UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
DI.C.O. – DIPARTIMENTO DI INFORMATICA E COMUNICAZIONE

Using Code Normalization for Fighting Self-Mutating Malware

Danilo Bruschi, Lorenzo Martignoni, Mattia Monga
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39/41, 20135 Milano - Italy
{bruschi,martign,monga}@dico.unimi.it

RAPPORTO TECNICO N. 08-06

Using Code Normalization for Fighting Self-Mutating Malware

Danilo Bruschi, Lorenzo Martignoni, Mattia Monga
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39/41, 20135 Milano - Italy
{bruschi,martign,monga}@dico.unimi.it

Abstract

Self mutating malware has been introduced by computer virus writers who, in '90s, started to write polymorphic and metamorphic viruses in order to defeat anti-virus products. In this paper we present a novel approach for dealing with self mutating code which could represent the basis for a new detection strategy for this type of malware. A tool prototype has been implemented in order to validate the idea and the results are quite encouraging, and indicate that it could represent a new strategy for detecting this kind of malware.

1 Introduction

Most of malware detection mechanisms are based on pattern matching, i.e. they recognize malware by looking for the presence of *malware signatures*[1] inside programs, IP packet sequences, email attachments, etc. One of the most obvious strategy adopted for circumventing them is based on the notion of self mutating malware, i.e. malicious code which continuously changes its own code and, consequently, makes signature based detection, completely useless.

Self mutation is a particular form of code obfuscation[2], which is performed *automatically* by the code itself. Some well known self mutating programs are METAPHOR [2], ZMIST [16] and EVOL [22]. The diffusion of this type of malware is quite worrying as, in some papers recently appeared in literature [8, 7], it has been shown that current commercial virus scanners can be easily circumvented by using simple obfuscation techniques.

In this paper we present a novel approach for dealing, with self mutating code which could represent the basis for a new detection strategy for this type of malware. Even if in [6] it has been proven that, in the general case, perfect detection of self-mutating code is a non computable problem, such a result leaves room for the implementation of techniques which, even if not perfect, can defeat a significant percentage of self mutating code; the strategy we devised is one of them.

Such a strategy, is based on a static analysis approach for verifying security properties of executable code and it is derived by the following observation (see also [19]). A self-mutating malware has to be able to analyze its own body and extract from it all the information needed to mutate into the next generation, which has to be able to perform the same process, and so on. As the mutation mechanisms adopted has to be particularly efficient, the mutations performed cannot be too complex and, in the general case, it is possible to iteratively reverse the mutation process until an *archetype*[3] has been derived. Such a consideration is enforced by experimental observations in which we have shown that malware produced by self mutating programs is characterized by highly unoptimized code, that is, it contains a lot of redundant and useless instructions. Thus, giving a piece of code M, our strategy for verifying whether is a malware code is the following. Initially we decode M and transform it into a normal form M', and during such a phase we also perform a code optimization by removing from M' all useless and redundant computations. Subsequently we verify whether M' shares common features with the normalized code of a known malware. In order to perform the normalization process we referred to well known code optimization techniques adopted in compilers' construction (for example see [3, 20]).

[1] By malware signature we mean peculiar sequences of bytes (usually represented by a regular expression) which characterize a specific malware with respect to any other program.

[2] Code obfuscation is a set of techniques adopted for transforming a program into an equivalent one which is more difficult to understand yet it is functionally equivalent to the original.

[3] The term archetype is used to describe the zero-form of a malware, i.e. the original and un-mutated version of the program from which other instances are derived.

The comparison between a normalized code is instead realized using *clone detection* techniques [18, 17, 5]. We briefly remember that clone detection is primarily used in software engineering to identify fragments of source code that can be considered similar despite little differences. We adapted such techniques to work on binary code instead of source code, since malware is normally available only as a compiled executable.

We implemented a tool prototype in order to validate our idea and the results are quite encouraging, and indicate that the current stable version of such tool could be considered as a basic component in the design of security architectures.

This paper is organized as follows. In Section 2 we describe mutation techniques adopted by malware writers. In section 3 we describe the optimization techniques we implemented for removing the mutations previously described. In section 4 we provide a brief overview of the clone detection techniques we decide to adopt. In Section 5 we describe the prototype implemented while in section 6 we describe the experiments we performed with our prototype and the preliminary results. Section 7 discusses related works. In the final section some conclusions on the work presented are drawn.

