Semantics-Aware Malware Detection

Mihai Christodorescu[*] Somesh Jha[*]
University of Wisconsin, Madison
{*mihai, jha*}*@cs.wisc.edu*

Sanjit A. Seshia[†] Dawn Song Randal E. Bryant[†]
Carnegie Mellon University
{*sanjit@cs., dawnsong@, bryant@cs.*}*cmu.edu*

Abstract

A malware detector is a system that attempts to determine whether a program has malicious intent. In order to evade detection, malware writers (hackers) frequently use obfuscation to morph malware. Malware detectors that use a pattern-matching approach (such as commercial virus scanners) are susceptible to obfuscations used by hackers. The fundamental deficiency in the pattern-matching approach to malware detection is that it is purely syntactic and ignores the semantics of instructions. In this paper, we present a malware-detection algorithm that addresses this deficiency by incorporating instruction semantics to detect malicious program traits. Experimental evaluation demonstrates that our malware-detection algorithm can detect variants of malware with a relatively low run-time overhead. Moreover, our semantics-aware malware detection algorithm is resilient to common obfuscations used by hackers.

1. Introduction

A *malware instance* is a program that has malicious intent. Examples of such programs include viruses, trojans, and worms. A classification of malware with respect to its propagation method and goal is given in [29]. A *malware detector* is a system that attempts to identify malware. A virus scanner uses signatures and other heuristics to identify malware, and thus is an example of a malware detector. Given the havoc that can be caused by malware [18], malware detection is an important goal.

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

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

[†]This work was supported in part by Army Research Office grant DAAD19-01-1-0485.

The goal of a malware writer (hacker) is to modify or morph their malware to evade detection by a malware detector. A common technique used by malware writers to evade detection is program obfuscation [30]. Polymorphism and metamorphism are two common obfuscation techniques used by malware writers. For example, in order to evade detection, a virus can morph itself by encrypting its malicious payload and decrypting it during execution. A *polymorphic virus* obfuscates its decryption loop using several transformations, such as `nop`-insertion, code transposition (changing the order of instructions and placing jump instructions to maintain the original semantics), and register reassignment (permuting the register allocation). *Metamorphic viruses* attempt to evade detection by obfuscating the entire virus. When they replicate, these viruses change their code in a variety of ways, such as code transposition, substitution of equivalent instruction sequences, change of conditional jumps, and register reassignment [28, 35, 36].

Addition of new behaviors to existing malware is another favorite technique used by malware writers. For example, the Sobig.A through Sobig.F worm variants (widespread during the summer of 2003) were developed iteratively, with each successive iteration adding or changing small features [25–27]. Each new variant managed to evade detection either through the use of obfuscations or by adding more behavior. The recent recurrence of the Netsky and B[e]agle worms (both active in the first half of 2004) is also an example of how adding new code or changing existing code creates new undetectable and more malicious variants [9, 17]. For example, the B[e]agle worm shows a series of "upgrades" from version A to version C that include the addition of a backdoor, code to disable local security mechanisms, and functionality to better hide the worm within existing processes. A quote from [17] summarizes the challenges worm families pose to detectors:

> Arguably the most striking aspect of Beagle is the dedication of the author or authors to refining the code. New pieces are tested, perfected, and then deployed with great forethought as to how to evade antivirus scanners and how to defeat network edge protection

devices.

Commercial malware detectors (such as virus scanners) use a simple pattern matching approach to malware detection, i.e., a program is declared as malware if it contains a sequence of instructions that is matched by a regular expression. A recent study demonstrated that such malware detectors can be easily defeated using simple program obfuscations [8], that are already being used by hackers. The basic deficiency in the pattern matching approach to malware detection is that *they ignore the semantics of instructions*. Since the pattern-matching algorithm is not resilient to slight variations, these malware detectors must use different patterns for detecting two malware instances that are slight variations of each other. This is the reason that the signature database of a commercial virus scanner has to be frequently updated. The goal of this paper is to design a malware-detection algorithm that uses semantics of instructions. Such an algorithm will be resilient to minor obfuscations and variations.

Suppose a program P is transformed by a compiler phase (such as register allocation) to produce a program P'. The *translation-validation* problem is the problem of determining whether P' "simulates" P [31,32]. Translation validation can be used to prove that various compiler phases preserve the semantics of a specific program P. By viewing hacker obfuscations as compiler phases, the malware-detection problem at the surface seems similar to the translation-validation problem. However, there are fundamental differences between the two problems. Translation-validation techniques determine whether the two programs are semantically equivalent. However, variants of malware are *not* equivalent because malware writers add additional functionality between variants. Moreover, translation-validation researchers make certain assumptions about the transformed program. For example, Necula [31] assumes that all branches in the target program must correspond to branches in the source program. In the context of malicious-code detection, such an assumption is not valid: an adversary is free to transform the malicious code as they wish. Although we use some ideas from the translation-validation literature in the context of malware detection (such as modeling semantics of instructions and using decision procedures), our algorithm is differs from those in the translation-validation literature.

We use the observation that certain malicious behaviors (such as a decryption loop in a polymorphic virus or a loop to search for email addresses in a user's mail folder) appear in all variants of a certain malware. The problem is that the specified malicious behavior (such as a decryption loop) appears in different guises in the different variants. We formalize this problem and describe an algorithm for discovering specified malicious behaviors in a given program. Since our algorithm is semantic rather than purely syntactic, it is resistant to common obfuscations used by hackers. Specifically, this paper makes the following contributions:

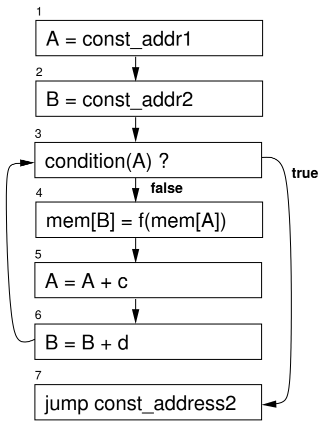
- **Formal semantics:** We formally define the problem of determining whether a program exhibits a specified malicious behavior. In general, this problem is undecidable. Our semantics is described in detail in Section 2. We believe that the semantics presented in Section 2 can be used as a "reference semantics" for other researchers working on malware detection.

- **A semantics-aware malware detection algorithm:** Since the problem of determining whether a program exhibits a specified malicious behavior is undecidable, we cannot hope to have a algorithm for the abovementioned problem. In Section 3 we present an algorithm for handling a limited set of transformations used by hackers. However, our evaluation (in Section 4) shows that our algorithm is very effective in discovering malicious behavior, as it detects multiple Netsky and B[e]agle variants with a single template, and its resilience to obfuscation is better than that of commercial virus scanners.

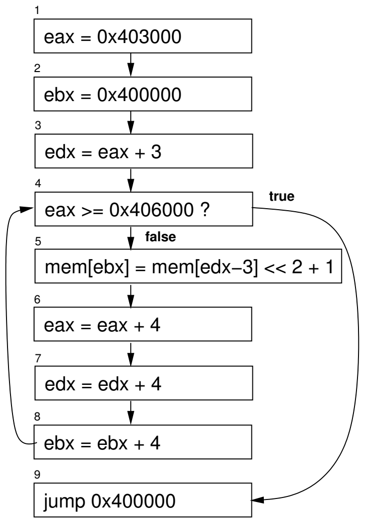
2. Semantics of malware detection

Before we present our formal semantics for malware detection, we will provide an intuitive explanation of the underlying ideas. We will use the example shown in Figure 1 as a running example. Figure 1(a) shows the control-flow graph (CFG) of a specification of malicious behavior and Figure 1(b) shows the CFG of an instruction sequence (which is a fragment of larger program). Instructions in Figure 1 are in the intermediate form (IR) used by our tool. The instructions in our IR are intuitive, and we give a formal description of the IR in Appendix B. We describe various components of our semantics.

Specifying the malicious behavior. In our framework, malicious behavior is described using *templates*, which are instruction sequences where variables and symbolic constants are used. Figure 1(a) describes a malicious behavior that is a simplified version of a decryption loop found in polymorphic worms. The malware specification described in Figure 1(a) decrypts memory starting from address $\mathsf{const\_addr1}$ and writes the decrypted data to memory starting at address $\mathsf{const\_addr2}$. The decryption function and the termination condition are denoted by the function $\mathsf{f}(\cdot)$ and predicate $\mathsf{condition}(\cdot)$ respectively. By abstracting away the names of specific registers and symbolic constants in the specification of the malicious behavior, our semantics and algorithm are insensitive to simple obfuscations, such as register renaming and changing the starting address of a memory block.



