Malware Normalization

Mihai Christodorescu* Johannes Kinder† Somesh Jha* Stefan Katzenbeisser† Helmut Veith†

*University of Wisconsin, Madison
{mihai,jha}@cs.wisc.edu

†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 by malware detectors, 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 normalizer* that undoes the obfuscations performed by a malware writer. Our experimental evaluation demonstrates that a malware normalizer can drastically improve detection rates of commercial malware detectors. Moreover, a malware normalizer can also ease the task of forensic analysis of malware.

1 Introduction

Malware is code that has malicious intent. Examples for 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 its propagation methods and goals is given in [25]. A *malware detector* identifies and contains malware before it can reach a system or network.

As malware detectors use advanced detection techniques, malware writers use better evasion techniques to transform their malware to avoid detection. For example, polymorphic and metamorphic viruses and, more recently, polymorphic shellcodes [12] are specifically designed to bypass detection tools. There is strong evidence that commercial malware detectors are susceptible to common evasion techniques used by malware writers: previous testing work 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 [1, 15, 26, 27, 33, 34].

A *malware normalizer* is a system that takes an obfuscated executable, undoes the obfuscations, and outputs a normalized executable. Therefore, a malware normalizer can be used to improve the detection rate of an existing malware detector. At an abstract level, a malware normalizer thus performs a functionality at the host that is analogous to the functionality provided by a traffic normalizer [13] at the network level. In this paper, we describe the design and implementation of a malware normalizer that handles three common obfuscations (code reordering, packing, and junk insertion) performed by malware writers. An executable that has been processed by a malware normalizer does not have these obfuscations. Since our malware normalizer produces another (normalized) executable, it can be used by any commercial malware detector. In fact, our experimental results demonstrate that the detection rates of four commercial malware detectors are dramatically improved by first processing an executable with our malware normalizer. Our malware

normalizer is also useful for forensics, as it can, for example, undo the packing to expose to a security analyst any additional code embedded in the malware, and can improve the human readability of the code.

Summarizing, this paper makes the following contributions:

- We present the design and implementation of a malware normalizer that handles three common obfuscations used by malware writers. The overview of our malware normalizer is given in Section 2. Details of our normalizing algorithms appear in Section 3. Existing related research work is discussed in Section 5.
- We evaluate the detection rates of four commercial virus scanners: Norton AntiVirus, McAfee VirusScan, Sophos Anti-Virus, and ClamAV. Our malware normalizer improves the detection rate of all four scanners. For example, the detection rate of Sophos Anti-Virus jumps from 5% to 100% when normalizing the code-reordering transformation. Details of our experimental evaluation appear in Section 4.

2 Overview of Malware Normalization

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 the malware writer, or automatic, executed during the replication phase of a virus. For a given set of obfuscation transformations, each possible composition of obfuscation transformations can possibly generate a distinct malware variant. We apply the malware normalization procedure to an obfuscated malware instance with the goal of obtaining the original (publicly known) malware instance. Figure 1 illustrates the threat model and the application of malware normalization. A malware instance M (i.e., a malicious executable binary program) is obfuscated to obtain a new malware instance M'; the malware normalizer processes the obfuscated malware M' to produce a normalized executable program M_N that is then checked by the malware detector.

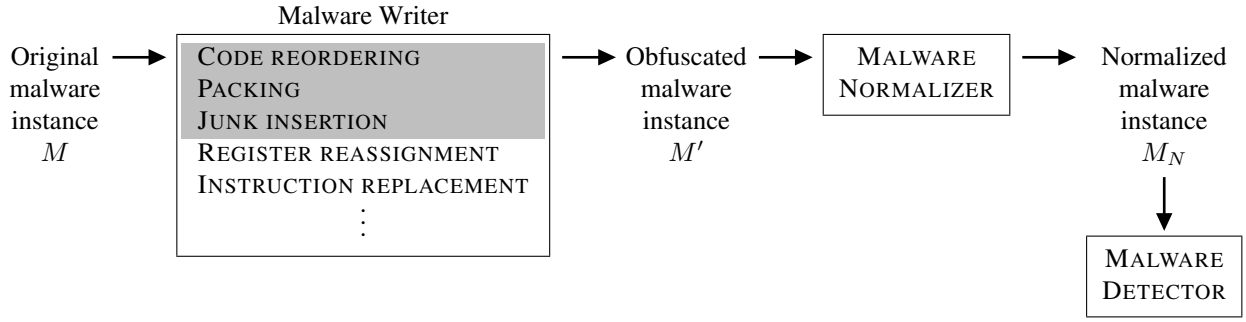

Figure 1: Malware normalization reverses the effects of obfuscations.

The normalization algorithms we describe in this paper target the three obfuscations highlighted in Figure 1. The *code reordering obfuscation* changes the syntactic order of instructions in a program while maintaining the execution order through the insertion of jump instructions. The *junk-insertion obfuscation* randomly adds code sequences that do not change the program behavior. We call such code sequences *semantic nops*. The *packing obfuscation* replaces a code sequence with a data block containing the code sequence in packed form (encrypted or compressed) and an unpacking routine that, at runtime, recovers the original code sequence from the data block. The result of the packing obfuscation is a *self-generating program*: a program that dynamically generates code in memory and then executes it.

