Journal in Computer Virology Preprint

This article is an expanded version of a paper in SCAM 2005. It has been revised and accepted for publication in Journal in Computer Virology and is scheduled to appear in 2008. Citations to and quotations from this work should reference that publication. If you cite this work, please check that the published form contains precisely the material to which you intend to refer.

Constructing Malware Normalizers using Term Rewriting

Andrew Walenstein, Rachit Mathur*, Mohamed R. Chouchane, Arun Lakhotia

Center for Advanced Computer Studies
University of Louisiana at Lafayette

Abstract A malware mutation engine is able to transform a malicious program to create a different version of the program. Such mutation engines are used at distribution sites or in self-propagating malware in order to create variation in the distributed programs. Program normalization is a way to remove variety introduced by mutation engines, and can thus simplify the problem of detecting variant strains. This paper introduces the "normalizer construction problem" (NCP), and formalizes a restricted form of the problem called "NCP=", which assumes a model of the engine is already known in the form of a term rewriting system. It is shown that even this restricted version of the problem is undecidable. A procedure is provided that can, in certain cases, automatically solve NCP= from the model of the engine. This procedure is analyzed in conjunction with term rewriting theory to create a list of distinct classes of normalizer construction problems. These classes yield a list of possible attack vectors. Three strategies are defined for approximate solutions of NCP=, and an analysis is provided of the risks they entail. A case study using the `W32.Evol` virus suggests the approximations may be effective in practice for countering mutated malware.

Key words Malware – program transformation – normalize – classify.

1 Introduction

Malicious programs—worms, viruses, Trojans and the like—are collectively known as "malware" [27]. In 1989, Cohen [10] anticipated that self-mutating malware would one day be created: the malicious program would be able to transform its own code so as to create variants of itself. Six years later, such self-mutating viruses began to appear [25]. Variants created by such malware might still behave like the original program, but their code could be different. For example, an early mutating virus called `W95.RegSwap` rewrote itself so that use of one general-purpose register was swapped for use of another [25]. More sophisticated program-to-program transformations have been used over the years, including substitution of equivalent code, insertion of irrelevant instructions, and reordering of code without ordering dependencies [30]. Apart from being carried by self-propagating malware, mutation engines can also be used at hosts distributing non-propagating malware.

Mutation was introduced in order to evade malware detectors. The variety introduced by the mutation has the potential to create tremendous detection challenges. In particular, if a detection technique relies on recognizing some pattern of features, and the mutation engine serves to modify these features, the detector can be defeated. For instance, if a detector relies on matching a "signature" (an identifying pattern) of bytes or system calls, then the mutating transformations may alter those bytes or system calls such that the pattern no longer matches. An approach to counteracting the effects of mutation is to *normalize* the input programs in order to try to remove the variety that challenges the pattern matching. Arguably, the "perfect" normalizer would transform all varieties of any family to a single form. We call the problem of creating a normalizer for a family of variants the "normalizer construction problem" (NCP).

* R. Mathur is presently at McAfee AVERT Labs

Several different malware normalizers have been created (Lakhotia *et. al* [21], Christodorescu *et. al* [9], Bruschi *et. al* [6]). To date, these efforts have proceeded without a theoretical basis from which to understand the prospects for success. The situation is troubling because it is already suspected, via Chess and White [7], that perfect detection of all variants of a mutated malware family is impossible even when one is provided a sample variant from the family. However, perfect normalizers are possible if one restricts conditions sufficiently well. As a simple existence proof, consider that perfect normalization is straightforward for self-mutating programs such as `W95.RegSwap` mentioned above. So important questions are: (1) can we define useful classes of normalization problems for sets of variants created by mutation engines?, (2) is perfect normalization possible in pragmatically interesting cases?, and (3) what can be done for classes that provably cannot be perfectly normalized? This paper addresses all three of these questions, and introduces a new approach to generating normalizers from models of mutation engines.

A restricted version of the NCP is formalized using the theory of term rewriting [3]. This restricted problem is denoted "NCP=", and is a restriction in two ways. First, it assumes one has an accurate model of the mutation engine in the form of a term rewriting system. In the unrestricted normalization problem, such a model is not assumed: it must either be unnecessary, or else it must first be created—by inferring the transformations from multiple samples, reverse-engineering the rules from the engine itself, etc. Second, NCP= is restricted to those cases where the rewrite rules are all semantics-preserving. Solving NCP= involves creating a TRS that induces the same *equivalence classes* as the self-mutating program such that it is *convergent*, i.e., that the normalizer is *terminating* and *confluent*. The theory of term rewriting is used to show that NCP= is undecidable.

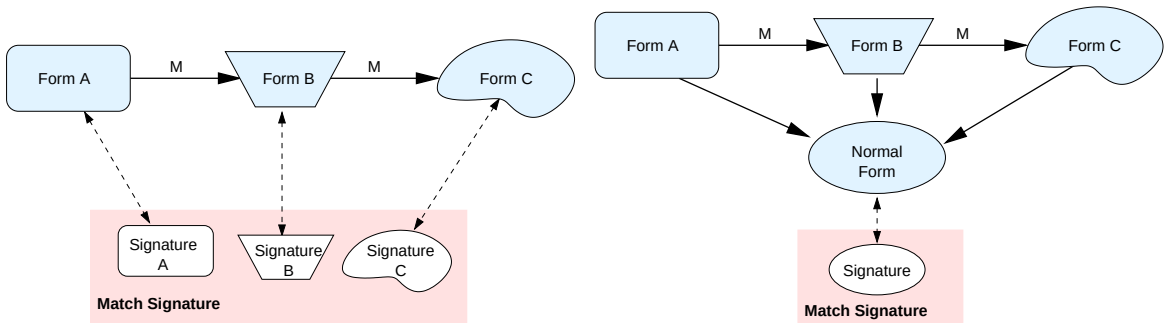
Because NCP= is undecidable, no procedure can exist which is guaranteed to halt and produce a correct normalizing transformer. In practical terms, however, all hope need not be lost. Procedures may be defined that solve it in some circumstances, so it may be practically important to have such procedures at hand. A two-phase normalizer construction procedure is defined that will work for certain classes of self-mutating programs. It is based on two well-known term rewriting procedures. A case study using the well-known `W32.Evol` virus demonstrates that this procedure can fail in realistic circumstances, while also demonstrating that is nevertheless possible to construct perfect normalizers—in our case by manual introduction of suitable rules that can be said to "complete" the rule set with respect to the normalization engine.

Three approximation approaches to NCP= are introduced. Approximate solutions may be desirable in cases where it is impossible or impractical to either construct an exact normalizer, or to correctly evaluate conditions in a conditional normalization rule system. The approximation approaches are: (1) using "incomplete" rule sets, (2) using a priority scheme, and (3) ignoring conditions in conditional normalization rule sets. A second part of the case study on `W32.Evol` demonstrates that the NCP= problem is not so restricted that it is uninteresting in practice, shows the problem of approximation is practically relevant, and illustrates the promise of the priority-based approach to approximation.

