Detecting self-mutating malware using control-flow graph matching

Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga

Dip. Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 35, I-20135 Milan, Italy
{bruschi,martign,monga}@dico.unimi.it

Abstract. Next generation malware will by be characterized by the intense use of polymorphic and metamorphic techniques aimed at circumventing the current malware detectors, based on pattern matching. In order to deal with this new kind of threat novel techniques have to be devised for the realization of malware detectors. Recent papers started to address such issue and this paper represents a further contribution in such a field. More precisely in this paper we propose a strategy for the detection of malicious codes that adopt the most evolved self-mutation techniques; we also provide experimental data supporting the validity of such a strategy.

1 Introduction

Malware detection is normally performed by *pattern matching*. Detectors have a database of distinctive patterns (the *signatures*) of malicious pieces of code and they look for that in possibly infected systems. This approach is fast and, up to now, quite effective when it is used to find known viruses.

Such defences will probably be circumvented by the next generation malicious code which will intensively make use of metamorphism. This type of malware is not yet appeared in the wild, but some prototypes have been implemented (see for example MetaPHOR [2], Zmist [12], Evol) which have shown the feasability and the efficacy of mutation techniques [18]. Some papers recently appeared in literature [8, 7], have shown that current commercial virus scanners can be easily circumvented by the use of simple mutation techniques.

Various levels of code mutation have been individuated in literature, ranging from simple modifications (e.g. useless instructions insertion, and registers swapping) to the complete mutation of the payload. Probably the most advanced prototype in such a context is represented by the `Zmist` virus, which beside a metamorphic engine which takes care of changing the static structure of the payload, inserts itself into an executable code and scatters its body among the benign instructions (that are updated to reflect relocations). Malicious fragments are then connected together using appropriate control flow transition instructions. The malicious code will be executed when the normal control flow reaches its first instruction: this is known as *Entry Point Obfuscation* [4]. Threats such as those represented by the `Zmist` virus, poses three serious challenges to malware detectors:

– the ability to recognize self-mutating code code;
– the ability to recognize malware which is randomly spread in the original code;
– the ability to recognize code which does not modify neither the behavior nor the properties of the infected program.

Note also that in order to be effective a malware detector has to be able to solve the above challenges simultaneously.

The only viable way for dealing with such a kind of threat is the construction of detectors which are able to recognize malware's dynamic *behavior* instead of some *static properties* (e.g. fixed byte sequences or strangeness in the executable header). Recent papers ([9, 7, 14, 17]) started to address such issues and this paper represents a further contribution in such a field. More precisely in this paper we propose a strategy for solving the problems above mentioned, and we will also provide experimental data which indicate that such a strategy can represent a significant step towards the identification of novel techniques for dealing with the new forms of malware.

Roughly speaking the strategy we propose works as follows. Given an executable program P we perform on it a disassembling phase which produces a program P', on P' we perform a normalization phase aimed at reducing the effects of most of the well known mutations techniques and at unveiling the flow connection between the benign and the malicious code, thus obtaining a new version of P namely P_N. Subsequently given P_N we build its corresponding *labelled inter-procedural control flow graph* CFG_{P_N} which will be compared against the control flow graph of a normalized malware CFG_M in order to verify whether CFG_{P_N} contains a subgraph which is isomorphic to CFG_M, thus reducing the problem of detecting a malware inside an executable, to the subgraph isomorphism problem[1]. Using such a strategy we will be able to defeat most of the mutations techniques (see also [5] for further details) adopted for the construction of polymorphic malware as well as code scattering. Obviously, the strategy still need improvements, but the experimental results we obtained are really encouraging.

The paper is organized as follows. Section 2 describes some of the techniques that can be adopted by a malware to accommodate its payload within a benign program stealthily. In Section 3 we describe the approach we followed in order to treat the kind of malicious code. Section 4 briefly describes how our prototype was realized and discusses the experimental results obtained. Section 5 discussed related works and in the last section we draw our conclusion about the work presented.

