# Software Transformations to Improve Malware Detection

Mihai Christodorescu[1], Somesh Jha[1], Johannes Kinder[2], Stefan Katzenbeisser[2], and Helmut Veith[2]

[1] University of Wisconsin, Madison, {mihai,jha}@cs.wisc.edu
[2] Technische Universität München, {kinder,katzenbe,veith}@in.tum.de

**Abstract.** Malware is code designed for a malicious purpose, such as obtaining root privilege on a host. A malware detector identifies malware and thus prevents it from adversely affecting a host. In order to evade detection, malware writers use various obfuscation techniques to transform their malware. There is strong evidence that commercial malware detectors are susceptible to these evasion tactics.
In this paper, we describe the design and implementation of a *malware transformer* that reverses the obfuscations performed by a malware writer. Our experimental evaluation demonstrates that this malware transformer can drastically improve the detection rates of commercial malware detectors.

**Key words:** malware detection, program transformation, deobfuscation

## 1 Introduction

*Malware* is code that has malicious intent. Examples of this kind of code include computer viruses, worms, Trojan horses, and backdoors. Malware can infect a host using a variety of techniques, such as exploiting software flaws, embedding hidden functionality in regular programs, and social engineering. A classification of malware with respect to propagation methods and goals was given by McGraw and Morrisett in 2000 [1]. A *malware detector* identifies and contains malware before it can reach a system or network.

Current malware detectors are based on *scan strings* [2] or signatures, i.e., suspicious byte sequences of instructions and data. Malware analysts extract a scan string from a virus sample in such a way that the scan string is typical of the virus but not likely to be found in benign programs. Scanners can very efficiently determine if a file shares a byte sequence with a known malware instance. In this case, the file is either infected by the malware or in fact a copy of the malware. Thus, scanning for malware reduces to a search for particular byte sequences in a suspicious file. Scan strings and slightly more general regular expressions over bytes are widely used in practice today, because detection is efficient and has a low false positive rate, with individual scan strings tailored for each known malware instance.

The high specificity of scan-string matching however opens the door for *malware obfuscation* where malware writers transform malicious code to make it unrecognizable. For example, polymorphic and metamorphic viruses and, more recently, polymorphic shellcodes [3] are specifically designed to bypass detection tools. Both polymorphic and metamorphic viruses manipulate their code and data in such a way that the

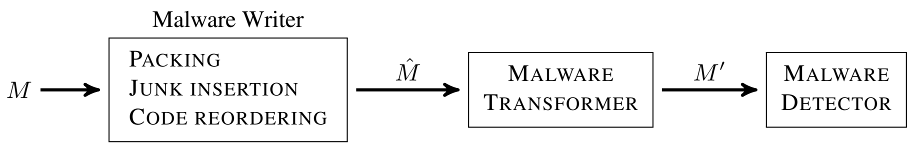


**Fig. 1.** Malware transformation reverses the effects of obfuscations. The malware instance $\hat{M}$ (an obfuscated variant of $M$) is transformed into $M'$ before detection.

malicious behavior is preserved but the current scan-string signatures no longer match. There is strong evidence that commercial malware detectors are susceptible to common evasion techniques used by malware writers. Malware testing has shown that malware detectors cannot cope with obfuscated versions of worms [4], and there are numerous examples of obfuscation techniques designed to avoid detection [5,6,7,8,9,10].

In this paper, we define a *malware transformer* as a system that takes an obfuscated executable and transforms it into an executable free of obfuscations. Therefore, a malware transformer can be used to *enhance the detection capabilities of existing malware detectors which use scan strings*. In the light of over 60000 known viruses, reusability of virus signatures becomes the sine qua non of advanced methods for malware detection. Since the problem of transforming the executable into an unobfuscated state is completely orthogonal to the scan string detection, our method can be used as a preprocessor or filter that improves the detection rate of all existing virus scanners. Obfuscation [11,12] increases the complexity [13,14,15] of a program to make reverse engineering harder. Techniques presented in this paper can be viewed as deobfuscation techniques designed to reduce the complexity of malware with the goal of improving the detection rates of scan-strings based malware detectors. We want to emphasize that our techniques are targeted towards three common obfuscations frequently used by malware writers. We describe the design and implementation of a malware transformer that handles three of the most important practical obfuscations, namely *packing*, *code reordering*, and *junk insertion*. Our malware transformer outputs another (deobfuscated) executable and is thus compatible with all commercial malware detectors. Our experimental results demonstrate that the detection rates of four commercial malware detectors are dramatically improved by our method, in particular, in those cases where the scan string describes instruction sequences rather than data sequences. This is no surprise, as the obfuscation methods considered—with the single exception of the packing obfuscation—affect the code, and not the data of the malware. Summarizing, this paper makes the following contributions:

1. We present the design and implementation of a malware transformer that handles three important obfuscations used by malware writers. In Section 2, we give an overview of our malware transformer; details of the algorithms used by the malware transformer appear in Section 3. The methods are presented in such a way as to decouple malware transformation and classical program analysis algorithms, which our malware transformer uses as oracles. We separately describe implementations for the analysis algorithms in Section 4. Thus, our malware transformer can be further improved by plugging in stronger program analysis capabilities.

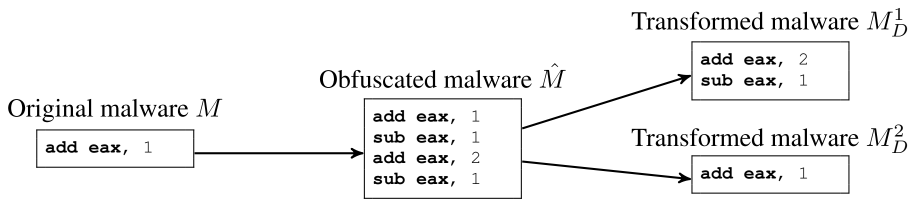


**Fig. 2.** Example of ambiguity in the malware transformation process. Original code (left) is obfuscated using junk insertion (center).

2. We evaluate the detection rates of four commercial virus scanners, Norton AntiVirus, McAfee VirusScan, Sophos Anti-Virus, and ClamAV, against variants of ten known viruses, listed in Section 5.2. Our experiments demonstrate that the malware transformer improves the detection rate of all four scanners. Most strikingly, in the case of Sophos Anti-Virus, the detection rate jumps from 7.5% to 100% after deobfuscating junk insertions. Details of our experimental evaluation appear in Section 5.

We finally note that we see other future applications for our malware transformer, for example in forensics where the deobfuscation transformations performed by our tool increase the human readability of the code.

## 2 Threat Model

We consider a threat model in which a known malware instance is obfuscated to obtain another, equally malicious instance. This obfuscation step can be manual, performed by a malware writer, or automatic, executed during the replication phase of a virus. By obfuscating an existing virus, one can generate an enormous amount of new and undetected pieces of malware. Obfuscation of a program can be achieved in numerous ways; in most cases, however, the set of obfuscations that are actually used is rather limited. In particular, most malware authors rely on the use of external tools that allow them to obfuscate their programs after compilation. We therefore focus on common obfuscations to demonstrate the effect of a malware transformer on the detection rate of commercial malware detectors.

