Code Normalization for Self-Mutating Malware

Next-generation malware will adopt self-mutation to circumvent current malware detection techniques. The authors propose a strategy based on code normalization that reduces different instances of the same malware into a common form that can enable accurate detection.

Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga
Università degli Studi di Milano

Most of today's commercial malware-detection tools recognize malware by searching for peculiar sequences of bytes. Such byte strings act as the malware's "fingerprint," enabling detection tools to recognize it inside executable programs, IP packet sequences, email attachments, digital documents, and so on. Thus, these patterns are usually called *malware signatures.* Detectors assume that these signatures won't change during the malware's lifetime. Accordingly, if the fingerprint changes (that is, if the malware code mutates), detection tools can't recognize it until the malware-detection team develops a new fingerprint and integrates it into the detector. To defeat signature-based detection, attackers have introduced *metamorphic malware*—that is, a self-mutating malicious code that changes itself and, consequently, changes its fingerprint automatically on every execution.[1] (*Code obfuscation* involves transforming a program into a semantically equivalent one that's more difficult to reverse engineer. Self-mutation is a particular form of obfuscation in which the obfuscator and the code being obfuscated are the same entity.) Although this type of malware hasn't yet appeared in the wild, some prototypes have been implemented (for example, W32/Etap.D, W32/Zmist.DR, and W32/Dislex), demonstrating these mutation techniques' feasibility and efficacy against traditional antivirus software. Recent work also demonstrates that attackers can easily circumvent current commercial virus scanners by exploiting simple mutation techniques.[2,3]

Perfect detection of self-mutating malware is an undecidable problem—no algorithm can detect with complete accuracy all instances of a self-mutating malware.[4] Despite such negative results, we strongly believe that from a practical viewpoint, reliable detection is possible. We base this belief on several considerations, which we outline in a later section. On the basis of these observations, and considering a series of advances in malware detection presented in the literature (see the sidebar), we developed our own approach for dealing with self-mutating code that uses static analysis to exploit the weaknesses of the transformations the self-mutating malware have adopted. The result is a prototype tool that, in the cases we've considered, can determine the presence of a known self-mutating malicious code, the guest, within an innocuous executable program, the host.

Mutation and infection techniques

Executable object mutations occur via various program-transformation techniques.[5–7] Basically, malware mutates itself and hides its instructions within the host's benign program code.

Mutation techniques

A malware can adopt several common strategies to achieve code mutation. As it applies these transformations several times randomly, each one introduces further perturbations locally, to a basic block, as well as globally to the control-flow graph. When the malware introduces a fake conditional jump, for example, subsequent transformations obfuscate the way in which the program computes the predicate, and so on.

Instruction substitution. A sequence of instructions is associated with a set of alternative instruction sequences that are semantically equivalent to the original one. The

Related work in malware-detection techniques

Frédéric Perriot first proposed the idea of leveraging code-optimization techniques to aid malicious code detection in 2003.[1] Building on that work, we show in the main article that the transformations current malicious codes adopt are weak and can be reverted to reduce different instances of the same malware into the same canonical form. We can then use this form as a pattern for detection.

The malware detector Mihai Christodorescu proposes[2] is based on the idea of semantic-aware signatures and leverages static analysis and decision procedures to identify common malicious behaviors that are shared among different malware variants but are generally obfuscated to protect them from standard detection techniques. Thus, a malware detector can use one signature to detect a big class of malicious codes that share the same common behaviors. Our work, instead, focuses on code normalization because we believe it's fundamental to identifying equivalent code fragments generated through self-mutation.

Other authors' works perform malware normalization via term rewriting.[3] The rewriting system proposed is suited to the rules used by self-mutating engines and can help normalize different malware instances generated using instruction substitution and irrelevant instructions insertion. Our approach doesn't guarantee perfect equivalence among different normal forms (that is, different malware instances can still perform the same operation using different machine instructions), but it can remove unnecessary computations and recover the original control-flow graph. Moreover, our approach is independent of the particular rules for self-mutation the malware adopts, so we don't need any knowledge about the malware to normalize its instances.

Different researchers propose the idea of assessing program equivalence by comparing control-flow or call graphs. In one work,[4] the similarity measure obtained from comparing the call graph helps automatically classify malicious code in families; this classification is coherent with the actual malware naming. In another,[5] researchers measure program-functions similarity by comparing a fingerprint generated from their control-flow graph to each other and then using it to identify the same function within different executables. Unfortunately, we can't use these approaches in our context for two reasons:

- The malicious code can be located anywhere, even inside another procedure, so we should formulate the detection as a subgraph isomorphism and not as a graph isomorphism (or a simplified instance, such as comparing graphs fingerprints).
- One main, observable difference among different instances of the same self-mutating malicious code is in the control graph's structure; thus, normalization is a fundamental step toward performing effective control-flow graph matching.

