Accepted to SCAM 2006: The 6th IEEE Workshop on Source Code Analysis and Manipulation, *Sep. 27–29, 2006, Philadelphia, PA, U.S.A.*

Normalizing Metamorphic Malware Using Term Rewriting

Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane & Arun Lakhotia
Center for Advanced Computer Studies
University of Louisiana at Lafayette
arun@louisiana.edu, walenste@ieee.org

Abstract

Metamorphic malware — including certain viruses and worms — rewrite their code during propagation. This paper presents a method for normalizing multiple variants of metamorphic programs that perform their transformations using finite sets of instruction-sequence substitutions. The paper shows that the problem of constructing a normalizer can, in specific contexts, be formalized as a term rewriting problem. A general method is proposed for constructing normalizers. It involves modeling the metamorphic program's transformations as rewrite rules, and then modifying these rules to create a normalizing rule set. Casting the problem in terms of term rewriting exposes key challenges for constructing effective normalizers. In cases where the challenges cannot be met, approximations are proposed. The normalizer construction method is applied in a case study involving the virus called "W32.Evol". The results demonstrate that both the overall approach and the approximation schemes may have practical use on realistic malware, and may thus have the potential to improve signature-based malware scanners.

1 Introduction

Malicious programs like worms, viruses, and Trojans are collectively known as "malware" [14]. In 1989, Cohen [8] anticipated that *metamorphic* malware would one day be created, that is, the malware would be able to transform its own code so as to create variants of itself [13]. Six years later, such metamorphic malware began to appear. Variants created by metamorphic malware might still behave like the original program, but their code would be different. For example, an early metamorphic virus called `W95.RegSwap` rewrote itself so that some of the general-purpose registers it used were swapped [13]. More recent metamorphic viruses perform a host of very complicated transformations, including code substitution, insertion of irrelevant instructions, reordering of instructions, and more [16].

The main reason metamorphism was introduced to malware, of course, was to try to evade detection from malware scanners. Malware scanners typically rely on *signatures* to detect malware. In general, a signature could be any sort of pattern of data, code, or behavior, although typical scanners use relatively simple patterns of bytes, code, or calling behavior. A signature is effective if it matches malware yet is extremely unlikely to match any benign program. If a benign program is matched, a *false positive* is created. Malware scanner writers typically try to avoid false positives. The signatures they use may thus be highly specific to the malicious program the signature is intended to match.

Metamorphic malware can potentially cause serious difficulties for signature-based scanners. A given signature may not match all variants of a metamorphic program, so multiple signatures may be required, as shown in Figure 1. In the worst case a different signature is required for every possible variant. However, metamorphic programs may create an unbounded number of variants. To be able to detect all variants, something must be done to keep the number of required signatures down to a tractable quantity.

One approach is to make the pattern matching more capable. Indeed, this was done to catch early metamorphic malware. For example, in the case of `W95.RegSwap` the scanners at the time were enhanced to use wild-card based matching, allowing them to match variants regardless of their specific assignment of registers [16]. Also, malware scanners started using emulation to match dynamic behavior, which can stay constant even if the code changes. Although dynamic behavior

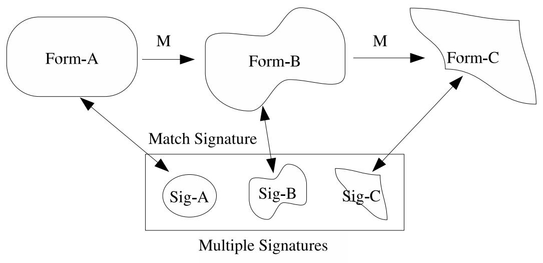

Figure 1. Detect using unique signatures

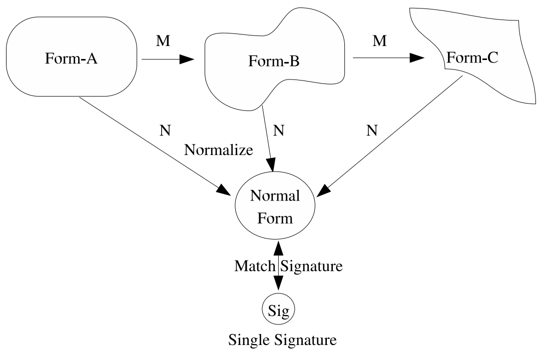

Figure 2. Detect using a normal form

pattern scanning has been successful, it is not without limitations. It is costly, and there are various tricks that can be used to foil the emulation-based scanners.

An alternative approach is exemplified by the methods of Lakhotia and Mohammed [12]. The general approach, illustrated in Figure 2, is to *normalize* the input to the scanner, i.e., to reduce the number of possible variants it needs to recognize, and thus reduce the power needed in the matching techniques, or else reduce the number of signatures needed to match the malicious programs. In the ideal case, the normalizer will transform all possible variants into a single "normal" form.