We apply the malware transformations to an obfuscated malware instance and attempt to recover the original, unobfuscated malware instance. Figure 1 illustrates the threat model and the application of malware transformations. A malware instance $M$ (i.e., a malicious executable binary program) is obfuscated to obtain a new malware instance $\hat{M}$; the malware transformer processes the obfuscated malware $\hat{M}$ to produce a deobfuscated executable program $M_D$ that is then checked by the malware detector.

We observe that it is possible that the original malware instance $M$ and the transformed malware instance $M_D$ are distinct. In other words, the obfuscations applied to the original malware instance $M$ cannot always be perfectly reversed. For example, consider the example from Figure 2 where a code fragment (left box) is obfuscated using junk insertion (middle box). The malware transformation could yield one of two equivalent possibilities. We resolve this ambiguity by having the transformation choose one of the outcomes in a consistent manner. In our description of the malware transformation algorithms (in Section 3), we outline the way these choices are made. As a result, the original instance $M$ and all its obfuscated variants $\hat{M}$ are transformed to the same malware instance $M_D$.

This approach to handling ambiguity during malware transformation requires a change in the method for producing scan-strings from malware samples. Since a malware transformation does not guarantee us the recovery of the original malware instance $M$, the scan-string signature cannot be based on the $M$. Rather, the malware transformation consistently outputs the transformed malware instance $M_D$ when given any obfuscated variant $\hat{M}$ or the original instance $M$. It is thus necessary to *base the scan-string signature on a transformed malware instance* $M_D$, obtained from a sample $M$ for which we wish to generate a signature. Other than this procedural change, our technique places no other requirements on scan-string malware detectors.

Our malware transformer is capable of handling three practically relevant kinds of obfuscation:

– The *code reordering obfuscation* changes the location of instructions in a program while maintaining the original execution order through the insertion of jump instructions. This permutation is randomized, resulting in a large possible number of obfuscated instances of the original program. Sequences of the original binary code cannot be found in the obfuscated instances, causing scan string-based malware detectors to fail.
– The *junk-insertion obfuscation* randomly inserts contiguous code sequences that do not affect program behavior. We call such code sequences *semantic nops*. Motivated by real world obfuscation engines we consider sequences that are self-contained, produce no output, and preserve all program variables.
– The *packing obfuscation* replaces a binary (code and data) sequence with a data block containing the binary sequence in packed form (encrypted or compressed) and a decryption routine that, at runtime, recovers the original binary sequence from the data block. The result of the packing obfuscation is a program that dynamically generates code in memory and then executes it. There are a large number of tools available for this purpose, which are commonly known as *executable packers*.

For illustration, consider the example in Listing 1.1. This code fragment from a virus of the Netsky family prepares to install a copy of the virus (under the name services.exe) into the Windows system directory obtained through a call to GetWindowsDirectory. After applying the packing, junk insertion, and reordering obfuscations (in this order), a malware writer could obtain the obfuscated malware in Listing 1.2. Comparing the two listings, we note the effect of obfuscation on the malware instance: the packing transformation has hidden the malicious code inside the data block starting at X and, as a result, only the code of the unpacking routine is visible. Any attempted matches against a signature that refers, for example, to the call to GetWindowsDirectory will fail. Furthermore, the unpacking routine was injected with junk code (instructions on lines 16, 2, 18, and 12) and was subjected to reordering. The result of these obfuscations is that signatures identifying the unpacking loop will no longer match, and thus

```
 lea  eax, [ebp+Data]
 push esi
 push eax
 call ds:GetWinDir
 lea  eax, [ebp+Data]
 push eax
 call _strlen
 cmp  [ebp+eax+v_1], 5Ch
 pop  ecx
 jz   short loc_40
 lea  eax, [ebp+Data]
 push offset asc_408D80
 push eax
 call _strcat
 pop  ecx
 pop  ecx
loc_40:
 lea  eax, [ebp+Data]
 push offset aSvcs_exe
 push eax
 call _strcat
```

**Listing 1.1.** Example of a malware code fragment.

```
1         jmp  lab
2   lan:  add  [esp], 1
3         jmp  lay
4   lop:  cld
5         jmp  lah
6   lac:  scasb
7         jmp  lam
8   laz:  mov  al, 99
9         jmp  lop
10  lav:  loop lop
11        jmp  law
12  las:  dec  edi
13        jmp  lav
14  lab:  mov  edi, offset X
15        jmp  laz
16  lam:  push edi
17        jmp  lan
18  lay:  pop  edi
19        jmp  las
20  lah:  xor  byte ptr [edi],1
21        jmp  lac
22  law:  jmp  short X
23        ...
24  X:  db 8c 84 d9 ...
25      db ...
26      db 01 98
```

**Listing 1.2.** Obfuscation of Listing 1.1.

```
 lea  eax, [ebp+data1]
 push esi
 push eax
 call ds:GetWinDir
 lea  eax, [ebp+data1]
 push eax
 call _strlen
 cmp  [ebp+eax+dat2], 5Ch
 pop  ecx
 jz   short label1
 lea  eax, [ebp+data1]
 push offset data3
 push eax
 call _strcat
 pop  ecx
 pop  ecx
label1:
 lea  eax, [ebp+data1]
 push offset data4
 push eax
 call _strcat
```

**Listing 1.3.** Deobfuscated version of Listing 1.2.

the malware writer will successfully evade detection. Note that in contrast to typical malicious code, the unpacking routines cannot be recognized by characteristic system calls.

Our malware transformation algorithm, when applied to the code in Listing 1.2, identifies the obfuscations that were applied to the malware instance and reverses them. In this case, all three obfuscation types are present. The transformer first undoes the reordering obfuscation by determining that the jump instructions on lines 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, and 21 are unnecessary. Therefore, these instructions are removed from the program and the transformer reorganizes the code into straight-line sequences in order to maintain the original behavior. Next, the junk code on lines 16, 2, 18, and 12 (identified using a semantic nop detector) is removed from the program. Finally, the packed code is extracted with the help of a dynamic analysis engine. The specific algorithms applied during the malware transformation are detailed in Section 3. The resulting code, shown in Listing 1.3, is syntactically equivalent (modulo renaming) to the original malware and can be passed to a malware detector for analysis.

## 3 Malware Transformation Algorithms

We present algorithms to transform malware that is obfuscated with the techniques most common in real-life malware-generation libraries [16]. For each of the three obfuscation techniques (packing, code reordering, and junk insertion), we describe a corresponding transformation algorithm that reverses the effects of that obfuscation.

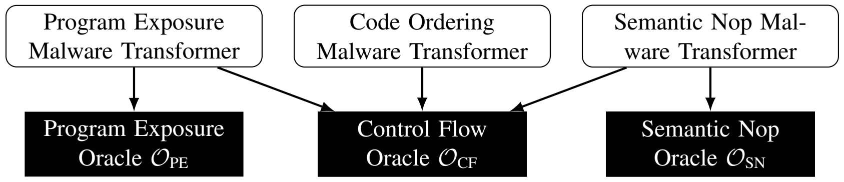
**Table 1.** The program analysis oracles and the corresponding query primitives used as building blocks for the malware transformation algorithms.

| *Oracle* | *Description of Return Value* |
|---|---|
| Program Exposure $\mathcal{O}_{\mathrm{EX}}(P)$ | The binary program $P'$, which is derived from $P$ by execution of a maximum of instructions, such that the relevant behavior of $P'$ and $P$ is equivalent. |
| Control Flow $\mathcal{O}_{\mathrm{CF}}(P, x)$ | The set of successors of the instruction at $x$ during any execution of program $P$. |
| Semantic Nop $\mathcal{O}_{\mathrm{SN}}(P, x, y)$ | True if the program locations $x$ and $y$ delimit a semantic nop for every execution of the program $P$. |

In order to factor out, and thus understand, the program analysis effort required for malware transformation, the algorithms are designed relative to several program analysis primitives. Each primitive operation takes the form of a query to a corresponding program analysis oracle. The program analysis oracles answer queries about various program structures (e.g., control flow graphs, data dependence graphs). Note that the tasks of the oracles are undecidable, as usually in program analysis, and therefore need good approximative implementations, which are presented in Section 4. The oracles and their primitives are listed in Table 1. For the remainder of this section, the oracles are black boxes that provide a query interface.

