Normalizing Metamorphic Malware Using Term Rewriting

Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, & Arun Lakhotia
University of Louisiana at Lafayette
arun@louisiana.edu

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 paper 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 in cases where exact solution is not feasible/possible. These are based on relaxing the confluence and equivalence preservation requirements. A simple priority scheme is outlined for reducing the number of false matches 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`".

1 Introduction

Malicious programs like worms, viruses, and trojans are collectively known as "malware" [16]. Malware is called "metamorphic" when it is able mutate or to construct offspring that differ from itself [20]. The collection of mechanisms a malware uses to mutate 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 [20].

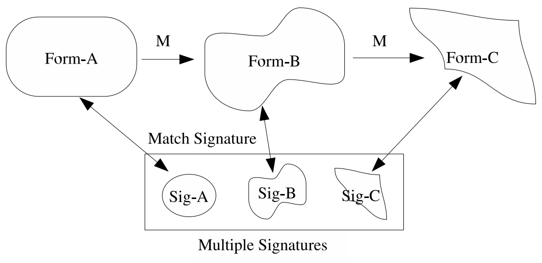

Figure 1: Detection using signature for each

The reason metamorphic engines were developed, of course, was to enable a malware to avoid detection by malware scanners [14]. 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 visualized in Figure 1. While `W95.RegSwap` could be matched using a more general signature [20], more powerful metamorphic engines could create many more variants—possibly unbounded quantities—each of which bear little obvious resemblence to the original. Indeed, metamorphic engines have been evolving in such a direction [10, 20]. Recent thought on the matter is that current scanning techniques, by themselves, will not be able to counteract more powerful self-transformation methods [19].

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. If the normalization is successful the prevalent signature-matching techniques can be leveraged to detect, as visualized in Figure 2.

This paper shows how to construct normalizers for metamorphic programs. Theories and techniques from the field of *term rewriting* [3] are employed. Term rewriting is a general model of computation involving sets of *rules*. A normalizer can be constructed from a term rewiting 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. The paper's main contributions are in (a) formalizing the normalizer construction problem (NCP) using term rewriting, (b) proposing a strategy for solving this problem, and (c) demonstrating that certain relaxations of correctness conditions may still yield an effective normalizer while avoiding potentially costly or fallible static program analyses.

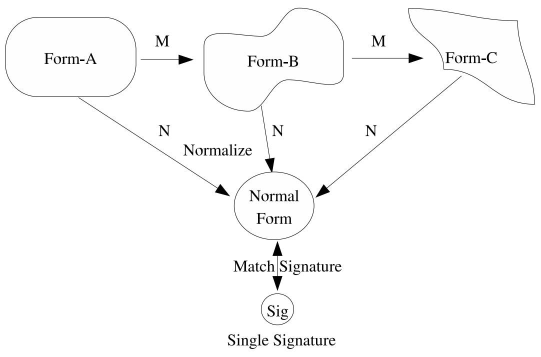

Figure 2: Detecting all variants using a normal form

Section 2 defines the NCP and introduces the necessary term rewriting definitions, it also lists the critical problems involved in creating a useful normalizer. Section 3 defines a strategy for solving the NCP. First, a method for *reorienting* rules is described which 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*. Section 4 discusses how to generalize the strategies from Section 3 in cases where one or more of the correctness properties cannot be assured. Section 5 reports on a case study using the normalizers for `W32.Evol` created using the proposed approach. It demonstrates the feasibility of the strategy and the potential usefulness of the approximated normalizer. Section 7 concludes the paper.

2 The NCP

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 [19]. 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 their code. This development had been expected. Cohen had observed that a program can reproduce either exactly or with "mutations" [8]. It is typically a goal for a metamorphic engine to ensure its mutations are programs that are 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 undecideable problem in the general case. Moreover, Chess and White [6] showed that detecting that a program has the property than any one of its instances ”potentially produces” any other instance is also undecidable. Spinellis [17] 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. Oue approach to handle metamorphics is to reduce the number of variants that need to be considered by *normalizing* the programs.

