Normalizing Metamorphic Malware Using Term Rewriting

A Thesis

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Master of Science

Rachit Mathur

Fall 2006

Normalizing Metamorphic Malware Using Term Rewriting

Rachit Mathur

APPROVED:

_____	_____
Arun Lakhotia, Chair Associate Professor of Computer Science	William R. Edwards Associate Professor of Computer Science
_____	_____
Anthony Maida Associate Professor of Computer Science	C. E. Palmer Dean of the Graduate School

To Mom, Dad and dearest brother

Acknowledgments

I thank my advisor, Dr. Arun Lakhotia, for his valuable guidance, extraordinary support, inspiration, and encouragement. This thesis would never have been conceptualized without the ideas and motivation that he provided me. I greatly appreciate his patience and the trust he showed in me throughout my thesis. He was always there to support me both morally and academically whenever I was in a fix. My gratitude for him cannot be expressed in a paragraph.

I am grateful to Dr. Andrew Walenstein and Mohamed Chouchane without whose timely advice this thesis would not have materialized. I thank my parents and my brother for their never-ending encouragement, trust, and support during my lows and my highs in the research period. Thanks to Aditya Kapoor, Eric Uday, Michael Venable, and Enamul Karim who gave me helpful feedback during all times in my thesis and shared intriguing discussions about the challenges I faced.

Finally, I would also like to thank my roommates Gautham Konthum, Arshad Azeem, and Rajesh Sharma for their constant words of support and encouragement in the last two years and the good food they cooked for me.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Malicious programs like worms, viruses, and trojans are collectively known as "malware" [1]. Malware is called "metamorphic" when it is able to construct offspring that differ from itself [2]. The method typically used by existing metamorphic malware is to create a copy of itself and then mutate the copy. The collection of mechanisms a malware uses to mutate this copy are referred to as its "metamorphic engine." `W95.RegSwap`, for example, was an early metamorphic virus whose metamorphic engine rewrote the code by consistently substituting the registers used throughout [2].

The reason metamorphic engines were developed, of course, was to enable malware to avoid detection by malware scanners [3]. One of the primary techniques used by malware scanners is to match invariant patterns of data, code, or behavior. Any such distinguishing pattern can be called a "signature." The threat that metamorphic malware poses for all signature matching techniques is that it reduces or removes invariants the match relies upon. In the worst case, the malware scanner would require a signature for every variant, as shown in Figure 1. While `W95.RegSwap` could be matched using a more general signature [2], more powerful metamorphic engines could create many more variants—possibly unbounded quantities—each of which bear little obvious resemblance to the original. Indeed, metamorphic engines have been evolving in such a direction [2, 4]. Recent thought on the matter is that current scanning techniques, by themselves, will not be able to counteract more powerful self-transformation methods [5].

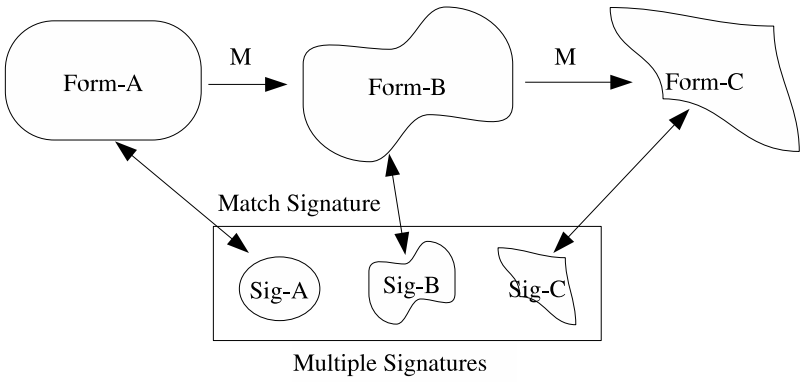

Figure 1: Detecting metamorphic worm variants using signature for each variant

1.2 Research objectives

One method for recognizing metamorphic malware is to re-transform every possible program variant into a single common form or, at least, a much smaller number of variants. As a first approximation one may think of the goal is to "undo" the metamorphic changes to recover the original program. However in reality the true aim is not to recover an original program, but rather a "normal" form which is representative of the class of all programs that are being considered equivalent; the process of creating this normal form is called "normalization." If the normalization is successful the prevalent signature-matching techniques can be leveraged to detect metamorphic malware, as shown in Figure 2.

1.3 Research contributions

This thesis shows how to construct normalizers for metamorphic programs. Theories and techniques from the field of *term rewriting* [6] are employed in this effort. Term rewriting is a general model of computation involving sets of *rules*, each of which specify how to rewrite some *terms* using other equivalent terms. A normalizer can be constructed from a term rewriting model of a metamorphic engine by judiciously reverting the direction of some of the rewrite rules and adding additional rules to guarantee a unique normal form.

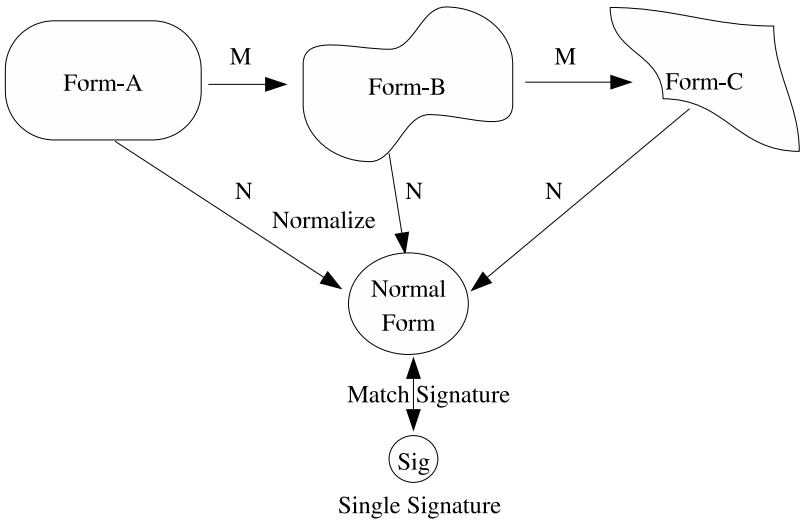

Figure 2: Detecting metamorphic variants using a normal form

The main contributions of this research are in:

1. formalizing the normalizer construction problem (NCP) using term rewriting;

2. proposing a strategy for solving this problem; and

3. demonstrating that certain relaxations of correctness conditions may still yield an effective normalizer while avoiding potentially costly or fallible static program analysis.

1.4 Organization of the thesis

Chapter 2 defines the NCP after reviewing related work and introducing the necessary term rewriting definitions. It lists the critical problems involved in creating a useful normalizer, including the problems of ensuring termination, confluence, and the correctness of the equivalence relations induced by the normalizing rule set. Chapter 3 defines a strategy for solving the NCP. First, a method for *reorienting* rules is described that creates a normalizing rule set such that termination is ensured. Second a strategy is described for making the normalizer confluent by applying a *completion procedure*. Chapter 4 discusses how to generalize the strategies from Chapter 3 in cases where one or more of the

correctness or confluence properties cannot be assured. The result is an approximated normalizer that may still yield sufficiently useful results for the purposes of signature matching. Chapter 5 reports on a case study using a normalizer for `W32.Evol` created using the proposed approach. It demonstrates the feasibility of the strategy and the potential usefulness of the approximated normalizer. Chapter 6 discusses the related work. Chapter 7 concludes the thesis and lists several important open research problems. The appendix provides the rules used by the mutation engine of `W32.Evol`.