(a) Template of malicious behavior.



(b) Malware instance.

const_addr1 $\leftarrow$ 0x403000
const_addr2 $\leftarrow$ 0x400000
condition(X) $\leftarrow$ X$\geq$0x406000
f(X) $\leftarrow$ X$\ll$2 + 1
c $\leftarrow$ 4
d $\leftarrow$ 4

(c) Execution context.

const_addr1 : F(0)
const_addr2 : F(0)
c : F(0)
d : F(0)
f : F(1)
condition : P(1)

(d) Symbolic constant types.

Figure 1. Malware instance (b) satisfies the template (a) according to our semantics.

When does an instruction sequence contain the malicious behavior? Consider the instruction sequence shown in Figure 1(b). Assume the symbolic constants in the template are assigned values shown in Figure 1(c). If the template and the instruction sequence shown in Figure 1(a) and 1(b) are executed from a state where the contents of the memory are the same, then after both the executions the state of the memory is the same. In other words, the template and the instruction sequence *have the same effect on the memory*. This is because whenever memory accesses are performed the addresses in the two executions are the same. Moreover, stores to memory locations are also the same. Thus, there is an execution of instruction sequence shown in Figure 1(b) that exhibits the behavior specified by the template given in Figure 1(a). In other words, *the malicious behavior specified by the template is demonstrated by the instruction sequence*. Note that this intuitive notion of an instruction sequence demonstrating a specified malicious behavior is not affected by program transformations, such as register renaming, inserting irrelevant instruction sequences, and changing starting addresses of memory blocks.

2.1. Formal semantics

A *template* $T = (I_T, V_T, C_T)$ is a 3-tuple, where I_T is a sequence of instructions and V_T and C_T are the set of variables and symbolic constants that appear in I_T. There are two types of symbolic constants: an n-ary function (denoted as $F(n)$) and an n-ary predicate (denoted as $P(n)$). Notice that a simple symbolic constant is 0-ary function or has type $F(0)$. A constant c of type τ is written as $c : \tau$. In the template shown in Figure 1(a) the variables V_T are $\{A, B\}$ and the symbolic constants C_T are shown in Figure 1(d). Let I be an instruction sequence or a program fragment. An example instruction sequence is shown in Figure 1(b). Memory contents are represented as a function $M : Addr \rightarrow Values$ from the set of addresses $Addr$ to the set of values $Values$, where $M[a]$ denotes the value stored at address a.

An *execution context* for a template T, with $T = (I_T, V_T, C_T)$, is an assignment of values of appropriate types to the symbolic constants in the set C_T. Formally, an execution context EC_T for a template T is a function with domain C_T, such that for all $c \in C_T$ the type of c and $EC_T(c)$ are the same. An execution context for the template shown in Figure 1(a) is shown in Figure 1(c). Given an execution context EC_T for a template T, let $EC_T(T)$ be the template obtained by replacing every constant $c \in C_T$ by $EC_T(c)$.

A *state* s^T for the template T is a 3-tuple denoted (val^T, pc^T, mem^T), where $val^T : V_T \rightarrow Values$ is an assignment of values to the variables in V_T, pc^T is a value for the program counter, and $mem^T : Addr \rightarrow Values$ gives the memory content. We follow the convention that a state and its component referring to the template are superscripted by T. Given a template state s^T, $val(s^T)$, $pc(s^T)$ and $mem(s^T)$ refer to the three components of the state. Similarly, a state for the instruction sequence I is a 3-tuple (val, pc, mem), where $val : Reg \rightarrow Values$ is an assignment of values to the set of registers Reg, pc is a value for the program counter, and $mem : Addr \rightarrow Values$ gives the memory contents. Let S^T and S be the state space for the template and the instruction sequence respectively.

Assume that we are given an execution context EC_T for a template T, and the template is in state s^T. If we execute an instruction i from the template $EC_T(T)$ in state s^T, we transition to a new state s_1^T and generate a system event e (in our context, events are usually system calls or kernel traps). We denote a state change from s^T to s_1^T generating an event e as $s^T \xrightarrow{e} s_1^T$. For uniformity, if an instruction i does not generate a system event, we say that it generates the null event, or $e(i) = null$. For every initial template state s_0^T, executing the template T in an execution context EC_T generates a sequence as follows:

$$\sigma(T, EC_T, s_0^T) \quad = \quad s_0^T \xrightarrow{e_1^T} s_1^T \xrightarrow{e_1^T} \cdots ,$$

where for $i \geq 1$, s_i^T is the state after executing the i-th instruction from the template $EC_T(T)$ and e_i^T is the event generated by the $(i-1)$-th instruction. Notice that if the template does not terminate, $\sigma(T, EC_T, s_0^T)$ can be infinite. Similarly, $\sigma(I, s_0)$ denotes the sequence when the instruction sequence I is executed from the initial state s_0.

Definition 1 We say that an instruction sequence I *contains a behavior specified by* the template $T = (I_T, V_T, C_T)$ if there exists a program state s_0, an execution context EC_T, and a template state s_0^T such that $mem(s_0^T) = mem(s_0)$ (the memory in the two states are the same), and the two sequences $\sigma(I, s_0)$ and $\sigma(T, EC_T, s_0^T)$ are finite, and the following conditions hold on the two sequences:

- **(Condition 1):** Let the two execution sequences be given as follows:

$$\begin{aligned} \sigma(T, EC_T, s_0^T) \quad &= \quad s_0^T \xrightarrow{e_1^T} s_1^T \xrightarrow{e_2^T} \cdots \xrightarrow{e_k^T} s_k^T \\ \sigma(I, s_0) \quad &= \quad s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_r} s_r \end{aligned}$$

  Let $affected(\sigma(T, EC_T, s_0^T))$ be the set of addresses a such that $mem(s_0^T)[a] \neq mem(s_k^T)[a]$, i.e., $affected(\sigma(T, EC_T, s_0^T))$ is the set of memory addresses whose value changes after executing the template T from the initial state. We require that $mem(s_k^T)[a] = mem(s_r)[a]$ holds for all $a \in affected(\sigma(T, EC_T, s_0^T))$, i.e. values at addresses that belong to the set $affected(\sigma(T, EC_T, s_0^T))$ are the same after executing the template T and the instruction sequence I.

- **(Condition 2):** Ignoring $null$ events, the event sequence $\langle e_0^T, \cdots, e_k^T \rangle$ is a subsequence of the event sequence $\langle e_0, \cdots, e_r \rangle$. In order for the two system events e_1 and e_2 to match, their arguments and return values should be identical.

- **(Condition 3):** If

$$pc(s_k^T) \in affected(\sigma(T, EC_T, s_0^T)),$$

  then $pc(s_r) \in affected(\sigma(T, EC_T, s_0^T))$. In other words, if the program counter at the end of executing the template T points to the affected memory area, then the program counter after executing I should also point into the affected memory area.

Consider the example shown in Figure 1. Assume that we use the execution context shown in Figure 1(c) for the template shown in Figure 1(a). Suppose we execute the template and instruction sequence shown in Figure 1 from states with the same memory contents. The state of the memory is same in both executions, so condition 1 is true. Condition 2 is trivially satisfied. Since the jumps have the same target, condition 3 is trivially true.

Definition 2 A program P *satisfies* a template T (denoted as $P \models T$) iff P contains an instruction sequence I such that I contains a behavior specified by T. Given a program P and a template T, we call the problem of determining whether $P \models T$ as the *template matching problem* or *TMP*.

Defining a variant family. Definition 2 can be used to define a variant family. The intuition is that most variants of a malware contain a common set of malicious behavior, such as a decryption loop and a loop to search for email addresses. Let $\mathcal{T}$ be a set of templates (this set contains specification of malicious behavior common to a certain malware family). The set $\mathcal{T}$ defines a *variant family* as follows:

$$\{P \mid \text{for all } T \in \mathcal{T} \ , \ P \models T\}$$

In other words, the variant family defined by $\mathcal{T}$ contains all programs that satisfy all templates in the set $\mathcal{T}$.

Theorem 1 TMP is undecidable.