For this approach to succeed in general, one must be able to construct suitable normalizers. We call this the "Normalizer Construction Problem" or NCP. As a proof-of-concept, Lakhotia and Mohammed [12] developed a generic normalizer for `C` programs. It removed program variations via program transformations such as expression reshaping, renaming of variables, and instruction reordering. While they were unable to reduce the programs to a single form, they reported 10^{183} to 10^{20} reduction in the space of variants. Though this is an important first step, the problem of constructing effective normalizers, in general, is still an open problem.

This paper presents a method for solving the NCP for a specific class of metamorphic programs. The method utilizes techniques from term rewriting [3]. In particular, the NCP is cast in terms of a problem of constructing term rewriting systems that meet specific criteria. This approach can be applied to metamorphic programs that (1) substitute instruction sequences with equivalent instruction sequences, or (2) insert irrelevant code, i.e., code that has no bearing on the overall computation performed by the program. When all of a metamorphic program's transformations are known, we show that in some cases it is possible to create a "perfect" normalizer: i.e., it transforms all variants into a single normal form, and no non-variant is transformed into the same form. The paper also introduces two approximations that can be made when such a perfect normalizer is either not possible or not feasible. A case study demonstrates the general feasibility of both the approaches.

Section 2 gives some background information on term rewriting, and outlines the critical issues that need to be addressed when solving the NCP. Section 3 proposes a strategy for solving the NCP by applying a *completion procedure* to an extracted metamorphic rule set. Section 4 introduces approximations that can be utilized when the method of Section 3 are not able to generate the "perfect" normalizer. Section 5 describes a case study that evaluates the general feasibility of the approach using the `W32.Evol` virus. Section 6 lists relations to other work. Conclusions are drawn in Section 7.

2 Term rewriting and the NCP

Metamorphic malware are typically decomposable into two components: a *metamorphic engine*, which performs source-to-source transformations on programs, and a *payload*, which is the body of code that the engine is applied to. This separation makes it possible to attach a single metamorphic engine to a variety of different payloads. Good metamorphic engines are known to be difficult to create, and this may explain why there are so few of them, and why they evolve slowly [15].

Given these facts, one might wonder whether it is possible to extract all the transformations of a given metamorphic engine and then simply "reverse" all the transformations. That is, if $A \rightarrow B$ appears in the metamorphic engine, could one not create a normalizer that simply applies $B \rightarrow A$? Assume that S is the set of possible variants created through transformations of some metamorphic engine M. The basic insight is that any element in S must have been created through some sequence of transformations $T = m_1, m_2, \ldots$, so if one simply reverses the transformations in T then one would remove the variations, yielding the original program again.

While the scheme may appear *prima facie* sound, it will not work in general. This is illustrated below in an example using the virus `W32.Evol`. Select examples of its transformations are shown in Figure 3. The disassembly of the parent's code is shown in the left column and the corresponding transformed offspring code in the right column. The parts of the code changed in the offspring are shown in bold face.

The transformation shown in Figure 3(a) replaces the `mov [edi], 0x04` instruction with a code segment that saves the value of register `ecx` by pushing it onto the stack, moves `0x04` into `ecx`, and then into the memory location pointed to by `edi`, and finally restores the previous value of `ecx` by popping it from the stack back into `ecx`. The transformation shown in Figure 3(b), replaces the `push 0x04` instruction with a code segment that moves `0x04` into register `eax`, which it then pushes onto the stack. The transformation shown in Figure 3(c) inserts an irrelevant statement (i.e. one that has no effect on the computation) `mov eax, 0x09`. The first transformation is unconditionally semantics-preserving: the meaning of the program stays the same no matter when the rule is applied. The last two are semantics-preserving under the condition that register `eax` is not live at the point where either of them is applied.

Inspection of the transformation rules shown in Figure 3 reveals some problems that can arise when one applies the rules in reverse. Consider, for instance, the

	Parent	**Offspring (transformed)**
	push eax	push eax
	mov [edi], 0x04	**push ecx**
(a)	jmp label	**mov ecx, 0x04**
		mov [edi], **ecx**
		pop ecx
		jmp label
	push 0x04	**mov eax, 0x04**
(b)	mov eax, 0x09	**push eax**
	jmp label	mov eax, 0x09
		jmp label
	mov eax, 0x04	mov eax, 0x04
(c)	push eax	push eax
	jmp label	**mov eax, 0x09**
		jmp label

Figure 3. Three sample rules from `W32.Evol`

$$add(1,1) \rightarrow 2\ ;\ add(1,2) \rightarrow 3\ ;\ add(0,3) \rightarrow 3$$

Figure 4. Sample rewrite rules

hypothetical sequence

```
mov eax, 0x04; push eax; mov eax, 0x09
```

which can be part of the output of either rule in Figure 3(b) or Figure 3(c). The normalizer must be able to decide which of rules (b) and (c) to apply in reverse. If it cannot, the result of its transformations may not be a single normal form. Worse still is if it applies the wrong rule and either transforms a non-variant into a variant, or a variant into a non-variant. For example, if the `mov eax, 0x09` in the hypothetical sequence is *not* a junk instruction, then applying rule (c) in reverse yields a different program, meaning it should not be in S, and the rule application is incorrect.