2 The normalizer construction problem (NCP)

Metamorphism in malware was developed as a way to defeat the signature-based methods used by malware scanners. If a virus or worm creates an exact copy of itself each time it propagates, the task of recognizing it is clearly made easier than if it at least occasionally changes. The biological analogy is that genetic variation within individuals of a species (or between related species) makes identification harder. In the case of biological viruses, for example, a virus with a single strain can be fought with a single antibody, but if it mutates into multiple strains it forces the body's immune system to fabricate new antibodies for each strain. The virus with multiple strains is also able to infect widely even if individuals have partial resistance to certain strains.

Early viruses and worms tried to increase genetic variation within their populations by using such tricks as selecting from multiple behaviors and modifying the encryption scheme used to encode or "pack" the programs. These were examples of what are termed "polymorphic" worms and viruses [5]; the word literally translates as "many forms." Although the number of forms increased via polymorphism, they were closely related, since these worms and viruses did not perform complicated transformation of their code bodies. Malware scanners were presented some new hurdles but these were possible to overcome since, at the very least, the fact that the code bodies were not rewritten means that they were at some point there and ready to be recognized. Beginning in 1998, however, metamorphic malware appeared that would rewrite the main bodies of the code. This development had been expected. Cohen had observed that a program can reproduce either exactly or with "mutations" [7].

Some metamorphic worms of viruses could conceivably use metamorphism to evolve in unpredictable ways; indeed, research in the artificial life paradigm seeks to enable

evolution in artificial life forms. Nevertheless, to date, nearly all of the metamorphic worms and viruses are not designed to truly evolve. Rather, metamorphism is employed primarily as a means to increase genetic variety while maintaining a common identity for all the forms. Specifically, it is typically a goal for a metamorphic engine to ensure its mutations are programs that are functionally equivalent to the original. That is, the metamorphic engines strive to be *semantics-preserving*. When a metamorphic engine is added to any given malicious payload program it can generate a collection of equivalent programs, each of which have potentially different code bodies.

Even if the metamorphic engines preserve semantics, they can create enough mischief to pose difficult problems for malware scanners. The fact that all variants are semantically equivalent provides limited help since deciding program equivalence is known to be an undecidable problem in the general case. Moreover, Chess and White [8] showed that detecting that a program has the property than any one of its instances "potentially produces" any other instance is also undecidable. Spinellis [9] offered a proof that correctly detecting evolving bounded-length viruses is NP complete. While a complete and general solution may be too costly or simply impossible, it does not follow that the simpler problem of merely recognizing some malware variants is infeasible.

One possible approach is to define pattern-based signatures of malicious activity and then build tools that can find such signature activity in every (or at least most) implementations of them. An example of such an approach is given by Christodorescu *et al.* [10]. A second possible approach is to generalize the matching so that at least certain metamorphic transformations will not destroy the match. The work by Karim *et al.* [11] is in this vein. They define a similarity function between programs that matches code even when certain permutations have been performed on it. While this is a specific form of generalizing the match to allow for code mutations, it nonetheless shows that it may be possible to

account for specific transformations by suitable alteration of the match methods.

A third general approach is to reduce the number of variants that need to be considered by *normalizing* the programs. The remainder of this chapter defines this problem. A motivating example is first described in Section 2.1; the required term rewriting background is introduced in Section 2.2, and NCP is formalized in 2.3.

2.1 The normalization problem

Semantics-preserving program-to-program transformations can be used to normalize programs and thus reduce the variation in the input to the signature matching schemes. Effectively, this prunes the search space for a pattern matcher, simplifying the recognition problem. Such an approach was introduced by Lakhotia and Mohammed [12]. They defined a collection of semantics-preserving transformations called "zeroing" transformations. These would impose order and regularity on the input program. Unfortunately, while their system can reduce the number of variations there may still be an enormous number of them. A potential weakness in their approach is that the specific transformations of the metamorphic engine are not considered, so the transformations are generic and not tailored to normalizing a specific collection of related programs.

Instead of trying to define generic transformations, however, it might be feasible to define transformers specific to a metamorphic engine. Once a metamorphic engine is released it can be studied, and it may be possible to use the knowledge gained to revert all of its outputs to a known variant. More specifically, the goal would be to build a normalizer that could take any program variant constructed by the metamorphic engine and transform it in such a way that if two normal forms are equal it implies the programs are equivalent. With such a normalizer in hand a single signature could be developed for the normal form of a virus or worm. Suspect programs could then first be normalized and if the normal form is

matched to the signature we can be sure that the suspect program was equivalent.

While the scheme is *prima facie* sound, there are several potential hurdles to this approach, since simply "reversing" the transformations of the metamorphic engine is not a sufficient strategy. We illustrate this fact here using an example from the metamorphic worm `W32.Evol`, which contains a metamorphic engine that has dozens of ways of rewriting code. It operates by selecting rules to perform in a randomized fashion. Examples of `W32.Evol` rewrite rules are shown in Figure 3. The disassembly of the parent's Intel x86 code is shown in the left column and the corresponding transformed offspring code in the right column. In the figure, the parts of the code that are changed in the offspring are shown in bold face.

	Parent	**Offspring (transformed)**

(a)	`push eax`
	`mov [edi], 0x04`
	`jmp label`
		`push eax`
		`push ecx`
		`mov ecx, 0x04`
		`mov [edi], ecx`
		`pop ecx`
		`jmp label`

(b)	`push 0x04`
	`mov eax, 0x09`
	`jmp label`
		`mov eax, 0x04`
		`push eax`
		`mov eax, 0x09`
		`jmp label`

(c)	`mov eax, 0x04`
	`push eax`
	`jmp label`
		`mov eax, 0x04`
		`push eax`
		`mov eax, 0x09`
		`jmp label`

Figure 3: Examples of rewrite rules from `W32.Evol`

In example (a), the metamorphic engine has replaced an immediate `mov` into a mov from a register. This transformation does not change the semantics of the code: both versions do the same thing, only in slightly different ways. In the transformed code, the register `ecx` needed to be disturbed in order to change the `mov` immediate instruction. If `ecx`

is known to be *dead* (its value is not needed later), at that point the transformation would not change how the program works. Any existing value of `ecx`, however, is preserved by the `push` and subsequent `pop` immediately surrounding the two middle `mov` instructions. `W32.Evol` adds these push/pop blocks each time it makes such a transformation because it does not first determine whether `ecx` is live or dead at any point. If it failed to add these blocks, the result could be semantically different and is likely to not work correctly.

In example (b), the `push` immediate instruction has been changed to a `mov` immediate into a temporary register followed by a `push` from that register. Like the transformation in (a), this transformation is semantics preserving. However, its correctness relies on the fact that the register `eax` is being overwritten before its *use* following the `push` in the original program and is therefore not live at that point.

In example (c), a "junk" statement (i.e., one with no effect on the computation) `mov eax, 0x09` is inserted. It is not possible to discern that this is a junk statement without knowing a specific property of the `W32.Evol` program as a whole: that immediately after a `push eax`, the program guarantees that the `eax` register is dead and can be altered in any way without affecting the running of the program.

The transformation examples in Figure 3 help introduce several potential problems in developing a normalizer. There is, at minimum, a technical hurdle of specifying an appropriate transformation system for this type of normalization problem. `W32.Evol`'s metamorphic engine can perform the transformation in (a) no matter the value of the immediate constant, so the normalization system must be able to express variant substitutions such as this. In addition, the transformations in (b) and (c) can be performed only when certain conditions can be guaranteed to hold. The transformation system must be able to formalize this condition-specific transformation.

Apart from the mechanics, though, there are potentially deeper problems as well.