2 Mutation Techniques

The mutation of an executable object can be performed using various types of program transformations techniques, exhaustively described in [11, 12]. For the sake of completeness we report in the following a brief summary of the most common strategies used by a malicious code to mutate itself.

Instructions substitution

A sequence of instructions is associated to a set of alternative sequences of instructions which are semantically equivalent to the original one. Every occurrence of the original sequence can be replaced by an arbitrary element of this set. For example, as far as the `eax` and `ebx` registers are concerned, the following code fragments are equivalent.

Machine instructions	Equivalent form
`mov eax,ecx`	`xor ebx,eax`
`mov ebx,eax`	`xor eax,ebx`
`mov ecx,ebx`	`xor ebx,eax`

Instructions permutation

Independent instructions, i.e., instructions whose computations do not depend on the result of previous instructions, are arbitrarily permutated without altering the semantic of the program. For example, the three statements `a = b * c`, `d = b + e` and `c = b & c` can be executed in any order, provided that the use of the `c` variable precedes its new definition.

Garbage insertion

Also known as dead-code insertion. It consists of the insertion, at a particular program point, of a set of valid instructions which does not alter the expected behavior of the program. For example given the following sequence of instructions `a = b / d`, `b = a * 2`; any instruction which modifies `b`, can be inserted between the first and the second instruction; moreover instructions that reassign any other variables without really changing their value can be inserted at any point of the program (e.g., `a = a + 0`, `b = b * 1`, ...).

Variable substitutions

The usage of a variable (register, memory address or stack element) is replaced by another variable belonging to a set of valid candidates preserving the behavior of the program.

Control flow alteration

The order of the instructions, as well as the structure of the program, is altered introducing fake conditional and unconditional branch instructions such that at run-time the order in which single instructions are executed is not modified. Furthermore, direct jumps and function calls can be translated into indirect ones whose destination addresses are camouflaged into other instructions in order to prevent an accurate reconstruction of the control flow.

3 Normalization Techniques

Code normalization is the process of transforming a piece of code into a canonical form more useful for comparison.

Most of the transformations used by malware to dissimulate its presence (see Section 2) led to a major consequence: the code size is highly increased. In other words, the various mutations of a piece of malware can be thought of as unoptimized versions of its archetype, since they contain some irrelevant computations whose presence has the only goal of defeating recognition. Normalization of a malware aims at removing all the "dust" introduced during the mutation process and optimization techniques can be used to reduce the "dust".

Usually optimization is performed by the compiler to improve the execution time of the object code or to reduce the size of the text or data segment it uses. The optimization process encompasses a lot of analysis and transformations that are well documented in the compilers literature [3, 20]. As shown by [21, 13] the same techniques can also be directly applied to executable in order to achieve the same goals. We deeply investigated such techniques and we found that some of them can be successfully used in order to fight the mutation engines adopted by self mutating malware. In the following we briefly describe such techniques.

Instructions meta-representation

All the instructions of a CPU can be classified in the following categories: (i) jumps, (ii) function calls and returns, and (iii) all the other instructions that have side effects on registers, memory and control flags. In the following we call the instructions in category (iii) *assignments*. Comparison instructions can be considered as assignments because they usually perform some arithmetic on their operands and then update a control register accordingly (e.g. `eflags` on IA-32). In order to ease the manipulation of object code, we used a high-level representation of machine instructions that expresses the operational semantics of every opcode, as well as the registers, memory address and control flags involved. A simple example follows:

Machine instruction	Meta-representation
`pop eax`	`r10 = [r11]`
	`r11 = r11 + 4`
`lea edi,[ebp]`	`r06 = r12`
`dec ebx`	`tmp = r08`
	`r08 = r08 - 1`
	`NF = r08@[31:31]`
	`ZF = [r08 = 0?1:0]`
	`CF = (~(tmp@[31:31]) ...`
	`...`

It is worth noting that even the simple `dec` instruction conceals a complex semantics: its argument is decremented by one and six control flags are updated according to the result (the above example is reduced for space constraints).