The above issues of transformation ordering and normal forms are dealt with elegantly by term rewriting systems. The following subsections review relevant background from term rewriting literature, and then the theory is recruited to formalize the NCP in terms of constructing a convergent rule set with three specific properties.

2.1 Term rewriting background

This section briefly reviews definitions and results from term rewriting literature [3] that we will use in later sections.

Terms, subterms, atomic, and ground. For appropriately chosen domains, *terms* are constants, variables, functions, or functions on terms. The term $multiply(2, add(3,1))$, for example, is built using the binary functions add and $multiply$ on integers and the constant integers $1, 2,$ and 3. A term t may contain other terms (called *subterms* of t). An *atomic* term is one that does not contain subterms. A *ground* term is one that does not contain variables.

Term rewriting system (TRS). A *term rewriting system* is a set of rewrite rules. A *rewrite rule* $s \rightarrow t$ maps term s to term t. A *conditional* TRS is one that has conditions attached to its rules. The notation $p|R$ means that rule R may be applied only when condition p holds. Figure 4 shows a simple example of an unconditional TRS.

Reduction relation ($\rightarrow_T$). A TRS T induces a relation $\rightarrow_T$ on terms, also denoted $\rightarrow$ where clear from the context. Given terms s and t, $\rightarrow_T$ is defined as follows: $s \rightarrow_T t$ holds iff, for some rewrite rule $s' \rightarrow t'$, s has as a subterm an instance of s' which, if replaced with its corresponding instance of t', turns s into t.

Normal form. If a term t is not related to any other term under $\rightarrow_T$, then t is said to be in *normal form* with respect the rewriting system T. $Norm_T(x)$ is the set of terms in $[x]_T$ which are in normal form. For the TRS in Figure 4, the term $add(2, 2)$ is in normal form , and $add(1, add(1, 1)) \rightarrow_T add(1, 2)$ by application of the rule mapping $add(1, 1)$ to 2.

Termination. A TRS T is terminating if there exists no infinite descending chain of the form $a \rightarrow b \rightarrow c \cdots$.

Confluence. Let $w,\ x,\ y\ and\ z$ denote arbitrary terms. Suppose there is a sequence of applications of rewriting rules that reduces x to y and another sequence that reduces x to z. The system is confluent if y and z are joinable. Two terms y and z are said to be *joinable* if there is a sequence of applications of rewriting rules that reduces y and z to some term w. Confluence of a TRS is in general undecidable.

Convergence. A TRS is *convergent* if it is confluent and terminating. If a TRS T is convergent then it can be used to decide membership in any of the equivalence classes defined by the reflexive symmetric transitive closure $\overset{\star}{\longleftrightarrow}$ of the relation $\rightarrow$ it induces on terms.

Equivalence relation ($\overset{\star}{\longleftrightarrow}$). The symbol $\overset{\star}{\longleftrightarrow}$ denotes the equivalence relation that is the reflexive symmetric transitive closure of the relation $\rightarrow$ induced by T. This equivalence relation partitions the set of terms into equivalence classes. Given a TRS T, $[t]_T$ denotes the equivalence class of term t under $\overset{\star}{\longleftrightarrow}$.

2.2 NCP as a term rewriting problem

The normalizer construction problem, introduced informally in Section 1, can now be formalized as follows.

Modeling the metamorphic engine. A metamorphic engine is modeled as a TRS. An instruction is a term that consists of a function (the opcode mnemonic) applied to one or more variables or constants (the register, variable, and immediate operands). A program, or a code segment, is a term obtained by applying a `concatenate` function to such terms. Figure 5 gives an example of how a transformation rule of a metamorphic engine is captured as a rewrite rule.

The rule in Figure 5 is not a conditional one: its left hand side, if interpreted as a code segment, is semantically equivalent to its right hand side, no matter its context. This is also true for the first rule of Figure 3. Other rewrite rules may need to be conditioned in order to accurately model the condition-sensitivity of transformations such as those shown in Figure 3(b) and Figure 3(c). We used a conditional rewriting system to capture the transformation rules in our case study (Section 5). For simplicity, we will henceforth write rules in assembly language with embedded term variables, rather than in the function application form shown in Figure 5.

```
        mov (reg1, imm)
          { push (reg2);
 ⟶          mov  (reg2, imm);
            mov  (reg1, reg2);
            pop  (reg2); }
```

Figure 5. Code substitution rewrite rule

Let M denote the TRS modeling the transformation engine of our target metamorphic program. The equivalence relation induced by M partitions terms (modeling programs) into equivalence classes. If M is convergent, then it can be used to decide whether two terms x and y belong to the same equivalence class by verifying that their normal forms with respect to M are equal. A convergent M implies that any sequence of transformations of any variant will eventually result in the a-priori computable normal form of the program.