Consider, for instance, the `mov eax, 0x04 ; push eax ; mov eax, 0x09` sequence. The transformations in both (b) and (c) can produce the sequence. If one were to choose to revert to the code in (b) instead of (c) (or vice versa), will this decision affect the results? Can we guarantee only a single normal form will be produced for any variant? Is it possible that transforming a non-malicious program would yield a false match? Without appropriate understanding of how the transformation systems work, it will be impossible to answer these questions or to build a correct normalizer. The following subsections provide an overview of some necessary background from the term rewriting literature and show that we can formalize the normalizing transformer construction problem.

2.2 Term rewriting background

Term rewriting systems are widely studied and good references exist (e.g., Baader and Nipkow [6]); this section only briefly reviews definitions and results needed for later sections.

Term rewriting system. A term rewriting system, T, consists of a set of *rewrite rules*. A rule is denoted $s \to t$, where s and t are terms (described below). Figure 4 shows a simple example of a term rewriting system.

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

Reduction relation ($\to_T$). Any term rewriting system T *induces a relation* $\to_T$ on terms,

also represented as $\rightarrow$ where obvious. Given terms s and t, $\rightarrow_T$ is defined as follows: $s \rightarrow_T t$ holds if and only if for some rewrite rule $s' \rightarrow t'$, s has, as a subterm, an instance of s' that, if replaced with its corresponding instance of t', turns s into t; that is, applying rule $s' \rightarrow t'$ to s transforms it into t. A conditional term rewriting system may have conditions attached to the rules. Conditions are normally expressed as predicates and written as $p|R$, where R is a rule. This means that rule R may be applied only when the condition p holds. Clearly an unconditional term rewriting system is simply the special case of a conditional term rewriting system with each predicate being true.

Equivalence relation ($\overset{\star}{\longleftrightarrow}$). The *equivalence relation*, $\overset{\star}{\longleftrightarrow}$, is the reflexive symmetric transitive closure of the relation $\rightarrow$ induced by T. Because $\overset{\star}{\longleftrightarrow}$ is an equivalence relation, it partitions the set of terms into equivalence classes. Given a T, we use the notation $[x]_T$ to refer to the equivalence class of a term x, as defined by $\overset{\star}{\longleftrightarrow}$.

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 $[x]_T$ in normal form. The term $add(2, 2)$ is in normal form with respect to the example rewriting system shown in Figure 4, and $add(1, add(1, 1))$ is related to $add(1, 2)$ under the relation induced by this system (by application of the second rule).

Termination. T is terminating if there exists no infinite descending chain of the form $a \rightarrow b \rightarrow c \cdots$. Determining termination is, in general, undecidable.

Confluence. Let w, x, y *and* z denote arbitrary terms. Suppose there is a sequence of

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

Figure 4: Example rule set transforming arithmetic expressions

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. Determining confluence is, in general, undecidable.

Convergence. A term rewriting system is *convergent* if it is confluent and terminating. If a term rewriting system is convergent, then membership in an equivalence class becomes decidable and each equivalence class has a unique normal form.

2.3 NCP as a term rewriting problem

Using the term rewriting theory of Section 2.2 we can formally restate the normalization problem introduced in Section 2.1. This is done by modeling the metamorphic engine as a term rewriting system and then posing the NCP problem as one of constructing an equivalent term rewriting system with three specific properties.

2.3.1 Modeling the metamorphic engine

It may be possible to formalize a metamorphic engine as a term rewriting system by considering assembly statements as terms, treating operations as functions, and considering its operands either constants (which must match exactly) or variables (which allow patterend matches). For example, consider the `mov ecx, 0x04` statement from Figure 3. When writing this program fragment in a rewrite rule it can be modeled using the term $mov(reg1, \texttt{0x04})$. In this case mov is the function, $reg1$ is a variable, and `0x04` is a constant. To match this rule, the specific constant would need to be matched. Figure 5 depicts an example of how rules might typically be written in a term rewriting formalism.

In modeling the metamorphic engine, we make the explicit assumption that the

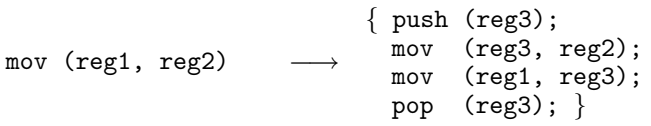

Figure 5: Sample metamorphic transformation pattern as a rewrite rule

metamorphic engine must preserve semantics and, furthermore, each rule preserves semantics. The rule in Figure 5 has no condition attached to it, meaning its conditions for firing are always true, and the left hand side must be equivalent to the right hand side at any time. This is true for the potential rules in Figure 3(a), modulo issues that are considered unimportant such as code size and location. Other rewrite rules need to be conditioned to be able to encode such transformations as the ones shown in Figure 3(b) and Figure 3(c). In particular, the conditions attached to the rules need to ensure that the rules cannot fire if by doing so they would not preserve semantics.

Using a scheme like this, we were able to formally model the metamorphic engine in `W32.Evol`. One of the rewrite rules is depicted in Figure 6. This rule corresponds to the transformation shown in Figure 3(a). For simplicity we will write such rules in the assembly language form rather than the term rewriting form of Figure 5. In the figure, *IMMED* is a variable, which can match a constant. In particular, it can match the `0x04` constant from Figure 3(a).

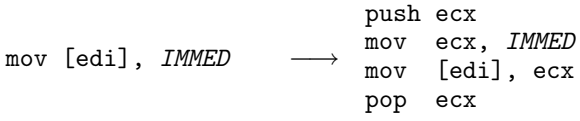

Figure 6: Metamorphic transformation as a rewrite rule

2.3.2 The normalizer construction problem (NCP)

Suppose one has formalized a metamorphic engine as a term rewriting system, as described above. Call this term rewriting system M. M induces an equivalence relation and so partitions programs into equivalence classes $[x]_M$. If M happens to be convergent then the problem of determining equivalence for variants of a metamorphic program is, in principle, solvable. Given a malicious sample s and a sample program p one can determine whether p is a variant of s, i.e., whether $p \in [s]_M$. To do this a term rewriting system can apply rewrite rules to p and s until they match. If M is convergent it is confluent, and if it is confluent then p and s will be joinable if they are equivalent and not joinable if not. This malware recognizer could therefore produce neither false positives (no non-equivalent programs would join s) nor false negatives (all equivalent p would eventually join).

In practice, however, it is unlikely that M is convergent; otherwise the metamorphic engine will only serve to progress the malware towards a single form, thereby defeating the goal of metamorphism. Moreover, even if M were convergent, the process of rewriting p and s until they join is not a feasible malware detection scheme not least because it requires the distribution of the original malicious sample s. However, M can be useful if it can be modified to create a new term rewriting system which is convergent.

The problem of doing this is what we call NCP: the normalizer construction problem. It involves constructing a convergent rule set N from M such that the the equivalence classes of M are equal to the equivalence classes of N. Using the definitions of convergent, this means the following properties must hold:

Equivalence. For all programs x, $[x]_M = [x]_N$. If M and N are not equivalent wrt x then one of the following conditions would hold: (a) $\exists k.k \in [x]_M \wedge k \notin [x]_N$ or (b) $\exists k.k \notin [x]_M \wedge k \in [x]_N$. The first condition leads to false negatives and the second one

to false positives.

Termination. Clearly a successful normalizer must halt, which means that N must be guaranteed to have no rules that could cause an infinite chain of application. The rule in Figure 5, for example, would do so since the right hand side contains a match to the left hand side.

Confluence. If N is confluent then the order of application is not important.