Finally, the idea of generating a fingerprint from the executable control-flow graph and using it to detect different instances of the same polymorphic worm is proposed in another work,[6] from which we adopted our labeling technique. Their fingerprinting technique, however, suffers the latter of the two problems just described.

References

1. F. Periot, "Defeating Polymorphism Through Code Optimization," *Proc. Virus Bulletin Conf. 2003*, Virus Bulletin, 2003, pp. 142–159.
2. M. Christodorescu et al., "Semantics-Aware Malware Detection," *Proc. 2005 IEEE Symp. Security and Privacy*, IEEE CS Press, 2005, pp. 32–46.
3. A. Walenstein et al., "Normalizing Metamorphic Malware using Term Rewriting," *Proc. Int'l Workshop on Source Code Analysis and Manipulation* (SCAM), IEEE CS Press, 2006, pp. 75–84.
4. E. Carrera and G. Erdélyi, "Digital Genome Mapping—Advanced Binary Malware Analysis," *Proc. Virus Bulletin Conf.*, Virus Bulletin, 2004, pp. 187–197.
5. H. Flake, "Structural Comparison of Executable Objects," *Proc. Conf. Detection of Intrusions and Malware & Vulnerability Assessment* (DIMVA), IEEE CS Press, 2004, pp. 161–173.
6. C. Kruegel et al., "Polymorphic Worm Detection using Structural Information of Executables," *Proc. Int'l Symp. Recent Advances in Intrusion Detection*, Springer, 2005, pp. 207–226.

malware can replace every occurrence of the original sequence with an arbitrary element from this set.

Instruction permutation. Independent instructions—that is, those whose computations don't depend on previous instructions' results—can be arbitrarily permuted without altering the program's semantics. For example, the malware can execute the three statements `a = b * c`, `d = b + e`, and `c = b & c` in any order, provided that the use of the `c` variable precedes its new definition.

Garbage insertion. This is also known as dead-code insertion, and involves the malware inserting, at a particular program point, a set of valid instructions that don't alter its expected behavior. Given the following sequence of instructions `a = b / d`, `b = a * 2`, for example, the malware can insert any instruction that modifies `b` 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 (for example, `a = a + 0`, `b = b * 1`, `...`).

Variable substitutions. The malware can replace a variable (register or memory address) with another variable belonging to a set of valid candidates preserving the program's behavior.

Control-flow alteration. Here, the malware alters the order of the instructions as well as the program's structure by introducing useless conditional and unconditional

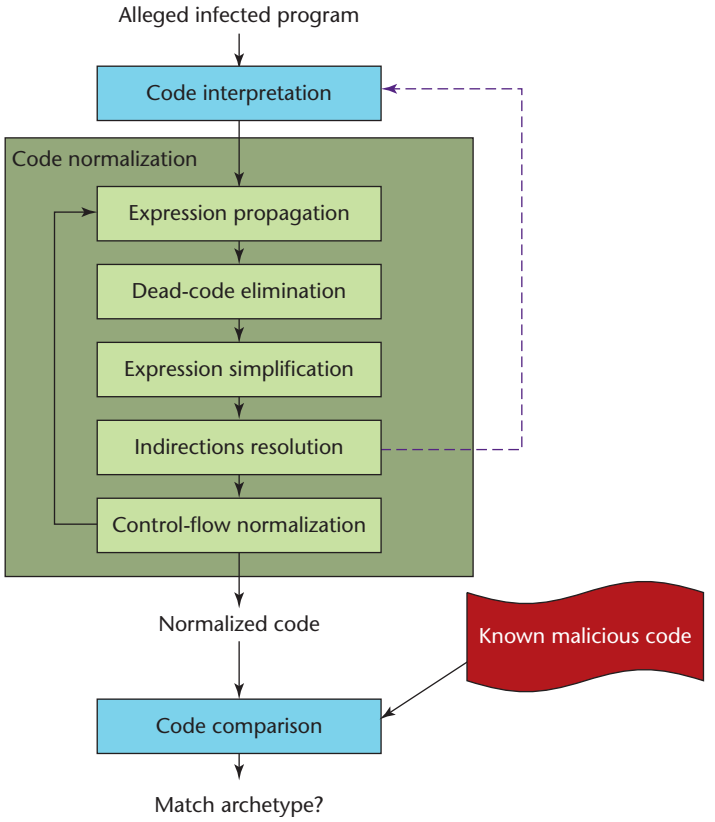

Figure 1. Overview of the detection process. Starting with a suspicious program, we translate it into the normal form, normalize it, and then compare the normalized version with known malicious programs.