A convergent M therefore essentially defeats the purpose of metamorphism, as the malicious program will fail to create distinct variants once it transforms itself into its normal form. A convergent M also provides a potential way for the scanner to recognize the program (i.e., by applying the malware's own M until it converges to the normal form). Thus it is reasonable to expect malicious engines to be non-convergent. To build a normalizer, it must be made convergent.

NCP in term rewriting terms. Given a metamorphic rewrite system M, construct a rewrite system N that satisfies the following properties:

Equivalence: $\forall x.[x]_M = [x]_N$.

Termination: N must be terminating.

Confluence: N must be confluent.

The equivalence condition states that for any term x, neither $\exists y.y \in [x]_M \wedge y \notin [x]_N$ nor $\exists y.y \notin [x]_M \wedge y \in [x]_N$ holds. This implies that, for any term x, the terms that are related to x under the reflexive symmetric transitive closure of M remain related to x under the reflexive symmetric transitive closure of N, and vice versa. The termination condition requires that the relation $\rightarrow_N$ have no cycles or that *any* sequence of applications of the rules of N to some term t will eventually halt. It also implies that N is normalizing, meaning that every term has at least one normal form. The confluence condition implies that if a normal form for some term is reached, then this normal form is unique.

3 A strategy for solving NCP

Henceforth, M denotes the rewriting system modeling the transformation module of our target metamorphic malware. A normalizer for the malware may be constructed by first applying a *reorienting procedure* to M to ensure termination and then a *completion procedure* [10] to the resulting system. If the completion procedure halts, it returns a rewriting system that satisfies the equivalence and confluence properties and hence is a solution to the NCP. Since M is modeled as a TRS, this NCP solution works for the class of metamorphic engines that can be modeled using a conditioned TRS, namely, those that implement semantics-preserving instruction substitution types of transformations, including junk-inserting transformations.

3.1 Ensuring termination

While a rewrite rule relates equivalent terms, the term-rewriting system may apply the rule only in one direction. A reorienting procedure determines the direction in which a rule may be applied such that the reduction procedure of the term-rewriting system is guaranteed to terminate. To ensure that a set of reoriented rules M^t is terminating it is sufficient to show that for every directed rule $x \rightarrow y \in M^t$, $x > y$, for some *reduction order* $>$ on terms [3].

We used the well-founded length-lexicographic ordering [3] to order the elements of our set of terms. The reorientation procedure traverses M and reorients the rules whose right hand sides are length lexicographically greater than their left hand sides. The resulting system M^t is terminating because any sequence of application of its rules will decrease the length-lexicographic size of the term being reduced. Figure 6(a) shows a fragment of an example rewriting system, and Figure 6(b) shows the output of the reorienting procedure on input that fragment. Rule M_1 was reoriented because r_1 is length-lexicographically greater than l_1.

Since the reflexive symmetric transitive closure of M^t is identical to that of M, the set of equivalence classes defined by the former closure is identical to that defined by the latter; in other words, $\forall x.[x]_M = [x]_{M^t}$. Hence, M^t satisfies the termination and equivalence properties which are part of the requirements for a rewriting system to solve the NCP.

Rule	**Post Condition**		l_i
M_i	C_i	$\rightarrow$	r_i
M_1	T		`mov [`*`reg1`*`+`*`imm`*`], `*`reg2`*

		$\rightarrow$	`push eax`
			`mov eax, `*`imm`*
			`mov [`*`reg1`*`+eax], `*`reg2`*
			`pop eax`

M_2	eax is dead		`push `*`imm`*

		$\rightarrow$	`mov eax, `*`imm`*
			`push eax`

M_3	eax is dead		`push eax`

		$\rightarrow$	`push eax`
			`mov eax, `*`imm`*

M_4	T		`NOP`
		$\rightarrow$	

(a) M, the original rule set

Rule	**Post Condition**		l_i
M_i^t	C_i	$\rightarrow$	r_i

M_1^t	T		`push eax`
			`mov eax, `*`imm`*
			`mov [`*`reg1`*`+eax], `*`reg2`*
			`pop eax`

		$\rightarrow$	`mov [`*`reg1`*`+`*`imm`*`], `*`reg2`*

M_2^t	eax is dead		`mov eax, `*`imm`*
			`push eax`

		$\rightarrow$	`push `*`imm`*

M_3^t	eax is dead		`push eax`
			`mov eax, `*`imm`*

		$\rightarrow$	`push eax`
M_4^t	T		`NOP`
		$\rightarrow$	

(b) M^t, the reoriented rule set

Figure 6. Reorienting example

3.2 Ensuring confluence

Confluence is decidable for finite terminating term rewriting systems [3]. If a TRS is not confluent then additional rules may be added to it to make the system confluent. The process of adding rules to make a TRS confluent is called a *completion procedure*. The resulting confluent TRS is called *completed*. The problem of completing a TRS is undecidable.