Proof: We will reduce the halting problem to TMP. Let M be a Turing machine, and P_M be a program that uses instructions in our IR that simulates M (since our IR is Turing complete, this can be accomplished). Without loss of generality, assume that P_M does not touch a special address $\mathsf{sp\_addr}$ while simulating the Turing machine M. Before starting to simulate M, P_M sets $mem[\mathsf{sp\_addr}]$ to 0. After simulating M, if P_M halts, it sets $mem[\mathsf{sp\_addr}]$ to 1. Consider the template T shown below:

$$\begin{aligned} mem[\mathsf{sp\_addr}] \ &= \ 0 \\ mem[\mathsf{sp\_addr}] \ &= \ 1 \end{aligned}$$

It is easy to see that $P_M \models T$ iff M halts. □

2.2. A weaker semantics

In some scenarios the semantics described in definition 1 is too strict. For example, if a template uses certain memory locations to store temporaries, then these memory locations should not be checked for equality in comparing executions of an instruction sequence and a template. Let $\sigma(T, EC_T, s_0^T)$ be the sequence generated when a template T is executed from a state s_0^T using the execution context EC_T. Define a set of *core memory locations* $core(\sigma(T, EC_T, s_0^T))$ which is a subset of $affected(\sigma(T, EC_T, s_0^T))$. Condition 1 in definition 1 can be changed as follows to give a weaker semantics:

- **(Modified condition 1):** We require that for all $a \in core(\sigma(T, EC_T, s_0^T))$, we have $mem(s_k^T)[a] = mem(s_r)[a]$, i.e., values in the addresses that belong to the set $core(\sigma(T, EC_T, s_0^T))$ are the same after executing the template T and the instruction sequence I. In other words, only the memory locations in the set $core(\sigma(T, EC_T, s_0^T))$ are checked for equality in the two executions.

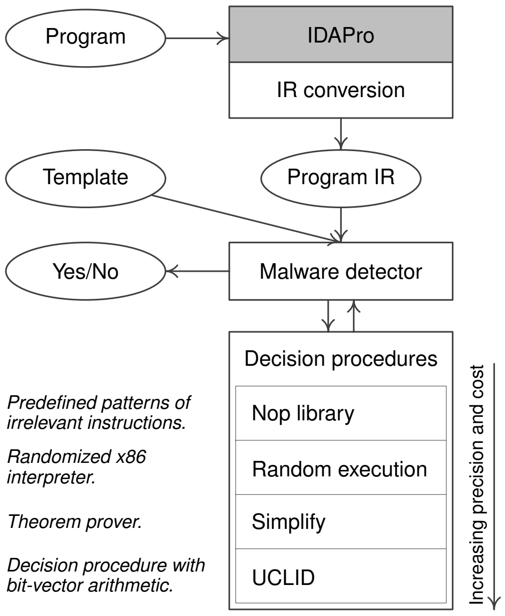
There are several ways to define core memory locations. We describe one possible definition of core memory locations. Assume that each instruction in the template T is labeled as temp or persistent. Intuitively if an instruction is labeled with temp, it performs temporary computation. Let $\sigma(T, EC_T, s_0^T)$ be the sequence generated when a template T executed from a state s_0^T using the execution context EC_T. Recall that $affected(\sigma(T, EC_T, s_0^T))$ is the set of addresses a such that $mem(s_0^T)[a] \neq mem(s_k^T)[a]$, where s_k^T is the last state in the sequence $\sigma(T, EC_T, s_0^T)$. Then the core memory locations $core(\sigma(T, EC_T, s_0^T))$ can be defined as any address in $affected(\sigma(T, EC_T, s_0^T))$ that is a target of a load or store instruction with label persistent.

3. Semantics-aware matching algorithm

We present an algorithm for checking whether a program P satisfies a template T; this algorithm is sound, but not complete, with respect to the semantics presented in Section 2.1, yet it can handle several classes of obfuscations (see Section 3.4). We have also implemented our algorithm as a malware-detection tool. The tool takes as input a template and a binary program for the Microsoft Windows on Intel IA-32 (x86) platform, and determines whether a fragment of the suspicious program contains a behavior specified by the template. The tool's output describes the matched fragment of the program together with information relating template variables to program registers and memory locations, and a list of "irrelevant" instruction sequences that are part of the matched program fragment.

3.1. Architecture

The tool is built on top of the IDA Pro disassembler [13]. Figure 2 illustrates the flow through the tool,



Program
IDAPro
IR conversion
Template
Program IR
Yes/No
Malware detector
Decision procedures
Predefined patterns of irrelevant instructions.
Nop library
Randomized x86 interpreter.
Random execution
Theorem prover.
Simplify
Decision procedure with bit-vector arithmetic.
UCLID
Increasing precision and cost

Figure 2. The architecture of the malicious code detector. Gray boxes represent existing infrastructure.

starting with the input binary program on the left and ending with an output indicating whether the program satisfies the template.

We first disassemble the binary program, construct control flow graphs, one per program function (as identified by IDA Pro), and produce an intermediate representation (IR). The IR is generated using a library of x86 instruction transformers, in such a way that the IR is architecture-independent. The IR can still contain library and system calls that are, by definition, platform-specific – we remove this last dependency by using summary functions expressed in IR form (see Appendix B for a description of the IR).

The rest of the toolkit takes advantage of the platform- and architecture-independent IR. We have initially focused our research on the Microsoft Windows on Intel IA-32 platform, with no loss of generality. By providing appropriate front-ends that translate the program into the IR format, one can use our malware-detector tool to process programs for different platforms.

3.2. Algorithm description

Our malware detection algorithm $\mathcal{A}_{MD}$ works by finding, for each template node, a matching node in the program. Individual nodes from the template and the program match if there exists an assignment to variables from the template node expression that unifies it with the program node expression. Once two matching nodes are found, we check whether the def-use relationships true between template nodes also hold true in the corresponding program (two nodes are related by a def-use relationship if one node's definition of a variable reaches the other node's use of the same variable). If all the nodes in the template have matching counterparts under these conditions, the algorithm has found a program fragment that satisfies the template and produces a proof of this relationship.

Formally, given a malicious code template T and a program P, we define the malware detection algorithm $\mathcal{A}_{MD}$ as a predicate on pairs of programs and templates, $\mathcal{A}_{MD}$: *Programs* $\times$ *Templates* $\rightarrow \{\text{yes}, \bot\}$, such that $\mathcal{A}_{MD}(P, T)$ returns "yes" when program P satisfies template T under the conditions below. We denote by $\bot$ (i.e. "don't know") the return value from the algorithm when it cannot determine that the program satisfies the template.

Consider the example in Figure 3, where a simpler version of the template T from Figure 1(a) uses expression X≥const_addr3 instead of the symbolic constant condition(X) and X^5 (where symbol '^' represents the bit-vector xor operator) instead of f(X). Similarly, the program P in Figure 3 is a simpler version of the malware instance from Figure 1(b). For P to satisfy T in algorithm $\mathcal{A}_{MD}$, i.e., $\mathcal{A}_{MD}(P, T) = \text{yes}$, the following two conditions have to hold:

- **Matching of template nodes to program nodes.** First, each node n in the template has to unify with a node m in P. In other words, there is an onto partial function f from the nodes of P to nodes in T, such that $f(m)$ in T and m in P are unifiable. We will denote by $B(X, n, m)$, where $n = f(m)$, the binding of variable X referred to in the template instruction at node n to an expression referred to in the program instruction at node m. In Figure 3, the gray arrows connect unified template nodes and program nodes: for instance, program node 7 unifies with template node 4.

- **Preservation of def-use paths.** For each node n in T, define $def(n)$ as the set of variables defined in n. Similarly, $use(n)$ is the set of variables used in node n. In Figure 3, template node 3 uses variable A, $use(3) = \{A\}$, while template node 6 both defines and uses variable B, $def(6) = use(6) = \{B\}$. Define synthetic nodes n_{pre} as predecessor to the template entry node and n_{post} as successor to all template exit nodes, such that $def(n_{pre}) = use(n_{post}) = V_T$. A *def-use path* is a sequence of nodes $\langle n_1, ..., n_k\rangle$ such that the following conditions are true:

  – For $1 \leq i < k$, there is an edge from n_i to n_{i+1} in T, i.e., $\langle n_1, \cdots, n_k\rangle$ is a valid path through the template T.

  – The instruction $I(n_1)$ (at node n_1) defines a variable that $I(n_k)$ uses, that is, $def(n_1) \cap use(n_k) \neq \emptyset$.

  – None of the instructions at the intermediate nodes redefine the value of the variables in $def(n_1)$, i.e., $def(n_1) \cap def(n_i) = \emptyset$ for $1 < i < k$.