[1] The subgraph isomorphism problem is a well known NP-complete problem, but in most of the instances we will consider it turns out to be tractable in an efficient way.

2 Concealing malicious code

In order to assess our approach, we experimented with a simulated malware that is able to insert its own payload (the *guest*) into another executable (the *host*). Our mock virus can inject its code in ELF executables, and it uses only basic modifications of the host code to accommodate the guest. The basic technique consists in localizing existing candidate insertion points. Moreover, new insertion points are added while guaranteeing that the host code continue to run mostly as in the past. The entry point of the guest is located somewhere in the executable code but its never referenced directly (in fact it is possible that it is never reached, and consequently, never executed), thus achieving complete entry point obfuscation. There are many different ways to perform code insertion and entry point obfuscation but we decided to analyze deeply only three of them, in order to keep our prototype simple enough, while being able to demonstrate that complete insertion, whose identification and reversal is not obvious, is possible.

A brief description of the techniques used by our prototype (which targets GNU/Linux IA-32 Elf executables) to achieve stealthiness follows.

2.1 Unused space between subsequent functions

The first technique we consider exploits a behaviour of most compilers which usually between a function epilogue and the next one prologue add some padding filled with NOPs. It is easy to find this unused space by trivial pattern matching: we used two different patterns: (i) `\x90{7,}` (i.e., more than 7 consecutive `nop`) and (ii) `\xc3\x90{7,}` (i.e. `ret` followed by more than 7 consecutive `nops`). The former is used to identify any type holes that do may accommodate a malicious code; the latter is used to identify holes that start just after the epilogue and that can potentially be reached by an execution flow[2]. Any hole of type (ii) can be used as the guest entry point by moving the `ret` instruction at the end of the `nops` padding and substituting `nops` with payload operations. (Figure 1 shows this kind of insertion). Holes of type (i) can also be used to insert arbitrary code, but this code must be reached by a control flow starting from an entry point created somewhere else, otherwise it will never be executed. This technique is known as *cavity insertion* [4].

During our experiments we discovered that insertion points of type (i) are pretty common (several occurrences per binary program), while insertion points of type (ii) are rather rare, although we found at least a candidate in virtually all the binaries we examined.

2.2 Manipulation of jump tables

Another technique we implemented for realizing entry point obfuscation is the jump-table manipulation. A *jump-table* is commonly used by compilers to im-

[2] This pattern does not correspond to a standard epilogue (i.e., `leave; ret`) because in several cases the `leave` instruction is substituted with some direct operations on the stack.

```
mov  %ebp,%esp        mov  %ebp,%esp
pop  %ebp             pop  %ebp
ret                   payload
nop                   payload
nop                   ...
...                   payload
nop                   payload
nop                   ret
push %ebp             push %ebp
mov  %esp,%ebp        mov  %esp,%ebp
```

Fig. 1. Insertion of the guest payload between two function boundaries

plement `switch`-like constructs. The right block of instructions is addressed by an indirect jump through the table, which is stored in an appropriate section of ELF executables, namely the `.rodata` section.

Jump tables can be exploited to inject malicious code in two conceptually analogous ways: (i) by replacing an entry with an address in which the new payload has been inserted and then link back the payload to the original target address or (ii) by moving a block of instructions addressed by an entry to a new location, while using the room just freed for the payload, augmented with a final jump to the original code.

We look for jump tables in executables by pattern matching in the text segment: we looked for `\x24\xff\d(addr)` (e.g. `jmp *addr(,%reg,4)`) where `addr`, or even simply `\d{4}`, must be an address belonging to the `.rodata` section. Once the absolute address of the jump table has been found, target addresses can be located just by extracting values starting from the beginning of the jump table and stopping when a value does not represent a valid `text` segment address.