branch instructions such that at runtime, the order in which the program executes single instructions isn't modified. Furthermore, the malware can translate direct jumps and function calls into indirect ones whose destination addresses are camouflaged through other instructions in order to prevent an accurate reconstruction of the control flow.

Infection techniques

To camouflage the link between the benign host and the malicious guest, the malware tangles each one's instructions together by exploiting smart techniques referred to as *entry-point obfuscation* (see http://www3.ca.com/securityadvisor/glossary.aspx). Once the malware achieves a seamless integration, the host invokes the new guest code in a way very similar to how it invokes other portions of its own code. Moreover, the host will continue to work exactly as it did before the infection.

The malware can adopt different techniques—which require minimal effort to implement—to achieve this goal.

Cavity insertion. Generally, executables contain several portions that aren't used to store data or code (compilers can introduce them to align code and data structure). The malware identifies these "cavities" and uses them to insert small pieces of malicious code that the host will execute after minor modifications of its code.

Jump-table manipulation. In a compiled program, high-level control-flow transfer constructs such as `switch` are implemented using jump tables to redirect the execution to the appropriate location according to the table's contents and the value that the control variable assumes. The malware can modify entries of such tables to get the host to redirect the execution anywhere.

Data-segment expansion. To create the required space inside the host code's address space, malware can expand some of the host segments as needed. Not all segments are suited for expansion because that would require relocating most of the code. Other segments, such as the one storing uninitialized data, seem to be more appropriate because their expansion allows the malware to insert malicious code without requiring further modification of the host code.

Proposed detection strategy

To be effective, malware mutations must be automatic and efficient; otherwise, their mutation computations would be too difficult to hide. More precisely,[8] a self-mutating malware must be able to analyze its own body and extract from it all the information needed to mutate itself into the next generation, which in turn must be able to perform the same process, and so forth. Because mutations occur directly on machine code, and the mutation engine is embedded directly into the malicious code itself, the applicable transformations must, in most cases, be rather trivial. Consequently, we can iteratively reverse the mutation process until we obtain an *archetype* (that is, the original and unmutated version of the program from which the malware derives other instances). Experimental observations show that self-mutating programs are basically highly unoptimized code, containing a lot of redundant and useless instructions.

Our detection process is thus divided into three steps: *code interpretation*, *code normalization*, and *code comparison*. Figure 1 illustrates the entire detection process: the detector's input is an executable program that hosts a known malicious code. The code interpreter then transforms the program into an intermediate form, which is then normalized. Finally, the code comparator analyzes the normalized form to identify possible matches of the known malicious code within the normalized program.

Code interpretation

To ease the manipulation of object code, our detector uses a high-level representation of machine instructions

to express every opcode's operational semantics, as well as the registers and memory addresses involved. Code interpretation considerably simplifies the subsequent subprocesses because normalization and comparisons will target a language with a very limited set of features and instructions. In fact, the instructions composing the languages are just assignments, function calls, jumps (conditional and unconditional), and function returns. The instruction operands can be only registers, memory addresses, and constants.

Table 1 shows a simple example, reduced from the original due to space constraints. In the table, the notation `[r10]` denotes the content of the memory whose effective address is in the register `r10`, and the notation `r10@[31:31]` denotes a slice of bits of the register `r10`, composed by the register's bit. Note that even the simple `dec` instruction conceals a complex semantics: its argument is decremented by one, and the instruction, or the subtraction, produces an update of six control flags according to the result.

Table 1. An example of code interpretation.

MACHINE INSTRUCTION	INTERPRETED INSTRUCTION
`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]))...`
	`...`

Table 2. A scenario generating a high-level expression.