The Knuth-Bendix completion procedure (KB) is the most prevalent method used in term-rewriting literature [10]. It adds rules to resolve *critical overlaps* between rules. For finite terminating ground term rewriting systems, the left hand sides of a pair of (not necessarily distinct) rules are said to critically overlap if the prefix of one is identical to the suffix of the other, or if one is a subterm of the other. Critical overlaps indicate conflicts in a rewriting system that may make the system non-confluent [3]. For the example in Figure 6(b), M_1^t and M_2^t critically overlap at `push eax`. The same is true for M_2^t and M_3^t. KB resolves such critical overlaps by repeatedly adding rules to the system in fashion similar to that shown in Figure 7. KB is not guaranteed to terminate. However, if it does terminate then the TRS it produces will be confluent. A detailed discussion of this procedure is available in [3, 10].

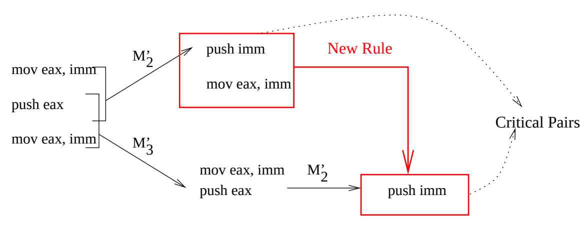

Figure 7. Completion step for M_2^t and M_3^t

For the TRS of `W32.Evol` if the left hand side of some rule has, as a suffix, the prefix of the left hand side of some other rule, it is not enough to conclude that the rules critically overlap. Neither is it sufficient for the left hand side of some rule to be a subterm of the left hand side of another. This is due to the fact that either of the rules may be conditional. It may even be the case that the condition of one is a negation of the other's. Rules M_1^t and M_3^t in Figure 6(b) overlap at `push eax ; mov eax, imm`. This overlap does not create any conflicts between the rules because M_3^t can be applied only when register `eax` is dead while M_1^t can be applied only when `eax` is live.

Figure 7 illustrates how the completion procedure is used to resolve the critical overlap between M_2^t and M_3^t of Figure 6. A rule mapping the irreducible term `push imm; mov eax,imm` to the irreducible term `push imm` is added to the system. The completion procedure terminated on input the rewriting system of Figure 6 and returned the confluent system of Figure 8.

4 Approximate solutions to the NCP

If the two-step procedure of Section 3 succeeds, it yields a tool for deciding membership in each of the equivalence classes defined by the reflexive symmetric transitive closure of N. Each of these equivalence classes, including that composed of the variants of the malware, contains a unique normal form. Given N and a suspect program, the scanner would simply use the rules of N, in any order, to reduce the suspect program until

Rule	**Post Condition**		l_i
N_i	C_i	$\rightarrow$	r_i

N_1	T		`push eax`
			`mov eax, imm`
			`mov [reg1+eax], reg2`
			`pop eax`

		$\rightarrow$	`mov [reg1+imm], reg2`

N_2	eax is dead		`mov eax, imm`
			`push eax`

		$\rightarrow$	`push imm`

N_3	eax is dead		`push eax`
			`mov eax, imm`

		$\rightarrow$	`push eax`
N_4	T		`NOP`
		$\rightarrow$	

N_5	eax is dead		`push imm`
			`mov eax, imm`

		$\rightarrow$	`push imm`

N_6	T		`push imm`
			`mov eax, imm`
			`mov [reg1+eax], reg2`
			`pop eax`

		$\rightarrow$	`mov eax, imm`
			`mov [reg1+imm], reg2`

Figure 8. Completed rules

a normal form is reached. If this normal form is equal to that of the metamorphic malware from which N was computed, the scanner would determine that the suspect program is, without a doubt, a variant of the malware. Non-variants of the metamorphic malware do not reduce to the malware's normal form.

Unfortunately, since KB is not guaranteed to terminate, the normalizer may or may not be generated by this method. Furthermore, since the conditions attached to each rule are extracted after a manual analysis of the engine's code, they may be inaccurate or incomplete. As a result, even if the normalizer is finally generated by the completion procedure, its correctness will depend entirely on that of these conditions. Moreover, the conditions may be costly, difficult, or impossible to calculate in the general case (e.g., accurate register liveness).

These observations motivate the search for approximate solutions to the NCP. Three potential approximations are introduced in this section, and arguments in their favor are raised. An initial evaluation of their feasibility was conducted in the study described in Section 5.

4.1 Failure to complete

Since the completion process—which repeatedly adds rules to M^t—may or may not halt, a running time limit and a maximum number of rule count is normally imposed on the completion procedure. If the completion procedure does not terminate within a "reasonable" amount of time, or if the repeated addition of rules yields a rule set that is simply too large to be useful for normalization purposes, then it may be reasonable to preempt the procedure and use the non confluent M^t as the normalizer. The non-confluence of the terminating system M^t implies that some, or all, of the equivalence classes induced by its reflexive transitive closure (which is also that of M) may contain multiple irreducible normal forms.

