, where Σ^n is the set of finite state sequences of length n. The maximal finite trace semantics $\mathbf{S}^+[\![P]\!]$ of program P is given as $\mathbf{S}^+[\![P]\!] \triangleq \cup_{n>0} \mathbf{S}^n[\![P]\!]$.

TABLE II.
AUXILIARY FUNCTIONS

Labels:	Variables:
$lab[\![L_1 : A \to L_2;]\!] \triangleq L_1$ $lab[\![L_1 : B \to \{L_T, L_F\};]\!] \triangleq L_1$ $lab[\![P]\!] \triangleq \{lab[\![C]\!] \mid C \in P\}$	$var[\![L_1 : A \to L_2;]\!] \triangleq var[\![A]\!]$ $var[\![L_1 : B \to \{L_T, L_F\};]\!] \triangleq var[\![B]\!]$ $var[\![P]\!] \triangleq \cup_{C \in P} var[\![C]\!]$
Action of a command:	**Successors of a command:**
$act[\![L_1 : A \to L_2;]\!] \triangleq A$ $act[\![L_1 : B \to \{L_T, L_F\};]\!] \triangleq B$	$suc[\![L_1 : A \to L_2;]\!] \triangleq L_2$ $suc[\![L_1 : B \to \{L_T, L_F\};]\!] \triangleq \{L_T, L_F\}$

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. The control flow graph (CFG) can be easily constructed as follows:

- For each command $C \in \mathbb{C}$, create a CFG node $v_{lab[\![C]\!]}$ annotated with that command. Let $C[\![v]\!]$ denote the command at CFG node v.
- For each command $C \in \mathbb{C}$, $L_1 = lab[\![C]\!]$, for each label $L_2 \in suc[\![C]\!]$, create a CFG edge (v_{L_1}, v_{L_2}).

For a given CFG $G=(V, E)$, $entry(G)$ denotes the set of the entry vertexes. $Path(G)$ denotes the set of all paths in G. $Path_{mn}(G) = \{\theta \in Path(G) \mid \theta = v_m \to \cdots \to v_n\}$ denotes the set of all paths from node v_m to node v_n in G. Consider a path θ in the CFG from node v_1 to node v_k, $\theta = v_1 \to v_2 \to \cdots \to v_k$. There is a corresponding sequence of commands in program P, written $P|\theta = \{C_1, \ldots, C_k\}$, where $C_i = C[\![v_i]\!]$. Then we can express the set of possible states after executing the sequence of commands $P|\theta$ as $\mathbf{C}^k[\![P|\theta]\!](\rho, C_1)$.