If all of these properties are met, a correct malware recognition system might be built on top of signature matching techniques. For any given program sample p, the rules of N can be applied until the result is irreducible. If N is terminating, the applications are guaranteed to stop. If N is both terminating and confluent, the same normal form will be produced for any two programs p and q that are equivalent under $[q]_N$, and different normal forms will be produced otherwise. If $[x]_M = [x]_N$ for all x, then no false positives or false negative matches would occur. With such an N, one applies it to a known virus or worm sample s to create its normal form $Norm_N(s)$ and then builds a signature for it. Then for any suspicious program p one creates $Norm_N(p)$ and checks for a match to a signature.

3 A strategy for solving NCP

Assuming one is given the term rewriting model of the metamorphic engine M, the challenge of NCP is to construct a new rule set N such that the equivalence classes induced by N and M are equal and N is convergent. Here we give a strategy for constructing such an N. The strategy involves the application of two procedures to M: a *reorienting procedure* which seeks to ensure it is terminating, and then a *completion procedure* [13] that seeks to ensure it is confluent. The completion procedure itself may not terminate. However, should it terminate it will yield a solution to the NCP.

3.1 Reorienting procedure: termination

The first step is to apply a reorienting procedure to M to create a terminating rule set M^t. Reorienting means changing the direction of a transformation. For example, $x \to y$ is reoriented by turning it into $y \to x$. Since $x \to y$ means that x is considered equivalent to y, reorienting the rule will not change the equivalence classes induced by $\to$. Reorienting any rule in M to construct an M' will therefore never result in a case where $[x]_M \neq [x]_{M'}$.

To solve the NCP, not just any reorienting procedure will do. The procedure needs to ensure that the reoriented M will always terminate on any given input. The naive strategy for constructing M^t is to reorient every rule from M. One can think of this as constructing the "undo" rule set: for every rule in M reversing the directions should "undo" each metamorphic change. This procedure, however, does not guarantee termination. For example, if $M = \{a \to b, b \to a\}$, then reorienting both rules will yield a rule set that is still non-terminating (a can be rewritten to b and vice versa forever). In order to ensure the result is terminating, the reorientation procedure must be based on a *well-founded reduction ordering* $>$ on the terms. Given such a $>$, M' can be constructed such that

$\forall a \rightarrow b \in M'.a > b$. The construction is simple and linear: go through the set and reorient any rule $a \rightarrow b$ if $b > a$. Since each rule in this M' would serve to reduce the terms (according to ordering $>$) and since $>$ is well founded, then M' is guaranteed to always terminate [6].

Really, any well-founded ordering would do. For the term language we have used in our prototype we can define a well-founded $>$ using a length-reducing lexicographic ordering, as follows. Let $len(x)$ be length of term x, where $len(x)$ is the number of atomic subterms it contains. Then leave $a \rightarrow b$ if $a > b$, otherwise reorient it. Thus for rules of unequal length we simply reorient the rule if the term on the right is longer than the term on the left. If both sides of a rule are of equal length, we consider the particular instance where the rule matches a string, which would of course be all ground terms, and reorient towards the lexicographically first pair. This definition ensures that the $>$ is a well-founded reduction order for our rule sets.

Figure 1 shows an example of reorienting a set of rules. Figure 1(a) shows the initial rules for M and 1(b) shows the M^t that results from reorienting M according to the length-reducing lexicographically first ordering. Note that the rules are conditional rewrite rules. The post condition column C_i specifies the condition that must be true at the end of corresponding l_i, for all conditional rewrite rules. The C_i for an unconditional rule is always `true` denoted by a T in the figure. As an example of reorientation, consider row M_1. The length of l_1 is 1 and that of r_1 is 4, so it must be reoriented.

3.2 Completion procedure: confluence

Given the terminating M^t, testing if it is confluent is decidable [6]. If the confluence test is successful, it would mean that the system is convergent. From the previous step we already know the equivalence constraint is satisfied, which means that both the constraints specified in the previous section are satisfied and such an M^t will solve the NCP and the

Table 1: Example application of the rule reorienting procedure

Rule M_i	**Post Condition** C_i	l_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 M_i^t	**Post Condition** C_i	l_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

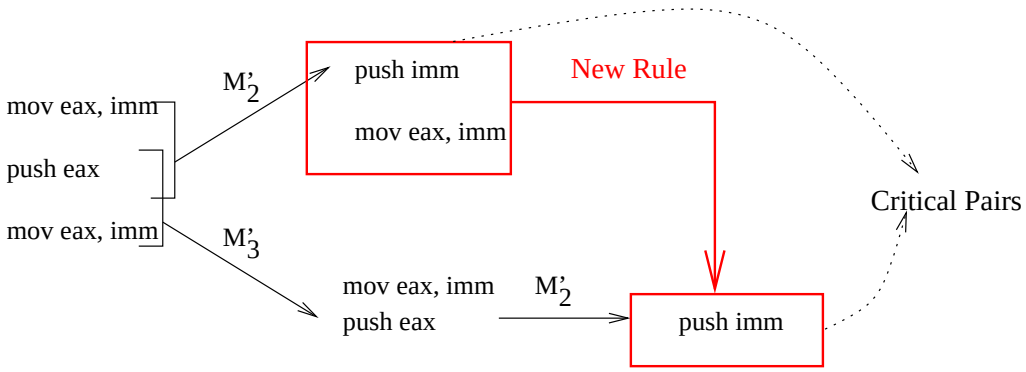

Figure 7: Completion step for M_2^t and M_3^t

output will be the desired N. If the confluence test fails then M^t would contain what are called *critical overlaps* [6, 13] in the rules. The left hand sides of a pair of (not necessarily distinct) rules are said to critically overlap if the prefix of one matches the suffix of the other or if one is a subterm of the other.

The M^t shown in Figure 1(b) has two critical overlaps: (1) M_1^t and M_2^t overlap at `push eax`, and (2) M_2^t and M_3^t overlap at `push eax`. These critical overlaps indicate conflicts in the rule set that may make the set non-confluent. In this example, a case can occur where either of M_2^t or M_3^t might be applied, and the resulting irreducible forms may not be equivalent. Note that while M_1^t and M_3^t may seem at first to have a critical overlap at `push eax ; mov eax,` *`Imm`*, it is not critical if we take into account the corresponding post conditions. In particular, M_3^t requires the `eax` register to be dead, yet for M_1^t, after the `mov eax,` *`Imm`* is performed it must be the case that `eax` is never dead, and hence there is no possibility for the potential overlapping rules to be applicable simultaneously.

In cases where M^t is terminating but non-confluent, a completion procedure can be applied to try to make it confluent. A completion procedure attempts to add rules to a rule set such that all members of the same equivalence class are joinable. It is an undecidable problem, in general, to ensure convergence of a rule set. However, it is possible to test for

Table 2: Rule set N, the result of the completion procedure

Rule N_i	**Post Condition** C_i	l_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 Imm1` `mov eax, Imm2`	$\rightarrow$	`push Imm1`

N_6	T	`push Imm1`
		`mov eax, Imm2`
		`mov [Reg1+eax], Reg2`
		`pop eax`
			$\rightarrow$	`mov eax, Imm1`
				`mov [Reg1+Imm2], Reg2`

confluence on a terminating set, so it is possible to apply a completion procedure and simply test to see whether it worked. This is the strategy suggested here.

In our case we applied the completion procedure of Knuth and Bendix [13]. This procedure successfully completed the full M^t for `W32.Evol`. The procedure searches for critical overlaps and then adds rules that connect the two potentially distinct and irreducible forms. Figure 7 illustrates the process of completing for the critical overlap between M_2^t and M_3^t. The new rule, as shown, connects the two irreducible terms `push Imm; mov eax, Imm` and `push Imm`. Repeatedly applying this procedure to all critical overlaps from Table 1 terminated, giving us the desired N as appears in Table 2.