Section 2 provides background on the NCP, and formalizes the NCP= using term rewriting theory. The section also introduces the fallible two step procedure for solving NCP=. Section 3 introduces the approximation solution strategies. Section 4 describes a case study using the `W32.Evol` virus which sought to evaluate the general feasibility of the term-rewriting based normalization approach, and to examine the efficacy of the different approximations. Section 5 documents relations to other work. Conclusions are drawn in Section 6.

2 The problem of normalizing mutants

Self-mutating malware may be conceptually decomposed into two components: a *mutation engine*, which performs program-to-program transformations, and a *payload*, which is the body of code that implements the malicious behavior. Many malicious programs are structurally decoupled in this way, since the separation makes it possible to easily reuse a given mutation engine by attaching it to a different payload. The scope of this paper are those mutation engines that transform the malware so that the program code itself changes; these were called "metamorphic" by Ször and Ferrie [30]. Existing mutation engines have targeted their attacks on malware detectors that utilize signatures defined over the program's form—i.e., its *syntax*. The engines modify the code bodies during replication with the intention of ensuring that signatures cannot be constructed which match all variants the engine can produce. Following Cohen [10], we call the set of all such variants the "viral set". For such malware, multiple signatures would be required, as illustrated in Fig. 1(a). When the number of signatures required grows sufficiently large, the signature-based methods may fail to provide adequate detection rates for the entire viral set. In the worst case an unbounded number of signatures are required to match all possible variants.



(a) Detection using multiple signatures (b) Detection using a normal form

Fig. 1 Intended effect of normalization: pattern space simplification

Several different approaches have been proposed for detecting mutating malware. One such approach attempts to detect variants within the viral set by matching some facet that may not be easily disguised by mutations. Perhaps the best known is the use of signatures based on some type of behavior rather than form. The merit of this approach stems from the empirical fact that current mutation engines tend to be *behavior-preserving*, that is, they modify their form while keeping the behavior constant or nearly constant (at least, when considered at some level). Signatures defined on behavior may be matched either by dynamic or static analysis. Dynamic analysis frequently entails emulating or tracing the programs while looking for telltale call sequences or data use [29]. Indeed, emulation-based signature matching was one of the techniques that commercial detectors used to detect `W32.Evol`, a parasitic self-mutating virus [29]. Published static techniques for matching behavior include those by Singh *et. al* [26], Christodorescu *et. al* [8], Kruegel *et. al* [19], and Kinder *et. al* [17]. While behavior-based signature detection has been successful, it also has its share of problems. Regardless of whether it is done statically or dynamically, it can be costly, error-prone, and introduces its own set of vulnerabilities and limitations.

A second approach to detecting mutating malware is to use more powerful pattern matching to define and match the signatures. The intent is to permit the use of more general signatures, namely ones that match any variant in the viral set. This strategy was used to catch early self-mutating malware. In the case of `W95.RegSwap`, for example, certain malware detectors were enhanced to use wild-card based matching, allowing them to match variants regardless of their specific assignment of registers [30]. While the benefits of more powerful matching are clear, adding the power can introduce its own set of problems also. These include: increased cost of matching, and difficulty in specifying patterns that precisely match all variants in the viral set, but no other programs.

A third approach to detecting mutating malware is to try to normalize the input programs. Because normalization removes unimportant variations, it may allow less powerful pattern matching to be (or remain) effective. As illustrated in Fig. 1(b), the goal is to shrink the effective input space—from the original viral set to a smaller set—thereby decreasing the variations a detector needs to consider during recognition. The normalization approach to matching self-mutating malware was exemplified by the methods of Lakhotia *et. al* [21]. As a proof-of-concept, Lakhotia *et. al* developed a "generic" normalizer for `C` programs—i.e., one not tuned to any particular mutation engine. It removed variations via program transformations such as expression reshaping and constant propagation. These are techniques common to optimizing compilers. It also employed the strategy of imposing order on unordered items, such as reordering instructions in a fixed way. While this approach was shown to be unable to reduce all variants in a family to a single form, Lakhotia *et. al* reported a massive reduction in the number of possible normalized forms: from 10^{183} possible forms to 10^{20}. Other malware normalization methods have also been proposed using compiler-like transformations, including Christodorescu *et. al* [9] and Bruschi *et. al* [6]. None of these approaches required—nor took advantage of—knowledge about the specific mutation engine producing the variants.

Though these normalizers are important first steps, questions arise as to how well one these approaches will fare. It might be hoped that the transformations will collectively result in only a few normal forms for all the variants. But what guarantees can be had, and in what conditions can a "perfect" normalizer be constructed? Exploration of such questions has been limited to either mathematical analysis of the reduction in space of variants (Lakhotia *et. al* [21]), or empirical studies of pragmatic effectiveness using limited case studies (Lakhotia *et. al* [21], Christodorescu *et. al* [9], Bruschi *et. al* [6]). No theoretical analyses of the problem have been able

to answer the key open questions: (a) When are perfect normalizers possible? (b) How can they be constructed? and (c) If they are not possible then what approximate solutions are feasible?

The remainder of this section introduces key problems in normalizing self-mutating programs, and formalizes the NCP= using term rewriting theory. Required definitions from term rewriting theory are recalled, and a two-step procedure is defined that can be used to try to solve the NCP=. Its possible application is discussed, and a selection of relationships between NCP= and other normalization problems are listed.

2.1 How not to "reverse" a transformation set

The normalizers of Lakhotia *et. al* [21] and Bruschi *et. al* [6] applied "generic" transformations in the sense that they are not tailored specifically for a given mutation engine. Unfortunately, at the moment it is not clear that they would normalize variants to a single normal form. Perhaps the generality of the transformation set is a liability? One might wonder whether it is possible to correctly normalize programs by choosing transformations tailored to an already-known mutation engine. While such normalizers would not provide a general solution, they might still be practical due to the limited number of engines needing normalizers. Powerful mutation engines have been written by few authors, are thus rare, and are not known to evolve rapidly [29]. This is likely due, in part, to the difficulty even experts have in designing and writing correct program transformers (cf. the extensive work on compiler verification [13]). So while mutation engine-specific normalizers would not provide a universal solution, they may nonetheless form a useful part of a practical defense regime.

If one already knows the transformations a mutation engine uses, perhaps the naive approach of simply "reversing" all the transformations would yield an effective normalizer? That is, if $A \rightarrow B$ appears in the mutation engine (i.e., statements A are transformed into statements B), could one not create an effective normalizer simply by applying $B \rightarrow A$ for all such rules? So, for example, suppose P is a self-mutating program, M is the mutation engine of P, and $S(P)$ is the set of possible variants of P that can be created through transformations of M. The essence of the naive approach is: since any element in $S(P)$ must have been created through some sequence of transformations $T = m_1, m_2, \ldots, m_k$, if one reverses the transformations that were applied then one would "undo" the variations, yielding the original program P again. That is, the idea is that the inverse of T, namely $T^R = m_k, m_{k-1}, \ldots, m_1$ could be performed by reversing the transformations of M. While the scheme may appear *prima facie* sound, and it might be made to work well enough in select circumstances [20], it will not work in general. The problem is that reversing the direction of the transformations of M fails to guarantee correct reversal of T.