Propagation

Propagation is used to propagate forward values assigned or computed by intermediate instructions. Whenever an instruction defines a variable (a register or a memory cell) and this variable is used by subsequent instructions without being redefined, then all its occurrences can be replaced with the value computed by the defining instruction. The main advantage of propagation is that, it allows to generate higher level expressions (with more than two operands) and eliminate all intermediate temporary variables that were used to implement high-level expressions. The following code fragment sketches a simple scenario where, thanks to propagation, a higher level expression is generated:

Before propagation	After propagation
`r10 = [r11]`	`r10 = [r11]`
`r10 = r10 | r12`	`r10 = [r11] | r12`
`[r11] = [r11] & r12`	
`[r11] = ~[r11]`	
`[r11] = [r11] & r10`	`[r11] = (~([r11] & r12)) & ([r11] | r12)`

Dead code elimination

Dead instructions are those whose results are never used. For example, if a variable is on the left hand side of two assignments but it is never used between them (i.e., does not appear on the right hand side of an expression), the first assignment is consider dead or useless (the second is called a *killing definition* of the assigned variable). For example, the first assignment of the little instructions sequence shown above (`r10 = [r11]`) is useless after propagation. Dead instructions, without side effects, can be safely removed from a program because they do not contribute to the final computation.

Algebraic simplification

Since most of the expressions contain arithmetical or logical operators, they can sometimes be simplified according to the ordinary algebraic rules. When simplification are not possible, variables and constants could be reordered to enable further simplifications after propagation. The following table shows some examples of the rules that can be used to perform simplification and reordering (c denotes a constant value and t a variable):

Original expression	**Simplified expression**
$c_1 + c_2$	the sum of the two
$t_1 - c_1$	$-c_1 + t_1$
$t_1 + c_1$	$c_1 + t_1$
$0 + t_1$	t_1
$t_1 + (t_2 + t_3)$	$(t_1 + t_2) + t_3$
$(t_1 + t_2) * c_1$	$(c_1 * t_1) + (c_1 * t_2)$

Control flow graph compression

The control flow graph can be heavily twisted with the insertion of fake conditional and unconditional jumps. A twisted control flow graph could impact on the quality of the whole normalization process because it can limit the effectiveness of other transformations. At the same time other transformations are essential to improve the quality of the normalization of the control flow graph.

Expressions which represent branch conditions and branch (or call) destinations could benefit from the results of previous transformations: to be able to calculate the result of a branch condition expression means to be able to predict whether a path in the control flow graph will be followed or not. If a path is never accessed then all paths that are originating from it and that have no other incoming paths will never be accessed too (i.e., they are unreachable) and can be removed from the original control flow graph. The same applies to expressions that represent the addresses of indirect function calls and of indirect branches. If an expression could be calculated then the indirection would be replaced with a fixed constant address and that means that new nodes can precisely be added to the control flow graph.

Chances for normalization may also arise even if an expression can not be fully calculated. For example, suppose that propagation into one of the operands of a branch condition is not possible because there are two different incoming definitions of the same variable coming from two different concurrent paths. Nothing can be told about the truth value of the condition because it still contains a variable. But, if looking back at the values that an incoming variable may assume, we find only fixed values, and not variables, we can evaluate the condition for all possible incoming values and if the result is always the same, one of the two outgoing paths can be removed from the control flow graph. The same approach can be used to determine the set of possible target addresses of an indirect jump (or of a function call).

4 Comparison techniques

Unfortunately we can not expect that, at the end of the normalization, all the samples of a self-mutating malware reduce to the same normal form. Differences that can not be handled (e.g., some temporary assignment that cannot be removed) usually remains. For these reasons it is not possible to compare two samples just comparing byte per byte their normalized form. For dealing with such a problem we need a method that is able to provide a measure of the similarity among different pieces of code, which allow us to capture the effective similarity of them.

The problem of comparing codes in order to discover common fragments among them is known as *clone detection*. Clone detection is primarily used by software engineers to detect equivalent blocks within source code, in order to factor out them

in macros and function calls, reducing the size of the code and improving its understandability and maintainability. Clone detection techniques can rely on the analysis of the text of the code [14], or on some more sophisticated representation of the program (see [4, 5, 17]).