Malware writers use obfuscations to their advantage, as any of these transformations effectively evades signature-based matching algorithms commonly used in malware detectors, and some can even evade heuristic detectors. Consider the example in Listing 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 GetWindowsDirectoryA. After applying the packing, junk insertion, and reordering obfuscations (in this order), a malware writer could obtain the obfuscated malware in Listing 2. Comparing the two listings, we note the effect of obfuscation on the malware instance, i.e., the packing transformation has hidden the malicious code inside the data block starting at der 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 GetWindowsDirectoryA 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 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.

```
   lea  eax, [ebp+Data]
2  push esi
   push eax
4  call ds:GetWindowsDirectoryA
   lea  eax, [ebp+Data]
6  push eax
   call _strlen
8  cmp  [ebp+eax+var_129], 5Ch
   pop  ecx
10 jz   short loc_40
   lea  eax, [ebp+Data]
12 push offset asc_408D80
   push eax
14 call _strcat
   pop  ecx
16 pop  ecx
loc_40:
18 lea  eax, [ebp+Data]
   push offset aServices_exe
20 push eax
   call _strcat
26
```

Listing 1: Example of a malware code fragment.

```
       jmp  lab
2  lan: add  [esp], 1
       jmp  lay
4  lop: cld
       jmp  lah
6  lac: scasb
       jmp  lam
8  laz: mov  al, 99
       jmp  lop
10 lav: loop lop
       jmp  law
12 las: dec  edi
       jmp  lav
14 lab: mov  edi, offset der
       jmp  laz
16 lam: push edi
       jmp  lan
18 lay: pop  edi
       jmp  las
20 lah: xor  byte ptr [edi],1
       jmp  lac
22 law: jmp  short der
       ...
24 der: db 8c 84 d9 ff fe fe
       db ...
26     db 01 98
```

Listing 2: Malware instance obfuscated from Listing 1.

```
   lea  eax, [ebp+data1]
2  push esi
   push eax
4  call ds:GetWindowsDirectoryA
   lea  eax, [ebp+data1]
6  push eax
   call _strlen
8  cmp  [ebp+eax+data2], 5Ch
   pop  ecx
10 jz   short label1
   lea  eax, [ebp+data1]
12 push offset data3
   push eax
14 call _strcat
   pop  ecx
16 pop  ecx
label1:
18 lea  eax, [ebp+data1]
   push offset data4
20 push eax
   call _strcat
26
```

Listing 3: Normalized malware instance derived from Listing 2.

Our malware normalization algorithm, when applied to the code in Listing 2, identifies the obfuscations that were applied to the malware instance and reverses them. In this case, all three obfuscation types are present. The normalizer 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 instruction are removed from the program and the normalizer 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. We give a step-by-step description of this process, along with the results of each intermediate stage, in Appendix A. The specific algorithms applied during the malware normalization are detailed in Section 3. The resulting code, shown in Listing 3, is syntactically equivalent (modulo renaming) to the original malware and can be passed to a malware detector for analysis.

3 Malware Normalization Algorithms

We present algorithms to normalize programs obfuscated with three of the techniques most common in malware-generation libraries found in the wild [14]. For each of the three obfuscation techniques (code reordering, junk insertion, and packing) we describe a corresponding normalization algorithm that transforms the program and eliminates the spurious code introduced through obfuscation. Each normalization algorithm detects the presence of a specific obfuscation and then undoes the obfuscation.

3.1 Normalization for Code Reordering

A program contains a *reordered code sequence* when the execution ordering of any two instructions in the code sequence differs from the syntactic ordering of the same instructions in the executable file. Consider the code in Listing 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 or conditional control-flow instructions with opaque predicates [8]. In this paper we focus on code reordering that uses unconditional jumps.

Any code sequence generated by the code reordering obfuscation necessarily contains unconditional jump instructions that are not needed. Intuitively, if all the predecessors of an instruction are unconditional jumps, then one of the unconditional 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 a normalized 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.

The normalization algorithm identifies the unconditional jump instructions that cause the program and execution orders to differ, removes them, and reorders the remaining instructions such that the program behavior is preserved. We analyze the CFG for each procedure in the program and, for each instruction violating the invariant above, we mark its unconditional-jump predecessor as superfluous. In the case of a violating instruction with $N > 1$ unconditional-jump predecessors, we have a choice of instructions to mark as superfluous. In such cases, we order the set of unconditional jump predecessors by their position in the executable file and we choose the last one in the ordered set as candidate for removal.

For example, the CFG for the first four non-control-flow instructions in execution order in Listing 2 is displayed in Figure 2. CFG nodes N_2, N_4, N_6, and N_8 violate the CFG invariant, i.e., each such CFG node has no fall-through immediate predecessor. In contrast, CFG node N_3 respects the CFG invariant. Analyzing the CFG nodes that violate the invariant, we find that nodes N_1, N_3, N_5, and N_7 are candidates for removal.