TABLE III.
THE SEMANTICS OF THE SIMPLE IMPERATIVE LANGUAGE

Value Domains	Boolean Expressions $\mathbf{B} : \mathbb{B} \times \mathfrak{E} \to \mathfrak{B}_\perp$
$\mathfrak{B}_\perp \triangleq \{\text{true,false}, \perp\}$ (truth values) $n \in \mathbb{Z}$ (integers) $\mathfrak{D}_\perp$ (variable values) $\rho \in \mathfrak{E} \triangleq \mathbb{X} \to \mathfrak{D}_\perp$ (environments) $\Sigma \triangleq \mathfrak{E} \times \mathbb{C}$ (program states)	$\mathbf{B}[\![true]\!]_\rho \triangleq true$ $\mathbf{B}[\![false]\!]_\rho \triangleq false$ $\mathbf{B}[\![E_1 < E_2]\!]_\rho \triangleq \mathbf{E}[\![E_1]\!]_\rho < \mathbf{E}[\![E_2]\!]_\rho$ $\mathbf{B}[\![\neg B]\!]_\rho \triangleq \neg \mathbf{B}[\![B]\!]_\rho$ $\mathbf{B}[\![B_1 < B_2]\!]_\rho \triangleq \mathbf{B}[\![B_1]\!]_\rho \vee \mathbf{B}[\![B_2]\!]_\rho$
Arithmetic Expressions $\mathbf{E} : \mathbb{E} \times \mathfrak{E} \to \mathfrak{D}_\perp$	**Actions** $\mathbf{A} : \mathbb{A} \times \mathfrak{E} \to \wp(\mathfrak{E})$
$\mathbf{E}[\![n]\!]_\rho \triangleq n$ $\mathbf{E}[\![X]\!]_\rho \triangleq \rho(X)$ $\mathbf{E}[\![E_1 - E_2]\!]_\rho \triangleq \mathbf{E}[\![E_1]\!]_\rho - \mathbf{E}[\![E_2]\!]_\rho$	$\mathbf{A}[\![skip]\!]_\rho \triangleq \{\rho\}$ $\mathbf{A}[\![X := E]\!]_\rho \triangleq \{\rho \mid X := \mathbf{E}[\![E]\!]_\rho\}$ $\mathbf{A}[\![X := ?]\!]_\rho \triangleq \{\rho' \mid \exists z \in \mathbb{Z}: \rho' = \rho[X := z]\}$ $\mathbf{A}[\![B]\!]_\rho \triangleq \{\rho' \mid \mathbf{B}[\![B]\!]_{\rho'} = true \wedge \rho' = \rho\}$
Commands $\mathbf{C} : \Sigma \to \wp(\Sigma)$	
$\mathbf{C}(\langle \rho, L_1 : A \to L_2; \rangle) \triangleq \{\langle \rho', C' \rangle \mid \rho' \in \mathbf{A}[\![A]\!]_\rho \wedge lab[\![C']\!] = L_2\}$ $\mathbf{C}(\langle \rho, L_1 : B \to \{L_T, L_F\} \rangle) \triangleq \left\{\langle \rho, C' \rangle \,\middle|\, lab[\![C']\!] = \begin{cases} L_T & \text{if } \mathbf{B}[\![B]\!]_\rho = true \\ L_F & \text{if } \mathbf{B}[\![B]\!]_\rho = false \end{cases}\right\}$	

III. FORMAL DESCRIPTION OF MALWARE DETECTION ALGORITHM $\mathcal{A}_{MOM}$

The semantics-based malware detection algorithm $\mathcal{A}_{MOM}$ proposed by Gao et al. [6] compares the semantics of a program with the semantics of the malware to identify the malicious behavior in the program and detect whether the program is a variation of the malware with respect to a class of obfuscation transformations of which the specifications are given, i.e. $\mathcal{A}_{MOM}(P, M) = 1$.

In order to reason about the ability of the algorithm $\mathcal{A}_{MOM}$ handling obfuscation transformations, we give a formal description of the algorithm from a semantic point of view as follows. The algorithm proceeds in four steps:

1. Collect program invariants of program P and malware M by symbolic executions. Let $G=(V, E)$ be the CFG of a program P, $Path_{mn}(G) = \{\theta_1, \theta_2, \ldots, \theta_{num}\}$, where $num = |Path_{mn}(G)|$ is the size of set $Path_{mn}(G)$. The invariant at program point v_n can be expressed formally as $\varphi(v_n) = \bigvee_{i=1}^{num}\left(\left(\bigwedge_{X \in var[\![P]\!]}\left(X = \mathbf{E}[\![X]\!]_{\rho_i}\right)\right) \wedge \left(\bigwedge_{b \in B_i} b\right)\right)$, where $\rho_i = env[\![s_i]\!]$ is the environment in the state s_i after executing the sequence of commands $P|\theta_i$ from the initial state $\langle \rho_0, C[\![v_m]\!] \rangle$, $\theta_i \in Path_{mn}(G)$, B_i is the set of predicates needed to be satisfied while executing the sequence of commands $P|\theta_i$. This step makes use of two oracles:

OR_{CFG} that returns the control-flow graph $G^P = (V^P, E^P)$ of a program P and OR_{PI} that returns the invariants at all program points $\Psi^P = \{\varphi^P(v) | v \in V^P\}$ of a program P.

2. Identify a control-flow map and a data-flow map between malware M and program P according the specification of the obfuscation algorithm $\mathcal{O}$. There is a map matching a malware node v^M to a program node v^P, denoted by $\mu: V^M \to V^P$. And this map μ induces a map $\upsilon: var[\![M]\!] \times V^M \to var[\![P]\!]$ from variables at a malware node to variables at the corresponding program node.