In this paper we are interested in the approach proposed in [18] where the structure of a code fragment is characterized by a vector of software metrics and a measure of *distance* between different fragments is proposed.

The metrics we decided to adopt during our experiments are the following:

1. m_1: number of nodes in the control flow graph;
2. m_2: number of edges in the control flow graph;
3. m_3: number of direct calls;
4. m_4: number of indirect calls;
5. m_5: number of direct jumps;
6. m_6: number of indirect jumps;
7. m_7: number of conditional jumps;

These metrics have been chosen such that they should capture the structure of an executable code. Thus, we have defined the fingerprint of a code fragment as the tuple $(m_1 : m_2 : m_3 : m_4 : m_5 : m_6 : m_7)$ and we used it for comparing code fragments. More precisely we say that two code fragments a and b are equivalent if the Euclidean distance, $\sqrt{\sum_{i=1}^{7} (m_{i,a} - m_{i,b})^2}$, where $m_{i,a}$ and $m_{i,b}$ are the i^{th} metric calculated respectively on fragment a and b, is below a given threshold. Ideally the threshold should be 0, but a higher value would allow to tolerate little imperfections in the normalization process.

The structure of different instances of the same malicious code is far from being constant, but we believe that, after normalization, most of the structural differences will be removed. Roughly speaking, we expect that most of fake conditional jumps get translated into unconditional jumps or removed, fake indirect function calls and jumps get translated into direct, because the destination addresses have been computed, and nodes in the control flow that are not reachable get simply removed.

Thus, the Euclidean distance of a code fragments before and after normalization allows to quantify the effectiveness of the normalization and the Euclidean distance between a normalized code fragments and the malicious code archetype allows to quantify their similarity.

5 The prototype

We have implemented the techniques described in Sections 3 and 4 in a tool which, given as input two code fragments a and b, will decide whether b is a mutated version of a. In this Section we describe the architecture of our tool and its main components.

Figure 1 represents the steps performed during the code analysis. The malware executable is processed through a disassembler and it is converted into an intermediate form. Subsequently, the malware is processed by a set of transformations in order to reshape the code and to remove, as much as possible, redundant and useless instructions while trying to compact their form. Each step of the normalization process (depicted in gray in Figure 1), is highly dependent on both prior and subsequent steps; for this reason they are repeated until the latest normalized form does not introduce new improvements over the previous one. At the end of the normalization process the resulting malware, that should resemble as much as possible its archetype, is processed by the evaluator in order to compare it with other samples.

We built our prototype on top of BOOMERANG [1], an open source decompiler. The aim of BOOMERANG is to translate machine code programs into an equivalent C source code and to do this it reconstructs, from low level machine instructions, high level concepts (e.g. compound statements like *while, for*, ... and high level expressions with more than two operands) which are subsequently translated into source code. BOOMERANG provides an excellent framework which can be used to perform any kind of machine code manipulation, so we decided to build out tool on top of it by modifying some of its native functionality and adding new ones. For example, the control flow compression routine has been enhanced in order to deal with obfuscated branches predicates and to remove pending unreachable nodes; the algebraic simplification routines has also

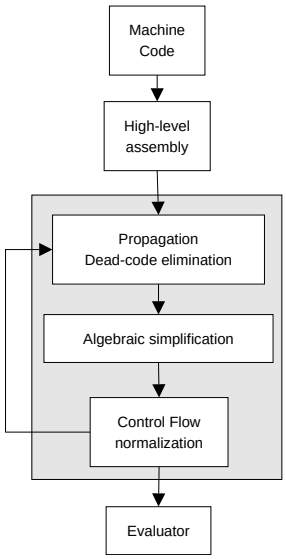

Figure 1: Normalization and identification process.

been enhanced to handle a bigger set of simplifications not previously handled and without which normalization would have been less effective. It is worth noting that none of the simplifications and the transformations implemented are malware specific. A brief description of our tool follows.

During the first phase of the analysis, called *decoding*, the code is translated from machine code to an intermediate representation. Each instruction is disassembled and it is expanded into the respective semantic, which is expressed through the SSL [10] language. When jump and call instructions are encountered the decoding is performed following the execution paths; this approach of disassembling is called *recursive traversal* because the program flow is followed. During decoding, sequential instructions are grouped together into blocks, called *basic block*s, which are then connected together, according to flow transition instructions, in order to construct the *control flow graph*. The instructions belonging to the same basic block satisfy the property that all of them is always executed before all the subsequent ones.