Once the superfluous control-flow instructions are identified, the reordering normalization algorithm 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. For example, to remove the instruction at program line 15 in Listing 2 (corresponding to CFG node N_3), we replace it with instructions at lines 8 and 9 (corresponding to CFG nodes N_4 and N_5).

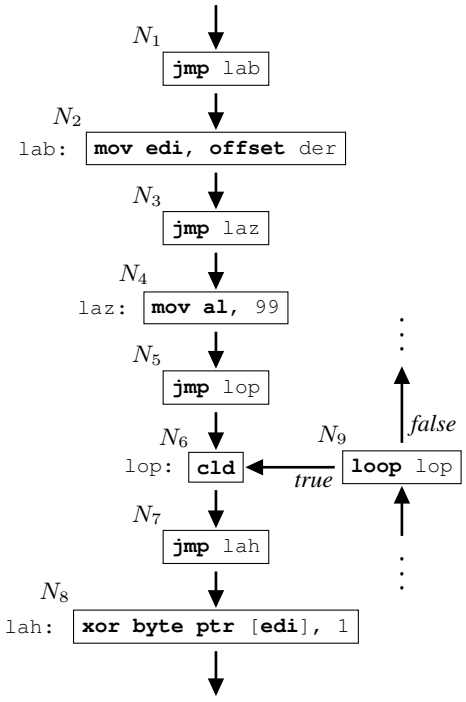

Figure 2: Control flow graph fragment containing the first four non-control-flow instructions in execution order from Listing 2.

Input: A control-flow graph $G = (V, E)$, where V is the set of vertices and E is the set of edges.

Output: The control-flow graph $G' = (V', E')$ with the reordered code of G normalized.

begin

$V' \longleftarrow V$; $E' \longleftarrow E$; $N \longleftarrow \emptyset$

// Collect in N the instructions violating the invariant.

foreach *node* $v \in V$ **do**

  if v *has one unconditional jump predecessor* **then**

    if v *has no fall-through predecessors* **then**

      $N \longleftarrow N \cup \{v\}$

    end

  end

// Replace unconditional jumps with their targets.

foreach *violating node* $n \in N$ **do**

  $j \longleftarrow$ *last jump node, in file order, from* $Predecessors(n)$

  $V' \longleftarrow V' \setminus \{j\}$

  $E' \longleftarrow E' \setminus \{e : e \in E' \wedge (e = (j, k) \vee e = (i, j))\}$

  $E' \longleftarrow E' \cup (Predecessors(j) \times \{n\})$

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

end

Algorithm 1**:** Reordering normalization of malware.

The normalization algorithm is described (in pseudocode format) in Algorithm 1. To produce a normalized CFG (where the invariant holds), the normalization algorithm is iteratively applied until no more violating nodes are identified. When applied to the program fragment in Listing 2, this algorithm identifies instructions on lines 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, and 21 as candidates for removal. The resulting program fragment, shown in Listing 4 of Appendix A, is easier for a human analyst to understand and allows the malware detector to employ simple instruction-sequence matching algorithms without the need to account for arbitrary control flow.

3.2 Normalization for Programs with Semantic Nops

A code sequence in a program is a *semantic nop* if its removal from the program does not change the behavior 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[1]. In an executable binary program, registers (including the status and control registers) and memory locations represent variables. Thus, identifying semantic nops requires deciding whether a code sequence preserves variable values. We use decision procedures to determine whether a code sequence is a semantic nop. This is similar to the use of decision procedures in semantics-aware malware detection [5].

The normalization algorithm analyzes each function in the program, enumerating standalone code se-

[1]We note that this is only one of the possible definitions for semantic nops. One could take into account the context of the code sequence, requiring, for example, that only the live variables are preserved.

quences that are candidates for semantic-nop checking. We use *hammocks*[2] [18] for standalone code sequences. A hammock is a CFG fragment with a single entry node and a single exit node: structured if, while, and repeat statements are examples of hammocks. Hammocks lend themselves to semantic-nop checks and are easy to remove from the program if necessary. The normalization algorithm proceeds as follows:

1. Query the decision procedures for every hammock in a program function.
2. If a hammock is found to be a semantic nop, it is removed from the program.

For the code fragment in Listing 4 of Appendix A, there are multiple hammocks that are checked using the decision procedures. The normalization algorithm identifies the hammock in lines 6–9 as a semantic nop and removes it to obtain the normalized result shown in Listing 5 of Appendix A.

3.3 Normalization for Self-Generating Programs

A *self-generating program* is a program whose execution can reach instructions generated in memory during that same execution. Self-generating programs are part of the larger class of self-modifying programs and include programs that decompress, decrypt, or optimize themselves at runtime. In this section, we focus on programs that generate new code at runtime. In general, such programs contain:

- an instruction sequence (the *code generator*) that generates code at runtime,
- a control-flow instruction (the *trigger instruction*) that transfers control to the dynamically generated code,
- a data area (the *unpacked area*) where the new code resides after being generated, and
- a data area (the *packed code*) from where the packed code is read.

The normalization 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 code generator before it reaches the trigger instruction. Formally, we make the following assumptions about self-generating programs.