This implies that the malware's equivalence class may have more than one normal form. The actual number of these normal forms depends entirely on the malware code and on its transformation system. This set may nevertheless be helpful as an approximate solution, for it may be the case that some, or all, of these normal forms are similar enough. In such a case, all but one of these similar normal forms can be dropped from the set of the malware's normal forms.

Alternatively, an analyst can complete M^t in an *ad-hoc* manner by manually choosing and adding rules to M^t to create some normalizer N' which the analyst believes will produce a single normal form.

4.2 Incorrect condition evaluation

The process of analyzing and extracting the conditions under which rules are to be applied is one that require costly program analyses, such as control flow analysis and points-to analysis [2]. These analyses may fail to return accurate results due to the undecidability of some of the problems, and due to the nearly ubiquitous use of obfuscation techniques designed specifically to thwart static analysis techniques.

It may be reasonable, however, to approximate the condition checking. For example, a default decision on liveness might be taken when the liveness is not calculable precisely within an allotted time. Or perhaps no condition checking is performed at all. The drawback to using such a normalizer is that its transformations may not be semantics-preserving and may transform malicious code into non malicious code and vice versa. In practical terms the issue is whether these errors are likely to create false positives or false negatives in the matching.

4.3 Priority scheme

According to Visser [18], a *rule application strategy* can be imposed on a non convergent term rewriting system to make it behave like a convergent one. This result motivated our design and use of a simple priority scheme to reduce the likelihood of false matches. A simple priority scheme was implemented for use in the case study described in Section 5. It works as follows.

First the initial set N' of rules is partitioned into two subsets N'_U and N'_C, where N'_U contains the unconditional rules of N' and N'_C the contains the conditional rules of N'. For the rule set in Figure 8, the set of unconditional rules is $\{N_1, N_4, N_6\}$ and the set of conditional rules is $\{N_2, N_3, N_5\}$. A suspect code segment is normalized with respect to N' by giving priority to rules of N'_U over the rules of N'_C. Whenever a rule from N'_C is applicable on a term, it is chosen for application only if no rule from N'_U is applicable.

The priority scheme capitalizes on our knowledge that the rules in N'_U preserve semantics, whereas those in N'_C may not. Assigning a lower priority to the latter guarantees that the former will be applied before any non-semantic-preserving transformation gets applied.

5 Case study

A case study of using our exact and approximate solutions to the NCP on the `W32.Evol` is presented below.

5.1 Subject and preparation

We obtained a copy of a 12,288-byte long variant of an executable, infected with `W32.Evol`, from the *VX Heavens* archive [1]. We refer to this variant as the *Eve*. The engine of this virus is a relatively sophisticated one. It substitutes instructions with equivalent code segments, inserts irrelevant code at some sites, and replaces immediate operands with arithmetic expressions computing them. We conservatively estimated that our Eve variant can generate at least 10^{686} second generation variants, $10^{1,339}$ third generation variants, and $10^{1,891}$ fourth generation variants. The payload of this particular virus makes it possible for some emulation-based techniques to detect its variants [16].

Several factors make `W32.Evol` a suitable study subject. First, we are able to make it replicate and safely experiment on it in our secure environment. Second, its metamorphic engine is capable of generating enormous numbers of variants, and the variants it creates are significantly different from each other. This makes it a realistic study subject in that it is nontrivial to develop signatures for the entire space of variants. Third, `W32.Evol`'s metamorphic engine uses a conditional transformation system that contains critical over-

Gen	Eve	2	3	4	5	6
ASO	2,182	3,257	4,524	5,788	6,974	8,455
MSNF	2,167	2,167	2,184	2,189	2,195	2,204
ASNF	2,167	2,167	2,177	2,183	2,191	2,204
LNC	0	0	10	16	24	37
PC	100.00	100.00	99.54	99.27	98.90	98.32
ET	2.5	3.0	4.3	6.3	8.0	11.2
TC	16	533	980	1,472	1,902	2,481

Gen=generation; ASO=average size of original (LOC); MSNF=maximum size of normal form (LOC); ASNF=average size of normal form (LOC); LNC=lines not in common; PC=percentage common; ET=execution time (CPU secs); TC=transformation count

Table 1. Evaluating normalizer P_0

laps; this makes `W32.Evol` a suitable candidate for evaluating and illustrating our normalization approaches.

5.2 Materials and protocol

We first extracted the transformation rules of `W32.Evol` by manually reading the code, and occasionally tracing its execution in a debugger. We then implemented these rules as a term rewriting system M. Next, we used the reorienting procedure to transform M into an initial normalizing rewriting system N_0. N_0 was not completed. N_0 consisted of 55 rules, five of which did not participate in any overlaps.