On top of the oracles, we build three algorithms that transform malware such that three common obfuscations are reversed. The first algorithm describes a malware transformation for code ordering that ensures that instructions appear in the program file in a natural order. The second algorithm implements a malware transformation for program exposure, such that a formerly packed program appears with all encrypted code and data exposed. The third algorithm addresses a malware transformation that identifies and eliminates all instruction sequences that form a semantic nop. As Figure 3 illustrates, each of these algorithms uses one or more of the oracles from Table 1.

We use the following model of binary programs to describe the functionality of the oracles. A binary program is a pair $P = (M, c)$, where $M$ is a binary string representing program memory (code and data) and $c$ is a pointer to a position in $M$ (the entry point). The binary program is executed by reading and interpreting the instruction at the target location of $c$. An instruction can read and modify any value in $M$, including other instructions. Finally, it sets $c$ to point to the next instruction to be executed. After execution of one instruction, the program reaches a new state, which itself represents a new binary program $P' = (M', c')$ on the modified memory $M'$ with the new entry point $c'$. Thus, we view the execution of an instruction as transition from one program to another. Since the interpretation of the instruction at location $c$ can depend on program input (including system state), there may be multiple outgoing transitions from one program state. All possible transitions define a relation $\overset{\delta}{\longmapsto} \subseteq \mathbf{P} \times \mathbf{P}$, where $\mathbf{P}$ is the set of all programs. We will use $\overset{\delta^*}{\longmapsto}$ and $\overset{\delta^+}{\longmapsto}$ to denote 0 or more and 1 or more transitions, respectively.



Program Exposure Malware Transformer
Code Ordering Malware Transformer
Semantic Nop Malware Transformer
Program Exposure Oracle $\mathcal{O}_{\text{PE}}$
Control Flow Oracle $\mathcal{O}_{\text{CF}}$
Semantic Nop Oracle $\mathcal{O}_{\text{SN}}$

**Fig. 3.** Dependencies of malware transformers on program analysis oracles.

Output can be generated during the execution of instructions. By *relevant outputs* we will refer to outputs such as user interaction, operations on files, or network communication, but not to side effects such as different cache contents, page tables, etc. Accordingly, we define the *relevant behavior* of a program as a mapping from inputs (including the empty input) to *relevant outputs*. For two programs $P$ and $P'$ with equivalent relevant behavior, we will write $P \simeq P'$.

## 3.1 Malware Transformation for Code Ordering

A common technique for obfuscation is code reordering, where the basic blocks of a program are split by the insertion of additional jump instructions, and subsequently reordered. Consider the code in Listing 1.2, where the first four non-control flow instructions executed are:

```
mov edi, offset der | mov al, 99 | cld | xor byte ptr [edi], 1
```

Their order in the program file is completely different, as illustrated by their corresponding line numbers 14–8–4–20. Obfuscation by code reordering can be achieved using unconditional control flow instructions (jumps) or conditional control flow instructions (branches) with opaque predicates [12]. Opaque predicates are calculations which have a deterministic result, but are statically hard to analyze. A conditional jump depending on the result of an opaque predicate is therefore in fact unconditional. The algorithm we are about to describe uses the control flow oracle $\mathcal{O}_{\text{CF}}$ that is able to identify all jump targets actually reachable during execution of the program:

**Definition 1.** [CONTROL FLOW ORACLE $\mathcal{O}_{\text{CF}}$]
*The oracle $\mathcal{O}_{CF}(P, x)$ determines the set of program locations that are successors of the instruction at location $x$ in a binary program $P$ during any possible execution of $P$:*

$$\mathcal{O}_{CF}(P, x) = \Big\{ y : \exists M, M'.\, P \xmapsto{\delta^*} (M, x) \wedge (M, x) \xmapsto{\delta} (M', y) \Big\}$$

By querying the oracle, we know for any control flow instruction $x$ where $|\mathcal{O}_{\text{CF}}(P, x)| = 1$, that it has only one possible target, and is thus semantically equivalent to an unconditional jump. In particular, by using the oracle, the algorithm is able to treat conditional jumps whose control condition is an opaque construct as unconditional. For the remainder of this section, we use the term "jump" to refer to unconditional control flow instructions as identified by the oracle $\mathcal{O}_{\text{CF}}$.

```
Input: A CFG $G = (V, E)$ of a program $P$, with $V$ a set of vertices and $E$ a set of edges.
Output: A CFG $G' = (V', E')$, the reordered version of $G$ respecting the CFG invariant.

begin
    $V' \longleftarrow V$ ;   $E' \longleftarrow E$ ;

    repeat
        $N \longleftarrow \emptyset$
        // Collect in $N$ the instructions violating the invariant
        foreach node $v \in V$ do
            if $\exists u \in Predecessors(v) \,.\, |\mathcal{O}_{CF}(P, u)| = 1$ and $v$ has no fall-through
            predecessors then  $N \longleftarrow N \cup \{v\}$

        // Replace unconditional jumps with their targets
        foreach violating node $n \in N$ do
            $j \longleftarrow$ select jump node from $Predecessors(n)$
            $V' \longleftarrow V' \setminus \{j\}$
            $E' \longleftarrow E' \setminus \{e : e \in E' \wedge (e = (j, k) \vee e = (i, j))\}$
            $E' \longleftarrow E' \cup (Predecessors(j) \times \{n\})$
    until $N = \emptyset$

    return $G' = (V', E')$
end
```

**Algorithm 1**: Code ordering transformation of malware.

Any code sequence generated by the code reordering obfuscation necessarily contains jump instructions that are not needed. Intuitively, if all the predecessors of an instruction are jumps, then one of the jump instructions is not needed and can be replaced with the target instruction itself. In the context of a control flow graph (CFG), we can formalize the concept of unneeded unconditional jumps as a CFG invariant: *in an ordered CFG, each CFG node with at least one unconditional-jump immediate predecessor also has exactly one incoming fall-through edge.* A fall-through edge is a CFG edge linking a non-control flow instruction with its unique immediate CFG successor or a CFG edge representing the false-path of a conditional control flow instruction.

We analyze the CFG for each procedure in the program. For each instruction violating the CFG invariant above, we mark its unconditional-jump predecessor as superfluous. Once all the superfluous control flow instructions are identified, the code ordering transformation removes them and reorganizes the program code such that the behavior is preserved. We edit the program code directly by removing each superfluous unconditional jump instruction and replacing it with the target basic block. The code ordering transformation is described in Algorithm 1 and illustrated in Figure 4. As CFG nodes $N_2$, $N_4$, $N_6$, and $N_8$ violate the CFG invariant, we find that nodes $N_1$, $N_3$, $N_5$, and $N_7$ are candidates for removal.

