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, 20135 Milano
{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.; the most obvious strategy for circumventing them was to write mutating malware, i.e. malicious codes which continuously change their own code and, consequently, render signatures 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, ZMIST and EVOL. 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 perfect detection of self-mutating code is a non computable problem, we believe that the strategy we present has very interesting implications on the practical point of view.

Our strategy, which can be classified as a static analysis approach for verifying security properties of executable code, is based on the following observation (see also [17]). Since we target *automatic mutations,* mutating programs have to be able to analyze their own bodies and extract from them all the information they need to mutate into next generations, and further generations must be able to do the same. Thus, mutation mechanisms adopted by this type of code have to be easily computable and in particular they have to be efficient. This implies that it should be possible to reverse the mutation process and derive the archetype from which mutations have been constructed. More precisely, we observed that self mutating programs for producing the next generation usually produce highly unoptimized code, that is code which contains a lot of redundant and useless instructions, on the other hand all the copies of the same malware have to share common pieces of code. Thus our idea is that of reducing any piece of code which we suspect to contain some malware to a normal form, roughly speaking a code which does not contain anything not needed for its correct functioning, and verify whether it shares common features with the normalized code of known malware. In order to implement the normalization process we referred to well known code optimization techniques adopted by compilers' writers (for example see [3, 18]).

Once the code has been normalized the next problem to solve is that of verifying its adherence to some malware, adopting a strategy which guarantees a low false positive percentage. For realizing such a phase we considered *clone detection* techniques [16, 15, 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

[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.

the current stable version of such tool should 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 Techniques used for mutations

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 ecx,eax`	`xor ebx,eax`
`mov eax,ebx`	`xor eax,ebx`
`mov ebx,ecx`	`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 their presence (see Section 2) led to a major consequence: the code size is highly increased. In other words, one could view the variants of a piece of malware as unoptimized versions of its archetype[3], since they contain some irrelevant computations whose presence has only the 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".

Generally speaking, 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, 18]. As shown by [19, 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.

[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.

Instructions meta-representation

In order to ease the manipulation of object code, we used a high-level representation of machine instructions that makes explicit their semantics.

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) *expressions*. Comparison instructions can be considered as expressions because they usually perform some arithmetics on their operands and then update a control register accordingly (e.g. `eflags` on IA-32). Our high-level representation expresses the operational semantics of every opcode: which registers, memory address and control flags they are using and modifying, etc. A simple example follows:

Machine instruction	**Intermediate form**
`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). A high-level representation makes easy to take into account all the side effects, e.g., a subsequent branch that relies on control flags to decide whether to jump or not.

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 assigned twice but it is never used between the two assignments (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 computation.

Algebraic simplification

Since most of the expressions contains 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 indirected function calls and of indirected 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 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. [4] uses tokens, [5] compares the abstract syntax trees of two fragments discarding the smallest subtrees that should represent variables and constants, [15] uses slicing to construct the biggest set of equivalent instructions of two code fragments. In [16] the structure of a code fragment is characterized by a vector of software metrics and a measure of *distance* between different fragments is proposed. In our approach, we exploited this last technique since it allows for choosing the appropriate accuracy level by selecting different thresholds. Moreover, the distance can be used to quantify the similarity of two code fragments.

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;

The fingerprint of a code fragment is then simply a tuple which elements are the metrics listed above: $(m_1 : m_2 : m_3 : m_4 : m_5 : m_6 : m_7)$. Code fragments are compared by comparing their fingerprints and fingerprints are compared calculating their 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.

The metrics have been chosen such that they should capture the structure of an executable code and it would be possible to evaluate the benefit gained from the normalization. We expect that after the normalization process most of the 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. We expected that most of the samples, after normalization, have the same fingerprint of the archetype.

5 The prototype

In this Section we describe in detail the architecture of our tools and its main components.

Figure 1 represents the steps of the analysis. The malware executable is processed through a disassembler and it is converted into an intermediate form. Further steps of the analysis work on the intermediate representation. 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 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,

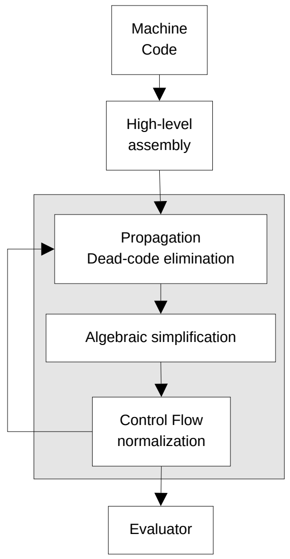

Figure 1: Normalization and identification process.