  Two nodes n_1 and n_2 are *def-use related* (denoted by $duse(n_1, n_2)$) iff there exists a path from n_1 to n_2 that is a def-use path. In Figure 3, template nodes 1 and 3 are def-use related, with the corresponding def-use path $\langle 1, 2, 3\rangle$.

  Consider two nodes n and n' in T that are def-use related. Let nodes n and n' in T be matched to nodes m and m' in program P, i.e. $f(m) = n$ and $f(m') = n'$. Suppose X is a variable that is defined in n and used in n', and $B(X, n, m) = r$ and $B(X, n', m') = r'$, i.e., X is bound to expression r program in node m and r' in program node m'. In this case, we need to make sure that the value of expression r after executing the instruction $I(m)$ is same as the value of expression r' before executing the instruction $I(m')$, for every program path corresponding (under the matching function f) to a template path from n to n'. In Figure 3, program nodes 1 and 5 match template nodes 1 and 3, with respective bindings $B(A, 1, 1) = eax$ and $B(A, 3, 5) = (ecx - 1)$. The condition we need to enforce is that the value of eax after program instruction $I(1)$ is same as the value of $(ecx - 1)$ before program instruction $I(5)$. In our algorithm, these value invariants are checked using a decision procedure.

Algorithm $\mathcal{A}_{MD}$ implements a conservative approximation of the formal semantics of malware detection from Section 2.1, as the following result demonstrates.

Theorem 2 $\mathcal{A}_{MD}$ is sound with respect to the *TMP* semantics, i.e. $\mathcal{A}_{MD}(P, T) = \text{yes}$ implies that $P \models T$.
□

The proof is given in Appendix A. Our algorithm is thus sound with respect to the *TMP* semantics, but it is not complete. We can show there exists a program P, and a template T, such that $P \models T$, but $\mathcal{A}_{MD}(P, T)$ returns $\bot$. Consider the following example (in C-like syntax) where the template captures the behavior of initializing an array of 10 elements to 0.

Template T	*Program P*

for (i =0; i <10; i ++)
    a[i]=0;
	for (i =0; i <10; i +=2)
	    a[i]=0;
	for (i =1; i <10; i +=2)
	    a[i]=0;

Both the program and the template initialize one array of 10 elements: the sets of affected memory locations are the same in the final states of the template and the program, thus $P \models T$. Our algorithm $\mathcal{A}_{MD}$ will not

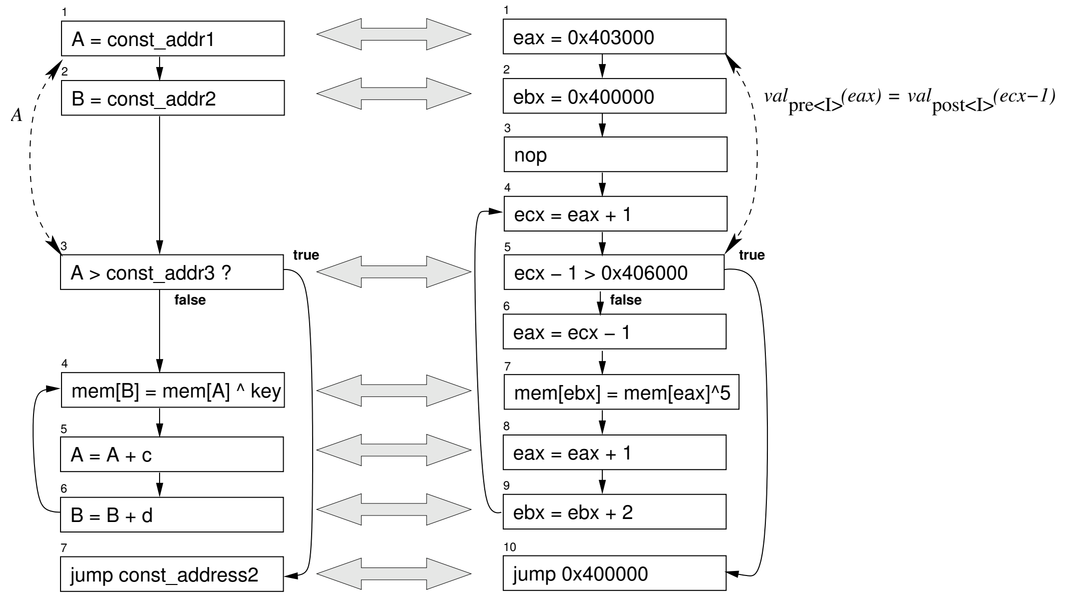


Figure 3. Example of program (on the right) satisfying a template (on the left) according to our algorithm $\mathcal{A}_{MD}$. Gray arrows connect program nodes to their template counterparts. The dashed arrow on the left marks one of the def-use relations that *does hold true* **in the template, while the corresponding dashed arrow on the right marks the def-use relation that** *must hold true* **in the program.**

match the program with the template as the ordering of memory updates in the template loop is different from that in the program loops.

Local unification. The unification step addresses the first condition of algorithm $\mathcal{A}_{MD}$, by producing an assignment to variables in an instruction at a template node such that it matches a program node instruction (if such an assignment exists). Since a program node expression contains only ground terms, the algorithm uses one-way matching that instantiates template variables with program expressions. In Figure 3, template node 3 matches program node 5, with the binding $\{$ A $\leftarrow$ ecx - 1, const_addr3 $\leftarrow$ 0x406000 $\}$. In the prototype we implemented, a template can contain expressions with variables, symbolic constants, predefined functions (see the operators in Appendix B), and external function calls. The matching algorithm takes these restrictions into account:

- A variable in the template can be unified with any program expression, except for assignment expressions.
- A symbolic constant in the template can only be unified with a program constant.
- The predefined function memory : $F(1)$ (traditionally written in array notation memory[...]) can only be unified with the program function memory : $F(1)$.
- A predefined operator in the template can only be unified with the same operator in the program.
- An external function call in the template can only be unified with the same external function call in the program.

Standard unification rules apply in all other cases: for example, an operator expression in the template unifies with an expression in the program if the program operator is the same as the template operator and the corresponding operator arguments unify. Template node 1 (A = const_addr1) can unify with program nodes 1 (eax = 0x403000) or 2 (ebx = 0x400000) but not with program nodes 8 or 9, since template node 1 expression has a symbolic constant as the right-hand side.

The result of the local unification step is a binding relating template variables to program expressions. The binding for the example in Figure 3 is shown in Table 1. Note that the bindings are different at various program points (at program node 1, template variable A is bound to eax, while at program node 5, template variable A is bound to (ecx - 1)). Requiring bindings to be consistent (monomorphic) seems like an intuitive approach, but leads to an overly restrictive semantics – any obfuscation attack that reassigns registers in a program fragment would evade detection. We want to check program expressions bound to the same template variable (e.g. eax and (ecx - 1) are both bound to A) and verify they change value in the same way the template variable changes values. We employ a mechanism based on def-use chains and value preservation to answer this problem.

Unified nodes		*Bindings*

T1	P1	A ← eax
		const_addr1 ← 0x403000

T2	P2	B ← ebx
		const_addr2 ← 0x400000

T3	P5	A ← ecx - 1
		const_addr3 ← 0x406000

T4	P7	A ← eax
		B ← ebx

T5	P8	A ← eax
		increment1 ← 1

T6	P9	B ← ebx
		increment2 ← 2

T7	P10	const_addr2 ← 0x400000

Table 1. Bindings generated from the unification of template and program nodes in Figure 3. Notation Tn refers to node n of the template, and Pm refers to node m of the program.

Value preservation on def-use chains. The second condition of algorithm $\mathcal{A}_{MD}$ requires template variables and the corresponding program expressions to have similar update patterns (although not necessarily the same values). For each def-use chain in the template, the algorithm checks whether the value of the program expression corresponding to the template-variable use is the same as the value of the program expression corresponding to the template-variable definition. Consider Figure 3, where template nodes 1 and 3 are def-use related. Program nodes 1 and 5 map to template nodes 1 and 3, respectively. Denote by R the program fragment between nodes 1 and 5, such that R contains only program paths from program node 1 to node 5 that correspond to template paths between template nodes 1 and 3 (i.e., any path in R has nodes that either map to template paths between 1 and 3, or have no corresponding template node). A def-use chain for template variable A connects template node 1 with template node 3: in the program, the expressions corresponding to template variable A after program node 1 and before program node 5 must be equal in value, across all paths in R. This condition can be expressed in terms of value predicates: for each path I in program fragment R (e.g. $I = \langle \text{P2}, \text{P3}, \text{P4} \rangle$), $val_{\text{pre}\langle I \rangle}(eax) = val_{\text{post}\langle I \rangle}(ecx - 1)$, where $val_{\text{pre}\langle I \rangle}$ represents the variable-valuation function for the program state before path I and $val_{\text{post}\langle I \rangle}$ is the variable-valuation function for the program state after path I. We can formulate this query about preserving def-use chains as a *value preservation problem*: given the program fragment R, we want to check whether it maintains the value predicate $\phi \equiv \forall I \in R \,.\, \big(val_{\text{pre}\langle I \rangle}(eax) = val_{\text{post}\langle I \rangle}(ecx - 1)\big)$.