Assumption 1 [NO SELF-MODIFICATION]: Self-generating programs do not perform self-modification, i.e., no dynamically generated code overwrites existing code.

Assumption 1 requires the code-generator sequence to execute before control flow reaches the data area for unpacked code. In any execution trace, instructions from the code generator must appear strictly before all instructions that transfer the control flow to the generated code. This assumption eliminates the case of the data area containing code that is initially executed, then overwritten with new code. We note that this corner case is quite rare in malware-obfuscation libraries, and thus is outside the normalization goals of this paper.

Assumption 2 [INPUT INDEPENDENCE]: The code generation does not depend on the program inputs or the runtime environment of the program.

Assumption 2 mandates that the code generator produces the same code sequence regardless of the program inputs. This assumption is consistent with the usage scenarios for viruses and similar malware, where the malware writer (the attacker) is not the same person as the one executing the malware (the victim) and cannot provide interactive inputs to the running malware. This requires the malware to be self-contained, and all the information necessary for running the malware has to be in the possession of the victim.

[2] A hammock is a subgraph of a CFG G induced by a set of nodes $H \subseteq Nodes(G)$ such that there is a unique entry node $e \in H$ where $(m \in Nodes(G) \setminus H) \wedge (n \in H) \wedge ((m,n) \in Edges(G)) \Rightarrow (n = e)$ and such that there is a unique exit node $t \in H$ where $(m \in H) \wedge (n \in Nodes(G) \setminus H) \wedge ((m,n) \in Edges(G)) \Rightarrow (m = t)$.

Input: An executable program P.

Output: An unpacked executable program P'.

```
begin
    while P is packed do
        T ⟵ ∅ ;   r ⟵ EntryPoint(P)
        while ¬(∃v.⟨r, v⟩ ∈ T) do
            I_c ⟵ P[r]    // Instruction at address r in P.
            Emulate(P, I_c)
            // Return unmodified program if no self-generating behavior found.
            if HasTerminated(P) then return P
            if IsMemoryWrite(I_c) then
                Let v be the value written by I_c and let a be the target memory address
                // Remove earlier writes.
                if ∃v'.⟨a, v'⟩ ∈ T then T ⟵ T − ⟨a, v'⟩
                T ⟵ T ∪ ⟨a, v⟩
            r ⟵ CurrentProgramCounter(P)
        // Identify all the writes that affect the static program space.
        m_1 ⟵ lowest address in P ;   m_2 ⟵ highest address in P
        // Construct the unpacked program.
        P' ⟵ P
        foreach a, m_1 < a < m_2 do  if ∃v.⟨a, v⟩ ∈ T then P'[a] ⟵ v
        EntryPoint(P') ⟵ r
        P ⟵ P'
    return P'
end
```

Algorithm 2**:** Unpacking normalization of malware.

Normalization of a self-generating program (called *unpacking*) consists of constructing a program instance which contains the embedded program, contains no code-generating routine, and behaves equivalently to the self-generating program. We capture parts of the program state that are constant across executions and hence independent of any particular input value. *The two program instances are equivalent because the code of the original program is preserved, with the exception of the code generator that has no effect on program output.* The new program instance exposes additional code to static analysis, thus reducing the chance of an attacker evading the malicious code detector.

Algorithm 2 describes, in pseudocode, the process of normalization (or unpacking) of a self-generating program. It consists of two basic steps. First, we execute the program in a controlled environment to identify the control-flow instruction that transfers control into the generated-code area. In this step we also capture all writes to the code area. Second, with the information captured in the previous step, we construct a normalized program instance that contains the generated code.

1. **Identify the first control transfer into generated code.** We execute the program in a modified version of the *qemu* system emulator [2], 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. Based on assumption 2, we know that the code generator and the instruction causing the control-flow transfer are reached in all program executions.

By emulating the program and monitoring each instruction executed, we can precisely identify the moment when execution reaches a previously written memory location. Since we do not employ any heuristics for determining this location, our algorithm has no false negatives.

2. **Construct a non-self-generating program.** Using the data captured in step 1, we can construct an equivalent program that does not contain the code generator. The data area targeted by the trigger instruction is replaced with the captured data.

   The memory writes captured in step 1 contain both the dynamically generated code and the execution-specific data, e.g. the state of the program stack and heap. To differentiate between the two types of writes, we consider all memory writes that affect the load-time image of the program as being part of the dynamically generated code. In Algorithm 2, this memory area is delimited by variables m_1 and m_2.

   The executable file of the new program P' is a copy of the executable file of the old program P, with the byte values in the virtual memory range $[m_1, m_2]$ set from the captured data. The program location where execution was terminated in step 1 is used as entry point for the new program P'.

This algorithm is iteratively applied to a self-generating program until all nested code-generating sequences have been removed. 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 [22, 24, 36].

We illustrate the normalization algorithm applied to the code fragment in Listing 5 of Appendix A. In the first step, the instruction on line 11 is identified as the control-flow transfer into the generated-code area, i.e., the first control-flow instruction whose target is written to. The program state right before we terminate execution is similar to Listing 6 of Appendix A. The final result is a program with no self-generating features, as shown in Listing 7 of Appendix A.