The limitation of this naive approach is illustrated using an example subset of transformations selected from the virus `W32.Evol`. The example selection is Fig. 2. The disassembly of the parent's code (in IA32 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 Fig. 2(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 Fig. 2(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 Fig. 2(c) inserts "`mov eax, 0x09`" which, because of the specifics of `W32.Evol`, is a "junk" or irrelevant statement—i.e. one that has no effect on the computation. Note that none of these transformations will affect program semantics in the ordinary sense (i.e., ignoring such nuances as new memory faults due to increase in code size, measurable differences in performance, self-inspecting or modifying code, etc.).

Inspection of the transformations shown in Fig. 2 reveals some problems that can arise if one naively applies the transformations in reverse. Consider, for instance, the hypothetical sequence

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

which can be part of the output of either of the transformations (b) or (c) from Fig. 2. The normalizer must be able to decide which of transformations (b) or (c) to apply in reverse; that is, it must correctly decide which of the transformations created the observed code. If it cannot, the inverse of T will not be created, and the normalization may be incorrect. More specifically, if it applies the wrong transformation then it may either transform a non-variant into a variant, or transform a variant into a non-variant. In such cases, no one unique normalized form is guaranteed for all variants. For example, if the `mov eax, 0x09` in the hypothetical sequence is *not* a junk

	Parent	Offspring (transformed)	Brief Description

(a)	`push eax`
	`mov [edi], 0x04`
	`jmp label`
		`push eax`
		`push ecx`
		`mov ecx, 0x04`
		`mov [edi], ecx`
		`pop ecx`
		`jmp label`
			Use temporary register to transfer immediate value

(b)	`push 0x04`
	`mov eax, 0x09`
	`jmp label`
		`mov eax, 0x04`
		`push eax`
		`mov eax, 0x09`
		`jmp label`
			Immediate push turned into push through dead register

(c)	`mov eax, 0x04`
	`push eax`
	`jmp label`
		`mov eax, 0x04`
		`push eax`
		`mov eax, 0x09`
		`jmp label`
			"Junk" code insertion

Fig. 2 Three sample transformations from `W32.Evol`

instruction, then applying transformation (c) in reverse removes a non-junk instruction, which yields a different program, which means it should not be in $S(P)$, and the transformation is incorrect. Similar transformation choice problems can also arise when transformations must occur in a specific order.

The above issues of transformation selection, ordering, unique normalized forms, and ensuring equivalence are dealt with elegantly by the theories of term rewriting systems. Using these, it is possible to formalize NCP= terms of constructing what is called a "convergent rule set" that satisfies specific equivalence properties.

2.2 NCP= as a term rewriting problem

Some definitions and results from term rewriting are required. The reader is referred to Baader *et. al* [3] for detailed explanations.

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; these are *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* $s \to t$ 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. Fig. 3 shows a simple example of an unconditional TRS.

Reduction relation ($\to_T$).

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

Equivalence relation ($\overset{\star}{\longleftrightarrow}$).

The $\to$ relation on terms induces an equivalence relation $\overset{\star}{\longleftrightarrow}$ defined by the reflexive symmetric transitive closure of $\to$. $\overset{\star}{\longleftrightarrow}$ partitions the set of terms into equivalence classes. Given a TRS T, $[t]_T$ denotes the equivalence class of term t under $\overset{\star}{\longleftrightarrow}$.

Normal form.

If a term t is not related to any other term under $\to_T$, 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 Fig. 3, the term $add(2, 2)$ is in normal form, and $add(1, add(1, 1)) \to_T add(1, 2)$ by application of the rule mapping $add(1, 1)$ to 2.

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

Fig. 3 Sample rewrite rules

Termination.

A TRS T is terminating if there exists no infinite chain of reductions ($t_1 \rightarrow t_2 \rightarrow t_3 \cdots$).

Confluence.

Let 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 every such 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, although it is decidable for finite, terminating TRSs [3]. In the general case, the problem of converting an arbitrary TRS into an equivalent one that is confluent is undecidable regardless of whether it is conditional or not.

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 $\overset{\star}{\longleftrightarrow}$. This can be done by repeatedly applying the rules of T (in arbitrary order) to any given input x; this process is guaranteed to result in the normal form that is unique to x's equivalence class. Testing membership in class c then becomes a matter of comparing the normal form of x to the normal form associated with all members of class c: if the normal forms of the two terms differ, they are in different equivalence classes.

For a certain class of semantics-preserving mutation engines, it will be possible to use a TRS to model the transformation behavior of the engine. This can be done for ordinary machine languages by modeling instructions as terms that consist of a function applied to one or more variables or constants. The function is the operation (`mov`, `push`, etc., complete with mode of the operation) and the variables and constants are the registers and immediate operands. A program (or a code segment) is a term obtained by applying a `concatenate` function (written ";", or using a new line) to such terms. We consider only those engines that are, in fact, semantics-preserving.

This formalization of the mutation engine closely matches the prior formalization of viral set constructors via formal grammars by Zuo *et. al* [15]. In their formulation, the viral set is generated by an unconditional rewriting rule set that transforms strings of both terminal and non-terminal characters (as compared to term rewriting entities of functions, variables, and constants). Naturally, real mutation engines always transform terminal strings to other terminal strings. Our approach models the mutation engine as first mapping terminal symbols to non-terminals, i.e., from concrete specific registers or constants to non-terminals. A similar approach in the formal language formalization would be to assume rules that transform terminals to non-terminals.

Fig. 4 gives an example of how a transformation of a mutation engine may be formalized as a rewrite rule. The rule in the figure 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 Fig. 2. Other rewrite rules may need to be conditional in order to accurately model the condition-sensitivity of the mutation engine's transformations. Examples of these are shown in Fig. 2(b) and Figure 2(c). Conditions are written as predicates to the left of the left hand side. For simplicity, we will henceforth write rules in assembly language with embedded term variables, rather than in the function application form shown in Fig. 4.

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

Fig. 4 Code substitution rewrite rule

Assume a mutation engine E is a *closed-world* engine, meaning that it operates without using external information. Assume further that E can be modeled as a term rewriting system as above. Let M denote a TRS modeling engine E. The equivalence relation induced by M partitions terms 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 mutation, 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 detector to recognize the program (i.e., by applying the malware's own M until it converges to the normal form). Thus one would normally expect malicious engines to be non-convergent.

Any suitable normalizer N will be convergent and therefore terminating. Moreover, the "perfect" normalizer will be such that the equivalence relation induced by N is the same as that induced by M. In particular, all members of the viral set reduce to one normal form and no non-member reduces to this same normal form. These observations are used to formalize NCP= in TRS terms and show it may be treated as a problem of transforming M into a suitable N.