The algorithm uses decision procedures to determine whether a value predicate holds. We discuss these decision procedures in Section 3.3; for now, we treat the decision procedures as oracles that can answer queries of the form "Does a program fragment R maintain an invariant ϕ?" For each match considered between a program node and a template node, the algorithm checks whether the def-use chain is preserved for each program expression corresponding to a template variable used at the matched node (see Appendix C for a listing of all the def-use chains checked for the example shown in Figure 3). This approach eliminates a large number of matches that cannot lead to a correct template assignment.

3.3. Using decision procedures to check value-preservation

A value-preservation oracle is a decision procedure that determines whether a given program fragment is a semantic nop with respect to certain program variables. Formally, given a program fragment P and program expressions e_1, e_2, a decision procedure $\mathcal{D}$ determines whether the value predicate $\phi(P, e_1, e_2) \equiv \forall I \in P \,.\, val_{\text{pre}\langle I \rangle}(e_1) = val_{\text{post}\langle I \rangle}(e_2)$ holds.

$$\mathcal{D}(P, e_1, e_2) = \begin{cases} \text{true} & \text{if } P \text{ is a semantic nop,} \\ & \text{i.e. } \phi(P, e_1, e_2) \text{ holds} \\ \perp & \text{otherwise} \end{cases}$$

Similarly, we can define decision procedures that determine whether $\neg\phi(P, e_1, e_2)$ holds (in this case, the result of $\mathcal{D}(P, e_1, e_2)$ is "false" or $\perp$). We denote by $\mathcal{D}^+$ a decision procedure for $\phi(P, e_1, e_2)$, and by $\mathcal{D}^-$ a decision procedure for $\neg\phi(P, e_1, e_2)$.

As the value preservation queries are frequent in our algorithm (possibly at every step during node matching), the prototype use a collections of decision procedures ordered by their precision and cost. Intuitively, the most naïve decision procedures are the least precise, but the fastest to execute. If a $\mathcal{D}^+$-style decision procedure reports "true" on some input, all $\mathcal{D}^+$-style decision procedures following it in the ordered collection will also report "true" on the same input. Similarly, if a $\mathcal{D}^-$-style decision procedure reports "false on some input, all $\mathcal{D}^-$-style decision procedures following it in the ordered collection will also report "false" on the same input. As both $\mathcal{D}^+$- and $\mathcal{D}^-$-style decision procedures are sound, we define the order between $\mathcal{D}^+$ and $\mathcal{D}^-$ decision procedures based only on performance.

This collection of decision procedures provides us with an efficient algorithm for testing whether a program fragment P preserves expression values: iterate through the ordered collection of decision procedures, querying each $\mathcal{D}_i$, and stop when one of them returns "true", respectively "false" for $\mathcal{D}^-$-style decision procedures. This algorithm provides for incrementally expensive and powerful checking of the program fragment, in step with its complexity: program fragments that are simple semantic nops will be detected early by decision procedures in the ordered collection. Complex value preserving fragments will require passes through multiple decision procedures. We present, in order, four decision procedures that are part of our prototype.

Nop Library $\mathcal{D}^+_{\textbf{NOP}}$. This decision procedure identifies sequences of actual `nop` instructions, which are processor-specific instructions similar to the *skip* statement in some high-level programming languages, as well as predefined instruction sequences known to be semantic nops. Based on simple pattern matching, the decision procedure annotates basic blocks as nop sequences where applicable. If the whole program fragment under analysis is annotated, then it is a semantic nop. The nop library can also act as a cache for queries already resolved by other decision procedures.

Randomized Symbolic Execution $\mathcal{D}^-_{\textbf{RE}}$. This oracle is based on a $\mathcal{D}^-$-style decision procedure using randomized execution. The program fragment is executed using a random initial state (i.e. the values in registers and memory are chosen to be random). At completion time, we check whether it is true that $\neg\phi(P, e_1, e_2)$: if true, at least one path in the program fragment is not a semantic nop, and thus the whole program fragment is not a semantic nop.

Theorem Proving $\mathcal{D}^+_{\textbf{ThSimplify}}$. The value preservation problem can be formulated as a theorem to be proved, given that the program fragment has no loops. We use the Simplify theorem prover [14] to implement this oracle: the program fragment is represented as a state transformer δ, using each program register and memory expression converted to SSA form. We then use Simplify to prove the formula $\delta \Rightarrow \phi(P, e_1, e_2)$, in order to show that all paths through the program fragment are semantic nops under $\phi(P, e_1, e_2)$. If Simplify confirms that the formula is valid, the program fragment is a semantic nop. One limitation of the Simplify theorem prover is the lack of bit-vector arithmetic, which binary programs are based on. Thus, we can query Simplify only on programs that do not use bit-vector operations.

Theorem Proving $\mathcal{D}^+_{\textbf{ThUCLID}}$. A second theorem proving oracle is based on the UCLID infinite-state bounded model checker [22]. For our purposes, the logic supported by UCLID is a superset of that supported by Simplify. In particular, UCLID precisely models integer and bit-vector arithmetic. We model the program fragment instructions as state transformers for each register and for memory (represented as an uninterpreted function). UCLID then simulates the program fragment for a given number of steps and determines whether $\phi(P, e_1, e_2)$ holds at the end of the simulation.

Obfuscation transformation	*Handled by $\mathcal{A}_{MD}$?*
Instruction reordering	✓
Register renaming	✓
Garbage insertion	✓
Instruction replacement	*limited*
Equivalent functionality	✗
Reordered memory accesses	✗

Table 3. Obfuscation transformations addressed by our malware detection algorithm and some limitations.

For illustration, consider Figure 3: the value preservation problem consists of the program fragment R, created from program nodes 2, 3, and 4, and the value predicate $\phi \equiv \forall I \in R \,.\, val_{\text{pre}\langle I\rangle}(eax) = val_{\text{post}\langle I\rangle}(ecx-1)$. To use the Simplify theorem proving oracle, the formula shown in Table 2 is generated from program fragment R.

3.4. Strengths and limitations

For the algorithm to be effective against real-life attacks, it has to "undo" various obfuscations and other program transformations that a malware writer might use. We list the strengths and weaknesses of our algorithm in Table 3. We discuss below in detail four classes of obfuscations algorithm $\mathcal{A}_{MD}$ can handle: code reordering, equivalent instruction replacement, register renaming, and garbage insertion.

Code reordering is one of the simplest obfuscations hackers use to evade signature matching. The obfuscation changes the order of instructions on disk, in the binary image, while maintaining the execution order using jump instructions. This obfuscation is handled by the use of control flow graphs. *Register renaming* is a simple obfuscation that reassigns registers in selected program fragments. As a result, a signature matching tool will fail to detect any obfuscated variant as long as it searches for a signature with a specific register. Our template matching algorithm avoids this pitfall by using templates. The uninterpreted variables are assigned corresponding program registers and memory locations only during unification and, thus, the matching algorithm can identify any program with the same behavior as the template, irrespective of the register allocation.

Garbage insertion obfuscates a program by inserting instruction sequences that do not change the behavior of the program. Algorithm $\mathcal{A}_{MD}$ tackles this class of obfuscations through the use of decision procedures to reason about value predicates on def-use chains. *Equivalent instruction replacement* uses the rich instruction set available on some architectures, notably on the Intel

Instruction sequence:

```
2: ebx  =  0x400000
3: nop
4: ecx  =  eax + 1
```

Value predicate:
$\forall I \in R \,.\, val_{\text{pre}\langle I\rangle}(eax) = val_{\text{post}\langle I\rangle}(ecx - 1)$

Simplify formula:

```
(IMPLIES (AND (EQ ebx_1 4194304)
              (EQ ecx_4 (+ eax_pre 1) )
              (EQ ecx_post ecx_4)
          )
         (EQ eax_pre (- ecx_post 1) )
  )
```

Table 2. Example of Simplify query corresponding to program fragment R and value predicate ϕ.