4 Implementation and Evaluation

The implementation of our malware normalization algorithms builds on top of publicly available tools for analysis and manipulation of executable code. The normalizations for code reordering and for semantic nops combine static analysis and rewriting of x86 assembly code. The malware instance is disassembled using IDAPro [9] and the resulting assembly language program is analyzed for code reordering and semantic nops. To identify semantic nops, a collection of decision procedures such as Simplify [11] and UCLID [20] is used. The assembly language program is rewritten to eliminate code reordering obfuscations and semantic nops and is then reassembled into an executable program using MASM. 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.

The normalization for self-generating programs uses the open-source processor emulator *qemu* [2] hosting a full installation of the Microsoft Windows 2000 operating system. We added tracing and dynamic analysis functionality to *qemu*. After emulation, the resulting trace is processed to construct a new executable.

Evaluation Methodology. The key metric in our evaluation of malware normalization is the change in the detection rate observed in commercial anti-virus tools. Malware normalization acts as a preprocessing step in the malware detection process. It transforms the program so that the malware detector has a higher detection rate than the detection rate on the original program. The evaluation proceeded as follows for each obfuscation technique: we applied the obfuscation technique to a known malware instance to create a set of new variants. The obfuscation transformations were modeled after transformation libraries in the wild. The code reordering obfuscator randomly reorders instructions in the program (considering up to 2×10^{39} variants) and then inserts the necessary jumps to maintain control flow. The semantic nop obfuscator randomly selects program locations and injects junk code. For the packing obfuscation we applied packers frequently used by malware writers. 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 8.1, McAfee VirusScan 4.0, Sophos Anti-Virus 3.96, and ClamAV 0.86, each with up-to-date signatures. Each variant in the obfuscation set passed through the malware normalization process to produce a set of normalized variants. The detection rate of the same commercial anti-virus tools was then measured on the set of normalized variants. By comparing the two detection rates, we determined to what extent malware normalization improved the detection ability.

We observe that the detection rate significantly improves when program normalization is applied. This result supports our intuition that normalization benefits the detection process regardless of the detection technique used. In the remainder of this section we detail the evaluation results for each particular obfuscation technique.

4.1 Evaluation of Normalization for Code Reordering

We generated a large number of variants of the Beagle.Y virus and measured the detection rate before and after normalization. Each variant was generated by applying code reordering to a randomly chosen set of program fragments. In Table 1 we list the detection rate when the obfuscation is applied to 10%, 50%, and 90% of the program. For each obfuscation rate we generated 1,000 variants, for a total of 3,000 variants. While the detection rate for these obfuscated variants dropped significantly in many cases, after we applied normalization the detection rate went back up to 100%.

Obfuscation Ratio	*Norton AntiVirus*		*McAfee VirusScan*		*Sophos Anti-Virus*		*ClamAV*	
	Standalone	*Normalized*	*Standalone*	*Normalized*	*Standalone*	*Normalized*	*Standalone*	*Normalized*
10%	100%	100%	36.83%	100%	33%	100%	100%	100%
50%	100%	100%	50.75%	100%	5%	100%	100%	100%
90%	100%	100%	49.69%	100%	0%	100%	100%	100%

Table 1: Detection rates for Beagle.Y variants obfuscated using code reordering. The *Normalized* column indicates the detection rate of commercial virus scanners when the input program was first normalized.

Note that, in Table 1, Norton AntiVirus and ClamAV are outliers. Both succeed in detecting all variants generated using the reordering obfuscation. The reason is that their signatures for Beagle.Y match data areas of the virus. Thus, any obfuscation that affects only code areas, such as the reordering obfuscation, does not impact the detection rates of Norton AntiVirus or ClamAV.

4.2 Evaluation of Normalization for Programs with Semantic Nops

Similar to the evaluation of the normalization for reordered programs, we generated a large number of variants of the Beagle.Y virus and measured the detection rate before and after normalization. Each variant was generated by inserting junk code into a randomly chosen set of program locations. We list in Table 2 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 normalization the detection rate went back up to 100%. The Norton AntiVirus and ClamAV detection rates stand out again for the same reason as before.

Obfuscation Ratio	*Norton AntiVirus*		*McAfee VirusScan*		*Sophos Anti-Virus*		*ClamAV*	
	Standalone	*Normalized*	*Standalone*	*Normalized*	*Standalone*	*Normalized*	*Standalone*	*Normalized*
10%	100%	100%	65.5%	100%	35.0%	100%	100%	100%
50%	100%	100%	44.0%	100%	27.0%	100%	100%	100%
90%	100%	100%	17.1%	100%	4.4%	100%	100%	100%

Table 2: Detection rates for malware variants obfuscated using junk insertion. The *Normalized* column indicates the detection rate of commercial virus scanners when the input program was first normalized.

4.3 Evaluation of Normalization for Self-Generating Programs

To evaluate malware normalization performance 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*, transform a program such that the program body (the code and/or the data) is compressed, and a new program is created to include the decompressor as well as the compressed data. At execution time, the decompressor extracts and transfers control to the original program body.