NCP= as a TRS transformation problem. Given a finite, possibly conditional rewrite system M that accurately models a semantics-preserving mutation engine E, construct a rewrite system N that satisfies the following properties:

1. **Equivalence:** $\forall x.[x]_M = [x]_N$.
2. **Termination:** N must be terminating.
3. **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 *any* sequence of applications of the rules of N to some term t will eventually halt. The confluence condition implies that if a normal form for some term is reached, then this normal form is unique.

Once NCP= is defined as above it is trivial to see that it is undecidable: it is known that creating an equivalent confluent TRS from a non-confluent one is undecidable in the general case, and since we have not otherwise restricted M then NCP= is also undecidable. If the problem is undecidable given the model M, it is surely no easier without M. Thus it can be said from the outset that the perfect generic normalizer will be unattainable for those classes of mutation engine that are formalizable using a TRS as above. This places at least an outer limit on what can be expected from efforts similar to Lakhotia *et. al* [21] or Christodorescu *et. al* [9].

While this is in a sense a negative result, it is also positive in the sense that it outlines some conditions when perfect normalizers are possible. From the theory, it is known that if such an N can be produced, and the conditions on its conditional rules can be statically computed, it will form a perfect normalizer for M. Recall that programs are modeled as terms. The rules of N can be applied to an input program I, in any order, and eventually any input will be transformed into the single normal form of its equivalence class with respect to N. Since N is equivalent to M and M is semantics-preserving, that means all programs normalizing to the same form as P will be semantically-equivalent to P. Given a single variant of P, one can therefore extract the unique normal form and test for an exact match to normalized input programs. This test will yield no false positives or negatives.

Formalizing NCP= in this way restricts discussion only to those cases where the mutation engine can be modeled as a TRS, as above. This class of malware is not so restricted that it defines an uninteresting problem space. The problem is not decidable and, as will be shown in Section 4, important realistic self-mutating programs fall in the class. Engines using instruction-substitution transformations can be modeled in this manner. Using conditional TRS it is possible to model mutation engines that perform transformations only under certain conditions. Probabilistic mutation engines can be subsumed in the framework as well; these engines fire transformations only probabilistically, that is, only when some random (or pseudorandom) condition occurs. By making rule application probabilistic, the engine can turn the path that an outbreak takes through the space of variants into a random walk. When E makes only semantics-preserving transformations, probabilities can be ignored for the purposes of normalization since the confluence property makes the probabilistic application issue moot.

2.3 A procedure for solving NCP=

Even though NCP= is undecidable, procedures can be defined that attempt to solve it—they just can never come with a guarantee to halt with a correct output. Using the term rewriting literature, we define a two-phase procedure that involves first applying a *reorienting procedure* to M to ensure termination, and then a *completion procedure* 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 NCP=. We introduce this procedure because: (1) it may be useful in practice, and (2) the analysis of when the procedure fails can be used to define boundaries between problem classes, and thus exposes potential attack vectors.

2.3.1 Reorientation: ensuring termination and equivalence. Although a rewrite rule relates equivalent terms, the term-rewriting system may apply the rule in only one direction, namely, the direction indicated by the arrow in the rule. A rule is said to be "reoriented" when the application direction is reversed, i.e., reorientation transposes the left hand side (l_i) for the right hand side (r_i). A *reorienting procedure* is a procedure that assigns orientations of the rules in a TRS such that the reduction procedure of the TRS 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].

The *well-founded length-lexicographic* ordering is frequently used to reorient string rewriting systems [3], i.e., on systems with only ground terms. The reorientation procedure traverses M and reorients the rules whose right hand sides are length-lexicographically greater than their left hand sides. So long as M has no identity rules (rules of the form $x \rightarrow x$) the resulting system M^t is terminating because any rule application will decrease the length-lexicographic size of the term being reduced, and any finite term cannot be endlessly reduced in length. In certain cases, the ordering can be extended so it can be used on systems using variables. In the case of the TRS in this paper, by inspection we know the reoriented systems will terminate (we can define an appropriate ordering on the variables being used; also they are equivalent to notational shorthands, so that rules with variables can be replaced by a finite number of rules using only ground terms). Table 1 shows a fragment of an example rewriting system. The last column shows the decision of whether to reorient the rule. Note that the conditions of the second column are *post* conditions for the code on the left hand side of the rules. That is, the conditions shown must hold whenever the end of the l_i block is reached. A "T" means the condition is always true, i.e., that the rule is effectively unconditional. Rule M_1 is to be reoriented because r_1 is length-lexicographically greater than l_1. So, for example, M_2^t (the reoriented rule M_2) is "`eax is dead | mov eax, imm ; push eax → push imm`". Here, a register is dead when its value is not needed before the register is assigned [2].

Label	Condition	Rule $l_i \rightarrow r_i$	Reorient?

M_1	T	`mov [reg1+imm], reg2` → `push eax`
		`mov eax, imm`
		`mov [reg1+eax], reg2`
		`pop eax`
			y

M_2	`eax` is dead	`push imm` → `mov eax, imm` `push eax`	y
M_3	`eax` is dead	`push eax` → `push eax` `mov eax, imm`	y
M_4	T	`nop` →	n

Table 1 Reorienting example

Since the reflexive symmetric transitive closure of M^t is identical to that of M (no rules were modified apart from their orientation), the set of equivalence classes defined by M^t is identical to that defined by M; 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=.

2.3.2 Completion: ensuring confluence. Confluence is decidable for finite terminating TRSs [3]. If a TRS is not confluent, then additional rules may be added to it to make the system confluent. A process of adding rules to make a TRS confluent is called a "completion procedure." The resulting confluent TRS is said to be "completed."

Recall that the problem of completing a TRS is undecidable in the general case, so that any procedure attempting it cannot be guaranteed success. In practical contexts, trying to ensure confluence may be a matter of selecting one or more completion procedures to try, and then choosing suitable results, if they complete. The Knuth-Bendix completion procedure (KB) [18] is the most prevalent method used in term-rewriting literature. Detailed discussions of this procedure are available elsewhere [3, 18]; it is explained below only to the degree needed to later enumerate sub-classes of NCP=.

The KB procedure works to resolve *critical overlaps* between rules by adding new rules. For finite terminating ground TRSs, the left hand sides of a pair of (not necessarily distinct) rules are said to critically overlap if the

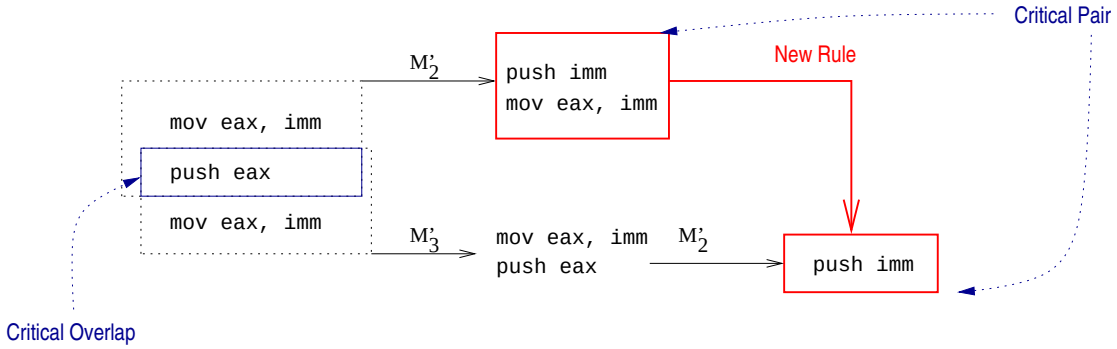


Fig. 5 Completion step for M_2^t and M_3^t

Label	Condition	Rule $l_i \rightarrow r_i$

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 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`

Table 2 N, the completed version of example rule set M^t

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 Table 1, 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 Fig. 5. The figure shows that KB adds the rule

`push imm`
`mov eax, imm` $\longrightarrow$ `push imm`

to the set. KB is not guaranteed to terminate. However, if it does terminate then the TRS it produces will be confluent.