ORIGINAL EXPRESSION	RECONSTRUCTED EXPRESSION
`r10 = [r11]`	`r10 = [r11]`
`r10 = r10 | r12`	`r10 = [r11] | r12`
`[r11] = [r11] & r12`	
`[r11] = ~[r11]`	
`[r11] = [r11] & r10`	`[r11] = (~([r11] & r12)) &` `([r11] | r12)`

Code normalization

The code normalizer's goal is to transform a program into a canonical form that's simpler in terms of structure or syntax, while preserving the original semantics. We observed that most of the transformations a malware uses to dissimulate its presence lead to underoptimized versions of the archetype. The mutated versions grow because they're stuffed with irrelevant computations whose presence only has the goal of avoiding recognition. Normalization thus aims to transform the code into a more compact form by reducing the number of useless instructions, as well as the number of indirect paths. We can consequently view normalization as an optimization of the malicious code aimed at simplifying the code structure.

The normalization process consists of a set of transformations that compilers adopt to optimize the code and improve its size.[9–11] They all rely on the information collected during a static analysis phase (control-flow and data-flow analysis), which we perform on the code at the beginning of the normalization process. The more accurate the static analysis, the higher the chances of applying these transformations.

Given that the malicious code can be anywhere inside the host program, we perform the normalization on the whole program, letting the normalization process also target the malicious guest. As Figure 1 shows, all transformations are repeated iteratively because they depend on each other; normalization will stop when we can no longer apply any of these transformations to the code.

Expression propagation. The Intel IA-32 assembly instructions denote simple expressions that generally have no more than one or two operands. Propagation carries forward values assigned or computed by intermediate instructions. This lets us generate higher-level expressions (with more than two operands) and eliminate all intermediate temporary variables that the malware used to implement high-level expressions. The code fragment in Table 2 shows a simple scenario in which, thanks to propagation, our detector generates a higher-level expression.

Dead-code elimination. Dead instructions are those whose results the program never uses. We remove them from a program because they don't contribute to the computation. In Table 2, the first instruction, after propagation, is completely useless because the intermediate result has been propagated directly into the next instruction.

We also classify assignments as dead instructions when they define a variable but don't change its value (for example, `r10 = r10 + 0`).

Expression simplification. Most of the expressions contain arithmetical or logical operators, so they can sometimes be simplified automatically according to ordinary algebraic rules. When simplification isn't possible, our tool can reorder variables and constants to enable further simplification after propagation.

Simplification becomes very useful when applied to expressions representing branch conditions and memory

```
1    xor    %ebx,%ebx
2    mov    $0x1000400c,%eax
3    mov    %eax,0x10004014
4    add    %ebx,%eax
5    test    %ebx,%ebx
6    jne    <T>
7    push    %ebx
8    mov    $0x0,%ebx
   T:
9    jmp    *%eax
10   leave
11   ret
12   nop
```

(a)

```
1    r11 := r11 ^ r11
2    r10 := 0x10004014
3    [0x1000400c] := r10
4    r10 := r10 + r11
5    tmp = r11 - r11
6    ZF = [tmp = 0?1:0]
7    jump (ZF = 1) T
8    [r15] := r11
9    r15 := r15 - 4
10   r11 := 0
   T:
11   jump r10
12   r15 := r16
13   r16 := m[r15]
14   r15 := r15 + 4
15   return
```

(b)

```
1    r11 := 0
2    r10 := 0x10004014
3    [0x1000400c] := 0x10004014
4    r10 := 0x10004014 + 0
5    tmp = 0 - 0
6    ZF = [0 = 0?1:0]
7    jump (ZF = 1) T
8    [r15] := 0
9    r15 := r15 - 4
10   r11 := 0
   T:
11   jump 0x10004014 + 0
12   r15 := r16
13   r16 := m[r16]
14   r15 := r15 + 4
15   return
```

(c)

```
1    r11 := 0
2    r10 := 0x10004014
3    [0x1000400c] := 0x10004014
4    ~~r10 := 0x10004014 + 0~~
5    ~~tmp = 0 - 0~~
6    ~~ZF = 1~~
7    ~~jump (ZF = 1) T~~
8    ~~[r15] := 0~~
9    ~~r15 := r15 - 4~~
10   ~~r11 := 0~~
   T:
11   jump 0x10004014
12   r15 := r16
13   r16 := m[r16]
14   r15 := r15 + 4
15   return
```

(d)

Figure 2. Malicious code and normalization. (a) A small fragment of malicious code and (b), (c), and (d) the corresponding steps of the transformation into the normalized form. To keep the figure compact, we simplified the instruction's semantics by removing irrelevant control flags updates. Instructions that are struck through are considered dead.

addresses because it lets us identify tautological conditions as well as constant memory addresses somehow camouflaged through more complex expressions.