We used the following packers: Petite, UPX, ASPack, Packman, UPack, PE Pack, and FSG. Each packer was run on several versions of the Netsky and Beagle viruses (which had been manually unpacked beforehand, if necessary), to obtain a total of 90 different new variants. The self-generating variants were then normalized to obtain the set of normalized variants. Both sets were scanned using the four anti-virus tools. The results are summarized in Table 3. As the packers we tested do not allow partial packing, these numbers reflect 100% (whole program) packing.

Norton AntiVirus		*McAfee VirusScan*		*Sophos Anti-Virus*		*ClamAV*	
Standalone	*Normalized*	*Standalone*	*Normalized*	*Standalone*	*Normalized*	*Standalone*	*Normalized*
60.0%	75.6%	57.8%	94.4%	58.9%	83.3%	28.9%	82.2%

Table 3: Detection rates for packed malware variants. The *Normalized* column indicates the detection rate of commercial virus scanners when the input program was first normalized.

The numbers show that normalization for self-generating programs 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 normalized 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 normalized instance. The majority of malware detectors already come with signatures of unpacked malware to support specialized unpacking engines for common executable packers. ClamAV, however, sometimes failed to detect the normalized instances due to a lack of signatures for the unpacked malware variants.

4.4 Normalization Times

We present here the average execution time of the malware normalizer for the various normalization steps. Since this is an unoptimized research prototype, we believe significant speed gains can be made through an optimized implementation. As shown in Figure 3, both the reordering normalizer and the unpacking normalizer perform in the 10-15 second range. The semantic-nop removal normalizer is significantly slower as it must query the decision procedures 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.

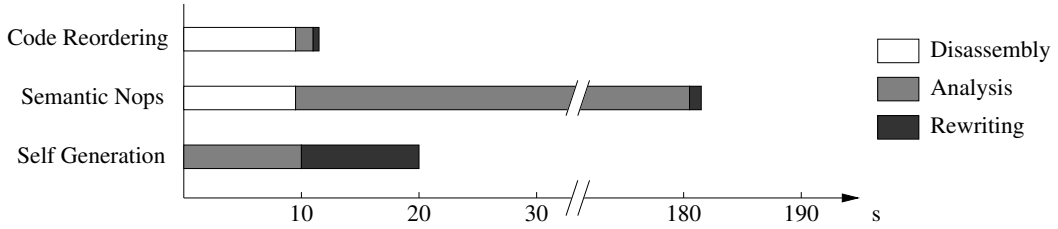

Figure 3: Average running times of the malware normalizer for different obfuscations. Times are broken down into phases for disassembly, analysis (static analysis resp. emulation), and rewriting of the executable.

4.5 Limitations

Our malware normalization has several limitations which we plan to address in future work. Normalization for code reordering can be foiled by unreachable code sequences that might add false fall-through edges to a control-flow graph. Simple unreachable code sequences are easily identifiable, while more complex sequences using opaque predicates [8] or branch functions [23] are subject to ongoing research.

The normalization for self-generating programs generally does not produce ready-to-run unpacked executables. While all the code is present in the normalized executable, the imports table listing the dynamically linked libraries used by the program is not recovered, since most packing obfuscations replace it by a custom dynamic loader. We note however, that in our experiments the integrity of the import table has not proven to be relevant for successful identification of a virus by the tested malware detectors.

5 Related Work

In the malware-writing world, obfuscations have long been used to evade detection. Polymorphic malware, which encrypts the code under a different key at each replication, and metamorphic malware, which morphs

itself at each replication, have presented increasingly more complex threats to malware detectors [14]. 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 [38]. Other tools such as the executable packer UPX [32] 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 [3, 6–8, 37], we note that the goal of our work is to address the particular obfuscations used by malware writers.

Deobfuscation tools appeared in response to particular attack instances. Detection of polymorphic malware requires the use of emulation and sandboxing technologies [28, 29]. The sandboxing engine is integrated into the malware detector, such that any malicious code can be detected in memory during execution. This approach is open to resource-consumption attacks and can have false negatives, since the execution time in the sandbox often has to be heuristically restricted for performance reasons [30, 31]. In contrast, our normalization for self-generating programs terminates the program as soon as it attempts to execute dynamically generated code. Thus we do not depend on preset execution limits to unpack a program.

Static analysis approaches for detecting and undoing obfuscations have been proposed by Udupa, Debray, and Madou [35], who examined simple techniques for defeating the effect of code obfuscation (specifically, control-flow flattening). Lakhotia and Mohammed [21] presented a method to order program statements and expressions in C source code.

When we apply our malware normalization algorithm, we assume that the code has been successfully disassembled. While recent work [23] showed that an attacker can make disassembly hard, we note that other researchers have already proposed solutions to make disassembly precise and correct. Kapoor [16] presented techniques for disassembly of malicious code. Kruegel, Robertson, Valeur, and Vigna [19] improved the success of the disassembly process in the presence of obfuscations targeted at the disassembly process.