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 [13]. Unfortunately, while their system can reduce the number of variations it may still be an enormous number (10^{20}) to have in a signature database. The reason being specific transformations of the metamorphic engine are not considered, so the transformations are 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 it's outputs to a normal form. 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 virus `W32.Evol`. Examples of its rewrite rules are shown in Figure 3. The disassembly of the parent's code is shown in the left column and the corresponding transformed offspring code in the right column, the parts of the code changed in the offspring are shown in bold face.

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

Figure 3: 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. In the transformed code, the register `ecx` needed to be disturbed in order to change the `mov` immediate instruction. Any existing value of `ecx`, however, is preserved by the `push` and subsequent `pop` immediately surrounding the two middle `mov` instructions. If a malware 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, but only under the condition that the register `eax` is not live at the point where transformation takes place. In example (c) a "junk" statement (ie. one with no effect on the computation) `mov eax, 0x09` is inserted. It is also semantics preserving under the same condition.

The transformation examples in Figure 3 help introduce several potential problems in developing a normalizer. 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? Can we always perform condition checks? So 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. In the following subsections we overview some necessary background from the term rewriting literature and show that we can formalize the NCP.

2.2 Term rewriting background

Term rewriting systems are widely studied and good references exist (e.g., Baader and Nipkow [3]); this section only briefly reviews definitions and results needed for later sections.

Term rewriting system (TRS). A TRS, T, consists of a set of *rewrite rules*. A rule is denoted $s \rightarrow t$, 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). An *atomic* term is one that does not contain subterms. A *ground* term is one that does not contain variables.

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

Figure 4: Example rule set transforming arithmetic expressions

Reduction relation ($\to_T$). Any term rewriting system T *induces a relation* $\to_T$ on terms, also represented as $\to$ where obvious. Given terms s and t, $\to_T$ is defined as follows: $s \to_T t$ holds if and only if, 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; that is, applying rule $s' \to t'$ to s transforms it into t. A conditional TRS may have conditions attached to the rules $(p|R)$. This means that rule R may be applied only when the condition p holds.

Equivalence relation ($\overset{\star}{\longleftrightarrow}$). The *equivalence relation*, $\overset{\star}{\longleftrightarrow}$, is the reflexive symmetric transitive closure of the relation $\to$ induced by T, 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 $\to_T$, 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. 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.

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

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

Convergence. A TRS is *convergent* if it is confluent and terminating. If a TRS is convergent then determining membership in an equivalence class is decidable since 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.

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). Figure 5 depicts an example of how rules might typically be written in a term rewriting formalism. For example, in this case mov is the function, $reg1$ and imm are variables.

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

Figure 5: Sample metamorphic transform as a rewrite rule

In modeling the metamorphic engine each rule should 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).[1] 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). Using a scheme like this we were able to formally model the metamorphic engine in `W32.Evol`. Henceforth, for simplicity we will write rules in simple assembly language rather than as shown in Figure 5.

The normalizer construction problem (NCP) Suppose one has formalized a metamorphic engine as a TRS, as described above. Call this TRS 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 TRS 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 no 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. However M can be useful if it can be modified to create a new TRS which is convergent. The problem of doing this is what we call NCP: the

[1]Modulo issues considered unimportant, such as code size and location.

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.

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 termination, and then a *completion procedure* [11] which seeks to ensure confluence. The completion procedure 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 to change the direction of a transformation. For example, $x \rightarrow y$ is reoriented by turning it into $y \rightarrow x$. Since $x \rightarrow y$ means that x is considered equivalent to y, reorienting the rule will not change the equivalence classes induced by $\rightarrow$. 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. It 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 $M = \{a \rightarrow b, b \rightarrow a\}$, then reorienting both rules will yield a rule set that is still non-terminating. 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 $>$, then a M' can be constructed such that $\forall a \rightarrow b \in M'.a > b$. Each rule in this M' would serve to reduce the terms and since $>$ is well founded then M' is guaranteed to always terminate [3].

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 then 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 6 shows an example of reorienting a set of rules. Figure 6(a) shows the initial rules for M, and 6(b) shows the M^t that results from reorienting M according to the length-reducing lexicographically first ordering. These 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 [3]. If the confluence test is successful then it would mean that the system is convergent. From previous step we already know the equivalence constraint is satisfied. Which means that both the constraints specified in previous section are satisfied and such a M^t will solve the NCP, and the output will be the desired N. If the confluence test fails then M^t would contain what are called *critical overlaps* [3, 11] in the rules. The left hand sides of a pair of, not necessarily distinct, rules is said to critically overlap if the prefix of one matches the suffix of the other, or if one is a subterm of the other.