In case a violating instruction has more than one unconditional-jump predecessor, we can freely choose which predecessors to mark as superfluous; this choice is a potential source of ambiguity as described in Section 2. Although we have not encountered it in our evaluation, we handle this case with a simple strategy for consistently selecting instructions to remove. Intuitively, we will order the set of unconditional jumps

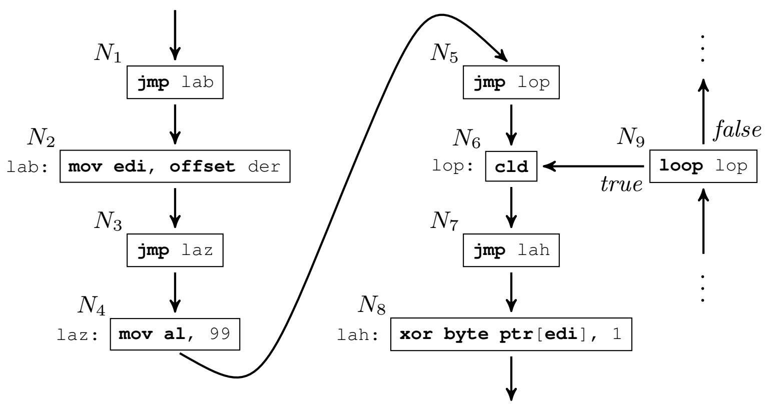


**Fig. 4.** CFG graph fragment containing the first four non-control flow instructions in execution order from Listing 1.2.

by comparing them using their sequences of predecessors. We will then select the first unconditional jump (in the ordered set) and mark it as superfluous. Formally, let $J = \{j_1, \ldots, j_N\}$ be a set of unconditional jumps that are all predecessors of the same instruction. Define $\mathit{Preds}(i)$ to be the set of predecessors of an instruction $i$ and extend it to sets of instructions, $\mathit{Preds}(\{i_1, \ldots, i_K\}) = \bigcup_{1 \leq k \leq K} \mathit{Preds}(i_k)$. Then compute $A = \{\mathit{PredSeq}(j_1), \ldots, \mathit{PredsSeq}(j_N)\}$, where $\mathit{PredSeq}$ is the sequence of predecessor instructions, $\mathit{PredSeq}(i) = \langle \mathit{Preds}(i), \mathit{Preds}^2(i), \ldots, \rangle$. We can order the set $A$ using any complete ordering relation over sequences of sets of instructions (e.g., lexicographic order over the set of opcodes). The ordered set $A$ induces a natural order on the set $J$ of unconditional jumps. The unconditional-jump instruction that is first in the ordered set $J$ is marked as superfluous and removed.

## 3.2 Malware Transformation for Program Exposure

Packing describes the process of encrypting a program and adding a runtime decryption routine to it, such that the behavior of the original program is preserved. By randomly choosing encryption keys, it is possible to create a multitude of instances from one original program. The encryption completely changes the binary footprint of a program, and malware authors commonly use packing to evade scan string-based malware detectors. The malicious code resides in the executable file in an encrypted form, and is not exposed until the moment the executable is run. Thus, a scan string algorithm will fail to detect the malware by reading the file, unless it is updated with a new scan string tailored towards this specific packed instance of the malware.

By analyzing programs obfuscated by packing, we found that they consist of (1) a *decryption routine* (an instruction sequence that generates code and data), (2) a *trigger instruction* that transfers control to the generated code, (3) an *unpacked area* (the memory area where the generated code resides), and (4) a *packed area* (the memory area

```
Input: A program file $P$.
Output: An unpacked program file $P'$.
begin
    $D \longleftarrow \emptyset$
    $P' \longleftarrow \mathcal{O}_{\text{EX}}(P)$
    $W \longleftarrow \{ProgramCounter(P)\}$ ;   $c \longleftarrow ProgramCounter(P')$
    if $W = \{c\}$ then return $P$
    while $W \neq \emptyset$ do
        Let $x \in W$
        $W \longleftarrow W \setminus \{x\}$ ;   $D \longleftarrow D \cup \{x\}$
        foreach $y \in \mathcal{O}_{CF}(P, x)$ do
            if $(y \neq c) \wedge (y\ \textit{not already processed})$ then $W \longleftarrow W \cup \{y\}$
        end
    foreach $d \in D$ do
        Delete location $d$ from $P'$
    return $P'$
end
```

**Algorithm 2**: Malware transformation for program exposure.

from where the packed original binary is read). The program transformation technique we present here works independently of the positioning of these four elements in the program. The only restriction we place is for execution flow to reach the decryption routine before it reaches the trigger instruction. Packing does not change the relevant behavior of a program. Hence, transformation of a packed program (called *unpacking*) consists of recovering the original program that has the same relevant behavior as the packed program.

According to our threat model described above, in a packed program the decryption routine precedes the execution of the original program. To ensure that the original program is correctly restored at runtime, the decryption routine generates the same results every time the program is run, regardless of any input to the program. Furthermore, the operations of the decryption routine affect only program memory. As a consequence, it is possible to create an equivalent program not containing the decryption routine by setting all the values in the unpacked area to the expected results of its computation beforehand.

We now define the Oracle $\mathcal{O}_{\text{EX}}$:

**Definition 2.** [PROGRAM EXPOSURE ORACLE $\mathcal{O}_{\text{EX}}$]
*Given a binary program $P$, the oracle $\mathcal{O}_{EX}$ determines a binary program $P'$, which exhibits the same relevant behavior as $P$, such that $P'$ is reachable from $P$ through transitions on $\delta$, and every binary program $P''$ reachable from $P'$ exhibits different relevant behavior. Formally, $\mathcal{O}_{EX}(P) = P'$ such that:*

$$P \overset{\delta^*}{\longmapsto} P' \wedge P \simeq P' \wedge \left(\forall P'' \,.\, P' \overset{\delta^+}{\longmapsto} P'' \Rightarrow P \not\simeq P''\right)$$

Intuitively, $\mathcal{O}_{\text{EX}}$ returns the binary program $P$ in the latest possible state where the relevant behavior is equivalent to the original program. Given a packed file $P$, $\mathcal{O}_{\text{EX}}(P) =$

$P'$ is the binary program which will be executed after the trigger instruction. For a program $P$ that has not been packed, $P' = P$, assuming that $P$ does not start with instructions not affecting the relevant behavior.

Using the oracle $\mathcal{O}_{\mathrm{EX}}$, we can now transform out the initial decryption in the program and produce a program file that visibly contains the unpacked code, as illustrated in Algorithm 2. To remove the unwanted decryption routine, we traverse the program control flow graph according to the control flow oracle $\mathcal{O}_{\mathrm{CF}}$. Starting with the instruction at the entry point of $P$, we delete all instructions preceding the trigger instruction in the control flow graph, as well as the trigger instruction itself.