3. Build an equivalent relation between the variables in the sets of nodes of the two CFGs. According to the specification of the obfuscation algorithm $\mathcal{O}$, the variable values in malware M should be equal to the corresponding variable values in program P. This equivalent relation, denoted by Q, can be expressed as $\forall v_k^M \in dom(\mu)$, $X_k^M \in var[\![C[\![v_k^M]\!]]\!]$, $\rho \in \mathfrak{E}$, $s^M \in \mathbf{C}^k[\![M | \lambda^M]\!](\rho, C[\![v_0^M]\!])$: $\mathbf{E}[\![X_k^M]\!]_{env[\![s^M]\!]} = \mathbf{E}[\![\upsilon(X_k^M, v_k^M)]\!]_{env[\![s^P]\!]}$, where $v_0^M = entry(G^M)$, $s^P = \mathbf{C}^l[\![P | \lambda^P]\!](\rho, C[\![v_0^P]\!])$, $v_0^P = entry(G^P)$, $v_l^P = \mu(v_k^M)$, $\lambda^P = \mu_{path}(\lambda^M)$.

4. Check whether the equivalent relation built in step 3 holds. Construct a verification condition, formally described as $\bigwedge_{v_k^M \in dom(\mu)} \left(\varphi^M(v_k^M) \wedge \varphi^P(\mu(v_k^M))\right) \Rightarrow Q$. Pass this verification condition to a theorem prover. If the condition holds, then identify the program P as a variant of the malware M with respect to the obfuscation algorithm $\mathcal{O}$, i.e. $\mathcal{A}_{MOM}(P, M) = 1$. This check is implemented in $\mathcal{A}_{MOM}$ as a query to oracle $OR_{Validation}$, which determines whether a verification condition holds.

IV. AN EQUIVALENT TRACE-BASED DETECTOR D_{Tr}

In the following, we first give the definitions of three abstractions α_{MOM}, α_{env} and α_r.

The abstraction α_{MOM}, when applied to a trace $\sigma \in \mathbf{S}^+[\![P]\!]$, with $\sigma = (\rho_1', C_1') \ldots (\rho_n', C_n')$, to a set of variable maps $\{\pi_i\}$, and a set of location maps $\{\gamma_i\}$, returns an abstract trace $\alpha_{\text{MOM}}(\sigma, \{\pi_i\}, \{\gamma_i\}) = (\rho_1, C_1) \cdots (\rho_n, C_n)$, if $\forall i, 1 \le i \le n$, $act[\![C_i]\!] = act[\![C_i']\!][X / \pi_i(X)]$, $lab[\![C_i]\!] = \gamma_i(lab[\![C_i']\!])$, $suc[\![C_i]\!] = \gamma_i(suc[\![C_i']\!])$, $\rho_i = \rho_i' \circ \pi_i^{-1}$, where $A[X / \pi(X)]$ represents actions A where each variable name X is replaced by the new name $\pi(X)$. Otherwise, if the condition does not hold, then $\alpha_{\text{MOM}}(\sigma, \{\pi_i\}, \{\gamma_i\}) = \varepsilon$. A map $\pi_i : var[\![P]\!] \to var[\![P']\!]$ renames program variables $var[\![P]\!]$ such that they match program variables $var[\![P']\!]$, $\gamma_i : lab[\![P]\!] \to lab[\![P']\!]$ reassigns program memory locations $lab[\![P]\!]$ to program memory locations $lab[\![P']\!]$.

Given a trace $\sigma = (\rho_1, C_1)\sigma'$, the abstraction α_{env} retains only the environments,

$$\alpha_{env}(\sigma) \triangleq \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ \rho_1 \alpha_{env}(\sigma') & \text{if } \sigma = (\rho_1, C_1)\sigma' \end{cases}. \tag{1}$$

Let $lab_r[\![P]\!] \subseteq lab[\![P]\!]$ be a restriction of a program P, the abstraction α_r propagates the restriction $lab_r[\![P]\!]$ on a given trace $\sigma = (\rho_1, C_1)\sigma'$ as

$$\alpha_r(\sigma, lab_r[\![P]\!]) \triangleq \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ (\rho_1{}^r, C_1)\alpha_r(\sigma') & \text{if } lab[\![C_1]\!] \in lab_r[\![P]\!] \\ \alpha_r(\sigma') & \text{otherwise} \end{cases}, \tag{2}$$

where $\rho_1{}^r \triangleq \rho_{1|var_r[\![P]\!]}$, $var_r[\![P]\!] \triangleq \cup\{var[\![C]\!] | lab[\![C]\!] \in lab_r[\![P]\!]\}$.