We implemented two prototype normalizers using the priority scheme of Section 4.3. The first prototype normalizer, P_0, used the N_0 rule set as-is, and did not perform any condition checking. The second prototype normalizer, P_1, used a rule set N_1, which was N_0 with rules added manually to complete it (i.e., ad-hoc completion). P_1 also ignored conditions.

The term rewriting systems were implemented in TXL [9].[1] `W32.Evol` contains two classes of rules: those that can be applied unconditionally and those that can be applied only when certain registers are not live. Our rewrite system was inaccurate in that it did not have the ability apply rules conditionally. To compensate for our inability to verify conditions before applying a rule, we used a prioritization scheme that applied conditional rules only when no unconditional rule could be applied.

The two normalizers were applied to 26 different variants spread across six generations.

[1]TXL system version 10.4 (8.1.05).

5.3 Results

Table 5.1 shows the evaluation results for normalizer P_0. The row labeled **ASNF** of the table contains the average length (instruction count) of the normal forms of the variants. The row labeled **LNC** lists how many lines, on average, differ. The row labeled **LNC** shows the average raw percentage of sequence commonality (as measured by the common program `diff`) between the normal form of the Eve, and the normal form of the sample variant. The ones labeled **ET** and **TC** record execution information for the prototype.

The second normalizer, P_1, was convergent. All variants reduced to the same 2,166-line normal form, with similar running times.

5.4 Discussion

Because the case study is limited, hard generalizations are impossible. Nonetheless, the study serves as a useful feasibility test, particularly of the approximations. Furthermore, `W32.Evol` is a good representative sample, so the positive results are at least suggestive of some usefulness for similar metamorphic engines. Other complex metamorphic viruses, like RPME, Zmist, Benny's Mutation Engine, Mistfall, Metaphor, etc [4, 17, 19, 20] have transformations similar to that of `W32.Evol`, and it appears likely that for some subset of metamorphic programs, a syntactic normalizer built according to the strategy in Section 4 will normalize all variants sufficiently well for ordinary signature matching to succeed well enough.

Regarding feasibility, Table 5.1 shows that, even without completion or condition checking, the prioritization scheme creates normal forms that are highly similar—more than 98% in common. The differences indicate the possibility of false positives or negatives. This result was expected, as priority scheme cannot be a complete substitute for an accurate condition-sensitive rule set evaluator. Nevertheless, the high level of similarity suggests the likelihood of false matches may be low in practice. Manually inspecting the difference, we found that incorrect rule application occurred at three and two sites for the priority-based prototypes with and without ad-hoc completion, respectively. The chances seem remote that a program would be found on an actual user's computer which is benign yet different from `W32.Evol` on only three lines.

Regarding practicality, the timing information reflect the fact that our prototypes are proofs-of-concepts: they work on disassemblies, and are unoptimized. The time growth curve is shallow for the sizes of samples involved, taking less than five times as long on the largest sample, which is almost four times as large. Practically speaking, while `W32.Evol` always grows in size, growing very large is not a good survival strategy, and recent metamorphic malware try to keep the size of their code within reasonable limits by applying 'code-shrinking' transforms.

One might find fault with the fact that the normalization technique depends upon having a formalization of the specific metamorphic engine. This means the technique cannot be expected to find malicious programs for which the metamorphic engine is unknown. While this certainly is an issue, the limitation may be tolerable. Signature matching cannot detect novel malware either, but it has proved to be a useful technology when the signature database can be updated frequently.

One might also argue that modeling the metamorphic engine can be too difficult, or too costly. In response, we first note that metamorphic engines evolve slowly—much slower than the worms and viruses themselves [15], so the number of new metamorphic engines released in a year is low enough to make them amenable for such analysis. Second, the metamorphic engines tend to be reused, often in the form of libraries. This is because, at least for now, only certain malware authors have both the motivation and capability of writing transformation engines with a sophistication level that forces the use of nontrivial normalizers.

6 Relations to other work

Lakhotia and Mohammed [12] describe a different method for constructing a static normalizer, one which methodically transforms programs by reordering its instructions, renaming its variables, and reshaping its expressions. Their approach does not require one to first extract the metamorphic rule set, but it also cannot generally normalize all variants to a single form. Bruschi *et. al* [5] constructed a malware normalizer which applies standard compiler optimization techniques to normalize suspect programs, and then uses "clone detection" techniques to match their normal form to that of known metamorphic malware. Christodorescu *et. al* [7] constructed a static malware normalizer targeted to undo the transformations made (by hand) by malware authors. It uses static analysis to detect junk code insertion, and applies obfuscation-undoing transformations for obfuscations it recognizes.