Indirection resolution. Assembly code is generally rich with indirect memory accesses and control transfers; for example, during compilation, `switch`-like statements are translated into jumps through an indirect jump table. When we encounter indirect control transfers during code interpretation, the currently available information isn't sufficient for us to estimate the set of admissible jump targets. During code normalization, however, we can "guess" some of the targets based on information collected through static analysis and subsequently elaborated during the transformations. Once identified, these new code segments are ready for analysis, we invoke the code interpretation on this code, and the normalizer then processes the output (purple line in Figure 1).

Control-flow normalization. A malware can significantly twist a program's control flow by inserting fake conditional and unconditional jumps. A twisted control flow can affect the quality of the entire normalization process because it can limit other transformations' effectiveness. At the same time, other transformations are essential for improving the quality of the control-flow graph's normalization (for example, algebraic simplifications and expression propagation). We can perform different types of normalizations on the control flow; for example, with *code straightening*, a straight sequence of instructions can replace a chain of unconditional jump instructions. During *spurious path pruning*, we can prune dead paths arising from fake conditional jump instructions from the control flow to create new opportunities for transformation.

An example. To better explain when and how we can apply the transformations composing the normalization process, we present a simple example. Figure 2 shows a small fragment of malicious code as well as the code obtained during the normalization process's intermediate iterations. The code in Figure 2a is unnecessarily complex, and we can translate it into a simpler form: the code contains a tautological conditional branch in addition to an undirected jump, whose target address we can evaluate statically.

Figure 2b shows the output of the normalization process's first step, which consists of translating the code into its corresponding intermediate form. We apply transformations such as algebraic simplifications and expression propagation directly to this new form; note that we can evaluate the value of some expressions statically and propagate these values into other expressions that use them. The instruction at line 4 in Figure 2c, for example, turns out to be completely useless because it doesn't alter the original value computed by the instruction at line 2. Moreover, through propagation, this constant value is also copied inside the expression that represents the jump target address (line 11), thus allowing us to translate the indirect jump into a direct one. In Figure 2d, which represents the normal form, we can see that the conditional jump (line 7) has been removed because the condition turns out to be always true; thus we can remove the fallback path from the program. Instructions that are struck through are considered dead.

Code comparison

Unfortunately, we can't expect normalization to reduce different instances of a malicious code to the same normal form—not every transformation is statically reversible (for example, when the order of instruction is permutable, or when the malware performs the same opera-

tion executing different instructions that are transformed into a different intermediate form). Thus, a bytewise comparison will likely lead to false negatives. During our experiments, we tried to classify the typologies of differences found in the normalized instances and discovered that they were all local to basic blocks. In other words, normalized instances of malicious code often share the same control flow.

To elicit the similarities, therefore, we decided to represent the malicious code and the alleged normalized host program through their *interprocedural control-flow graphs*, which are all procedures' control-flow graphs combined. Under this assumption, we can formulate the search for malicious code as a subgraph isomorphism decision problem: given two graphs G_1 and G_2, is G_1 isomorphic to a subgraph of G_2? (Even though subgraph isomorphism is in general NP-complete,[12] with our particular problem, control-flow graphs are very sparse, and finding a solution is usually tractable.) Figure 3 shows the two graphs just mentioned: Figure 3a models the malicious code, whereas Figure 3b matches the suspicious program (nodes highlighted are those that match).

Comparison through interprocedural control-flow graphs lets us abstract the information from local to basic blocks. This is beneficial in that we're throwing away every possible source of difference, but it could be a disadvantage if we consider that the information we're losing can help identify pieces of code that share the same structure but have different behaviors. Thus, we've augmented these graphs by *labeling* both nodes and edges: we labeled nodes according to the instruction properties belonging to them and edges according to the type of flow relations among the nodes they connect. A similar labeling method is proposed elsewhere.[13] We group instructions with similar semantics into classes and assign a number (label) to each node that represents all the classes involved in the basic block. Table 3 shows the classes in which we've grouped instructions and flow transitions. We also represent calls to shared library functions with the same notation: the caller node is connected to the function that's represented with just one node and labeled with a hash calculated on the function name.