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. Rules M_1^t and M_3^t from our example (see Table 1) 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. The completion procedure terminates on the TRS of Table 1, and returns the confluent system of Table 2.

2.4 Classes of mutation engines

Section 2 showed that it will not be possible to automatically construct a perfect normalizer for certain mutation engines. Nonetheless, certainly the viral sets for *some* types of mutation engines can be normalized, and some mutation engines present more challenges for constructing normalizers than others do. The definition of NCP= makes it possible to partition normalization construction problems into well-defined problem classes. Several distinct classes of solvable NCPs can be delimited by noting conditions on rule sets which make it possible to prove that NCP= can be solved. Note that a condition on a rule set identifies a subset of the universe of TRSs. Classes of program normalization problems that fall outside the scope of NCP= can be defined according to how they fail to meet conditions that permit automatic solutions. Knowing about these classes is important in defense since each class can indicate when a given technology or approach can work, and each class formalizes a possible attack vector for defeating (automatic) normalizer construction defenses.

Several classes of NCPs are listed in the following sub-sections; each is defined by noting when NCP= is known to be solvable and by recognizing conditions in which mutation engines are not formalizable using TRSs, as above. There may be other restrictions or extensions that yield further sub-classes (e.g., using different rewriting systems such as string, graph, or constrained rewriting systems), however the present list is not intended to be comprehensive in this respect. Rather the aim is utilize the definition of NCP= to explore the contours of the local problem space surrounding NCP=. Listed in each class are possible attacks on the normalizer constructor that can be associated with the class. That is, regardless of whether the normalizer construction is automated or not, these are attacks that can make the problem more difficult.

2.4.1 NCP=^{NV}: Formalizable without using variables. Consider the class of TRS as in NCP=, but without variables. For any mutation engine that can be modeled using such a TRS (using a finite set of rules) the two-phase procedure of Section 2.3—i.e., the length-lexicographic ordering followed by Knuth-Bendix completion procedure—is guaranteed to complete, and the result will create a perfect normalizer. This NCP problem subspace, denoted by "NCP=^{NV}", is therefore decidable.

A strong attack would ensure that transformations are complex enough that variables must be used to correctly model it.

2.4.2 NCP=^{NO}: No critical overlaps. Consider the class of TRS as in NCP=, but without overlaps defined as critical in the Knuth-Bendix procedure. For any mutation engine that can be modeled in this way, the two-phase procedure of Section 2.3 is guaranteed to terminate, and thus will create a perfect normalizer. This NCP problem subspace, denoted by "NCP=^{NO}", is therefore decidable.

A strong attack would ensure that critical overlaps are present.

2.4.3 NCP=^{IC}: Conditions or assumptions are not computable. In defining NCP=, no explicit guarantee was made as to whether the conditions attached to the rules of N would be feasibly computable. However the procedure described in Section 2.3 for constructing N from M does not alter the conditions, so if the conditions in M are feasibly and statically computable conditions in the general case, then those of N are too. Significantly though, it can be the case that E can determine the appropriate conditions yet, for practical purposes the normalizer may not be able to, or be permitted to. Specifically, E can make use of assumptions or encoded data rather than compute the conditions explicitly, whereas the normalizer may not be able to [33]. In fact, `W32.Evol` does not calculate its conditions at all; it only makes use of a carefully crafted property of the code. Specifically, it utilizes implicit indications of register liveness to avoid the need to extract it from its own code. In theory, any assumption E makes, the normalizer could make; in practice, if multiple similar engines use opposing assumptions it could lead to a new and potentially unsolvable problem of deciding on the correct assumptions. When these conditions are undecidable, the mutation engine falls outside of the class defined by NCP=.

One may also wish to restrict NCP= to only those cases where it is feasible for the normalizer to evaluate the conditions during normalization. In practice, the question of feasibility may not be the same for the engine as the normalizer, as the normalizer may be required to be significantly faster. The engine E may gain an additional computational advantage in that it can directly inspect the state of the running system; this may make certain conditions much easier to check as compared to static analyses. While the normalizer is free to try to emulate the program, of course, to discover the same information the same way, this may entail such a high overhead as to make the approach infeasible for the normalizer.

A strong attack would ensure that hard-to-compute conditions are attached to some of the rules. An even stronger attack would ensure that these conditions are non-computable using known techniques.

2.4.4 NCP≠: Semantics non-preservation. The formulation of NCP= explicitly considers only mutation engines whose rules preserve program semantics. In part this is because the conditions attached to rules can relate to operational semantics. For example, conditions based on register liveness are defined in terms of the computational interpretation of the program statements. The terms in the rewrite rules themselves, however, are purely syntactic. We required, however, that the rewrite rules do ensure that term equivalence maps to program equivalence. In this way the NCP= approach is both like and unlike the purely syntactic formal language-based approach of Zuo *et. al* [15] and Filiol [14]. In their formulation, the rewrite rules are syntactic transformations between (uninterpreted) strings of terminals and non-terminals. In the NCP= formalization, the left and right hand sides of the rules of M and N are, by definition, intended to be considered equivalent and replaceable both in the term rewriting sense and the program semantics sense.

It would be possible to discard the requirement that the rules of M be semantics-preserving, as many mutation engines that do not preserve semantics can be modeled in the way proposed in the paper. For example, an engine might turn a `push` into a `pop`, and nothing would be amiss from the term rewriting point of view. The equivalence class under M (i.e., the viral set) would merely contain programs that are not equivalent—as it should. However doing so can make it difficult or impossible to reason meaningfully about the problem of malware detection. For this reason, our formulation of NCP= requires that semantics be preserved in the rule set of M. Engines that modify program semantics are relegated into the superclass NCP≠.