We can model the algorithm $\mathcal{A}_{MOM}$ using these three abstractions α_{MOM}, α_{env} and α_r. The abstraction α that characterizes the trace-based detector D_{Tr} is given by the composition of these three abstractions $\alpha_{env} \circ \alpha_{MOM} \circ \alpha_r$. We will show that D_{Tr} is equivalent to $\mathcal{A}_{MOM}$, when the oracles it uses are perfect.

Definition 1: Malware detector D_{Tr} is an α-semantic malware detector defined on the abstraction α, it classifies a program P as a variation of a malware M, i.e. $D_{Tr}(\mathbf{S}^+[\![P]\!], \mathbf{S}^+[\![M]\!]) = 1$, if

$$\begin{aligned} &\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]), lab_r[\![M]\!] \in \wp(lab[\![M]\!]), \\ &\{\pi_i : var[\![P]\!] \to var[\![M]\!]\}_{i \ge 1}, \{\gamma_i : lab[\![P]\!] \to lab[\![M]\!]\}_{i \ge 1} : \\ &\alpha(\mathbf{S}^+[\![M]\!], lab_r[\![M]\!], \{\pi_i\}, \{\gamma_i\}) = \alpha(\mathbf{S}^+[\![P]\!], lab_r[\![P]\!], \{\pi_i\}, \{\gamma_i\}) \end{aligned}, \tag{3}$$

where $\alpha = \alpha_{env} \circ \alpha_{MOM} \circ \alpha_r$.

Proposition 1: The semantics-based malware detection algorithm $\mathcal{A}_{MOM}$ is equivalent to the $\alpha_{env} \circ \alpha_{MOM} \circ \alpha_r$-semantic malware detector D_{Tr}, i.e.

$$\forall P, M \in \mathbb{P} : \mathcal{A}_{MOM}(P, M) = 1 \Leftrightarrow D_{Tr}(\mathbf{S}^+[\![P]\!], \mathbf{S}^+[\![M]\!]) = 1. \tag{4}$$

One of the most important requirements of a robust malware detection algorithm is to handle obfuscation transformations. For a malware detector, this can be formalized in terms of soundness and completeness properties [4]. Intutitively, a malware detector is sound if it never erroneously claims that a program is infected (no false positives) and it is complete if it always detects program that are infected (no false negatives). When a program P is a variation of a malware M with respect to an obfuscation $\mathcal{O}$, it can be denoted by $P \approx \mathcal{O}(M)$. A malware detector D is sound (complete) for an obfuscation $\mathcal{O} \in \mathbb{O}$ if and only if

$$\forall M, P \in \mathbb{P} : D(P, M) = 1 \Rightarrow P \approx \mathcal{O}[\![M]\!] \; (P \approx \mathcal{O}[\![M]\!] \Rightarrow D(P, M) = 1). \tag{5}$$

Most malware detectors are built on top of other static analysis techniques for problems that are hard or undecidable. So Ref. [4] introduced the notion of relative soundness and completeness with respect to algorithms that a detector uses. A malware detector $D^{\mathcal{OR}}$ is oracle sound (complete) with respect to an obfuscation $\mathcal{O}$, if $D^{\mathcal{OR}}$ is sound (complete) for that obfuscation when all oracles in the set $\mathcal{OR}$ are perfect.

Following we define a class of obfuscations $\mathbb{O}_{MOM}$ which preserve the variation relation, namely, there is an variation relation between the original program M and obfuscated program $\mathcal{O}(M)$, formally expressed as