## 3.3 Malware Transformation for Semantic Nops

A code sequence in a program is a *semantic nop* if its execution preserves all variable values in the program and it does not generate any relevant output. In the control flow graph of the program, semantic nops created by the junk insertion obfuscation form *hammocks*[3] [17]. Hammocks are subgraphs of the control flow graph with single entry and exit nodes. The transformation algorithm for semantic nop removal uses the oracle $\mathcal{O}_{\mathrm{SN}}$ to determine whether a hammock is a semantic nop:

**Definition 3.** [SEMANTIC NOP ORACLE $\mathcal{O}_{\mathrm{SN}}$]
*The oracle $\mathcal{O}_{SN}$ determines whether a hammock given by its entry and exit nodes $x$ and $y$ is a semantic nop; specifically, it returns true iff (i) in all executions of $P$, memory contents at the time the hammock is entered are identical to memory contents at the time the hammock is left, and (ii) execution of the hammock does not change relevant behavior. Expressed in terms of transitions on binary programs, we have:*

$$\mathcal{O}_{SN}(P,x,y) = \forall M, M'.\ \left(P \overset{\delta^*}{\longmapsto} (M,x) \overset{\delta^*}{\longmapsto} (M',y) \Rightarrow M = M'\right) \wedge$$
$$\forall M.\ \left(P \overset{\delta^*}{\longmapsto} (M,x) \Rightarrow (M,x) \simeq (M,y)\right)$$

The transformation algorithm shown in Algorithm 3 analyzes each function in the program, enumerating all hammocks as candidates for semantic nop checking. The hammocks are derived from control flow dominators and post-dominators. A hammock's entry node dominates a hammock's exit node, and the exit node post-dominates the entry node. To compute dominators in a control flow graph we use standard algorithms from compiler literature [18], which depend on the control flow oracle $\mathcal{O}_{\mathrm{CF}}$.

We resolve any ambiguity arising from overlapping semantic-nop hammocks (such as those in Figure 2 of Section 2) by selecting the largest hammock(s) and then choosing the hammock with a non-overlapping entry node. This does not guarantee the recovery of the code in the original malware instance, but fixes one malware instance as a normal form for all obfuscated variants of the original malware instance.

[3] A hammock is a subgraph of a CFG $G$ induced by a set of nodes $H \subseteq \mathit{Nodes}(G)$ such that there is a unique entry node $e \in H$ where $(m \in \mathit{Nodes}(G) \setminus H) \wedge (n \in H) \wedge ((m,n) \in \mathit{Edges}(G)) \Rightarrow (n = e)$ and such that there is a unique exit node $t \in H$ where $(m \in H) \wedge (n \in \mathit{Nodes}(G) \setminus H) \wedge ((m,n) \in \mathit{Edges}(G)) \Rightarrow (m = t)$. Structured if, while, and repeat statements are examples of hammocks.

```
Input: A program file $P$.
Output: A program file $P'$ with no semantic nops.
begin
    $P' \longleftarrow P$
    foreach $H \in Hammocks(P')$ do
        if $\mathcal{O}_{SN}(P', H) = true$ then
            Remove $H$ from $P'$
            Recompute $Hammocks(P')$
        end
    end
    return $P'$
end
```

**Algorithm 3**: Semantic nop transformation of malware.

## 4 Implementation

Our implementation of the malware transformation algorithms builds on top of publicly available tools for analysis and manipulation of executable code. The program exposure transformation operates on the binary code representation, whereas the semantic nop and code ordering transformations rewrite x86 assembly code. The suspicious program is disassembled using IDAPro [19] and the resulting assembly language program is transformed according to the answers provided by the program analysis oracles. The assembly language program is rewritten and then reassembled into an executable program. Currently, the reassembly step of this process requires manual intervention, as IDAPro sometimes produces imprecise disassembly results. We plan to automate this process as much as possible and we are exploring possible alternatives to IDAPro.

In the remainder of this section we present our implementations approximating the three program analysis oracles. Each oracle implementation can make use of a static analysis engine built on top of IDAPro as well as a dynamic analysis engine based on the open-source processor emulator *qemu* [20] to which we added tracing and dynamic analysis functionality.

### 4.1 An Implementation of the Program Exposure Oracle $\mathcal{O}_{EX}$

For the implementation of the program exposure oracle, we introduce a technique based on dynamic analysis. Dynamic analysis is limited to information gained in the observed executions. However, since a packer has to ensure the integrity of the generated binary in every execution of the packed program, the result of the unpacking routine needs to be invariant and deterministic. Thus, dynamic analysis of the unpacking routine is guaranteed to yield results representative for every execution of the packed program.

Oracle 1 describes, in pseudocode, the process of identifying all the instructions of a self-generating program. We emulate the program in a controlled environment until the program counter reaches the generated-code area. During emulation, we capture all data the program writes to memory. Finally, we modify the original program using the captured data to reflect its last emulated state.

```
Input: A program file $P$ and a program location $x$.
Output: The program $P$ in the state after decryption has finished.
begin
    $T \longleftarrow \varnothing$ ; // history of program writes
    $r \longleftarrow EntryPoint(P)$ ; // current program counter
    /* Termination condition: control flow reaches a previously written location */
    while $\neg(\exists v.\langle r, v\rangle \in T)$ do
        $I_c \longleftarrow P[r]$ ; // Instruction at location $r$ in $P$.
        $Emulate(P, I_c)$
        if $HasTerminated(P)$ then break
        if $IsMemoryWrite(I_c)$ then
            Let $v$ be the value written by $I_c$
            Let $a$ be the target memory location
            if $\exists v'.\langle a, v'\rangle \in T$ then $T \longleftarrow T \setminus \{\langle a, v'\rangle\}$ // Remove earlier writes
            $T \longleftarrow T \cup \langle a, v\rangle$
        $r \longleftarrow ProgramCounter(P)$
    /* Construct the unpacked program */
    $P' \longleftarrow P$
    foreach location $a$ in $P$ do if $\exists v.\langle a, v\rangle \in T$ then $P'[a] \longleftarrow v$
    $ProgramCounter(P') \longleftarrow r$
    return $P'$
end
```

**Oracle 1**: Program exposure oracle $\mathcal{O}_{\text{EX}}$.

In our current prototype we execute the program in a modified version of the *qemu* system emulator [20], collect all the memory writes (retaining for each address only the most recently written value), and monitor execution flow. If the program attempts to execute code from a memory area that was previously written, we capture the target address of the control flow transfer (i.e., the trigger instruction) and terminate execution.

Our current implementation does not have an automated way to check whether a program is self-generating. We are exploring various techniques that can identify the presence of self-generating code. One promising approach is the use of static analysis to locate the control flow instruction that directs execution into dynamically generated code. Another possible approach is based on the byte value entropy of the executable file, as previous work has shown that compressed files have statistically different byte distributions compared to uncompressed files [21,22,23].

## 4.2 An Implementation of the Control Flow Oracle $\mathcal{O}_{\text{CF}}$

The goal of the control flow oracle $\mathcal{O}_{\text{CF}}(P, x)$ is to determine the program locations succeeding the instruction at program location $x$ in actual executions of the program $P$. The set of an instruction's successors is fixed and can be determined statically if the instruction is a non-control flow instruction, a direct jump, or a call. In our current implementation, we use the IDAPro disassembler [19] to identify these control flow edges.

Indirect jumps and indirect calls significantly raise the complexity of the oracle. Both indirect jumps and indirect calls use a computed value (in register or a memory location) to determine the target of the control flow transfer. The computation of the target location can be arbitrarily complex (e.g., using branch functions [24]). The current implementation of the control flow oracle handles a limited class of indirect jumps. Based on heuristics of the IDAPro toolkit, the control flow oracle can determine the target locations for control flow transfers that use a jump table.