For example, two critical overlaps in M^t shown in Figure 6(b) are: (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

Rule	Post Condition		l_i
M_i	C_i	$\rightarrow$	r_i
M_1	T		`mov [reg1+imm], reg2`
		$\rightarrow$	`push eax`
			`mov eax, imm`
			`mov [reg1+eax], reg2`
			`pop eax`
M_2	eax is dead		`push imm`
		$\rightarrow$	`mov eax, imm`
			`push eax`
M_3	eax is dead		`push eax`
		$\rightarrow$	`push eax`
			`mov eax, imm`
M_4	T		`NOP`
		$\rightarrow$	

(a) M, the original rule set

Rule	Post Condition		l_i
M_i^t	C_i	$\rightarrow$	r_i
M_1^t	T		`push eax`
			`mov eax, imm`
			`mov [reg1+eax], reg2`
			`pop eax`
		$\rightarrow$	`mov [reg1+imm], reg2`
M_2^t	eax is dead		`mov eax, imm`
			`push eax`
		$\rightarrow$	`push imm`
M_3^t	eax is dead		`push eax`
			`mov eax, imm`
		$\rightarrow$	`push eax`
M_4^t	T		`NOP`
		$\rightarrow$	

(b) M^t, the reoriented rule set

Figure 6: Example application of reorienting procedure

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.

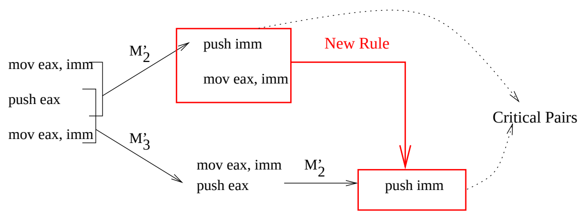

Figure 7: Completion step for M_2^t and M_3^t

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 adds 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 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.

Rule	Post Condition		l_i
N_i	C_i	$\rightarrow$	r_i
N_1	T		`push eax`
			`mov eax, imm`
			`mov [reg1+eax], reg2`
			`pop eax`
		$\rightarrow$	`mov [reg1+imm], reg2`
N_2	eax is dead		`mov eax, imm`
			`push eax`
		$\rightarrow$	`push imm`
N_3	eax is dead		`push eax`
			`mov eax, imm`
		$\rightarrow$	`push eax`
N_4	T		`NOP`
		$\rightarrow$	
N_5	eax is dead		`push imm`
			`mov eax, imm`
		$\rightarrow$	`push imm`
N_6	T		`push imm`
			`mov eax, imm`
			`mov [reg1+eax], reg2`
			`pop eax`
		$\rightarrow$	`mov eax, imm`
			`mov [reg1+imm], reg2`

Figure 8: Rule set N, the result of the completion procedure

In our case we applied the completion procedure of

Knuth and Bendix [11]. This procedure successfully completed the full M^t for `W32.Evol`, giving the 'ideal' normalizing set. The procedure searches for critical overlaps and then adds rules which 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 Figure 6 terminated, giving us the desired N as appears in Figure 8.

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, N would also come with a guarantee that each equivalence class has a distinct normal form, eliminating the potential for false matches. Two more 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 which 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. A strategy is outlined for dealing with the approximation at the term rewriting system implementation level by implementing a priority scheme.

4.1 Failure to complete

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. So 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' which the analyst believes will produce a single normal form under the priority scheme discussed in Section 4.3. There are, ofcourse, risks of manual errors.

4.2 Incorrect condition evaluation

Conditional analysis requires complicated analyses including control flow and points-to information [2]. In the general case these costs are likely to be exorbitant for an ordinary desktop malware scanner to perform. Perhaps worse yet, it might not be feasible to calculate the required condition information correctly for obfuscated malware using static analysis techniques.

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., $[x]_M \neq [x]_{N'}$ and we might have false negatives and false positives.

4.3 Priority scheme

As surveyed by [22], in the case when system is not convergent one can define a rule application strategy to reach the optimal solution. We have experimented with a simple priority scheme, that tries to compensate for the above approximations, for rule application which 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 8. 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 done 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. 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 it's application.