Definition 2: The obfuscation $\mathcal{O} \in \mathbb{O}_{MOM}$ preserves variation relation, if $\forall M \in \mathbb{P}$, such that

$$\begin{aligned} &\exists lab_R[\![M]\!] \in \wp\left(lab[\![M]\!]\right), lab_R[\![\mathcal{O}(M)]\!] \in \wp\left(lab[\![\mathcal{O}(M)]\!]\right), \\ &\quad \left\{\xi_i : var[\![\mathcal{O}(M)]\!] \to var[\![M]\!]\right\}_{i\ge1}, \left\{\vartheta_i : lab[\![\mathcal{O}(M)]\!] \to lab[\![M]\!]\right\}_{i\ge1} : \\ &\qquad \alpha_{env}\left(\alpha_{MOM}\left(\alpha_r\left(\mathbf{S}^+[\![M]\!], lab_R[\![M]\!]\right), \{\xi_i\}, \{\vartheta_i\}\right)\right) \\ &\qquad\quad = \alpha_{env}\left(\alpha_{MOM}\left(\alpha_r\left(\mathbf{S}^+[\![\mathcal{O}(M)]\!], lab_R[\![\mathcal{O}(M)]\!]\right), \{\xi_i\}, \{\vartheta_i\}\right)\right) \end{aligned} \quad (6)$$

Therefore, the following properties can be easily obtained, showing that the malware detector D_{Tr} which is equivalent to $\mathcal{A}_{MOM}$ is oracle sound and oracle complete with respect to this class of obfuscations.

Property 1: Malware detector D_{Tr} is oracle sound for the obfuscation $\mathcal{O} \in \mathbb{O}_{MOM}$, i.e.

$$D_{Tr}\left(\mathbf{S}^+[\![P]\!], \mathbf{S}^+[\![M]\!]\right) = 1 \Rightarrow \exists \mathcal{O} \in \mathbb{O}_{MOM} : P \approx \mathcal{O}(M). \quad (7)$$

Property 2: Malware detector D_{Tr} is oracle complete for the obfuscation $\mathcal{O} \in \mathbb{O}_{MOM}$, i.e.

$$\exists \mathcal{O} \in \mathbb{O}_{MOM} : P \approx \mathcal{O}(M) \Rightarrow D_{Tr}\left(\mathbf{S}^+[\![P]\!], \mathbf{S}^+[\![M]\!]\right) = 1. \quad (8)$$

V. CONCLUSIONS

The semantics-based malware detector $\mathcal{A}_{MOM}$ proposed by Gao et al. can detect whether a program is a variation of a malware with respect to some commonly used obfuscations, and largely reduce the updating of malware definition databases. In this paper, a formal description of the malware detection algorithm $\mathcal{A}_{MOM}$ proposed by Gao et al. is given from a semantic point of view and an equivalent trace-based detector D_{Tr} is constructed by abstract interpretation. At last, the oracle soundness and oracle completeness of the detector D_{Tr} for a restricted class of obfuscation transformations which preserve the variation relation have shown. Our work provides a formal basis for addressing the ability of the malware detection algorithm $\mathcal{A}_{MOM}$. The properties of $\mathcal{A}_{MOM}$ such as soundness and completeness can be proved using the result of this paper in the framework proposed by Preda et al. [4] which can be used to reason about and evaluate the resilience of malware detectors to various kinds of obfuscation transformations. Also our work can be a reference for designing effective malware detection algorithms.

ACKNOWLEDGMENT

The authors wish to thank X. Y. Luo, C. F. Yang, and L. Bin for comments and suggestions. This work was supported in part by grants from the National Natural Science Foundation of China under Grant Nos. 60970141, 60902102.

REFERENCES

[1] P. Szor, *The Art of Computer Virus Research and Defense*. Boston, MA: Addison-Wesley Professional, 2005.

[2] C. Nachenberg, "Computer virus-antivirus coevolution," *Comm. ACM*, vol. 40(1), pp. 46–51, 1997.

[3] R. Perdisci, A. Lanzi, and W. Lee, "Classification of packed executables for accurate computer virus detection," *Pattern Recognition Letters*, vol. 29(14), pp. 1941–1946, 2008.

[4] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," *ACM Trans. Program. Lang. Syst*, vol. 30(5), pp. 1–54, 2008.

[5] 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)*, IEEE Computer Society, pp. 32–46, 2005.

[6] Y. Gao, Y. Y. Chen, and B. J. Hua, "A semantics-based malware detector for obfuscated malware," *Journal of Chinese Computer Systems*, vol. 28(1), pp. 1–8, 2007.

[7] C. X. Wang, "A security architecture for survivability mechanisms," *PhD Dissertation*, University of Virginia, Department of Computer Science, 2000.

[8] P. Cousot, and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, ACM Press, pp. 238–252, 1977.

[9] 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)*, ACM Press, pp. 178–190, 2002.