4 Approximated solutions to NCP

When the strategy described in Section 3 works, the resulting normalizer is guaranteed to be convergent. This was shown to be, in many respects, an ideal situation: not only would there be only one normal form to build signatures for, but N would also come with a guarantee that each equivalence class has a distinct normal form, eliminating the potential for false matches. Nevertheless, in some cases it may not be feasible or desirable to construct or correctly execute a convergent normalizer.

Two cases are considered here: (1) when the completion procedure fails to terminate with a confluent rule set and (2) when one uses an implementation of the normalizer that does not calculate the conditions correctly for the conditional rules. We call these "approximated" solutions to NCP because they fail to meet at least one of the requirements for an exact solution. The advantages and disadvantages of these approximations are analyzed, and a strategy is outlined for dealing with the approximation at the term rewriting system implementation level by implementing a priority scheme.

4.1 Completion procedure failure and alternate completion methods

Section 3 noted that the completion procedure is not guaranteed to terminate with a confluent rule set. Even though it completed successfully in the case of `W32.Evol`'s metamorphic engine, one may still encounter a case where an alternative approach is required.

One possibility is to simply use the (possibly partially completed) M^t as the normalizer without a guarantee of confluence. One cost of doing so is that the equivalence classes induced by M^t may have multiple irreducible normal forms; that is, there may exist

an x such that $|Norm_{M^t}(x)| > 1$. Whether this fact poses a serious problem for application in a malware scanner may depend upon the normalizer or the particular metamorphic program itself. For instance, it might happen that all of the irreducible forms in $Norm_{M^t}(s)$ for some malware sample s are highly similar. The similarity between the normal forms may allow all of them to be matched using a small number of signatures—possibly a single signature. Hence while having a confluent normalizer is a laudable goal, it may not always be necessary to achieve it. The case study in Section 5 provides some support for this possibility.

A second approach is to apply an *ad hoc* completion. An analyst would have to examine M^t and add rules to create N' that the analyst believes will produce a single normal form for each equivalence class. There is a risk that the analyst will be unable to complete the rule set and, because of this, fail to guarantee confluence. In this case the risk may be the same as not completing it at all. There is also a risk that the analyst will mistakenly add a rule that compromises the equivalence constraint leading to the creation of false positives or false negatives. The consequences of this risk are described in more detail below.

4.2 Normalizer that calculates conditions incorrectly

Modeling `W32.Evol`'s metamorphic engine required the use of a conditional term rewriting formalism. In `W32.Evol`'s case the conditions attached to the rules were all either `true` or register liveness conditions. Register liveness is an analysis that can typically be performed statically, although in the general case it requires complicated analyses, including control flow and points-to information [14]. In the general case, these costs are likely to be exorbitant for an ordinary malware scanner to perform. Perhaps worse, it might not be feasible to calculate the required condition information correctly using known static analysis techniques.

In the case of register liveness, for example, accurate control flow information may be needed, yet this is not always possible to extract statically. That said, Lakhotia and Singh [15] note that the metamorphic engine must be able to perform the necessary analysis if it is to behave correctly on itself as input. This constraint may serve to limit the conditions that need to be checked in the normalizer.

`W32.Evol`'s metamorphic engine does not, in fact, perform any liveness analysis even though its engine correctly knows about the liveness conditions. It avoids having to worry about costly or complicated liveness analysis by making use of *hidden properties* of the malicious payload program it is attached to. For example, M_3 of Figure 1 has a condition attached to it that stipulates `eax` must be dead at the end of the `push` or the rule cannot be applied. In fact, `W32.Evol` makes assumption that allow it to ignore the condition. Specifically, it assumes that `eax` must be dead after such a push, and that nowhere in the body of `W32.Evol` is the value in `eax` needed after it is pushed and before it is set again. It also makes use of the fact that none of its other rewrite rules invalidate this very assumption. For `W32.Evol` this is a reasonable tactic since it is not expected to apply the rule set to any arbitrary program, only the malicious payload which has been carefully crafted to ensure these assumptions hold. A normalizer, on the other hand, is expected to be run on arbitrary programs, so it must calculate the conditions correctly. And we already noted that this is not always feasible or possible to do.

For the above reasons, one may wish to develop a normalizer N' for M that is not guaranteed to calculate the conditions correctly. The effective result is that the induced equivalence relationship no longer correctly reflects the true program equivalences, i.e., $[x]_M \neq [x]_{N'}$. There are several important cases.

In the case where no conditions are checked, $[x]_M \subseteq [x]_{N'}$. To see this, note that for any term a, b, if a reduces to b under M, it must also reduce to b under N' since every rule

executable under M is also executable under N' since the conditions are not checked. Thus $[x]_M$ is at least contained in $[x]_{N'}$. This guarantees that no false negatives will occur: the set of programs considered equivalent for a given malicious program s under M will also be considered equivalent under N'. $[x]_{N'}$ may also contain programs not equivalent under $[x]_M$. To see this, note that there may exist a rule $r = p|a \rightarrow b$ in M that cannot execute in some condition because p does not hold. If p is not checked by N' then r can be executed when the condition does not hold. This can add a new term to $[x]_{N'}$ and so $[x]_{N'} \geq [x]_M$. As a result, N' may lead to false positives since programs not equivalent according to M are considered equivalent according to N'.

In the case where conditions are checked erroneously, we can only say that $[x]_M \neq [x]_{N'}$. The case where conditions are not checked at all is merely a special case where the condition is considered `true` all the time. If, however, a condition check is erroneous and reports `false` when it should report `true`, then there might exist conditions where certain rules executable in M that are not executable in N'. In such cases, there may be programs equivalent under M that are not equivalent under N. Thus a normalizer that cannot check conditions correctly can yield both false negatives as well as false positives.

4.3 Priority scheme for implementing approximated solution

It was noted above that if conditions are not checked then false positives may result. We have experimented with a simple priority scheme for rule application that is designed to reduce the likelihood of producing false results. This priority scheme was used in the prototype in the case study described in the following section.

The priority scheme is constructed as follows. The initial set of rules in N' is first partitioned into two subsystems such the one has all unconditional rules while the other has conditional ones. Call these rule subsets N'_U and N'_C, respectively. Consider the rule set in

Figure 2. Then $N'_U = \{N_1, N_4, N_6\}$ and $N'_C = \{N_2, N_3, N_5\}$. The normalization process proceeds by applying rules from N'_U until the result is irreducible. Then N'_C is checked for a rule that can apply. If one (any one) can be applied, it is and the procedure loops back into applying all N'_U until the result is again irreducible. The process loops in this fashion until no more rules from either set can be applied.

This system is equivalent to a scheme with a priority-sensitive application order such that all the rules in N'_U have higher priorities than any rule in N'_C (any rule in N'_U that is still applicable will be fired before any rule in N'_C). There is a simple intuitive justification to this simple priority scheme: we know the rules in N'_U preserve semantics, whereas application of any rule in N'_C may not. Keeping the unsafe rules at a lower priority means that every safe applications will be tried before an unsafe one. As a result, some improper rule applications may be avoided because a higher priority rule will block its application. This should reduce the number of programs incorrectly considered equivalent to the malicious program.

5 Case study

A case study was performed to evaluate the effectiveness of the two approximated solutions described in Chapter 4. Specifically, we sought to evaluate how well the methods worked in the case where critical overlaps prevented the system from generating a unique normal form, and how well the priority scheme worked when applying the rules without checking conditions. The study sought to quantitatively measure the number of distinct normal forms created due to the approximations and to qualitatively assess the similarity of the resulting normal forms.