5 Case study

Even though Knuth and Bendix completion procedure terminated in the case of `W32.Evol`, a case study was performed to evaluate the effectiveness of the two approximated solutions described in Section 4.

5.1 Subject and preparation

`W32.Evol` was selected as a test subject. We obtained a copy of the 12288-byte long first generation sample from the "VX Heavens" archive [1]; henceforth we will refer to this sample as the "Eve" sample. `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 [20]. Some of its transforms 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 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 representative example of a virus with a sophisticated mutating engine. Third, as we noted in the examples from previous sections, `W32.Evol`'s metamorphic engine includes conditional transformations and critical overlaps. This provides a good test for the sufficiency of the proposed NCP solution strategies.

5.2 Materials and Protocol

The procedure we used is as follows: 1) construct a set of term rewriting rules M that represent `W32.Evol`'s mutating operations; 2) construct the normalizing set N by reorienting the rules in M as described in Section 3, but not completing it; 3) implement the N in a prioritized term-rewriting system that does not check conditions, as described in Section 4.3 (this is prioritized with no completion); 4) complete N in an ad-hoc manner to get N_C and implement in a second prototype using the priority scheme and not checking conditions (this is prioritized with ad-hoc completion); and 5) feed samples to the two prototypes to generate the normal forms, and collect results

Extraction of the mutating rules was done by hand using a combination of code reading and code tracing in a debugger. The normalizing set consisted of **55** rules of which **15** did not participate in any overlap. Prototype transformers for N and N_C were implemented using the `TXL` programming language [9].[2] We choose 26 different variants spread across six generations to perform the tests.

5.3 Results

Table 1 quantifies the results from the prioritized with no completion normalization prototype. Row four of Table 1 indicates the average length of the normal forms of the variants after normalizing with the prioritized scheme. Note that the normal form of the 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 wrongly since we are not performing any analysis to check for conditions. Row five simply lists how many lines, on average, differ. It was calculated using the `diff` program and counting the number of lines that differed from the normal form of the Eve. Row six shows the average raw percentage of sequence commonality between the normal form of the Eve, and the normal form of the sample variant. Rows seven and eight of Table 1 record execution information for the prototype.

The second prototype N_C, prioritized with ad-hoc completion, created a single normal form of 2166 lines for all samples.

5.4 Discussion

The results suggest that for a realistic metamorphic virus, even by using these approximated methods, it is possible to effectively normalize the variations sufficiently well that simple signature matching can be made effective. We were able construct a convergent normalizer for `W32.Evol`, which is not only bound to give a single normal form for all its

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

1	Generation	Eve	2	3	4	5	6
2	**ASO**	2182	3257	4524	5788	6974	8455
3	**MSNF**	2167	2167	2184	2189	2195	2204
4	**ASNF**	2167	2167	2177	2183	2191	2204
5	**LNC**	0	0	10	16	24	37
6	**PC**	100.00	100.00	99.54	99.27	98.90	98.32
7	**ET**	2.469	3.034	4.264	6.327	7.966	11.219
8	**TC**	16	533	980	1472	1902	2481

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

Table 1: Results using prioritized, non-completed normalizer

variants but also guarantees that there will be no false positives or negatives. Even though we did not implement the convergent normalizer, these claims directly follow from term rewriting theory. The main question now about the normalization scheme is whether the approximation methods would be sufficient in normalizing realistic metamorphic programs, containing difficult overlapping and conditional rules. Table 1 shows that even the prioritized scheme without completion creates, more than 98%, 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 variants of `W32.Evol`. Better yet, the priority scheme with ad-hoc completion gave single normal form.

As noted in the previous section, the normal forms of both the normalizers are smaller in size than the original program partly because of incorrect application of conditional rules. This may result in semantically different programs (normal forms) which opens up the possibility of false matches. We expected this result as priority scheme cannot be a complete substitute for conditional analysis. However, out of the 2182 original lines of code, by manual inspection we found incorrect conditional application to occur at two and three sites for priority without and with ad-hoc completion respectively. The probability of finding programs semantically different on only three lines from `W32.Evol` on an actual users desktop is extremely low and such false matches should not be a problem. This result also shows the effectiveness of the priority scheme.