The above three works share certain common attributes: they require complex static analysis (e.g., control flow or liveness), and utilize transformations that are not specific to a particular strain of malware. While these approaches do not depend on any a-priori information of a malware, they are nonetheless limited by the specific techniques they utilize. These methods do not theoretically guarantee that equivalent variants will be mapped to the same normal form. For example, there is no guarantee that the compiler optimization techniques will yield the same optimized program for any two arbitrary variants. A malware author can easily take advantage of the knowledge of their techniques and defeat them. In contrast, the present work is specific to a metamorphic engine, but proposes that deeper semantic analysis may not be necessary. An interesting research question arises as to the tradeoffs and benefits of general normalization rules versus ones targeted towards specific metamorphic engines. It is also an interesting question as to whether the precision offered by the completed normalizers offsets the initial cost of developing the normalizer.

The static techniques introduced in the paper can be contrasted with static detection techniques that use generic behavior patterns that can detect malicious programs even in the presence of variations in their code. Classic emulation-based techniques also look for behavior patterns, but they do so through dynamic methods, which may be attacked. Rather than emulation, Christodorescu *et. al* [6] and Kruegel *et al.* [11] proposed the use of static program analysis methods for detecting potentially obfuscated variants of specified behavior patterns. Their work is, effectively, a pursuit of a more capable pattern matcher, rather than a normalization approach. The normalization and behavior-match approaches are complementary and can be used together.

Acknowledgements

This research was funded, in part, by a grant from the Louisiana Governor's Information Technology Initative. The authors also wish to thank Michael Venable for his help in running the case study.

7 Conclusions

This paper presents an approach to construct a normalizer for a particular class of metamorphic malware by leveraging concepts and results from term-rewriting literature [3]. It was shown that metamorphic malware which use instruction substitution transformations or insert irrelevant instructions can be modeled as a conditional rewrite system. The problem of constructing a normalizer for this system then maps to the problem of constructing a convergent rewrite system by starting from the metamorphic engine's rule set. The latter problem has been well-studied: its problems and requirements for solution are known.

A general method was proposed for constructing either exact or approximated normalizers. When the rule set is completed, all variants are transformed into a single normal form. This proves that it is sometimes possible to develop "perfect" normalizers for the nontrivial class of metamorphic programs that perform semantics-preserving instruction sequence substitutions. The case study results suggest that this may be feasible in practice. Thus, the method has the potential to augment current static signature based scanners to detect metamorphic variants. That said, it was noted that not every rule set can be feasibly completed using an automated completion method. This identifies a weakness in any metamorphic normalizer that might potentially be exploited by malware authors. Research is still needed to understand the potential attacks and their possible remedies.

Finally, the approximations show that the general approach may have practical merit even when completion and accurate condition calculation cannot be guaranteed. Even without completion, and even without correctly calculating conditions, the prioritization approach yielded encouraging results on the test case. Though the normalizer did not map the 26 variants to a single normal form, there was over 98% similarity between the normal forms and the original program. Since the approximated normalizers forgo expensive analysis, they may be better suited in a scanner requiring real-time performance. Further research is needed to understand the practicality of using uncompleted rule sets, and for approximating the rule conditions.

References

[1] VX heavens. `vx.netlux.org`.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] Benny. Benny's metamorphic engine for Win32. `vx.netlux.org/29a/29a-6/29a-6.316`.

[5] D. Bruschi, L. Martignoni, and M. Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of International Symposium on Secure Software Engineering*, Washington, DC, USA, 2006. IEEE.

[6] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy*, pages 32– 46, 2005.

[7] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, Nov. 2005.

[8] F. Cohen. Computational aspects of computer viruses. *Computers & Security*, 8(4):325–344, 1989.

[9] J. R. Cordy. TXL – a language for programming language tools and applications. In *ACM 4th International Workshop on LTDA*, volume 110 of *Electronic Notes in Theoretical Computer Science*, pages 3–31. Springer-Verlag, Dec. 2004.

[10] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 342–376. Springer, 1983.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In A. Valdes and D. Zamboni, editors, *Recent Advances in Intrusion Detection: 8th International Symposium (RAID 2005)*, volume 3858 of *Lecture Notes in Computer Science*, pages 206–226. Springer-Verlag, 2006.

[12] A. Lakhotia and M. Mohammed. Imposing order on program statements and its implications to AV scanners. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, pages 161–171, Nov. 2004.

[13] C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):47–51, Jan 1997.

[14] E. Skoudis. *Malware: Fighting Malicious Code*. Prentice-Hall, 2004.

[15] P. Ször. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[16] P. Ször and P. Ferrie. Hunting for metamorphic. In *11th International Virus Bulletin Conference*, 2001.

[17] The Mental Driller. Metamorphism in practice. `vx.netlux.org/29a/29a-6/29a-6.205`.

[18] E. Visser. A survey of rewriting strategies in program transformation systems. In *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, 2001.

[19] Z0mbie. Automated reverse engineering: Mistfall engine. `vx.netlux.org/lib/vzo21.html`.

[20] Z0mbie. Some ideas about metamorphism. `vx.netlux.org/lib/vzo20.html`.