Several new research approaches propose semantics-based malware detectors that can reliably handle malware variants derived through obfuscation or program evolution [5,17]. While these methods are promising, 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 normalizer provides an easier upgrade path to more advanced detection techniques, as the malware normalizer works in conjunction with existing detectors.

6 Conclusion

This paper presented a malware normalizer 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 normalizer. An additional benefit of malware normalization is the separation of concerns between the malware normalization stage and the malware detection stage. This leads to more maintainable software and allows for independent improvements in malware normalization techniques and malware detection algorithms. In the future, we will expand the set of obfuscations handled by our malware normalizer and investigate performance improvements.

References

[1] AVV. Antiheuristics. *29A Magazine*, 1(1), 1999.

[2] F. Bellard. Qemu. Published online at http://fabrice.bellard.free.fr/qemu/. Last accessed on 16 Aug. 2005.

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

[4] 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 SIGSOFT, ACM Press.

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

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

[7] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the International Conference on Computer Languages 1998 (ICCL'98)*, pages 28–39, Chicago, IL, USA, May 1998. IEEE Computer Society.

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

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

[10] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elseview and MIT Press, 1990.

[11] 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.

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

[13] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th USENIX Security Symposium (Security'01)*, pages 115–131, Washington, D.C., USA, Aug. 2001. USENIX.

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

[15] L. Julus. Metamorphism. *29A Magazine*, 1(5), 2000.

[16] A. Kapoor. An approach towards disassembly of malicious binary executables. Master's thesis, The Center for Advanced Computer Studies, University of Louisiana at Lafayette, Nov. 2004.

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

[18] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 155–169, New York, NY, USA, 2000. ACM Press.

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

[20] 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.

[21] A. Lakhotia and M. Mohammed. Imposing order on program statements and its implication to AV scanners. In *Proceedings of the 11th IEEE Working Conference on Reverser Engineering (WCRE'04)*, pages 161–171, Delft, The Netherlands, Nov. 2004. IEEE Computer Society Press.

[22] W.-J. Li, K. Wang, and S. J. Stolfo. Fileprints: Identifying file types by n-gram analysis. In *Proceedings of the 6th Annual IEEE Information Assurance Workshop*, pages 64–71, United States Military Academy, West Point, NY, USA, June 2005.

[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)*. ACM Press, Oct. 2003.

[24] M. McDaniel and M. H. Heydari. Content based file type detection algorithms. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICCSS'03)*, Jan. 2003.

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

[26] Mental Driller. Metamorphism in practice. *29A Magazine*, 1(6), 2002.

[27] D. Mohanty. Anti-virus evasion techniques and countermeasures. Published online at http://www.hackingspirits.com/eth-hac/papers/whitepapers.asp. Last accessed on 18 Aug. 2005.

[28] C. Nachenberg. Polymorphic virus detection module. *United States Patent # 5,696,822*, Dec. 1997.

[29] C. Nachenberg. Polymorphic virus detection module. *United States Patent # 5,826,013*, Oct. 1998.

[30] K. Natvig. Sandbox technology inside AV scanners. In *Proceedings of the 2001 Virus Bulletin Conference*, pages 475–487. Virus Bulletin, Sept. 2001.

[31] K. Natvig. Sandbox II: Internet. In *Proceedings of the 2002 Virus Bulletin Conference*, pages 1–18. Virus Bulletin, 2002.

[32] M. F. Oberhumer and L. Molnár. The Ultimate Packer for eXecutables (UPX). Published online at http://upx.sourceforge.net/. Last accessed on 16 Aug. 2005.

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

[34] P. Ször. *The Art of Computer Virus Research and Defense*, chapter Advanced Code Evolution Techniques and Computer Virus Generator Kits. Symantec Press. Addison Wesley Professional, 1st edition, Feb. 2005.

[35] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th IEEE Working Conference on Reverse Engineering (WCRE'05)*, Pittsburgh, PA, USA, Nov. 2005. IEEE Computer Society Press.

[36] S. Wehner. Analyzing worms and network traffic using compression. Published online at http://arxiv.org/abs/cs.CR/0504045. Last accessed on 21 Aug. 2005.

[37] G. Wroblewski. *General method of program code obfuscation*. PhD thesis, Institute of Engineering Cybernetics, Wroclaw University of Technology, Wroclaw, Poland, 2002.

[38] z0mbie. Automated reverse engineering: Mistfall engine. Published online at http://z0mbie.host.sk/autorev.txt. Last accessed: 16 Jan. 2004.

A Step-by-Step Normalization Example

This section describes the detailed steps of the normalization algorithm applied to the program fragment in Listing 2 on page 3. This program fragment is a variant of the code in Listing 1 with the packing, junk insertion, and reordering obfuscations applied. The goal of malware normalization is to remove the obfuscations, and we want the normalized code to be as close as possible to the original code of Listing 1.