These are just proof of concept normalizers that we have implemented which work on assembly code instead of binaries. So, the timing information is just to give the reader an idea of the size vs time curve, actual values can be made lower using efficient implementations. Also, `W32.Evol` always grows in size, which is not good for the virus itself as the code for higher generations becomes too large, recent metamorphic malware try to keep the size of their code in reasonable limits (almost constant) by applying some ‘code-shrinking’ transforms, which means that the time to normalize will remain in reasonable limits as well (almost constant), even for their higher generations, since its a function of code size.

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. Signature matching already 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. First, we note that metamorphic engines evolve slowly—much slower than the worms and viruses themselves [19], the number of new metamorphic engines released in a year is low enough to make them amenable for such analysis. Second, there are metamorphic engine libraries available, since its not an easy technique even for malware authors to implement, a lot of malware just use existing libraries for their metamorphic engine. This means that we just need to construct normalizers for these relatively few metamorphic libraries to be able to catch all malware families who use it.

It is not possible to generalize from a single case study with any confidence. Nonetheless other complex metamorphic viruses like RPME, Zmist, Benny’s Mutation Engine, Mistfall, Metaphor, etc [4, 21, 23, 24] have transformations similar to that of `W32.Evol` and it appears likely that for some subset of metamorphic programs, a syntactic normalizer built according to the strategy in Section 4 will normalize all variants sufficiently well for ordinary signature matching to succeed well enough.

6 Relations to other work

Ször and Ferrie [20] 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 retargeting a binary to the host CPU [18]. However emulation based techniques are known to be slow and vulnerable to well known anti-emulation techniques. More recent work by Ször [19] gives a exhaustive list of metamorphic transformations and suggests that most current technologies would not be able to handle the threat posed by metamorphic viruses. Perriot and Bruschi *et. al* suggest using code optimization to normalize metamorphic variations [5, 15], similar to Lakhotia and Mohammed [13], and is similarly limited. The normalizers we construct are able to determine whether a program belongs to the equivalence class of a semantically equivalent program. It therefore relates to other efforts that try to find malicious behaviors in arbitrary code by looking for semantically or behaviorally equivalent fragments. An example of such an approach is given by Christodorescu *et. al* [7]. The work by Kruegel *et al.* [12] 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 (CFGs). However malware tend to be obfuscated so it is not easy to extract CFGs from unfriendly code.

7 Conclusions

This paper defined the NCP and defined strategies that can be employed to construct normalizing transformers. The normalizer construction strategy can be fallible in the sense that it may not lead to a convergent normalizer, even though it did for `W32.Evol`. But the case study provided a demonstration that in certain cases this may not matter, since multiple normal forms may not present a problem so long as the normal forms associated with the malicious program are similar enough.

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. Normalization, nonetheless, 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. Our case study demonstrates that even approximated normalizers might yield sufficiently good results, meaning they have some potential to be turned into efficient implementations in real malware scanners. This strategy also has the potential to be completely automated.

References

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

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

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

[4] Benny. Benny's metamorphic engine for win32. `vx.netlux.org/29a/29a-6/29a-6.316`.

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

[6] D. Chess and S. White. An undetectable computer virus. In *In Proceedings of Virus Bulletin Conference*, Sept. 2000.

[7] 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.

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

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

[10] M. Jordon. Dealing with metamorphism. *Virus Bulletin*, pages 4–6, 2002.

[11] 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.

[12] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *8th Symposium on RAID*, Lecture Notes in Computer Science. 2005.

[13] A. Lakhotia and M. Mohammed. Imposing order on program statements and its implication to av scanner. In *11th IEEE WCRE*, pages 161–171, Nov. 2004.

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

[15] F. Perriot. Defeating polymorphism through code optimization. In *Proceedings of Virus Bulletin 2003*, 2003.

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

[17] D. Spinellis. Reliable identification of bounded-length viruses is np-complete. *IEEE Transactions on Information Theory*, 49(1):280– 284, Jan. 2003.

[18] A. E. Stepan. Defeating polymorphism: Beyond emulation. In *Virus Bulletin Conference*. Virus Bulletin, October 2005.

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

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

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

[22] 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.

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

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