5.1 Subject and preparation

`W32.Evol` was selected as a test subject. It is an experimental Windows virus which appeared in 2000. We obtained a copy of the 12288-byte long first generation sample from the "VX Heavens" archive [16]; henceforth we will refer to this sample as the "Eve" sample. Its main body of code consists of 2182 assembler lines.

`W32.Evol` was considered to be a suitable subject for our study. First, it is not a serious threat to handle in our secure environment. Second, it creates sufficient variation in its mutated offspring that static signature based techniques fail: at the time of this writing we believe it is being matched by emulation [2]. It employs a metamorphic engine which generates mutations by instruction substitution. Some of these substitutions introduce arithmetic on immediate constants; these mutation rules, alone, can yield 2^{32} different varieties at each possible mutation site. Even ignoring all variations in constants and registers, by examining its possible mutation sites and its mutation methods we conservatively estimate that it could reasonably generate on the order of 10^{686}, 10^{1339}, and 10^{1891} variations in its second, third, and fourth generations, respectively. It is a

Table 3: Test set used for case study

Generation	Eve	2	3	4	5	6
Sample count	1	9	7	4	4	1
Average size (LOC)	2182	3257	4524	5788	6974	8455

representative example of a virus with a sophisticated mutating engine. Third, as we noted in the examples from previous sections, `W32.Evol`'s metamorhpic engine includes transformations that are sensitive to semantic conditions and contains rules with critical overlaps. This provides a good test for the sufficiency of the proposed NCP solution strategies.

One drawback of using `W32.Evol` is that it can be difficult to prepare enough mutated samples. As noted by Symantec [17], the mutated forms are tough to obtain because the virus is buggy and is highly selective in its mutation. Manipulating it in a debugger in a secure environment allowed us to bypass its various bugs and checks (such as an entry point calculation bug), and in that way we were able to coax it to generate 72 different variants, spread into six different generations. We created an ad hoc sample of 26 of the 100 variants from multiple families across the six generations as follows. Each of the samples was prepared by disassembling the main code using the `objdump`[1] disassembler. The average length of each of the sampled generations is shown in the above table in terms of number of non-blank lines of assembly. Our prototype required the immediate-mode constants to be represented in decimal format, which we achieved by altering `objdump` appropriately.

5.2 Materials and protocol

The procedure we used is as follows:

[1]GNU Binutils, version 2.16.1, see `gnu.org`.

1) construct a set of term-rewriting rules M that represent `W32.Evol`'s mutating operations;

2) construct the normalizing set N using the methods described in Section 3, but without completing it;

3) implement the N in a prioritized term-rewriting system that does not check conditions, as described in Section 4.3;

4) complete N in an ad-hoc manner and implement in a second prototype using the priority scheme and not checking conditions; and

5) feed each of the samples to the two prototypes to generate the normal forms, and collect output and timing information.

Our materials and more procedural details are described below.

Construct M: Extraction of the mutating rules was done by hand using a combination of code reading and code tracing in a debugger. This yielded **55** rules of which only **1** rule was length reducing. The normalizing set consisted of **55** rules of which **15** did not participate in any overlap; the rest had **84** overlaps among each other. There were only three unique overlaps; for example, a lot of the **84** overlaps were over push eax, i.e., involving push eax suffix of one rule and prefix of another, and were counted as one unique overlap.

Construct N: The normalizing set was constructed as described in Sections 3 and 4. Specifically, M^t was constructed from M using the reorienting procedure. The rule set was partitioned into conditional and unconditional subsets as described in Section 4.3. Next M^t was completed in an ad-hoc manner to form N_C. All of the overlaps in N^t were inspected to determine how to manually complete the rule set. It was determined that adding the two rules shown in Figure 8 would complete N and generate a single normal form for the species

members. The first rule reduces sequences of immediate moves to register `eax` to the final move. It is normally semantics preserving in any context since the moves removed have no effect. The second rule removes a move and is not semantics preserving in general; i.e., it is conditional. As a result, while the normal form for the species is expected to be unique, if such patterns exist in the Eve sample, they will be missing in the normal form as is noted in Section 4.2.

[1]
```
mov  eax, immed1
mov  eax, immed2   ⟶  mov eax, immed2
```

[2]
```
push immed
mov  eax, op       ⟶  push immed
```

Figure 8: Two rules needed to complete the normalizing set for `W32.Evol`.

Implement normalizers: A prototype transformer was implemented using the `TXL` programming language [18].[2] Translating most of the rules of N into TXL was straightforward, as they either contained ground terms or their non-ground terms could be turned into non-terminals in `TXL`. Several of the rules in N required arithmetic processing. These were successfully translated using `TXL`'s conditional term rewriting facilities. For example, an arithmetic substitution expression of

`mov eax, x` $\longrightarrow$ `mov eax,x-`y `; add eax,` y

could be translated into a TXL rule similar to

```
mov eax, X  ⟶ mov eax, Y ; add eax, Z where X=Y+Z
```

The transformer for the completed rule set N_C was implemented with minor changes to the first prototype.

[2] `TXL` system version 10.4 (8.1.05).

Table 4: Results using prioritized, non-completed normalizer

1	**Generation**	Eve	2	3	4	5	6
2	**Average size of original (LOC)**	2182	3257	4524	5788	6974	8455
3	**Maximum sizes of normal form (LOC)**	2167	2167	2184	2189	2195	2204
4	**Average size of normal form (LOC)**	2167	2167	2177	2183	2191	2204
5	**Lines not in common**	0	0	10	16	24	37
6	**Percentage in common**	100.00	100.00	99.54	99.27	98.90	98.32
7	**Execution time (CPU seconds)**	2.469	3.034	4.264	6.327	7.966	11.219
8	**Transformation counts**	16	533	980	1472	1902	2481

5.3 Results

Table 4 quantifies the results from the uncompleted normalization prototype. Row four of Table 4 indicates the average length of the normal forms of the variants after normalizing with the prioritized scheme. Note that the normal form of Eve is smaller than the original Eve. This is due to two factors. First, much of the reduction in size is due to transformations in N that happen to be length reducing. These are unconditionally semantics-preserving, which means that the equivalence class is preserved. A simple example is `nop` removal. Second, there are conditional length reducing rules applied and we are not performing any analysis to check for conditions. By manual inspection, we found these to occur at 2 sites out of the 2182 original lines of code. Row five simply lists how many of these extra lines, on average, are left. It was calculated using the `diff` program and counting the number of lines that differed from the normal form of Eve. Row six shows the average raw percentage of sequence commonality between the normal form of Eve and the normal form of the sample variant.

Rows seven and eight of Table 4 record execution information for the prototype. Row 7 of the table lists timing statistics collected by averaging the times for 10 runs for a single

sample from each generation. Row 8 of the table records the average number of transformation rules that were executed in normalizing the sample.

The second prototype with the completed normalizing rule set N_C created a single normal form of 2166 lines for all 26 samples.

5.4 Discussion

The results suggest that for a realistic metamorphic virus it is possible to effectively normalize the variations sufficiently well that simple signature matching can be made effective. The discussion that follows centers on the sufficiency of the two prototypes, the tradeoffs illustrated by the two normalization choices, and our own qualitative assessments of the difficulty of constructing the normalizing rule sets.

The main question about the normalization scheme is whether the approximation methods would be sufficient in normalizing realistic metamorphic programs containing difficult overlapping and conditional rules. Table 4 shows that the prioritized scheme creates similar normal forms for all of the samples. To us, it seems likely that relatively simple signature matching would be sufficient to recognize the normalized `W32.Evol`s.