Finally, branches (i.e., conditional jumps) pose problems as well when used in conjunction with opaque predicates. Opaque predicates always evaluate to the same truth value, with the additional property that determining this truth value is statically hard to do. Opaque predicates have been discussed in research literature [12,25], but have yet to make their way into real-world obfuscation tools. We plan to explore the handling of branches based on opaque predicates in the future, for example through the use of dynamic analysis to identify opaque predicate candidates.

### 4.3 An Implementation of the Semantic Nop Oracle $\mathcal{O}_{\mathrm{SN}}$

The semantic nop oracle has to determine whether a code hammock preserves the state of the program. This condition can be expressed, equivalently, in terms of program variables before and after the code sequence, i.e., a semantic nop preserves all program variable values (memory, registers). Our implementation is inspired by the use of decision procedures for semantics-aware malware detection [26]. We use a stack of decision procedures with varying degrees of power and cost. First, a simple pattern matcher identifies sequences of instructions known to be semantic nops. Second, a random execution engine can prove that the hammock is *not* a semantic nop. If after one random execution the state is not identical to the initial state, then the hammock is not a semantic nop.

The third decision procedure is a theorem prover, Simplify [27], which determines whether the hammock, expressed as a state transformer using variables in SSA form, implies that the values of all the state variables are preserved. If the negation of the formula is not satisfiable, then the hammock is a semantic nop. The decision procedure using the theorem prover does not handle hammocks that contain loops.

Our fourth decision procedure is based on a bounded model checker, UCLID [28]. The hammock is converted into a sequence of state-transition rules. We use UCLID to unwind the code up to a certain, heuristically determined depth, and then query whether the resulting state is identical to the initial state. If the model checker provides a counterexample, then the hammock is no semantic nop.

### 4.4 Discussion of Limitations

We now consider the limitations of our oracle implementations compared to the ideal oracles outlined in Section 3. The control-flow oracle $\mathcal{O}_{\mathrm{CF}}$ has the largest impact on the accuracy of our malware transformation system since all three component transformers depend on it. If the program is obfuscated such that the code cannot be disassembled or the control flow cannot be recovered accurately, then the transformation process can fail to deobfuscate the malware instance. The second limitation that could be exploited by a malware writer is the incompleteness of the semantic-nop oracle $\mathcal{O}_{\mathrm{SN}}$. Because

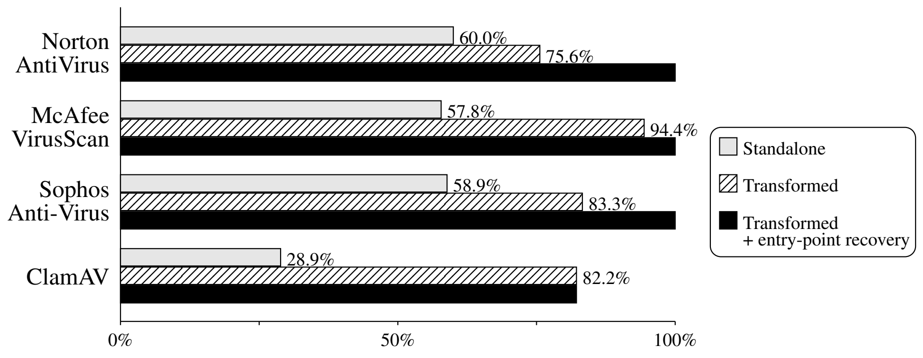


**Fig. 5.** Detection rates for packed malware variants. The *Transformed* bars indicate detection rates after the input programs were preprocessed by the malware transformer. The *Transformed + entry-point recovery* bars indicate the detection rates if, in addition, the entry-point of the unpacked program was manually recovered.

checking whether a code fragment is a semantic nop is undecidable, our implementation errs on the side of soundness. As a result, complex semantic nops that involve loop constructs might not be identified or removed.

## 5 Evaluation

The key metric in our evaluation of malware transformations is the improvement in the detection rate (percentage of variants detected) observed in commercial anti-virus tools. For each obfuscation technique, we obfuscated a known malware instance to create a set of new variants. We then measured the detection rate of four commercial anti-virus tools on this set of variants—this established the baseline for comparison. The four anti-virus tools are Norton AntiVirus version 10.0.2, McAfee VirusScan 4.40, Sophos Anti-Virus 3.96, and ClamAV 0.87, each with up-to-date signatures. Each variant in the obfuscation set passed through the malware transformation process to produce a set of transformed variants. The detection rate of the same commercial anti-virus tools was then measured on this set. By comparing the two detection rates, we determined to what extent our malware transformations improved the detection ability.

We observe that the detection rate significantly improves when malware transformations are applied. This result supports our intuition that malware transformations benefit the detection process.

### 5.1 Evaluation of the Transformation for Program Exposure

To evaluate the efficacy of malware transformations in the context of self-generating programs, we made use of a set of seven existing tools for code compression. These tools, commonly known as *packers*, obfuscate a program such that the program body (the code and/or the data) is compressed, and a new program is created to include the

**Table 2.** Detection rates for malware variants obfuscated by code reordering. The *Std.* column indicates the detection rate of standalone virus scanners. The *Xfrm.* column indicates the detection rate of virus scanners after the input program was first transformed.

| *Obfuscation Ratio* | *McAfee VirusScan* | | *Sophos Anti-Virus* | | *Norton AntiVirus* | | *ClamAV* | |
|---|---|---|---|---|---|---|---|---|
| | *Std.* | *Xfrm.* | *Std.* | *Xfrm.* | *Std.* | *Xfrm.* | *Std.* | *Xfrm.* |
| 10% | 32.8% | 100% | 40% | 100% | 75% | 100% | 80% | 100% |
| 50% | 40.2% | 100% | 33% | 100% | 68.5% | 100% | 82% | 100% |
| 90% | 40.7% | 100% | 10% | 100% | 67% | 100% | 75.5% | 100% |

decompressor as well as the compressed data. At execution time, the decompressor extracts and transfers control to the original program body. For evaluation, we used the packers Petite, UPX, ASPack, Packman, UPack, PE Pack, and FSG. Each packer was run on several versions of the Netsky and Beagle viruses, to obtain a total of 90 different new variants. The self-generating variants were then transformed to obtain the set of unpacked variants. Both sets were scanned using the four anti-virus tools. The results are summarized in Figure 5.

The numbers show that the program exposure transformation improved detection rates significantly, but did not achieve 100% detection. We discovered that some malware detectors use signatures depending on the entry point value, which in some cases differed between the initial and the unpacked malware instance. If we manually set the entry point for the unpacked executable to the correct value in the respective cases, the malware detector was able to identify the viruses reliably. The change in the entry point value is due to the fact that some packers not only dynamically generate the original program but also add additional fixup code. These code portions are executed before the original program, changing the entry point of the reconstructed program.

Furthermore, if a malware detector uses only signatures tailored towards a specific obfuscated malware instance, it will fail to detect the transformed instance. The majority of malware detectors already come with signatures for unpacked malware to support specialized unpacking engines for common executable packers. ClamAV, however, sometimes failed to detect the unpacked instances due to a lack of signatures for the unpacked malware variants.