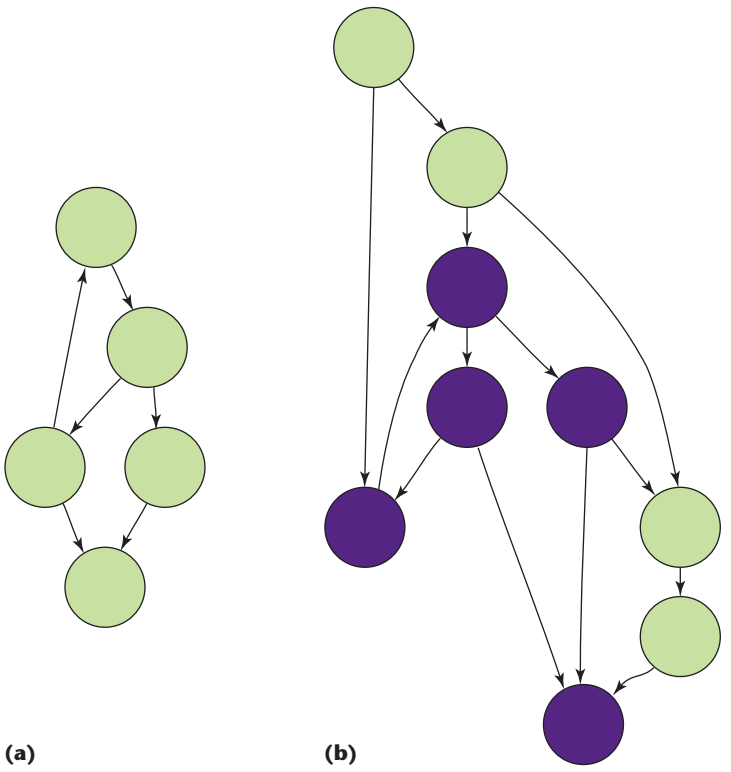

Figure 3. Two control flow graphs. We present (a) a malicious code M and (b) a normalized program P_N. The nodes highlighted in purple are those from P_N that matching ones in M.

Table 3. Instructions and flow transitions classes.

INSTRUCTIONS CLASSES	FLOW TRANSITIONS CLASSES
Integer arithmetic	One-way
Float arithmetic	Two-way
Logic	Two-way (fallback or false)
Comparison	*N*-way (computed targets of indirect jumps or calls)
Function call	
Indirect function call	
Branch	
Jump	
Indirect jump	
Function return	

Prototype implementation

To experimentally verify our approach, in terms of both correctness and efficiency, we developed a prototype. We built the code normalization module on top of Boomerang (http://boomerang.sourceforge.net), an open source decompiler that reconstructs high-level code by analyzing binary executables. Boomerang performs the data- and control-flow analysis directly on an intermediate form[14] automatically generated from machine code. We adapted it to our needs and used the engine to undo the previously described mutations. Using the information collected with the analysis, our tool decides which set of transformations to apply to a piece of code based on control- and data-flow analysis results. The analysis framework can also accommodate the resolution of indirections and performs jump- and call-table analysis[15] (further details are available elsewhere[16]).

After the prototype has normalized the code, it builds a labeled control-flow graph of the resulting code, along with the malware's control-flow graph, and we give the graph in input to a subgraph isomorphism algorithm in order to perform the detection. The prototype of our tool performs subgraph matching, leveraging the VF2 algorithm from the VFLIB library (http://amalfi.dis.unina.it/graph/db/vflib-2.0/).[17]

Experimental results

To evaluate the presented approach, we performed a set of independent experimental tests to assess the normalization procedure's quality and the code comparator's precision. The results demonstrate how effective our approach is but also highlight that we still need to do a lot of work to build a code normalizer that we can use with real-world executables.

Code normalization evaluation

We evaluated the code normalization's effectiveness by analyzing two different self-mutating malicious programs: W32/Etap.D (also known as Metaphor) and W32/Dislex. The former is considered one of the most interesting self-mutating malicious codes and evolves through five steps. The malware

1. disassembles the current payload;
2. compresses the payload according to a set of predefined rules to avoid size explosion;
3. mutates the payload by introducing fake conditional and unconditional branches;
4. expands the payload by applying step 2's rules in reverse; and
5. assembles the mutated payload.

W32/Dislex is slightly simpler because it just inserts useless code and permutes the payload by inserting fake control-flow transitions.

We collected different instances of the malicious programs by executing them in a virtual environment and forcing them to infect a set of predefined test programs. We repeated this step several times consecutively in order to infect new programs, using a newly generated malware instance every time. We collected 115 different samples of W32/Etap.D and 63 of W32/Dislex. We manually identified each malware's starting point in the various hosts—for W32/Etap.d, we chose an address stored at a fixed location in the executable import address table, whereas for W32/Dislex we chose the image start address—to focus the analysis only on the malicious payload. (In both cases, the code fragment analyzed seems to belong to a decryption routine that decrypts the real malicious payload.) We then compared the code before applying normalization and after.