IA-32 (x86), to replace groups of consecutive instructions with other short instruction sequences that have the same semantics. For example, to add 1 to register X in the x86 architecture, one can use the `inc` X (increment) instruction, or the `add` X`, 1` instruction, or the `sub` X`, -1` instruction. We handle a limited kind of instruction replacement by normalizing the code into an intermediate representation (IR) with semantically-disjoint operations.

Limitations. Our tool has few limitations. First, the current implementation requires all of the IR instructions in the template to appear in the same form in the program. For example, if an IR node in the template contains "`x = x * 2`", the same operation (an assignment with a multiplication on the right hand side) has to appear in the program for that node to match. This means that we will not match the equivalent program instruction "`eax = eax << 1`" (arithmetic left shift). Attacks against this requirement are still possible, but are fairly hard, as the bar has been raised: the attacker has to devise multiple equivalent, yet distinct, implementations of the same computation, to evade detection. This is not an inherent limitation of the semantics, and can be handled using an IR normalization step.

The second limitation comes from the use of def-use chains for value preservation checking. The def-use relations in the malicious template effectively encode a specific ordering of memory updates – thus, our algorithm $\mathcal{A}_{MD}$ will detect only those program that exhibit the same ordering of memory updates. We note that automatically generating functionally-equivalent variants is a hard problem. Handling obfuscations that reorder instructions to achieve a different ordering of memory updates is one goal of our ongoing research.

4. Experimental results

We evaluated our implementation of algorithm $\mathcal{A}_{MD}$ against real-world malware variants. The three major goals of our experiments were to develop malicious behavior templates that can match malware variants, to measure the false positive rates the algorithm generates with these templates, and to measure the detection algorithm's resilience to obfuscation. We used malware instances in the wild both for developing behavior templates and as starting point for obfuscation transformations in the obfuscation resilience testing. Highlights of the evaluation are:

- *The template-based algorithm detects worms from the same family*, as well as unrelated worms, using a single template.
- *No false positives* were generated when running our malware detector on benign programs, illustrating the soundness of our algorithm in its current implementation.
- The algorithm exhibits *improved resilience to obfuscation* when compared to commercial anti-virus tool McAfee VirusScan.

4.1. Variant detection evaluation

We developed two templates and tested malware samples and benign programs against them. One template captures the decryption-loop functionality common in many malicious programs, while the other template captures the mass-mailing functionality common to email worms. While throughout this section we use only the decryption-loop template as example, the results from using the mass email template were similar. For malware samples we used seven variants of Netsky (B, C, D, O, P, T, and W), seven variants of B[e]agle (I, J, N, O, P, R, and Y), and seven variants of Sober (A, C, D, E, F, G, and I). All of them are email worms, each with many variants in the wild, ranging in size from 12 kB to 75 kB.

Malware family	*Template detection*		*Running time*	
	Decryption loop	*Mass-mailer*	*Avg.*	*Std. dev.*
Netsky	100%	100%	99.57 s	41.01 s
B[e]agle	100%	100%	56.41 s	40.72 s
Sober	100%	0%	100.12 s	45.00 s

Table 4. Malware detection using algorithm $\mathcal{A}_{MD}$ for 21 e-mail worm instances.

The decryption-loop template describes the behavior of programs that unpack or decrypt themselves to memory: the template consists of (1) a loop that processes data from a source memory area and writes data to a destination memory area, and (2) a jump that targets the destination area. Many binary worms use this technique to make static analysis harder: the actual code of the worm is not available for inspection until runtime. To construct the template, we analyzed the Netsky.B email worm and manually extracted the code that performs the decryption and the jump. The template was then generalized from this code by replacing actual registers and memory addresses with variables. The mass-mailing template was developed in a similar manner: it describes the custom SMTP functionality present in the worm, including email composition and transmission.

We applied our algorithm on each malware instance and each decryption-loop template. We achieved 100% detection of the Netsky and B[e]agle worm instances using only the two templates (the mass-mailing template required minor tweaks to detect B[e]agle). The Sober worm instances did not match the mass-mailing template, requiring a separate template. Due to limitations in the current prototype implementation, matching calls into the Microsoft Visual Basic runtime library (used by Sober worm instances) is not supported. Nonetheless, we have shown that algorithm $\mathcal{A}_{MD}$ can detect multiple worm instances from the same family using one template, in contrast with commercial virus scanners that require many signatures (McAfee VirusScan uses 13 signatures to detect the worms in our test set). Our detection results come with one caveat: 3 of the Netsky variants could not be processed by IDA Pro. Since we build our tool on the assumption that disassembly can be performed successfully, we do not consider those 3 variants as false negatives. Running times for the tool (listed in Table 4) are satisfactory for a prototype and suggest that real-time performance can be achieved with an optimized implementation.

4.2. False positive evaluation

We evaluated the algorithm and the templates against a set of benign programs, in order to measure the false positive rate and to evaluate the templates we developed. We used a set of 2,000 programs with sizes between <1 kB and 150 kB, from standard Microsoft Windows system programs, to compiler tools, to Microsoft Office programs. We have also tried to test larger benign binaries, with sizes up to 5 MB, but they did not disassemble successfully. For successfully disassembled programs, the false positive rate was 0%, meaning that our implementation $\mathcal{A}_{MD}$ correctly classified all programs as benign (none of the test programs matched the templates).

In Figure 4, we present the results of this evaluation: programs are grouped by size, in 5 kB increments, and the disassembly and detection rates are plotted for each group. For example, the bar to the right of the 71,680 B point represents programs between 71,680 B and 76,799 B in size, and indicates that 97.78% of the programs in this group were disassembled successfully and were detected as benign, while 2.22% failed to disassemble (either because the disassembler crashed, or because it failed to group program code into functions). The running time per test case ranged from 9.12 s to 413.78 s, with an average of 165.5 s.

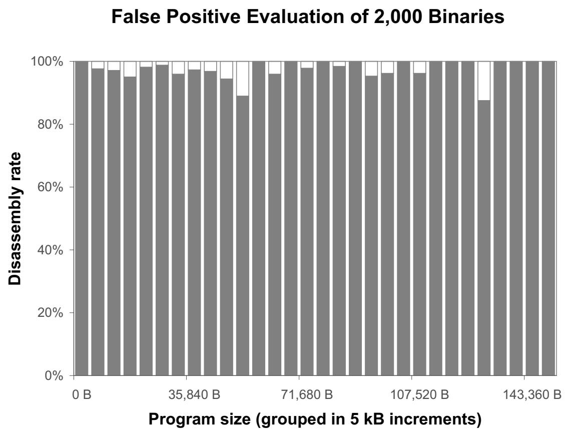


Figure 4. Evaluation of algorithm $\mathcal{A}_{MD}$ on a set of 2,000 benign Windows programs yielded no false positives. Gray bars indicate the percentage of programs that disassembled correctly and were detected as benign; white bars indicate the percentage of programs that failed to disassemble.

4.3. Obfuscation resilience evaluation

To test resilience to obfuscation, we applied garbage insertion transformations of varying degrees of complexity to worm variant B[e]agle.Y, and compared the detection rate of our malware detection tool against McAfee VirusScan. The garbage insertion transformation adds code fragments to a program, such that each code fragment is irrelevant in its insertion context. We considered three types of garbage insertion: (1) *nop insertion* adds simple sequences of nop instructions; (2) *stack-op insertion* adds sequences of stack operations; and (3) *math-op insertion* adds sequences of arithmetic operations. We generated 20 variants for each type of obfuscation. To test the limits of our implementation, one of the *math-op insertion* transformations actually replaced single instructions of the form x = x + const with equivalent sequences. The results are tabulated in Table 5: our tool catches all obfuscations with the exception of the math-op replacement transformation; in such a case, since the algorithm seeks to exactly match a template instruction of the form x = x + const, any equivalent instruction sequences are not detected.

Obfuscation type	*Algorithm $\mathcal{A}_{MD}$*		*McAfee VirusScan*
	Average time	*Detection rate*	
Nop	74.81 s	100%	75.0%
Stack op	159.10 s	100%	25.0%
Math op	186.50 s	95%	5.0%

Table 5. Evaluation of algorithm $\mathcal{A}_{MD}$ on a set of obfuscated variants of B[e]agle.Y. For comparison, we include the detection rate of McAfee VirusScan.