### 5.2 Evaluation of the Code Ordering Transformation

We generated a large number of variants from 10 viruses (Beagle.Y, Triplix-C, Triplix-D, Firstborn, Integrator, Fiasko, Halen, Terronia, Dammit, Idele). We created approximately 70 variants for each original virus, for a total of 712 variants. Then, we measured the detection rate over the new variants with and without code ordering.

Each variant was generated by applying code reordering to a randomly chosen set of program fragments. In Table 2 we list the detection rate when the obfuscation is applied to 10%, 50%, and 90% of the program. A ratio of 10% means that a tenth of the original virus code (measured in number of instructions) was randomly selected and obfuscated.

**Table 3.** Detection rates for malware variants obfuscated by junk insertion. The *Std.* column indicates the detection rate of standalone virus scanners. The *Xfrm.* column indicates the detection rate after the input program was first transformed.

| *Obfuscation Ratio* | *McAfee VirusScan* | | *Sophos Anti-Virus* | | *Norton AntiVirus* | | *ClamAV* | |
|---|---|---|---|---|---|---|---|---|
| | *Std.* | *Xfrm.* | *Std.* | *Xfrm.* | *Std.* | *Xfrm.* | *Std.* | *Xfrm.* |
| 10% | 62.5% | 100% | 35% | 100% | 80% | 100% | 90% | 100% |
| 50% | 45% | 100% | 22% | 100% | 78.5% | 100% | 78% | 100% |
| 90% | 25% | 100% | 7.5% | 100% | 69% | 100% | 76.5% | 100% |

While the detection rate for these obfuscated variants dropped significantly in many cases, it went back up to 100% after we applied the code ordering transformation.

Note that, in Table 2, Norton AntiVirus and ClamAV are outliers. For some of the viruses, both succeed in detecting all variants generated using the reordering obfuscation for a simple reason: their signatures match data areas, which are unchanged by obfuscations that work on code. Because of this, polymorphic viruses commonly employ code obfuscation in combination with packing.

## 5.3 Evaluation of the Semantic Nop Transformation

Similar to the evaluation of the code ordering malware transformation, we generated a large number of variants of the 10 viruses and measured the detection rate before and after applying the semantic nop transformation. Each variant was generated by inserting junk code into a randomly chosen set of program locations, for a total of 570 variants. We list in Table 3 the detection rate when the obfuscation is applied to 10%, 50%, and 90% of the program. In addition to varying the location of the junk code insertion, we also varied the types of junk code generated, from simple `nop` instructions to semantic nops that use stack and arithmetic operations. The detection rate for these obfuscated variants again dropped significantly in many cases; using the semantic nop transformation the detection rate went back up to 100%. Since junk-insertion is a code-only obfuscation, the detection rates of Norton AntiVirus and ClamAV stand out again for the same reason as before.

## 5.4 Malware Transformation Times

Figure 6 shows the average execution time for the various transformation steps. Since this is an unoptimized research prototype, we expect that significant speed gains can be achieved through an optimized implementation. Both the code ordering transformer and the program exposure transformer finish in about 10-20 seconds. The semantic nop transformer is significantly slower as it must query a decision procedure for all possible hammocks in the program. We believe that better strategies for semantic nop detection are possible; one of our goals for future work is to improve performance.

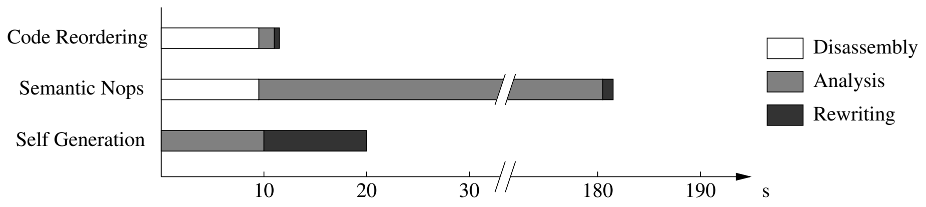


**Fig. 6.** Average running times of the malware transformers for different obfuscations. Times are broken down into phases for disassembly, analysis (static analysis resp. emulation), and rewriting of the executable.

## 6 Related Work

Obfuscations have been used for a long time to evade detection of malware. Polymorphic malware, which encrypts the code under a different key at each replication, and metamorphic malware, which morphs itself at each replication, are increasing threats to malware detectors [16]. Numerous obfuscation toolkits are freely available, some with advanced features. For example, *Mistfall* is a library for binary obfuscation specifically designed to blend malicious code into a host program [29]. Other tools such as the executable packer UPX [30] are available as open-source packages, making it easy for malware writers to custom-develop their own versions. Free availability of obfuscation toolkits is spawning new malware variants rapidly, whereas commercial malware detectors are commonly incapable of handling the plethora of obfuscations found in the wild [4]. While obfuscation has found commercial application in protecting intellectual property in software [11,12,31], the goal of our work is to address the particular obfuscations used by malware writers. Recent work presented obfuscations that aim to thwart both static and dynamic analysis by combining code encryption with expensive decryption strategies [32]. We note a detector need not identify malicious code before the program runs, but only before the program harms the system. A possible transformation strategy would monitor the program until the payload is decrypted and then make use of the techniques in our present work to transform the code before detection. Achieving the low overhead required to make such a strategy practical for broad deployment is the goal of future research.

Deobfuscation tools appeared in response to particular attacks. Detection of polymorphic malware requires the use of emulation and sandboxing technologies [33], which integrated emulation into the malware detector. This approach is open to resource–consumption attacks and is prone to false negatives, since the execution time in the sandbox has to be restricted for performance reasons [34,35]. In contrast, our code-identification oracle terminates the program as soon as it attempts to execute dynamically generated code. Another approach to unpacking is PolyUnpack [36], which executes the program inside a debugger until it reaches an instruction sequence that does not appear in the static disassembly of the program. PolyUnpack suffers from limitations inherent in static disassembly, limitations that we avoid in our program exposure transformer by using a purely dynamic technique. Other work on deobfuscation has proposed heuristics to recover the import address tables from a packed binary [37]. We will investigate the addition of such techniques to our implementation of the program exposure oracle.

Static analysis approaches for detecting and undoing obfuscations have been proposed for particular obfuscations, such as techniques for defeating the effect of control flow flattening [38] or for handling a given metamorphic engine through the use of term rewriting [39]. More general approaches, similar in intent to ours, attempt to normalize C programs by ordering program statements and expressions [40] and to apply compiler-optimization techniques to eliminate obfuscated code [41,42]. These algorithms complement ours as they perform local deobfuscation.

When we apply our malware transformations, we assume that the code in the program file has been successfully disassembled. While recent work showed that an attacker can make disassembly hard [24], we note that other researchers have already proposed solutions to counter such techniques [43,44].

Several new research approaches propose semantics-based malware detectors that can reliably handle malware variants derived through obfuscation or program evolution [45,26]. These methods are promising, but they face an uphill battle in terms of deployment opportunities due to the fact that the current commercial malware detectors have a large install base. We believe that our malware transformer provides an easier upgrade path to more advanced detection techniques, as it works in conjunction with existing detectors.