2.3 Data segment expansion

The last technique we considered is based on creation of an hole in the `data` segment, hole which can then be used for any kind of purposes as the benign code is not aware of its presence and instructions in this segment can be normally executed on my architectures. In the following a brief description of such a technique is provided.

Figure 2 depicts the simplified layout of an ELF executable; the left picture shows the layout of the file while the one on the right shows the layout once the executable is loaded in memory; the `text` segment is depicted in white while the `data` segment in gray. The `data` segment of an executable is divided in several sections, the most important ones are `.data` and `.bss`. The former is used to hold initialized data and it is loaded from the executable file while the latter holds uninitialized data and has no file counterpart.

Since the `.bss` section is neither initialized nor stored on the file, it can be easily shifted in order to increase the space available for the `.data` section which always precedes `.bss`. Such a modification however would require that all the

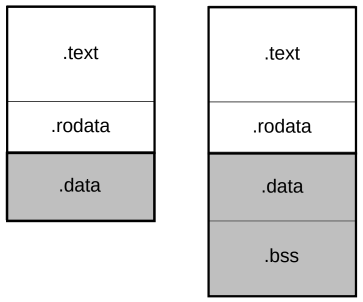

Fig. 2. Simplified layout of an executable.

instructions that reference the `.bss` section being updated. In order to avoid such an operation, an empty space of the same size of the original `.bss` is preserved in the expanded `.data`, and a new `.bss` section is mapped into a higher set of addresses. In such a way the code continues to refer to the old `.bss` section (see Figure 3). The new `.bss` and the hole created in `.data` can instead be used by the guest code for any kind of purpose.

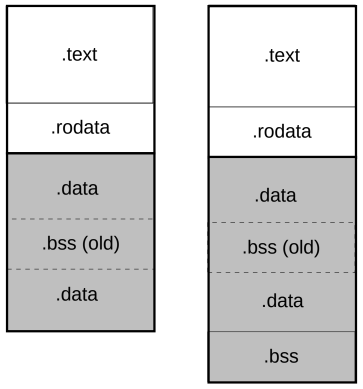

Fig. 3. Simplified layout of a manipulated executable with an expanded `.data`

3 Unveiling malicious code

The techniques described in the previous Section make malware detection rather problematic with respect to current anti-virus technology ([8, 7] witnessed the problem experimentally). In fact:

– *pattern matching* fails, since fragmentation and mutation make hard to find signature patterns;

– *emulation* would require a complete tracing of analyzed programs because the entry point of the guest is not known. Moreover every execution should be traced until the malicious payload is not executed;
– even *heuristics* based on predictable and observable alterations of executables could become useless when insertion is performed producing almost no alteration of any of the static properties of the original binary.

The core of the problem is that the malicious code seamlessly becomes part of the host program, thus making very difficult to distinguish between the two. In order to find out malware code we have to deal with both mutations and scattering.

As far as mutation is concerned, we aim at *normalizing* different instances of the same malicious code into a canonical and minimal version. Our previous experiments [5] showed that, by exploiting well known techniques of code optimization, it is possible to revert most of the mutations commonly used by malware. However, the lack of an easily guessable entry point makes things much more complicated. In fact, the detection can not be restricted to a limited set of instructions to check whether they can be considered equivalent (up to an acceptable threshold of accuracy) to malware code. The detection must consider every instruction in order to analyze if some groups of them, logically connected but physically separated by malicious scattering, match with the canonical version of malware under analysis.

In order to perform such tasks we devised a detection process which is composed by two different components: the *code normalizer* and the *code comparator*. The following sections describe them in details.

3.1 Code normalizer

The goal of the code normalizer is to *normalize* a program, i.e. transform it into a *canonical form* which is simpler in term of structure or syntax while preserving the original semantic. Most of the transformations used by malware to dissimulate their presence led to unoptimized versions of its archetype[3], since they contain some irrelevant computations whose presence has the only goal of hurdling recognition. Normalization aims at removing all the trash code introduced during the mutation process and thus can be viewed as an optimization of their code.