The results permit evaluation in terms of added and removed information. The prioritizing prototype leaves extra lines of code in the normal form. These were not removed in normalization in order to avoid removing meaningful lines of code. They are all junk and a def-use analysis should be enough to detect this and remove them. As seen in row six of Table 4 the spurious lines in the normal form are few in relation to the overall size. The unique normal form for the complete rule set is missing many lines and is not semantically equivalent to `W32.Evol` because context-sensitive transformations were applied in inappropriate contexts. For these, too, the overall difference from the original form is very low in the end.

Regarding the normalization choices, the differences in normal form length illustrate the tradeoff between the prioritization and rule completion strategies. The normal form of the Eve of the prioritized version is 2167 lines long, while the length of the normal form for the complete rules, N_C, is 2166. The difference indicates the number of locations in which meaningful lines of code from Eve were removed due to the completion rules of N_C. While this has the potential to create false positives, it seems unlikely to us that any benign program would yield an exact sequence match to this normal form of `W32.Evol`.

Our subjective experience was that once M was constructed, the strategy from Section 4 was relatively easy to perform. The key was understanding the need to focus on the critically overlapping and conditional rules. The manual completion rules were generated over the course of two days after discussing the possibilities and briefly experimenting with different options.

One might be tempted to find fault with the fact that the normalization technique depends upon having a formalization of the metamorphic engine to begin with. This means the technique cannot be expected to find malicious programs for which the metamorphic engine is unknown. This may not be be a significant problem. Already, signature matching cannot detect novel malware either, but it has proved to be a suitable technology when signature database are able to be updated frequently. One could also argue that the construction of the model of the metamorphic engine can be difficult and costly. This also is likely to be cost that is either acceptable or unavoidable. First, we note that metamorphic engines evolve slowly—much slower than the worms and viruses evolve—since they are very difficult to write correctly and few malicious programmers worldwide have done so [5]. Once an initial model is constructed, any changes can be tracked more easily. Second, the cost of constructing M is most of the cost of constructing N; we cannot imagine how the normalization rules could be constructed more cheaply than by studying the metamorphic

engine first to learn its secrets.

It is not possible to generalize from a single case study with any confidence. Nonetheless 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.

6 Relations to other work

Ször and Ferrie [2] give a list of both current and potential metamorphic transformations. They describe emulation based detection techniques used by industrial anti-virus scanners. Stepan improves upon dynamic emulation by dynamically retargeting a binary to the host CPU [19]. Recent work by Ször [5] gives a more exhaustive list of metamorphic transformations and suggests that most current technologies would not be able to handle the threat posed by metamorphic viruses. Perriot [20] and Bruschi *et al.* [21] suggest using code optimization to normalize metamorphic variations, similar to Lakhotia and Mohammed [12], and since they do not model the rules of metamorphic engine, they are similarly limited.

The normalizers we construct are able to determine whether a program belongs to the equivalence class of a semantically equivalent program (according to the rewrite rules of the metamorphic virus). Our work therefore relates well to other efforts that try to find malicious behaviors in arbitrary code by looking for semantically- or behaviorally-equivalent fragments. Section 2 already noted that the work by Christodorescu *et al.* [10] is related in this way. The work by Kruegel *et al.* [22] is similarly related in that they try to find mutated variants of identical behavior in polymorphic worms. Their approach differs in that they use structural information and perform matching based on signatures comprised of control flow graphs. Unlike the techniques in this thesis, their technique has the potential to find matches in programs not "genetically" related by transformation by a particular metamorphic engine. However, their techniques have little hope of finding equivalent forms with different control flow graphs, whereas the normalization methods explored here have that potential. Sung *et al.* [23] present a static signature based malware detection scheme; they use the sequence of Windows API calls to form a signature and then use similarity measures to match the

signature of a given executable against that of known malicious executables from an existing signature database. They have shown promising results in detecting variants of malware such mass mailing worms, but their approach is not suitable for metamorphic variant detection.

7 Conclusions and future work

This thesis defined the NCP and defined strategies that can be employed to construct normalizing transformers when given a formal description of a metamorphic engine. These techniques take a step forward towards being able to counteract the threat metamorphic engines pose to malware scanning. Traditional signature matching methods do not have a suitable solution to counter the metamorphic engines, and without a significant advance in program matching they cannot be expected to have a suitable solution. Normalization, however, has the potential to counter a metamorphic engine by normalizing all possible variants into a single normal form, from which standard signature matching can be, once again, effective.

The normalization strategy was shown to be fallible in the sense that the strategy does not invariably lead to a convergent normalizer. Yet the case study also provided a demonstration that in certain cases this may not matter, since multiple normal forms may not present a problem as long as the normal forms associated with the malicious program are similar enough.

As future work, we would like to investigate further the implications of condition evaluation while applying conditional rules; such a system might not require multiple priority stages. In particular it will be interesting to compare the run-times for normalizers that perform conditional analysis with multiple stage normalizers that do not check for conditions. A possible approach could be to perform conditional analysis only once and then leave it up to the rules in rewrite system to carry forward the results of conditional analysis as the code is transformed. A metamorphic rule cannot replace random instructions by another set of random instructions; this limits the number of ways in which rules can be formed. Work can be done in this direction to investigate the implications of this constraint

on possible overlaps and completion procedure for metamorphic assembly transforms modeled as rewrite rules.

In conclusion, the term rewriting approach to constructing normalizers has the potential to counter the threat posed by metamorphic malware and make tried-and-true signature matching techniques applicable once again. Our case study demonstrates that the strategy being proposed can yield effective normalizers (it has the potential to be completely automated). The case study suggests that even approximated normalizers might yield sufficiently good results, meaning they have some potential to be turned into efficient implementations in real malware scanners.

Appendix

Table 5, Table 6 and Table 7 list the rules of the metamorphic engine of `W32.Evol`; the corresponding rules of the normalizer can be easily obtained by reorienting these rules as described in Chapter 3.

Table 5: Rule set used by the mutation engine of `W32.Evol`

P.C. C_i	l_i	$\rightarrow$	r_i
T	`A` *Reg1*`|[`*Reg1+Offset*`],` *Val*`|`*Reg3*	$\rightarrow$	`push` *Reg2* `mov` *Reg2*`,` *Val*`|`*Reg3* `A` *Reg1*`|[`*Reg1+Offset*`],` *Val*`|`*Reg3* `pop` *Reg2*
T	`stos(b|d)`	$\rightarrow$	`mov [edi], al|eax` `add edi, 1|4`
T	`lods(b|d)`	$\rightarrow$	`mov al|eax, [esi]` `add esi, 1|4`

T	`movs(b|d)`	$\rightarrow$	`push eax`
			`mov al|eax, [esi]`
			`add esi, 1|4`
			`mov [edi], al|eax`
			`add edi, 1|4`
			`pop eax`

T	`mov [`*Reg1+|-Offset*`],` *Reg2*	$\rightarrow$	`push` *Reg3* `mov` *Reg3*`,` *Reg1* `add|sub` *Reg3*`,` *Val1* `mov|cmp [`*Reg3*`|`*Val2*`],` *Reg2* `pop` *Reg3*
T	`mov [`*Reg1+Offset*`],` *Reg2*	$\rightarrow$	`push` *Reg3* `mov` *Reg3*`,` *Reg2* `mov` *Reg1+|-Offset*`],` *Val|Reg3* `pop` *Reg3*
T	`mov [`*Reg1*`]|`*Reg1*`,` *Reg2*	$\rightarrow$	`push` *Reg3* `mov` *Reg3*`,` *Reg2* `mov` *Reg1*`]|`*Reg1*`,` *Reg3* `pop` *Reg3*
T	`mov` *Reg1*`|[`*Reg1+Offset*`],` *Val1*	$\rightarrow$	`push` *Reg2* `mov` *Reg2*`,` *Val1* `mov` *Reg1*`|[`*Reg1+Offset*`],` *Reg2* `pop` *Reg2*