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 source code and to do this it needs to reconstruct, from low level machine instructions, high level concepts that can be translated into source code. BOOMERANG provides an excellent framework which can be used to perform any kind of machine code manipulation, so we adapted it in order to pursue our goals. A brief description about how the normalization process is implemented follows.

During the first phase of the analysis the executable code is loaded and processed by a recursive disassembler. The disassembler translates each machine instruction into an internal representation that is constructed starting from a given specification expressed in the SSL [10] language that describes each machine instruction in term of their semantic. Moreover sequential instructions, thanks to the recursive behavior of the disassembler, are grouped into blocks and blocks are connected together according to jump instructions to construct the control flow graph of each function or fragment. Subsequently, instructions are converted into *static single assignment form*[4] on which is then performed the dataflow analysis. At the end of dataflow analysis propagation, algebraic simplification, control flow compression and removal of unreachable and dead instructions take place. Finally the last normalized form of the input sample is emitted, still represented into the intermediate form, provided with the fingerprint based on the metrics 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 a returns a number representing their similarity degree.

Our prototype takes advantage of BOOMERANG facilities to perform the whole normalization; whenever necessary they have been fixed and improved. 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 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.

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 evolutes 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.

Unfortunately, the whole mutation process is applied only on the malware payload that is stored encrypted in the binary. We then performed our experiments on the decryptor that is mutated only using step (iv)[5], 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 following. 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

[4]*Static single assignment* form, or simply *SSA*, is an intermediate representation in which every variables is defined only once and every definition generates a new variable. *SSA* is used to simplify the dataflow analysis.

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

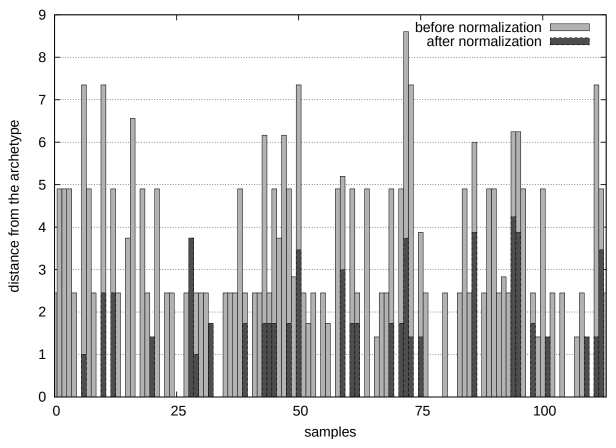

Figure 2: Distance of the analyzed samples from the malware archetype before and after the normalization.

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 2. We manually inspected the normalized samples that still presented a different structure and noticed that the 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 3 and in the following summary:

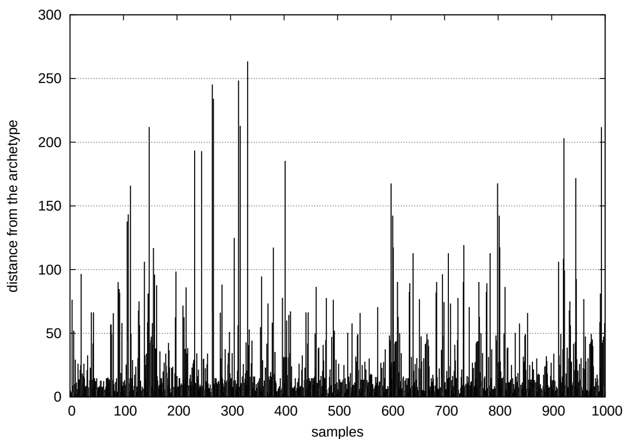

Figure 3: Distance of some harmless functions from the malware archetype.

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 characteristic 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 obfuscation malware 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 new malware detection algorithm received in input a program (converted into an intermediated form and which control flow is normalized removing useless unconditional jumps) and a template describing a malicious behavior and tried to identify bindings that connected program instructions to 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 algorithm returned *true* if the whole template had been found in the program, *don't know* otherwise. In both papers some experimental results obtained with system prototypes showed that the systems worked pretty well, especially the second one because the class of obfuscation transformations was wider than the one handled by the first: the system was able to detect using only a template different hand-made variants of the same malware. 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 especially 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 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] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–??, 2001.

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

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

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

[19] 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.