5. Related work

We compare our work to existing research in the areas of malware detection, translation validation, and software verification.

5.1. Malware detection

In previous work [8], we demonstrated that current commercial virus scanners can be easily evaded by using simple obfuscations used by hackers. Many malware-detection techniques are based on static-analysis of executables. In this area, we previously described a malware-detection algorithm called SAFE [7]. SAFE can only handle very simple obfuscations (only nops can appear between matching instructions), e.g., the example shown in Figure 1 cannot be handled by SAFE. Moreover, the formal semantics of malware detection was not explored by the earlier work. Static analysis of binaries is used by Kruegel *et al.* [21] to detect kernel-level rootkits, which are attack tools that are used by hackers to hide their presence from system administrators. They look for suspicious instruction sequences using symbolic execution. The detection algorithm presented in this paper is more powerful because we use multiple decision procedures (symbolic execution being just one of them). Our specification of malicious behavior is richer than the one used by Kruegel *et al.* Therefore, using our algorithm can result in a more powerful tool for detecting kernel-level rootkits. We will explore this interesting application of our algorithm in the future. Singh and Lakhotia [33] provide an annotated bibliography of papers on malware analysis and detection.

An essential step in statically analyzing executables is disassembly, which translates machine code to assembly code. Linn and Debray [23] demonstrate that simple obfuscations can thwart the disassembly process. The abovementioned research demonstrates the importance of handling obfuscations in malware detection. Kruegel *et al.* [20] present techniques for disassembling obfuscated executables. In this paper, we assume that the malware being analyzed can be disassembled.

The theoretical limits of malicious code detection (specifically of virus detection) have been the focus of many researchers. A virus is a malware that replicates itself by copying its code to other program files. Cohen [10] and Chess-White [6] showed that in general the problem of virus detection is undecidable. Spinellis [34] proved that problem of reliably identifying a bounded-length virus is NP-complete. These results are similar to the result presented in this paper (see Theorem 1). However, our formal model of malware detection is different than the one used by these researchers: we present a semantics and an algorithm for detecting *specific* malicious behaviors, avoiding the problem of undecidability (as proven by Cohen) and the problem of undetectable malware classes (as introduced by Chess and White).

Dynamic monitoring can also be used for malware detection. Cohen [10] and Chess-White [6] propose a virus detection model that executes code in a sandbox. Another approach rewrites the binary to introduce checks driven by an enforceable security policy [15] (known as the *inline reference monitor* or the *IRM* approach). We believe static analysis can be used to improve the efficiency of dynamic analysis techniques, e.g., static analysis can remove redundant checks in the IRM framework. In the future we will explore hybrid static-dynamic approaches to malware detection.

5.2. Translation validation

A translation-validation [12, 16, 31, 32] system attempts to verify that the semantics of a program is not changed by an optimizing transformation. Differences between malware detection and translation validation were discussed in the introduction.

5.3. Software verification

Our work is closely related to previous results on static analysis techniques for verifying security properties of software [1, 3–5, 19, 24]. In a larger context, our work is similar to existing research on software verification [2, 11]. However, there are several important differences. First, since these systems analyze benign code, they do not have to handle obfuscations. Second, to our knowledge, all existing work on static analysis techniques for verifying security properties analyzes source code. On the other hand, our analysis technique works on executables.

6. Conclusion

We observe that certain malicious behaviors (such as decryption loops) appear in all variants of a certain malware. Based on this intuition, we gave a formal semantics for malware detection. We also presented a malware-detection algorithm that is sound with respect to our semantics. Experimental evaluation demonstrated that our algorithm can detect all variants of certain malware, has no false positives, and is resilient to obfuscation transformations generally used by hackers.

There are several opportunities for further work. In the future we will address the limitations discussed in Section 3 (see Table 3). We also plan to optimize our tool to reduce the execution times.

Acknowledgments. We are thankful to our anonymous reviewers for their invaluable comments. We would also like to express our gratitude to Giovanni Vigna, our shepherd throughout the revision process, for his feedback.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (Oakland'02)*, pages 143–159, May 2002.

[2] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, Toronto, Ontario, Canada, 2001. Springer-Verlag Heidelberg.

[3] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2), 1996.

[4] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *9th ACM Conference on Computer and Communications Security (CCS'02)*. ACM Press, November 2002.

[5] B. Chess. Improving computer security using extending static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (Oakland'02)*, pages 160–173, May 2002.

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

[7] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186. USENIX Association, USENIX Association, Aug. 2003.

[8] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2004 (ISSTA'04)*, pages 34–44, Boston, MA, USA, July 2004. ACM Press.

[9] M. Ciubotariu. Netsky: a conflict starter? *Virus Bulletin*, pages 4–8, May 2004.

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

[11] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448. ACM Press, 2000.

[12] D. W. Currie, A. J. Hu, and S. Rajan. Automatic formal verification of DSP software. In *Proceedings of the 37th Annual ACM IEEE Conference on Design Automation (DAC'00)*, pages 130–135. ACM Press, 2000.

[13] DataRescue sa/nv. IDA Pro – interactive disassembler. Published online at `http://www.datarescue.com/idabase/`. Last accessed on 3 Feb. 2003.

[14] D. Detlefs, G. Nelson, and J. Saxe. The Simplify theorem prover. Published online at `http://research.compaq.com/SRC/esc/Simplify.html`. Last accessed on 10 Nov. 2004.

[15] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (Oakland'00)*, pages 246–255, May 2000.

[16] X. Feng and A. J. Hu. Automatic formal verification for scheduled VLIW code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES'02)*, pages 85–92. ACM Press, 2002.

[17] J. Gordon. Lessons from virus developers: The Beagle worm history through April 24, 2004. In *SecurityFocus Guest Feature Forum*. SecurityFocus, May 2004. Published online at `http://www.securityfocus.com/guest/24228`. Last accessed: 9 Sep. 2004.

[18] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. 2004 CSI/FBI computer crime and security survey. Technical report, Computer Security Institute, 2004.

[19] T. Jensen, D. Métayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Oakland'99)*, May 1999.

[20] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium (Security'04)*, San Diego, CA, Aug. 2004.

[21] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, Tucson, AZ, Dec. 2004.

[22] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In R. Alur and D. A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478, Boston, MA, USA, July 2004. Springer-Verlag Heidelberg.

[23] 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)*, Oct. 2003.

[24] R. Lo, K. Levitt, and R. Olsson. MCF: A malicious code filter. *Computers & Society*, 14(6):541–566, 1995.

[25] LURHQ Threat Intelligence Group. Sobig.a and the spam you received today. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig.html. Last accessed on 16 Jan. 2004.

[26] LURHQ Threat Intelligence Group. Sobig.e - Evolution of the worm. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig-e.html. Last accessed on 16 Jan. 2004.

[27] LURHQ Threat Intelligence Group. Sobig.f examined. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig-f.html. Last accessed on 16 Jan. 2004.

[28] A. Marinescu. Russian doll. *Virus Bulletin*, pages 7–9, Aug. 2003.

[29] G. McGraw and G. Morrisett. Attacking malicious code: report to the Infosec research council. *IEEE Software*, 17(5):33 – 41, Sept./Oct. 2000.

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

[31] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 83–94. ACM Press, June 2000.

[32] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag Heidelberg, Mar. 1998.

[33] P. Singh and A. Lakhotia. Analysis and detection of computer viruses and worms: An annotated bibliography. *ACM SIGPLAN Notices*, 37(2):29–35, Feb. 2002.

[34] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, Jan. 2003.

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

[36] z0mbie. z0mbie's homepage. Published online at http://z0mbie.host.sk. Last accessed: 16 Jan. 2004.

A. Proof of Theorem 2

Let P be a program and T a template such that $\mathcal{A}_{MD}(P,T)$ returns yes. Let f be a function from the set of nodes of P to nodes in T that satisfy the two conditions given in section 3 (such a function exists because the $\mathcal{A}_{MD}(P,T)$ returned yes). We will prove that $P \models T$.

There are two set of states - one for the program (which we simply refer to as state) and other for the template (which we refer to as template state). Recall that a state s of program is a 3-tuple comprising of an assignment $val(s)$ of values to registers, value $pc(s)$ for the program counter, and the memory contents $mem(s)$. Consider a node n in the template T and a node m in P such that $f(m) = n$. Let s be a state such that $pc(s) = m$ (in this proof we equate the program counter with the CFG node it corresponds to), and $B(\cdot, n, m)$ be the binding. We define a template state corresponding to s (denoted by $T(s)$) as follows:

- Let $B(val(s))$ be the assignment implied by the binding B, i.e., $B(val(s))(X)$ is equal to the value of the expression $B(X,n,m)$ in the state s.
- Define $pc(T(s))$ to be the program counter corresponding to node n.
- Define $mem(T(s))$ as $mem(s)$.