Table 6: Rule set used by the mutation engine of `W32.Evol`

P.C. C_i	l_i	$\rightarrow$	r_i

T	`mov Reg1, [Reg2+Val]`	$\rightarrow$	`push Reg3`
			`mov Reg3, [Reg2+Val]`
			`mov Reg1, Reg3`
			`pop Reg3`

T	`mov Reg1, [Reg2]|Reg2`	$\rightarrow$	`push Reg3`
			`mov Reg3, [Reg2]|Reg2`
			`mov Reg1, Reg3`
			`pop Reg3`

T	`mov Reg, Val`	$\rightarrow$	`mov Reg, Val1` `add|sub|xor Reg, Val2`

T	`lea|cmp Reg1, [Reg2+Val]`	$\rightarrow$	`push Reg3`
			`mov Reg3, Reg2`
			`add|sub Reg3, Val1`
			`lea|cmp Reg1, [Reg3+Val2`
			`pop Reg3`

T	`lea Reg1, [Reg2+Val]`	$\rightarrow$	`push Reg3`
			`mov Reg3, Reg1`
			`lea Reg3, [Reg2+Val]`
			`mov Reg1, Reg3`
			`pop Reg3`

T	`lea Reg1, [Reg2]|Reg2`	$\rightarrow$	`push Reg3`
			`mov Reg3, Reg1`
			`lea Reg3, [Reg2]|Reg2`
			`mov Reg1, Reg3`
			`pop Reg3`

T	`A Reg1, Reg2`	$\rightarrow$	`push Reg3`
			`mov Reg3, Reg1`
			`A Reg3, Reg2`
			`mov Reg1, Reg3`
			`pop Reg3`

eax is dead	`push Imm|[Reg+Offset]`	$\rightarrow$	`mov eax, Imm|[Reg+Offset]` `push eax`

Table 7: Rule set used by the mutation engine of `W32.Evol`

P.C. C_i	l_i	$\rightarrow$	r_i
RegSp is dead	`push` *RegSp*	$\rightarrow$	`push` *RegSp* `A` *RegSp*`,` *Val*
RegSp is dead	`push` *RegSp*	$\rightarrow$	`push` *RegSp* `mov` *RegSp*`,` *Val*`|[ebp+`*Val*`]`
T	`NOP`	$\rightarrow$	

RegSp refers to registers eax, ecx, edx. *A* refers to add, or, adc, sbb, and, sub, xor and cmp. A push-pop block will never use esp or ebp.

References

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

[2] P. Ször and P. Ferrie, "Hunting for metamorphic," *Proc. of the 11th International Virus Bulletin Conference*, 2001, pp. 123–144.

[3] C. Nachenberg, "Computer virus-antivirus coevolution," *Communications of the ACM*, vol. 40, no. 1, Jan. 1997, pp. 47–51.

[4] M. Jordon, "Dealing with metamorphism," *Virus Bulletin*, Oct. 2002, pp. 4–6.

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

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

[7] F. Cohen, "Computational aspects of computer viruses," *Computers & Security*, vol. 8, no. 4, Jun. 1989, pp. 325–344.

[8] D. Chess and S. White, "An undetectable computer virus," *Proc. of Virus Bulletin Conference*, Sept. 2000.

[9] D. Spinellis, "Reliable identification of bounded-length viruses is np-complete," *IEEE Transactions on Information Theory*, vol. 49, no. 1, Jan. 2003, pp. 280–284.

[10] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," *Proc. of the 2005 IEEE Symposium on Security and Privacy*, May 2005, pp. 32–46.

[11] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Computer Virology*, vol. 1, no. 1–2, Nov. 2005, pp. 13–23.

[12] A. Lakhotia and M. Mohammed, "Imposing order on program statements and its implication to av scanner," *Proc. of 11th IEEE Working Conference on Reverse Engineering*, IEEE Computer Society, Nov. 2004, pp. 161–171.

[13] D. E. Knuth and P. B. Bendix, "Simple word problems in universal algebras," *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, J. Siekmann and G. Wrightson, eds., Springer, 1983, pp. 342–376.

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

[15] A. Lakhotia and P. K. Singh, "Challenges in getting formal with viruses," *Virus Bulletin*, vol. 9, no. 1, Sept. 2003, pp. 14–18.

[16] "VX heavens," Apr. 2006; http://www.vx.netlux.org/.

[17] P. Ször, "W32.evol," Jul. 2000; http://securityresponse.symantec.com/avcenter/venc/data/w32.evol.html.

[18] J. R. Cordy, "TXL – a language for programming language tools and applications," *Proc. of the ACM 4th International Workshop on Language Descriptions, Tools and Applications (LTDA'2004)*, ser. Electronic Notes in Theoretical Computer Science, Elsevier, Dec. 2004, vol. 110, pp. 3–31.

[19] A. E. Stepan, "Defeating polymorphism: Beyond emulation," *Virus Bulletin Conference*, Virus Bulletin, Oct. 2005, pp. 40–48.

[20] F. Perriot, "Defeating polymorphism through code optimization," *Proc. of Virus Bulletin 2003*, Sept. 2003.

[21] D. Bruschi, L. Martignoni, and M. Monga, "Using code normalization for fighting self-mutating malware," *Proc. of International Symposium on Secure Software Engineering*, IEEE Computer Society, Mar. 2006.

[22] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," *Proc. of the 8th Symposium on Recent Advances in Intrusion Detection (RAID'2005)*, ser. Lecture Notes in Computer Science, Springer-Verlag, Sept. 2005, pp. 207–226.

[23] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static analyzer of vicious executables (SAVE)," *Proc. of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, Washington, DC, USA: IEEE Computer Society, Dec. 2004, pp. 326–334.

Mathur, Rachit. Bachelor of Engineering, Jai Narayan Vyas University, Spring 2004;
Master of Science, University of Louisiana at Lafayette, Fall 2006
Major: Computer Science
Title of Thesis: Normalizing Metamorphic Malware Using Term Rewriting
Thesis Director: Dr. Arun Lakhotia
Pages in Thesis: 55; Words in Abstract: 191

ABSTRACT

A malicious program is considered metamorphic if it can generate offspring that are different from itself. The differences between the offspring make it harder to recognize them using static signature matching, the predominant technique used in malware scanners. One approach to improving the ability to recognize these metamorphic programs is to first "normalize" them to remove the variations that confound signature matching. This thesis proposes modeling the metamorphic engines of malicious programs as term rewriting systems and then formalizes the normalization construction problem as a problem of constructing a normalizing term rewriting system such that its rule set maintains three properties: termination, confluence, and equivalence-preservation. Risks associated with failing to assure these three properties are outlined. A strategy is proposed for solving the normalization construction problem. Two approximations are also defined that can help improve normalizer performance. These are based on relaxing the confluence and equivalence preservation requirements. A simple priority scheme is outlined for reducing the number of false positives the approximations may produce. The results of a case study are reported; the study demonstrates the feasibility of the proposed approaches by normalizing variants of a metamorphic virus called "`W32.Evol`."

Biographical Sketch

Rachit Mathur was born in Kota, India, on April 09, 1982. He graduated with a Bachelor of Engineering degree with honors in Information Technology in June 2004 from Mugneeram Bangurh Memorial Engineering College, Jai Narayan Vyas University, Jodhpur, India. He entered the Master of Science program in Computer Science at the University of Louisiana at Lafayette in Fall 2004. Following completion of this degree, Rachit Mathur will be pursuing a Doctoral degree in the area of software security.