An example may help illustrate the problem. Suppose that one has a semantics-preserving mutation engine M, to which one adds the following probabilistic rule:

$$r_B\text{:}\quad \texttt{rand()} < 2^{-31} \mid \texttt{P} \longrightarrow \texttt{B}$$

where P is the whole program itself and B is a known benign program (perhaps the common program `notepad.exe`). Call this new augmented TRS "$M+$". The probabilistic condition on the left hand side assures that the likelihood that this rule fires is extremely small. Nonetheless, with respect to $[t]_{M+}$, the benign file B is equivalent to the original malware P, since there is a rule (r_B) that makes them equivalent. Any normalizer $N+$ for this set will induce the same equivalence relation as $M+$, so the normal form for B will be identical to the normal form of P. Practically, though, one may wish to say that matching B should be considered a false positive, yet from a theoretical point of view it is not. As a second example, consider adding probabilistic rules of the form

$$\texttt{rand()} < 2^{-31} \mid \texttt{op(arg1,arg2)} \longrightarrow \lambda$$

for every possible operation *op*, i.e., low probability rules that will map any possible program instruction into the empty string. The normal form for such a program is the empty string. Granted, these are perhaps imaginary examples and, if a human were creating the rule set is modeling the engine, then she might choose to exclude rules such as r_B. Still, the examples serve to illustrate the problems that can occur when rule equivalence does not map to program equivalence.

Requiring semantic equivalence restricts the problem space that NCP= defines in significant ways since certain important classes of mutation engine cannot be formalized by it. These include:

1. Engines that introduce *irrelevant* externally observable behavior, that is, computations that do not affect the malicious behavior, but which nonetheless can be detected without examining the internal workings of the executing program. For example, an engine may contain a transformation that inserts code to: create a file, write random content to it, and then remove the file some time later. While one might reasonably argue that the inserted code does not change the essential nature of the program, it certainly changes the behavior. If the temporary file is not removed, or if the file writing causes observable side effects (writing to error logs on disk failures, for example) the variants are not functionally or semantically equivalent.
2. Engines with bugs or limitations that prevent them from correctly performing semantics-preserving transformations. This is a significant omission in the sense that it may be rare to find complicated mutation engines that are completely bug free.
3. Evolutionary engines, i.e., ones that make changes to the functionality of the program as they reproduce. Such changes could be changes to the payload, the engine itself, or both.
4. Open-world engines. A closed-world engine operates with only the information contained in its own program. An open-world engine may utilize information from the environment. An example is an engine that downloads transformation rules from an external source.

In some of these cases, no approach based on term rewriting will be satisfactory because it is not feasible to model the mutation engine using a reasonable rule set (e.g., the open-world engines). In cases where it is possible to model the engine as a term rewriting system that does not preserve program semantics, the normalizer construction approaches of this paper will work as expected: with the possible exception of the first class above, they may just fail to map meaningfully onto malware detection problems.

From the list it appears that, when semantics are allowed to change, a Pandora's box is opened in terms of attacks to normalization.

3 Approximate solutions to the NCP

There exist classes of mutation engines for which perfect normalizers can be constructed. However even when a perfect normalizing rule set can be constructed, it may be infeasible to implement the normalizer in a practical implementation. The conditions attached to the rules may be too costly, difficult, or even impossible to calculate in the general case. Moreover, there are mutation engines in NCP= for which it is difficult to construct a normalizing rule set (e.g., the procedure of Section 2.3 does not halt).

These observations motivate the search for approximate solutions to NCP=. Precise solutions to NCPs preserve the equivalence classes of M and produce only a single normal form for each equivalence classes. Approximate solutions may create multiple normal forms for any given equivalence class, or they may fail to ensure that the equivalence classes of M are preserved by N. Three approximations are introduced in this section. For each, implications (in terms of errors introduced) are explained, and practical considerations are outlined for application in malware detection.

3.1 Using an non-completed (non confluent) rule set

Since the completion process of KB—which repeatedly adds rules to M^t—may or may not halt, restrictions are normally imposed: 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 seek alternative means. If these also fail, the non-confluent M^t might be considered for the normalizer.

Normalizers whose rule sets are not confluent will be approximate since non-confluence of the terminating system M^t implies that there may be some equivalence classes of M whose members reduce to different normal forms under M^t. More specifically, the members of the malware's equivalence class—the variant offspring—may have more than one normal form. The actual number of these normal forms depends entirely on the specifics of the malicious program and its transformation system.

It may be practically useful to use a non-confluent M^t for malware detection. It is important to note that while using the non-completed set may result in multiple normal forms, so long as condition checking is done correctly it is not possible for a two members of different equivalence classes under M be normalized to the same normal form. This is because terms that were unequal under M are still never joinable under M^t since none of its rules break equivalence. This is important for detectors because even though it is approximate it can never create a false positive. Moreover, even when M^t yields more than one normal form for the malware variants, it may still be able to reduce the number of variants from a *vast* number to a tractable number. In addition, the different normal forms for the equivalent set of P (i.e., that of $P(S)$) may be similar enough that matching them using conventional means (e.g., signatures) may be feasible even if there are many of them. In any case, it is an empirical question as to whether the results are sufficient for whatever purposes the normalization is being used for: the reduction in the number of variants to consider, and the possibly increased similarity of variants may make it possible to use detection methods that would not otherwise work without the normalization.

3.2 Incorrectly evaluating conditions

Ordinarily, the term rewriting process requires that any conditions attached to the rules of N must evaluate to true before they are allowed to fire. These conditions can require knowing certain program properties in order to evaluate them correctly. Such properties include control or data flow, register liveness, and points-to

information [2]. Such information can be challenging to extract [22]. Most interesting extraction problems are undecidable. Known analyses may fail to return accurate results, particularly if obfuscation is used specifically to thwart static analysis techniques [11]. 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.

Such a normalizer will be approximate because it may fire a rule when its condition does not hold, in which case the program semantics may be altered. Such changes can result in multiple normal forms for a given equivalence class. In addition, the changes can alter the equivalence class of the program. In some cases, therefore, a program in the malicious class might be transformed into a non malicious class, and vice versa. In some cases a default rule or simple heuristic might keep the number of incorrect rule applications tolerably small.

It may be practically useful to use such a normalizer for malware detection. In some cases the improper rule firings may not change equivalence classes between malicious and non-malicious, meaning no false matches occur. Moreover, the false matches might fall primarily into only one class (false positives or negatives) and there may be a greater tolerance in practice for that class of error. Finally, it is a practical issue of whether the error rates for the normalizer are suitable enough for application. The practical implications for multiple normal forms are the same as when using an non-completed rule set.

3.3 Priority scheme

If a TRS is confluent then rule application order does not affect the eventual result. Conversely, in a non-confluent system there exist different rule application orders that yield distinct terms which are not joinable. According to Visser [32], a *rule application strategy* can be imposed on a non-convergent TRS to make it behave like a convergent one. If successful, the strategy chooses rule orderings that yield only one normal form for any given equivalence class. This possibility is important if one wishes to use a non-convergent rule in cases where the correctness of the condition checks cannot be guaranteed. The rule application strategy may allow one to choose the ordering of rules that minimize the extent of errors introduced by the approximation. This motivated our design and use of a priority scheme that seeks to reduce the likelihood of false matches due to non-checking of conditions.