Let n_0 be the initial node of the template and m_0 be a node in the program such that $f(m_0) = n_0$. Let s_0 be a state such that $pc(s_0) = m_0$, and define s_0^T as $T(s_0)$. Recall that $I(n)$ denotes the instruction corresponding to node n. Given a state s, $I(n)(s)$ denotes the state obtained by executing instruction $I(n)$ from state s, and $e(I(n), s)$ denotes the event generated by executing instruction $I(n)$ from state s. Let s_1' be equal to $I(m_0)(s_0)$ and e_1 be equal to $e(I(m_0), s_0)$. Let $s_1^T = I(n_0)(s_0^T)$ and $e_1^T = e(I(n_0), s_0^T)$. Since n_0 and m_0 are unifiable, it is easy to see that $e_1^T = e_1$. Let n_1 be the template node corresponding to $pc(s_1^T)$. Assume that in program P, from state s_1' there is a feasible path $path(s_1', m_1)$ from $pc(s_1')$ to a node m_1, such that $f(m_1) = n_1$, i.e., executing $path(s_1', m_1)$ from state s_1' yields a state s_1, such that $pc(s_1) = m_1$. We denote this fact by $s_0 \overset{e_1}{\rightsquigarrow} s_1$, i.e., there is an execution sequence α that starts at s_0 and ends at s_1 and the first event generated by α is e_1. Thus we obtain the following execution sequences in P and T:

$$\begin{aligned} \sigma &= s_0 \overset{e_1}{\rightsquigarrow} s_1 \\ \sigma_T &= s_0^T \xrightarrow{e_1^T} s_1^T \end{aligned}$$

Recall that $use(n_1)$ is the set of variables used by instruction $I(n_1)$. For every variable $X \in use(n_1)$, value of expression $B(X, n_1, m_1)$ is unchanged by $path(s_1', m_1)$. Therefore, executing instruction $I(n_1)$ in the template from the states $T(s_1)$ and $s_1^T = I(n_0)(s_0)$ results in same memory accesses, generate the same event, and transitions to the same node n_2 in the template. We continue augmenting the execution sequences σ and σ_T until we obtain a template state s_k^T such that $pc(s_k^T)$ corresponds to the final node of the template. At the end, we have two execution sequences

$$\begin{aligned} \sigma &= s_0 \overset{e_1}{\rightsquigarrow} s_1 \overset{e_2}{\rightsquigarrow} s_2 \cdots \overset{e_k}{\rightsquigarrow} s_k \\ \sigma_T &= s_0^T \xrightarrow{e_1^T} s_1^T \xrightarrow{e_2^T} s_2^T \cdots \xrightarrow{e_k^T} s_k \end{aligned}$$

where σ and σ_T are execution sequences in the program and the template. Moreover, for $1 \le i \le k$, $e_i = e_i^T$ and template states s_i^T and $T(s_i)$ satisfy the following property:

- Executing instruction $I(pc(s_i))$ in the program P from the state s_i and instruction $I(pc(s_i^T))$ in the template from the state $T(s_i)$ results in same memory accesses and generate the same event.
- In the template T, executing instruction $I(pc(s_i^T))$ from the state s_i^T and from the state $T(s_i)$ results in same memory accesses and generate the same event.

The proof of abovementioned property uses the condition that def-use paths are preserved by the program. From the above two observations it is easy to see that $P \models T$.

B. Intermediate representation language

The intermediate representation (IR) of a program is structured as a collection of control flow graphs (CFGs), each node in the graph corresponding to one IR instruction. Edges in a CFG are unlabeled, with the exception of the outgoing edges of a conditional control flow instruction. In such a case, the IR instruction is labeled jump, and the outgoing edges are labeled with the corresponding condition that holds true along that branch.

IR instructions are elements in a language with simple rules, illustrated in Table 6. Each IR instruction (or IR term) is either an assignment that updates state variables or a call to other functions in the same program, or to external functions (library or system routines). An IR expression uses arithmetic, bit-vector, and relational operators, as well as the special memory addressing operator.

term	::	*expr*$_1$ **ASSIGN** *expr*$_2$
	|	**CALL** *expr*
expr	::	*expr*$_1$ *op* *expr*$_2$
	|	ident **(** *expr*, … **)**
	|	**memory [** *expr* **]**
	|	ident | number | string
op	::	**OR** | **AND** | **XOR** | **NOT** | **BITAND**
	|	**BITOR** | **BITXOR** | **BITNOT**
	|	**EQUAL** | **LESSTHAN**
	|	**GREATERTHAN** | **LEQ** | **GEQ**
	|	**BITSHIFTLEFT** | **BITSHIFTRIGHT**
	|	**PLUS** | **MINUS** | **STAR** | **DIV** | **MOD**

Table 6. Term algebra for intermediate representation (IR) expressions.

C. Def-use paths and value predicates

We list here the complete set of value predicates and corresponding program fragments that have to be proven as semantic nops for the program in Figure 3 to satisfy the template in the same figure. In Table 7, we list the 10 def-use chains in the template, the derived value predicates that relate program expressions, and the corresponding program fragments. The notation $n \xrightarrow{x} n'$ describes the variable corresponding to the def-use path, such that $def(n) \cap use(n') = \{x\}$.

Def-use chain and value predicate	*Program fragment*
T1 $\xrightarrow{A}$ T3 $val_{\text{post}\langle I\rangle}(ecx - 1) = val_{\text{pre}\langle I\rangle}(eax)$	2: ebx = 0x400000 3: nop 4: ecx = eax + 1
T5 $\xrightarrow{A}$ T3 $val_{\text{post}\langle I\rangle}(ecx - 1) = val_{\text{pre}\langle I\rangle}(eax)$	9: ebx = ebx + 1 4: ecx = eax + 1
T1 $\xrightarrow{A}$ T4 $val_{\text{post}\langle I\rangle}(eax) = val_{\text{pre}\langle I\rangle}(eax)$	2: ebx = 0x400000 3: nop 4: ecx = eax + 1 6: eax = ecx - 1
T5 $\xrightarrow{A}$ T4 $val_{\text{post}\langle I\rangle}(eax) = val_{\text{pre}\langle I\rangle}(eax)$	9: ebx = ebx + 1 4: ecx = eax + 1 6: eax = ecx - 1
T2 $\xrightarrow{B}$ T4 $val_{\text{post}\langle I\rangle}(ebx) = val_{\text{pre}\langle I\rangle}(ebx)$	3: nop 4: ecx = eax + 1 6: eax = ecx - 1
T6 $\xrightarrow{B}$ T4 $val_{\text{post}\langle I\rangle}(ebx) = val_{\text{pre}\langle I\rangle}(ebx)$	4: ecx = eax + 1 6: eax = ecx - 1
T1 $\xrightarrow{A}$ T5 $val_{\text{post}\langle I\rangle}(eax) = val_{\text{pre}\langle I\rangle}(eax)$	2: ebx = 0x400000 3: nop 4: ecx = eax + 1 6: eax = ecx - 1 7: mem[ebx] = mem[eax] ˆ 5
T5 $\xrightarrow{A}$ T5 $val_{\text{post}\langle I\rangle}(eax) = val_{\text{pre}\langle I\rangle}(eax)$	9: ebx = ebx + 1 4: ecx = eax + 1 6: eax = ecx - 1 7: mem[ebx] = mem[eax] ˆ 5
T2 $\xrightarrow{B}$ T6 $val_{\text{post}\langle I\rangle}(ebx) = val_{\text{pre}\langle I\rangle}(ebx)$	3: nop 4: ecx = eax + 1 6: eax = ecx - 1 7: mem[ebx] = mem[eax] ˆ 5 8: eax = eax + 1
T6 $\xrightarrow{B}$ T6 $val_{\text{post}\langle I\rangle}(ebx) = val_{\text{pre}\langle I\rangle}(ebx)$	4: ecx = eax + 1 6: eax = ecx - 1 7: mem[ebx] = mem[eax] ˆ 5 8: eax = eax + 1

Table 7. Def-use paths, value predicates, and program paths for the example in Figure 3.