Once a code fragment is completely decoded and its control flow graph is built, it is transformed into *static single assignment form* [15] (or simply *SSA*). The particularity of this representation is that every definition of a program variable gives rise to a new variable, thus the same variable is defined only once. For example the first definition of the variable A generates the variable A_1 and the j^{th} definition generates the variable A_j. The main advantage of the SSA form is that it allows to enormously simplify the data-flow analysis process because it makes explicit the relation between the use of a variable (when it appears inside an expression) and its definition (when it appears on the left hand side of an assignment). A trick has to be used when more than one definition reach the same use, through different paths: a special statement, called ϕ-statement, is inserted at the beginning of the basic blocks that uses these definitions, and it is used to define a new variable which indicates the use of multiple concurrent definitions. For example if both $A_i := 0$ and $A_j := 1$ reach the instruction $B := A + 1$ of block k the following instruction $A_k := \phi(i, j)$ is inserted before B definition, which is then translated into $B := A_k + 1$, in order to explicits that the assignment uses either A_i or A_j.

Figure 2 shows the assembly code of a dummy code fragment and Figure 3 shows the same program whose instructions are translated into the intermediate form plus SSA, and grouped in order to construct the control flow graph. The numbers between curly brackets near each variable use indicate, for clearness, the number of the instruction at which a variable is defined, instead of the incremental number of the variable definition, `m[0x1000400c]` is used to indicate the memory address `0x1000400c`, and `EAX@[31:31]` means the 31^{st} bit of the register EAX.

Further transformations are applied directly on the SSA form. Chances for propagation and dead instruction removal can be easily identified thanks to the SSA form. For example the flag ZF (Zero Flag) definition at instruction 8 (Figure 3), ZF_8, is dead because no instruction uses it, instead ZF_{12} is used by instruction 14. The same applies for definitions CF_5 (Carry Flag), OF_6 (Overflow Flag) and SF_7 (Sign Flag), they are all dead. Propagation is applied in the same way, every use of a variable A_i can be substituted with its definition whenever it is unique (i.e. it is not defined by a ϕ-statement).

Simplifications instead are performed on each expression through pattern matching. Each expression, and all of its sub-

```
8048354: mov    $0x1000400c,%eax
8048359: mov    %eax,0x10004014
804835e: add    %eax,%ebx
8048360: test   %ebx,%ebx
8048362: jne    804836a
8048364: push   %ebx
8048365: mov    $0x0,%ebx
8048365: inc    %eax
804836c: jmp    *%ebx
```

Figure 2: Assembly code of a dummy code fragment.

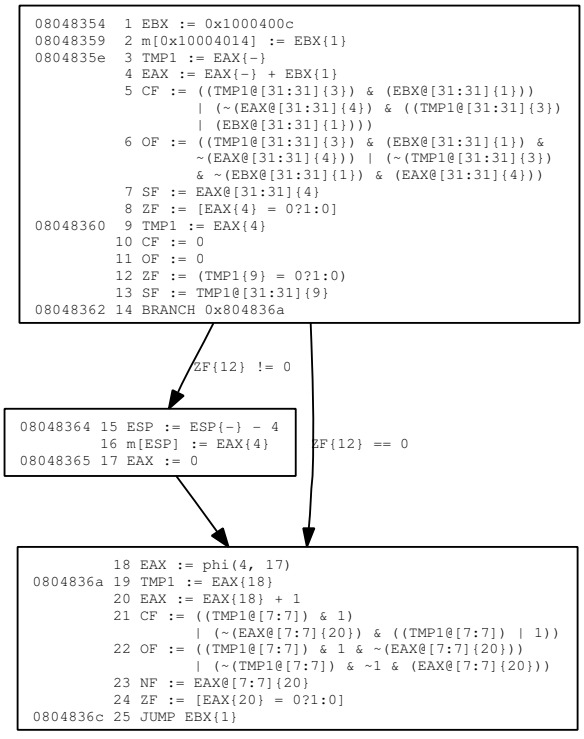

Figure 3: Single Static Assignment Form of the code fragment of Figure 2; instructions are expanded in order to expose their semantic and are grouped according to control flow relations.