Our proposed priority scheme 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. For the rule set in Table 2, $N'_U = \{N_1, N_4, N_6\}$ and $N'_C = \{N_2, N_3, N_5\}$. When considering a rule from N'_C, assume the system uses a fallible or heuristic condition check; for instance, perhaps the condition is not checked at all but is instead assumed always to be true. A suspect code segment is normalized with respect to N' by giving priority to rules of N'_U over the rules of N'_C. That is, 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 (potentially) semantics-altering transformation is applied. The intent is to avoid the case where a conditional rule should have fired before an unconditional rule. Such error conditions occur on critical overlaps between rules of N'_U and N'_C. They will occur more frequently when conditions are not being precisely evaluated since, if the condition were evaluated correctly, the rules might not ever overlap critically. The priority scheme does nothing to avoid the problems caused when the overlaps are within either N'_C or N'_U; using this priority scheme with an exact condition checker, therefore, cannot be expected to improve the approximation.

The practical implications for using normalizers with such a priority scheme are the same as for using non-completed rule sets and approximate checking of conditions. That is, multiple normal forms may be produced for equivalence classes, and false matches may occur due to erroneous application of rules that join terms that are not joinable under $\overset{\star}{\longleftrightarrow}$. These approximations occur whenever the priority scheme does not result in the correct rule application order, or when the condition is incorrectly calculated.

4 Case study

A case study was performed in order to explore the feasibility of the introduced normalization techniques in realistic settings, and to help quantify the impact of the proposed approximation techniques. For the study we

Normalizer	Rule Set	Priority Scheme	Condition checking?
I^0	N_0	no priority scheme	no
I^0_p	N_0	scheme of Section 3.3	no
I^1_p	N_1	scheme of Section 3.3	no

Table 3 The three normalizers used in the study

selected a self-mutating virus and then constructed several normalizers for it using the different methods outlined in previous sections. Variants of the virus were collected by allowing it to replicate in a controlled environment, and the normalizers were applied to the variants. Measures were then taken of the number of normal forms and the amount of differences between equivalent normal forms. These measures were then inspected for indications of the impact of the approximations, particularly with respect to their likely practical utility.

4.1 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]. The length of this sample matches the published size of the "first generation" variant according to Symantec's report [28]. We refer to this variant as the *Eve*. The engine of this virus is a relatively sophisticated one—it is the first entry listed by Ször in his text on anti-virus defense as under the section "More Complex Metamorphic Viruses and Permutation Techniques" [29]. It substitutes instructions with equivalent code segments, inserts irrelevant code at some sites, and replaces immediate operands with arithmetic expressions computing them [20]. By calculating the number of distinct choices for register and constant assignments in the places these can vary, we conservatively estimated that the 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 behavior of the payload of this particular virus makes it possible for some emulation-based techniques to detect its variants [30].

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 mutation 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 form-based signatures for the entire viral set. Third, `W32.Evol`'s mutation engine uses a conditional transformation system that contains critical overlaps; this makes `W32.Evol` a suitable candidate for illustrating and evaluating the normalization approaches. Furthermore, the conditions attached to certain rules require knowledge of register liveness in order to be evaluated. Since register liveness is undecidable (and costly to even approximate) it is a realistic situation in which approximation may be required.

Over 50 offspring were generated spanning 6 generations. 26 of these were selected, taking multiple samples randomly from each generation, except for the Eve and 6th generation, for which there was but one sample to choose from.

4.2 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 of Section 2.3 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. A second normalization set N_1 was constructed by manually adding rules to complete the rule set with respect to the priority scheme of Section 3.3. That is, when using the priority scheme for rule application, the system is convergent. In total, two rules were added. We selected this completion approach because it was apparent that the Knuth-Bendix procedure would not terminate on the rule set, as each rule addition created new critical overlaps.

Three prototype normalizers were implemented using the `TXL` [12] system version 10.4 (2005/01/05). These are named I^0, I^0_p and I^1_p; they differ in the rule sets used and the priority scheme employed. I^0 and I^0_p both used the N_0 rule set, i.e., the non-completed one. I^0_p and I^1_p both used the priority scheme. None implemented condition checking. Table 3 summarizes these normalizers. Additional implementation information may be found in Mathur [23].

Each of the normalizers was applied to the 26 variant samples, and three different groups of measurements were collected. The first group relates to the sizes of the normal forms. These are: **ASNF**, the average length

of the normal forms, and **MSNF**, the maximum size of the normal forms. Both measures are defined in terms of instructions and are averaged over a given generation. The second group relates to how different the normal forms are, on average, from the normal form of the Eve sample. These are: **LNC**, the number of lines, on average, that the normal forms differ (as measured by the common program `diff`), and **PC**, the average raw percentage of sequence commonality between the normal form of the Eve, and the normal form of the sample variant, that is, they list the average of $ASNF(Eve)/ASNF(x)$ for all samples x within any given generation. The third group are simple performance measures of execution time (**ET**) and the count of the number of rule applications performed during normalization (**TC**).

4.3 Results

Table 4 lists the results split into three sections. The top section lists the measures relating to differences in normal forms compared to the Eve's normal form, the middle section provide measurements of the normal forms created using the prioritized normalizer, and the bottom section provides execution information for this prototype. I_p^1 was convergent: all variants in all generations reduced to the same 2,166-line normal form. As a result, the measures are not listed in Table 4. The running times were similar for all prototypes, so only the prioritized version is listed.

	Eve	**2**	**3**	**4**	**5**	**6**
ASO	2,182	3,257	4,524	5,788	6,974	8,455
LNC I^0	0	0	108	316	803	1129
LNC I_p^0	0	0	10	16	24	37
PC I^0	100.00	100.00	95.25	87.27	72.96	65.74
PC I_p^0	100.00	100.00	99.54	99.27	98.90	98.32
MSNF I_p^0	2,167	2,167	2,184	2,189	2,195	2,204
ASNF I_p^0	2,167	2,167	2,177	2,183	2,191	2,204
ET I_p^0	2.5	3.0	4.3	6.3	8.0	11.2
TC I_p^0	16	533	980	1,472	1,902	2,481

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

Table 4 Results of normalizers on various `W32.Evol` generations

4.4 Discussion

Because the case study is limited, any generalizations must be tentative. 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 mutation engines. Other complex viruses, like RPME, Zmist, Benny's Mutation Engine, Mistfall, Metaphor, etc [5, 31, 34, 35] have transformations similar to that of `W32.Evol`, and it appears likely that for some subset of self-mutating programs, a syntactic normalizer built according to the strategy in Section 2.2 will normalize all variants sufficiently well for ordinary signature matching to succeed in practice.

Regarding feasibility, Table 4 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 the priority scheme could not 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. We manually inspected the differences between the normal forms of various generations and the normal form of the Eve (and of the Eve itself) to assess the amount and type of differences. We found that incorrect rule application occurred at three and two sites for the I_p^0 and I_p^1 prototype, 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 the impact of the priority scheme, it appears that it can make a relatively sizable difference in the amount of code that is incorrectly normalized, particularly for later generations of Eve. For instance, for the fifth generation, the priority increased the accuracy over 25 percentage points over the non-prioritized version. This relates to the number of sites at which the conditional rule may be fired incorrectly.