## 7 Conclusion

This paper presented a set of malware transformations that undoes the effects of three common obfuscations used by malware writers. We demonstrated that the detection rates of four commercial malware detectors can be improved by first processing an executable by a malware transformer. An additional benefit of malware transformations is the separation of concerns between the malware transformation stage and the malware detection stage. This leads to more maintainable software and allows for independent improvements in malware transformation techniques and malware detection algorithms. In the future, we will investigate improvements to our oracle implementations. In particular, we will address the handling of opaque predicates and the reconstruction of import tables. We also plan to expand the set of obfuscations handled by our malware transformer and to improve overall performance.

## Acknowledgments

We thank Saumya Debray for his helpful comments and suggestions.

## References

1. McGraw, G., Morrisett, G.: Attacking malicious code: Report to the Infosec research council. IEEE Software **17**(5) (Sept./Oct. 2000) 33 – 41
2. Szor, P.: 11. In: The Art of Computer Virus Research and Defense. Addison-Wesley (2005) 425–494
3. Detristan, T., Ulenspiegel, T., Malcom, Y., von Underduk, M.S.: Polymorphic shellcode engine using spectrum analysis. Phrack **11**(61) (August 2003) published online at http://www.phrack.org. Last accessed on 14 Apr. 2006

4. Christodorescu, M., Jha, S.: Testing malware detectors. In: Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2004 (ISSTA'04). (July 2004) 34–44
5. Mohanty, D.: Anti-virus evasion techniques and countermeasures. Published online at http://www.hackingspirits.com/eth-hac/papers/whitepapers.asp. Last accessed on 18 Aug. 2005.
6. AVV: Antiheuristics. 29A Magazine **1**(1) (1999)
7. Rajaat: Polymorphism. 29A Magazine **1**(3) (1999)
8. Julus, L.: Metamorphism. 29A Magazine **1**(5) (2000)
9. Mental Driller: Metamorphism in practice. 29A Magazine **1**(6) (2002)
10. Ször, P.: Advanced Code Evolution Techniques and Computer Virus Generator Kits. Symantec Press. In: The Art of Computer Virus Research and Defense. 1st edn. Addison Wesley Professional (February 2005)
11. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, Univ. of Auckland, New Zealand (July 1997)
12. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proc. of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98). (January 1998)
13. Henry, S.M., Kafura, D.G.: Software structure metrics based on information flow. IEEE Transactions on Software Engineering **7**(5) (1981)
14. McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering **2**(4) (1976)
15. Munson, J.C., Khoshgoftaar, T.M.: Measurement of data structure complexity. Journal of Systems and Software **20**(3) (1993)
16. Jordan, M.: Dealing with metamorphism. Virus Bulletin (October 2002) 4–6
17. Komondoor, R., Horwitz, S.: Semantics-preserving procedure extraction. In: Proc. of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00). (2000) 155–169
18. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
19. DataRescue sa/nv: IDA Pro – interactive disassembler Published online at http://www.datarescue.com/idabase/. Last accessed on 14 Apr. 2006.
20. Bellard, F.: Qemu. Published online at http://fabrice.bellard.free.fr/qemu/. Last accessed on 14 Apr. 2006.
21. Wehner, S.: Analyzing worms and network traffic using compression. Published online at http://arxiv.org/abs/cs.CR/0504045. Last accessed on 14 Apr. 2006.
22. McDaniel, M., Heydari, M.H.: Content based file type detection algorithms. In: Proc. of the 36th Annual Hawaii International Conference on System Sciences (HICCSS'03). (January 2003)
23. Li, W.J., Wang, K., Stolfo, S.J.: Fileprints: Identifying file types by n-gram analysis. In: Proc. of the 6th Annual IEEE Information Assurance Workshop, United States Military Academy, West Point, NY, USA (June 2005) 64–71
24. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proc. of the 10th ACM Conference on Computer and Communications Security (CCS'03). (October 2003)
25. Collberg, C., Thomborson, C., Low, D.: Breaking abstractions and unstructuring data structures. In: Proc. of the International Conference on Computer Languages 1998 (ICCL'98). (May 1998) 28–39
26. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proc. of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005). (May 2005) 32–46
27. Detlefs, D., Nelson, G., Saxe, J.: The Simplify theorem prover Published online at http://www.hpl.hp.com/downloads/crl/jtk/download-simplify.html. Last accessed on 14 Apr. 2006.
28. Lahiri, S.K., Seshia, S.A.: The UCLID decision procedure. In Alur, R., Peled, D.A., eds.: Proc. of the 16th International Conference on Computer Aided Verification (CAV'04). Volume 3114 of Lecture Notes in Computer Science. (July 2004) 475–478
29. z0mbie: Automated reverse engineering: Mistfall engine. Published online at http://z0mbie.host.sk /autorev.txt. Last accessed: 16 Jan. 2004
30. Oberhumer, M.F., Molnár, L.: The Ultimate Packer for eXecutables (UPX). Published online at http://upx.sourceforge.net/. Last accessed on 14 Apr. 2006.
31. Chow, S., Gu, Y., Johnson, H., Zakharov, V.: An approach to the obfuscation of control-flow of sequential computer programs. In Davida, G., Frankel, Y., eds.: Proc. of the 4th International Information Security Conference (ISC'01). Volume 2200 of Lecture Notes in Computer Science. (October 2001) 144–155
32. Beaucamps, P., Filiol, E.: On the possibility of practically obfuscating programs towards a unified perspective of code protection. Journal in Computer Virology **3**(1) (April 2007) 3–21
33. Nachenberg, C.: Polymorphic virus detection module. United States Patent # 5,826,013 (October 1998)
34. Natvig, K.: Sandbox technology inside AV scanners. In: Proc. of the 2001 Virus Bulletin Conference. (September 2001) 475–487
35. Natvig, K.: Sandbox II: Internet. In: Proc. of the 2002 Virus Bulletin Conference. (2002) 1–18
36. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In: the 22nd Annual Computer Security Applications Conference (ACSAC'06). (December 2006) 289–300
37. Josse, S.: Secure and advanced unpacking using computer emulation. Journal in Computer Virology (2007)
38. Udupa, S.K., Debray, S.K., Madou, M.: Deobfuscation: Reverse engineering obfuscated code. In: Proc. of the 12th IEEE Working Conference on Reverse Engineering (WCRE'05). (November 2005)
39. Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Normalizing metamorphic malware using term rewriting. In: Proc. of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '06). (September 2006) 75–84
40. Lakhotia, A., Mohammed, M.: Imposing order on program statements and its implication to AV scanners. In: Proc. of the 11th IEEE Working Conference on Reverser Engineering (WCRE'04). (November 2004) 161–171

41. Perriot, F.: Defeating polymorphism through code optimization. In: Proc. of the 2003 Virus Bulletin Conference (VB2003). (September 2003) 1–18
42. Bruschi, D., Martignoni, L., Monga, M.: Using code normalization for fighting self-mutating malware. In: Proc. of the International Symposium of Secure Software Engineering (ISSSE'06). (March 2006)
43. Kapoor, A.: An approach towards disassembly of malicious binary executables. Master's thesis, The Center for Advanced Computer Studies, University of Louisiana at Lafayette (November 2004)
44. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proc. of the 13th Usenix Security Symposium (USENIX'04), San Diego, CA, USA (August 2004)
45. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In Julisch, K., Krügel, C., eds.: Proc. of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05). Volume 3548 of Lecture Notes in Computer Science. (July 2005) 174–187