The malware normalizer starts by checking for the presence of any reordering obfuscations. The reordering normalizer will identify the instructions on lines 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, and 22 as anomalous as they violate the CFG invariant. All except the instruction on line 4 are straightforward to identify as violators. Each has one immediate predecessor and that immediate predecessor is an unconditional jump. Thus, the immediate predecessors (instructions on lines 1, 3, 5, 7, 11, 13, 15, 17, 19, and 21) are candidates for removal. The instruction on line 4 has two immediate predecessors, both of them jumps. While one jump is unconditional, the other one is conditional and cannot be considered for removal. Then the immediate predecessor that is an unconditional jump (i.e., the instruction on line 9) is also a candidate for removal.

The rewriting step of the reordering normalization takes the list of instructions that are candidates for removal and edits the instruction sequence such that the execution semantics of the program are preserved. Each removed instruction is replaced with the basic block it targets. The instruction on line 1 is replaced by instructions on lines 14 and 15, the instruction on line 3 is replaced by instructions on lines 18 and 19, and so on. The end result is a code sequence (in Listing 4) with a reduced number of unconditional jumps, such that the effects of the reordering obfuscation are eliminated.

```
            mov     edi, offset der
2           mov     al, 99
loop_start: cld
4           xor     byte ptr [edi], 1
            scasb
6           push    edi
            add     [esp], 1
8           pop     edi
            dec     edi
10          loop    loop_start
            jmp     short der
12          ...
der:        db 8c 84 d9 ff fe fe 57 51
14          ...
            db 01 98
```

Listing 4: Reordering normalization applied to Listing 2.

```
            mov     edi, offset der
2           mov     al, 99
loop_start: cld
4           xor     byte ptr [edi], 1
            scasb
6           loop    loop_start
            jmp     short der
8           ...
der:        db 8c 84 d9 ff fe fe 57 51
10          ...
            db 01 98
```

Listing 5: Semantic nop normalization of Listing 4.

The next stage of the malware normalization algorithm is the identification and removal of semantic nops. As described in Section 3.2, the algorithm locates all the hammocks in the code and queries the decision procedures about each hammock. In Listing 4 there are multiple hammocks, as each individual instruction is a hammock and most sequences of consecutive instructions are also hammocks. For example, the instructions on lines 1–2 form a hammock. Similarly, the complete loop on lines 3–9 forms a hammock. Note that the instructions on lines 1–3 are not a hammock, since there are *two* entry points into this would-be hammock: one entry point is through the instruction on line 1, another entry point is through the instruction on line 3. The decision procedures report the hammock composed of lines 6–9 as a semantic nop. The malware normalizer then simply removes the corresponding code from the program, to obtain the sequence in Listing 5.

The last stage is the unpacking normalization. The program is run inside the *qemu* processor emulator, customized to capture all memory writes to a trace file. In our example, the instruction on line 4 in Listing 5 is the only memory write operation. The trace records the destination address and the new data value for each memory write. At the same times, the emulator monitors the program counter and detects whether control flow reaches any previously written memory area. During the execution of the code in Listing 5 the jump instruction on line 7 transfers control flow to the address `der` which was previously written. At this point the emulator stops the program execution; the memory space of the stopped program is similar to Listing 6. The original program code is at lines 1–7, while the newly generated code resides at lines 9–29.

The next step in the unpacking normalization is the construction of a new executable using the trace data. The values of the most recent memory writes with destinations inside the original memory space (i.e., between lines 1 and 11 in Listing 5) are used to create the new executable. The existing code (i.e., decryption loop and the associated initialization code) is effectively removed by setting the entry point to the beginning of `der`, to obtain the sequence in Listing 7. The resulting program is identical to the initial program (Listing 1) obfuscated by the malware writer. If a malware detector identifies Listing 1 as malicious, it will also identify Listing 7 as malicious.

```
                mov     edi, offset der
                mov     al, 99
loop_start:     cld
                xor     byte ptr [edi], 1
                scasb
                loop    loop_start
                jmp     short der
                ...
der:            lea     eax, [ebp+Data]
                push    esi
                push    eax
                call    ds:GetWindowsDirectoryA
                lea     eax, [ebp+Data]
                push    eax
                call    _strlen
                cmp     [ebp+eax+var_129], 5Ch
                pop     ecx
                jz      short loc_402F56
                lea     eax, [ebp+Data]
                push    offset asc_408D80
                push    eax
                call    _strcat
                pop     ecx
                pop     ecx
loc_402F56:
                lea     eax, [ebp+Data]
                push    offset aServices_exe
                push    eax
                call    _strcat
```

Listing 6: One of the program instances obtained using the normalization algorithm for self-generating programs applied to Listing 5.

```
                lea     eax, [ebp+Data]
                push    esi
                push    eax
                call    ds:GetWindowsDirectoryA
                lea     eax, [ebp+Data]
                push    eax
                call    _strlen
                cmp     [ebp+eax+var_129], 5Ch
                pop     ecx
                jz      short loc_402F56
                lea     eax, [ebp+Data]
                push    offset asc_408D80
                push    eax
                call    _strcat
                pop     ecx
                pop     ecx
loc_402F56:
                lea     eax, [ebp+Data]
                push    offset aServices_exe
                push    eax
                call    _strcat
```

Listing 7: Result of the normalization algorithm for self-generating programs applied to Listing 5. The code-generation loop was removed.