Regarding practicality, the timing information reflects the fact that our prototypes are proofs-of-concepts: they work on ordinary textual 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. Moreover, it may happen that the upper end of the asymptotic curve is not problematic in certain practical cases where input growth is constrained. For instance, while `W32.Evol` always grows in size, growing very large is not a good survival strategy for malware, so some recent self-mutating malware include transformations that try to keep the size of their code within reasonable limits by applying 'code-shrinking' transforms [29].

One might find fault with the fact that the normalization technique depends upon having a formalization of the specific mutation engine. This means the technique cannot be expected to find malicious programs for which the mutation engine is unknown. While this certainly is an issue, the limitation may be tolerable. Signature-based techniques generally cannot detect novel malware either, and signatures are instance-specific rather than family specific, yet these techniques have proved to be a useful technology when the signature database can be updated at the same rate as malware production. Given that new mutation engines appear at a much lower rate than ordinary malware instances, an engine-specific scheme seems at least plausible.

One might also argue that modeling the mutation engine can be too difficult, or too costly. In response, we first note that mutation engines evolve slowly—much slower than the worms and viruses themselves [29], so the number of new mutation engines released in a year is low enough to make them amenable for such analysis. Second, the mutation 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.

5 Relations to other work

Normalization of input is a concern common to many domains, including databases, and text and speech processing. Program normalization is commonly performed on source code in the context of plagiarism detection and so-called "code clone" detection. The normalization aids matching copied code by removing detail that is considered unimportant. Variable names, white space, and syntactic forms are all commonly normalized. `CCFinder` [16], for example, normalizes input code by *tokenizing* many features. As with the present work, each of these normalizations attempt to defeat attempts at obfuscating the fact that two programs are variants in disguise. The main difference is that these normalizations generally attack relatively superficial differences in the code, and are not expected to aid in normalizing comprehensive semantics-preserving obfuscating transformations performed on the code.

In the plagiarism and code clone literature, some more complicated obfuscations are accounted for by abstracting to a comparison domain in which the differences can be normalized out (e.g., Baxter *et. al* [4]). Müller *et. al* [24] present an approach to matching programs which attempts to account for obfuscating transformations. They define a program similarity metric based on the similarity of specific tree-structured data flow structures. Unlike the approach in the present paper, their approach is motivated by the supposition that these data flow structures will be relatively constant even when the program has been transformed via obfuscators. They do not explicitly consider transformations in the style of known mutation engines. However they also define normalizations on their structures which they hope will allow successful matches to variants created through the obfuscators.

Specifically malicious program normalization approaches are surveyed in Section 2. The works by Lakhotia *et. al* [21], Bruschi *et. al* [6], and Christodorescu *et. al* [9] share several 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 *a priori* knowledge of the mutation engines 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. They leave open the possibility of defeat by introducing variations that ensure the optimizations yield different optimized programs. In contrast, the present work is specific to a mutation engine, but suggests that deeper semantic analysis may not always be necessary. An interesting research question arises as to the tradeoffs and benefits of general normalization rules versus ones targeted towards specific mutation 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* [8] and Kruegel *et al.* [19] proposed the use of static program analysis methods for detecting potentially obfuscated variants of specified behavior patterns. Works in this vein constitute pursuits of a more capable pattern matcher, rather than a normalization approach. The normalization and behavior-match approaches are complementary and can be used together.

6 Conclusions

This paper presents an approach to construct a normalizer for a particular class of mutating malware by leveraging concepts and results from term rewriting literature [3]. It was shown that mutating 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 mutation 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 mutating. 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 automatically-constructed mutants. That said, it was noted that not every rule set can be feasibly completed using an automated completion method. An analysis of the conditions when the completion procedure breaks down revealed attack points 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.

Acknowledgments

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

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. I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the 1998 International Conference on Software Maintenance (CSM '98)*, pages 368–377, 1998.
5. Benny. Benny's metamorphic engine for Win32. `vx.netlux.org/29a/29a-6/29a-6.316`.

6. 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.
7. D. Chess and S. White. An undetectable computer virus. In *In Proceedings of Virus Bulletin Conference*, Sept. 2000.
8. 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.
9. M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, Nov. 2005.
10. F. Cohen. Computational aspects of computer viruses. *Computers & Security*, 8(4):325–344, 1989.
11. C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, Aug 2002.
12. 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.
13. M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.
14. É. Filiol. Metamorphism, formal grammars, and undecidable code mutation. *International Journal of Computer Science*, 2(1), Nov. 2007.
15. Z. hong Zuo, Q. xin Zhu, and M. tian Zhou. On the time complexity of computer viruses. *IEEE Transactions on Information Theory*, 51(8), Aug. 2005.
16. T. Kamiya, S. Kusumoto, and K. Inoue. A multilinguistic token-based code clone detection system for large scale source code. *Transactions on Software Engineering*, 8(7):654–670, 2002.
17. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In K. Julisch and C. Kruegel, editors, *Intrusion and Malware Detection and Vulnerability Assessment: Second International Conference (DIMVA 2005)*, Lecture Notes in Computer Science, page 174. Springer-Verlag, 2005.
18. 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.
19. 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.
20. A. Lakhotia, A. Kapoor, and E. U. Kumar. Are metamorphic viruses really invincible? - part II. *Virus Bulletin*, pages 9–12, January 2005 2005.
21. 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.
22. W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
23. R. Mathur. Normalizing metamorphic malware using term-rewriting. Master's thesis, Center for Advanced Computer Studies, University of Louisiana at Lafayette, Dec. 2006.
24. A. J. Müller and T. Shinohara. On approximate matching of programs for protecting libre software. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, pages 21–36, New York, NY, USA, 2006. ACM Press.
25. C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):47–51, Jan 1997.
26. P. K. Singh, M. Moinuddin, and A. Lakhotia. Using static analysis and verification for analyzing virus and worm programs. In *Proceedings of the 2nd European Conference on Information Warfare and Security*, pages 281–292, 2003.
27. E. Skoudis. *Malware: Fighting Malicious Code*. Prentice-Hall, 2004.
28. Symantec. W32.Evol security response writeup. `www.symantec.com/security_response/writeup.jsp?docid=2000-122010-0` checked 2007/08/15.
29. P. Ször. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.
30. P. Ször and P. Ferrie. Hunting for metamorphic. In *11th International Virus Bulletin Conference*, 2001.
31. The Mental Driller. Metamorphism in practice. `vx.netlux.org/29a/29a-6/29a-6.205`.
32. 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.
33. A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. The design space of metamorphic malware. In *Proceedings of the 2nd International Conference on Information Warfare*, Monterey, CA, U.S.A., Mar. 2007.
34. Z0mbie. Automated reverse engineering: Mistfall engine. `vx.netlux.org/lib/vzo21.html`.
35. Z0mbie. Some ideas about metamorphism. `vx.netlux.org/lib/vzo20.html`.