expressions, are compared against a set of known reducible expressions and, when a match is found, simplification rules are applied until no changes are applicable. The set of simplification applied are just well known arithmetic and logical simplification rules. Whenever a branch condition is simplified into an expression that always evaluates to true or false, the unreachable path is removed from the control flow graph. If the deletion of a path has consequences on the instructions already processed (for example a block that was previously reached by two concurrent definitions it is now reached by one only) then new chances for propagation and simplification may arise, so the process must be repeated again. Other control flow simplifications may occur when, thanks to propagation and simplification, it is possible to determine the set of values that a variable, used as the target address, may assume. The discovery of new paths can have consequences on the whole analysis and thus it must be repeated until there are no new improvements.

For example (Figure 3) the propagation of EAX_4 into instruction 9, and then of $TMP1_9$ into instruction 12, leads to $ZF_{12} := (EAX\_ + 0x1000400c == 0\,?\,1 : 0)$ ($EAX\_$ means that EAX has not yet been defined) which can then be propagated into instruction 14, thus substituting the branch condition into $(EAX\_ + 0x1000400c == 0\,?\,1 : 0) == 0$, which is true when $EAX\_ \neq -0x1000400c$. Propagation of EBX_1 into instruction 25 instead allows to resolve the indirection and thus to translate the instructions into a direct jump to the address $0x1000400c$. This new address, if not yet decoded and analysed, must be treated as any other program instructions and, consequently, processed in the same way.

At the end the last normalized form of the input sample is emitted, represented into the intermediate form and provided with the fingerprint based on the metrics (described in Section 4) calculated at the end of the process. The similarity of the samples, instead, is calculated by a separate tool that receives in input a set of fingerprints and returns a number representing their similarity degree.

6 Experiments and results

We performed our experiments on the METAPHOR [2] virus which is one of the most interesting malware from the mutation point of view. METAPHOR evolves itself through five steps: (i) disassembling of the current payload, (ii) compression of the current payload using some transformation rules, (iii) permutation of the payload introducing fake conditional and unconditional branches, (iv) expansion of the payload using some transformation rules and (v) assembling of the new payload.

The malware analyzed applied the whole mutation process only on its payload that is stored encrypted in the binary; the decryption routine is instead is mutated only using step (iv)[4]. We then performed our experiments on this routine, but we expect similar results on the payload since our normalization is able to cancel most of the transformations introduced in steps (ii) and (iii). Moreover, compression is only used to avoid explosion of the size during evolution and its influence on the structure of the code is analogous to expansion, just the inverse way.

The experiments we performed are the followings. The malware archetype was executed in a controlled environment in order to infect some dummy programs; after the first infection the newly infected program were executed to infect new dummy programs, and so on, until we collected 114 samples. The infected dummy programs were compared with their uninfected copy such that it was possible to extract the decryption routine and the encrypted payload. The extracted decryption routines were then given in input to our prototype in order to perform the normalization and the comparison. Firstly we computed the fingerprint of each decryption routine without performing any normalization. Subsequently, we performed the normalization process and computed the fingerprints of the rearranged samples. The calculated fingerprints were then compared with the one that resembles the archetype. We noticed that the fingerprints of the original samples (i.e., without having carried out normalization) were almost different from the archetype and that just a little subset matches exactly, probably because during mutation weak transformations were chosen. After the normalization most of the samples, instead, matched perfectly the archetype:

Distance range	# of samples (before norm.)	# (after norm.)
0.0 - 0.9	29	85
1.0 - 1.9	9	19
2.0 - 2.9	38	2
3.0 - 3.9	4	7
4.0 - 4.9	21	1
> 4.9	13	0

The results of comparison of the similarity degree with and without normalization are depicted in figure 4. We manually inspected the normalized samples that still presented a different structure (similarity degree $\neq 0$) and noticed that the

[4]Instructions are not reordered but the control flow graph is mutated with the insertion of new branches.