For W32/Etap.d, we noticed that, in some cases, there was a correspondence between the archetype and the instance even before normalization. Given that the type of transformations applied during self-mutation are randomly chosen, we believe that the malware applied very weak transformations while generating those instances. After normalization, we noticed that all the samples matched the chosen archetype. We also observed an average reduction in the code size of about 57 percent.

Further experimentation with W32/Dislex confirmed these results. We noticed that before normalization, some graphs corresponded to each other, but none matched the archetypes; a deeper investigation revealed that the instances generated during the same infection shared the same payload. After normalization, we noticed that the control-flow graphs were perfectly isomorphic if we didn't consider node labels. Through labeling, we identified four different types of archetypes that differed only in the labels of one or two nodes. In some cases, the nodes ended with a jump absent in the others; during normalization the tool couldn't remove these extra jumps because they were located exactly at the end of a real basic block. Overall, thanks to normalization, we observed an average reduction in the graph size of roughly 65 percent and the elimination of approximately half the payload instructions.

In our experiments, we have applied normalization directly to the malicious code. In the real world, however, the malware is tangled into the host code—we would perform its normalization implicitly when normalizing the entire host code. Unfortunately, our prototype isn't mature enough to handle big executables and, although we believe normalization will be quite effective on the executable that hosts the malicious instructions, we were unable to make this assessment. In fact, two problems might reduce normalization's effectiveness:

- Our tool might not be able to explore the benign host code completely, and the code that invokes the malicious guest lies in the unexplored region.
- Our tool could explore the benign host code completely, but couldn't resolve the links with the malicious code region.

Heuristics to maximize code exploration already exist, and we can adopt them to overcome the first problem (Boomerang already handles common cases[17]). The second problem is less worrying because normalization appears to be rather effective in reconstructing an obfuscated control flow, and malware uses the same techniques to hide malicious code among host instructions.

Code comparison evaluation

We've evaluated the code comparator via an independent test to measure its precision (more details are available

elsewhere[18]). First, we collected a huge set of system binary executables and constructed their augmented interprocedural control-flow graph. We then split this graph in order to construct a graph for each program function. We used the functions identified to simulate malicious code and searched within all the sample set's programs; we threw away graphs with fewer than five nodes because they're too small to unambiguously characterize a particular code fragment. We then divided the code comparator output into three sets:

- equivalent graphs generated from equivalent functions (two functions were equivalent if the hashes computed on their opcodes matched);
- equivalent graphs generated from functions with different hashes; and
- different graphs generated from functions with different hashes.

We then compared a small number of random elements of the last two sets to verify the presence of false positives and false negatives. Table 4 shows the results obtained through this manual inspection. Besides a bug found in the prototype, manual inspection highlighted that, in most cases, the compared functions were semantically equivalent even when the hashes didn't match (we suspect that the same function was compiled with slightly different options). False positives arose only when we compared very small graphs (fewer than seven nodes). Manual inspection also revealed that all graphs reported to be different were generated from different functions.

Table 4. Results from manual evaluation of a random subset of the code comparator results.

POSITIVE RESULTS (EQUIVALENT GRAPHS)	#	%
Equivalent code	35	70
Equivalent code (negligible differences)	9	18
Different code (small number of nodes)	3	6
Unknown (too big to compare by hand)	1	2
Bugs	2	4
NEGATIVE RESULTS (DIFFERENT GRAPHS)	**#**	**%**
Different code	50	100

Despite theoretical studies demonstrating that it's possible in principle to build undetectable malicious code, we've demonstrated that the techniques malicious code writers currently adopt to achieve perfect mutation don't let them get too close to the theoretical limit.

We believe that the experimental results we obtained regarding our normalization process demonstrates that it adequately treats techniques that self-mutating malware currently adopts. Unfortunately, we expect that in the near future, such transformations will be replaced with more sophisticated ones, which could seriously undermine the effectiveness of static analysis and, consequently, our proposed approach as well. These transformations could include the use of function calls and returns to camouflage intrafunction control-flow transitions; the introduction of opaque predicates; the introduction of junk code containing useless memory references that will create spurious data dependencies; and the adoption of antidisassembling techniques.