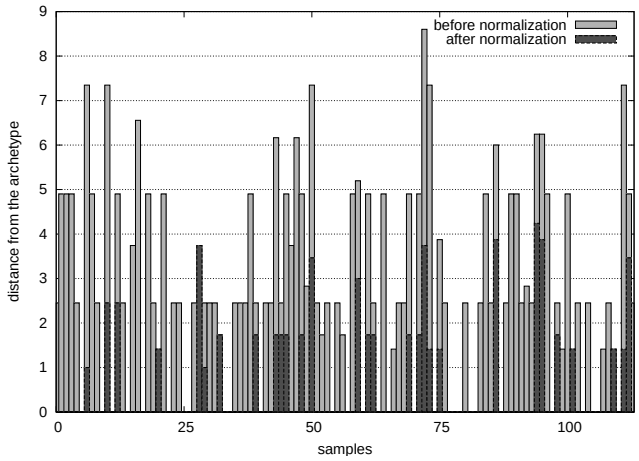

Figure 4: Distance of the analyzed samples from the malware archetype before and after the normalization.

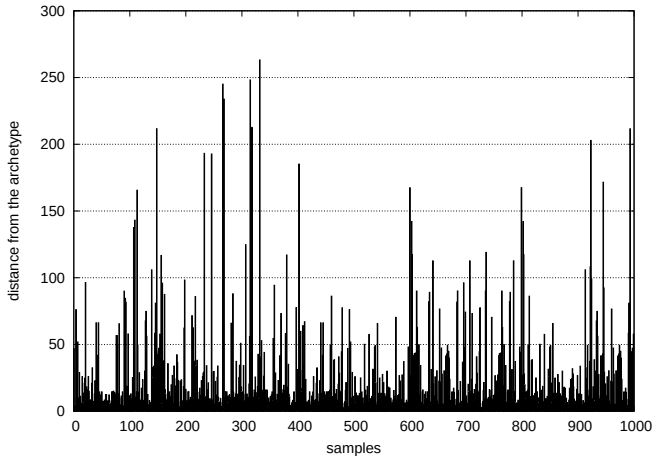

Figure 5: Distance of some harmless functions from the malware archetype.

differences were due to some garbage located after the end of the decryption routine that were recognized as valid machine instructions and became part of the control flow graph. This is a limitation of the comparison function chosen, because it is susceptible to garbage in the control flow graph. We also measured the difference between the number of instructions in the original samples and in the normalized ones, they were about 57% less.

Furthermore, we collected randomly a small number of system executable binaries, normalized them and compared their functions fingerprints with the archetype one, just to see what would have been their degree of similarity. The results, for a total of 1000 functions, are shown in figure 5 and in the following summary:

Distance range	**# of functions**
0.0 - 0.9	0
1.0 - 1.9	4
2.0 - 2.9	28
3.0 - 3.9	44
4.0 - 4.9	46
> 4.9	827

The current prototype implementation imposes some restrictions on the type of program characteristics that can be used to measure the similarity of the samples, mostly because of the presence of dead code. A manual inspection of the normalized samples highlighted that the main problem was the presence of temporary defined intermediate memory cells that could not be removed from the code because it was not possible to tell if they were used or not by other instructions. We can not

tell if they are dead until we are not able to add to our prototype a smarter alias analysis algorithm; actually the memory is handled in a conservative way. The use of more rigorous and uncorrelated metrics would allow to have a more accurate evaluation of the quality of the normalization process. Another problem we envisioned, even if we did not found evidence of it in our experiments, is that one could introduce branches conditional on the result of a complex function. It could be difficult to evaluate statically that the value of the function falls always in a given range, but this could be exploited by a malicious programmer. However, the complexity of the function is still bounded by the constraint that it has to be introduced *automatically* (see previous discussion in Section 1).

Although the normalization of our samples was quite fast (less than a second each) performances could be an issue with big executables; our attempts to normalize common executables of a UNIX system took an enormous amount of time and system resources. The running time, calculated on an Intel 3.0Ghz machine, ranges from 0.2 to 8 seconds per functions.