Another important issue we must address in the future is reducing the resources requested for the analysis. The static analysis we perform on the malicious code is in general quite expensive, so we believe that it's necessary to perform the analysis on the smallest portion of code possible, but this means that in the future, we must be able to identify which part of the code to focus on. □

References

1. P. Ször and P. Ferrie, "Hunting for Metamorphic," *Proc. Virus Bulletin Conf.*, Virus Bulletin, 2001, pp. 123–144.
2. M. Christodorescu and S. Jha, "Testing Malware Detectors," *Proc. 2004 ACM SIGSOFT Int'l Symp. Software Testing and Analysis* (ISSTA 04), ACM Press, 2004, pp. 34–44.
3. M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," *Proc. Usenix Security Symposium*, Usenix Assoc., 2003, pp. 169–186.
4. D.M. Chess and S.R. White, "An Undetectable Computer Virus," *Proc. Virus Bulletin Conf.*, Virus Bulletin, 2000; www.research.ibm.com/antivirus/SciPapers/VB2000DC.htm.
5. F.B. Cohen, *A Short Course on Computer Viruses,* 2nd ed., Wiley, 1994.
6. C. Collberg, C. Thomborson, and D. Low, *A Taxonomy of Obfuscating Transformations*, tech. report 148, Dept. of Computer Science, Univ. of Auckland, July 1997.
7. P. Ször, *The Art of Computer Virus Research and Defense*, Addison-Wesley, 2005.
8. A. Lakhotia, A. Kapoor, and E.U. Kumar, "Are Metamorphic Viruses Really Invincible?" *Virus Bulletin*, Dec. 2004, pp. 5–7.
9. S.K. Debray et al., "Compiler Techniques for Code Compaction," *ACM Trans. Programming Languages and Systems*, vol. 22, no. 2, 2000, pp. 378–415.
10. S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
11. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

12. J.R. Ullman, "An Algorithm for Subgraph Isomorphism," *J. ACM*, vol. 23, no. 1, 1976, pp. 31–42.
13. C. Kruegel et al., "Polymorphic Worm Detection using Structural Information of Executables," *Proc. Int'l Symp. Recent Advances in Intrusion Detection*, Springer, 2005, pp. 207–226.
14. C. Cifuentes and S. Sendall, "Specifying the Semantics of Machine Instructions," *Proc. 6th Int'l Workshop on Program Comprehension* (IWPC 98), IEEE CS Press, 1998, pp. 126–133.
15. C. Cifuentes and M.V. Emmerik, "Recovery of Jump Table Case Statements from Binary Code," *Proc. 7th Int'l Workshop on Program Comprehension*, IEEE CS Press, 2001, pp. 171–188.
16. D. Bruschi, L. Martignoni, and M. Monga, "Using Code Normalization for Fighting Self-Mutating Malware," *Proc. Int'l Symp. Secure Software Engineering*, IEEE CS Press, 2006, pp. 37–44.
17. L.P. Cordella et al., "A (Sub)graph Isomorphism Algorithm for Matching Large Graphs," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, 2004, pp. 1367–1372.
18. D. Bruschi, L. Martignoni, and M. Monga, "Detecting Self-Mutating Malware Using Control Flow Graph Matching," *Proc. Conf. Detection of Intrusions and Malware & Vulnerability Assessment* (DIMVA), Springer, 2005, pp. 129–143.

Danilo Bruschi *is a professor of computer sciences at Università degli Studi di Milano, Italy, where he is also director of the Master Program in ICT Security, director of the Laboratory for Security (LASER), and teaches computer and network security and operating systems. His main research interests include computer and network security, reliability, and survivability, computer forensics, social implications, and privacy. Bruschi has a PhD in computer sciences from the Università degli Studi di Milano. Contact him at bruschi@dico.unimi.it.*

Lorenzo Martignoni *is currently enrolled in the PhD program in computer science at Università degli Studi di Milano, Italy. His research interests include computer security and the analysis of malicious code and computer forensics in particular. Martignoni has an MS in computer sciences from Università degli Studi di Milano-Bicocca, Italy. Contact him at martign@dico.unimi.it.*

Mattia Monga *is an assistant professor in the Department of Computer Science and Communication at the Università degli Studi di Milano. His research activities are in software engineering and security. Monga has a PhD in computer and automation engineering from Politecnico di Milano, Italy. He is a member of the IEEE Computer Society and is on the steering committee of CLUSIT, an Italian association promoting awareness, continuous education, and information sharing about digital security. Contact him at monga@dico.unimi.it; http://homes.dico.unimi.it/~monga/.*