7 Related works

The idea of using object code static analysis for dealing with malware code obfuscation has been firstly introduced by Christodorescu and Jha in [7]. In such a paper they introduced a system for detecting malicious patterns in executable code, based on the comparison of the control flow graph of a program and an automaton that described a malicious behavior. More precisely, they generalized a program P translating it into an annotated control flow graph using a set of predefined patterns and then performed detection by determining whether there existed, or not, a binding such that the intersection between the annotated control flow graph and the malicious code automaton (i.e., the abstract description of the malware archetype) was not empty; if a binding existed then P was classified as malicious. The obfuscation techniques they were able to deal with using such an approach were registers reassignments, control flow rearrangement through unconditional jumps plus a limited set of dead-code instructions as they were detected through patterns during annotation. A further refinement of the their work has been presented in [9]. The malware detection algorithm presented in such a work received as input a program (converted into an intermediate form and which control flow is normalized removing useless unconditional jumps) and a template describing a malicious behavior and tried to identify bindings between program instructions and template nodes. Whenever a binding inconsistence was found (e.g. two different expression are bounded to the same template variable) a decision procedure (based on pattern matching, random execution and theorem proving) was used to determine if the inconsistence was due to an absence of the malicious behavior in the program or because the binding had been obfuscated inserting garbage. The class of obfuscation transformations was wider than the one handled by their first work: the system was able to detect using only a template different hand-made variants of the same malware. The algorithm returned *true* if the whole template had been found in the program, *don't know* otherwise. In both papers some experimental results were reported and they showed that the systems worked pretty well, especially the second one because; at the moment the real problem of these approaches is speed.

We decided to concentrate our efforts on malware that is able to obfuscate itself autonomously so our approach deals with mutations in a way that is closer to the ways in which it is generated. Thus, we can revert lot of the modifications that malware suffered during its life cycle by reverting the mutation process. The works described above concentrate on comparison but do not try to fight the obfuscation, so if the malicious code is highly obfuscated the proposed detection techniques could become useless. We, instead, have proposed that deobfuscation becomes a fundamental step in the analysis process and we also have shown which techniques can be successfully used.

8 Conclusions and future works

We presented a strategy, based on static analysis, that can be used to pragmatically fight malicious codes that evolve autonomously in order to circumvent detection mechanisms. To verify our ideas we developed a prototype and successfully used it to show that the transformations used by malware can be reverted and that a malware that suffers a cycle of mutations can be brought back to a canonical shape that is highly similar to its original one. The similarities among the analyzed samples were measured to determine the quality of the whole process. The same approach was also used to compare generic executables with the malware in analysis.

Our original intentions were to use more metrics, for example the number of different used and defined variables, the total number of statements and the number of concurrent variable definitions reaching program assignments. The improvement of the prototype will be targeted by future works. We are also planning to work on more accurate techniques for the comparison

of pieces of code that are not susceptible to undesired garbage and that could provide a more reliable way for the detection of known malicious codes in generic executables, even when they are located within already existing and harmless functions.

References

[1] Boomerang. `http://boomerang.sourceforge.net`.

[2] MetaPHOR. `http://securityresponse.symantec.com/avcenter/venc/data/w32.simile.html`.

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

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the second working conference on reverse engineering*, pages 86–95, Los Alamitos, CA, USA, 1995. IEEE.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

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

[7] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of USENIX Security Symposium*, Aug. 2003.

[8] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 34–44, Boston, MA, USA, July 2004. ACM Press.

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

[10] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *6th International Workshop on Program Comprehension - IWPC'98, Ischia, Italy, June 24-26 1998*, pages 126–133. IEEE Computer Society, 1998.

[11] F. B. Cohen. *A Short Course on Computer Viruses, 2nd Edition*. Wiley, 1994.

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

[13] S. K. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[14] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicating code. In *Proceedings of the International Conference of Software Maintenance*, pages 109–118, Sept. 1999.

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[16] P. Ferrie and P. Ször. Zmist opportunities. *Virus Bullettin*, 2001.

[17] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126, 2001.

[18] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M.Bernstein. Pattern matching techniques for clone detection. *Journal of Automated Software Engineering*, 1996.

[19] A. Lakhotia, A. Kapoor, and E. U. Kumar. Are metamorphic viruses really invincible? *Virus Bulletin*, Dec. 2004.

[20] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[21] B. Schwarz, S. K. Debray, and G. Andrews. Plto: A link-time optimizer for the intel ia-32. In *Proceedings of the 2001 Workshop on Binary Translation*, 2001.

[22] Symantec. W32.evol. `http://www.symantec.com/avcenter/venc/data/w32.evol.html`.