Decoding The executable machine code P is translated into a new representation P' that allows to describe every machine instruction in term of the operations it performs on the cpu. The goal is to increase, as much as possible, the level of abstraction and to express the program in a form that is more suitable for deeper analyses. P' will be the standard input to all the subsequent phases.

[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.

Control-flow and Data-flow analysis *Control-flow analysis* detects control flow dependencies among different instructions, such as dominance relations, loops, and recursive procedure calls. *Data-flow analysis* collects information about data relationship among program instructions. Particularly, all the definitions which can possibly *reach* each program instruction and all the definitions that are *live* before and after each instruction.

Code transformation Information collected through control-flow and data-flow analysis are used to identify which kind of transformations can be applied at any program point, in order to reduce it to the normal form. The transformation that can be successfully used to achieve our goal are compiler optimizations that are particularly suited for the reduction of the size of the code [3, 16], which, although developed to be used on source code, they have been showed to be suited also for machine executable code [11].

More practically, normalization allows to:

– identifying all the instructions that do not contribute to the computation (dead and unreachable code elimination);
– rewriting and simplifying algebraic expressions in order to statically evaluate most of their sub-expressions that can be often removed;
– propagating values assigned or computed by intermediate instructions, and assigned to intermediate variables into the instructions that make use of these values in order to get rid of the intermediate variables previously needed only for their temporary storage (constant and expression propagation);
– analyze and try to evaluate control flow transition conditions to identify tautologies, and rearrange the control flow removing dead paths;
– analyze indirect control flow transitions to discover the smallest set of valid targets and the paths originating. It is worth nothing that the connections between the benign and the malicious code are concealed behind these layers of indirections.

Although the analysis involves every program instructions, we expect that most of the candidate transformation targets are those that belong to the malicious code since host programs are usually already optimized during compilation.

Limitations of static analysis As just mentioned, more accurate the code normalizer is, major are the chances of recognizing a given malware. Unfortunately there exist transformations that can be very difficult to revert and situations in which normalization can not be performed on the entire code.

The use of *opaque predicates* [10] during the mutation can complicate the detection because the code produced, once normalized, may have different shapes. A predicate is defined opaque if its value is known a priori during obfuscation but it is difficult to deduce statically after obfuscation has been applied. Opaque predicates, which are generally used in code obfuscation and watermarking, allow to distort the control flow graphs inserting new paths that will not be removed during normalization unless the predicate can be evaluated, and the evaluation usually is very expensive or unfeasible.

The adoption of anti-analysis techniques by the malware, is a further problem for malware detection. Within this category fall anti-disassembling techniques [15] which can be employed to prevent a precise decoding of programs. We voluntarily neglected this problem because we assumed that self-mutating malicious codes need to be able to analyze their own code in order to generate the new one, thus they must be able to decode themselves and if they can, at least a decoder must exist.

The presence of indirection (where by indirection we mean a control flow transition that reference the target through a variable), in the analyzed code, could lead to an incomplete exploration of the code itself. In such a case if the malicious code, or at least its entry point, resides in the unexplored region, the corresponding control flow graph will not be complete and the presence of the malicious code will never be detected. The data-flow analysis performed during normalization plays a fundamental role in the resolution of indirections but it may miss to solve some of them; some heuristics could be adopted in order to exhaustively identify code regions and explore them.

Notwithstanding these limitations our experiments (see [5]) showed that normalization can be used effectively in most of the cases.

3.2 Code comparator

Given a program P and a malicious code M as input the code comparator answers to the following question: *is the program P hosting the malware M?* or more precisely, *is an instance of M present inside P?* The code comparator does not work directly on the native representation of the two inputs but instead it works on the normalized form P, namely P_N. Obviously we cannot expect to find, a perfect matching of M in P_N, as M is self-mutating, and even if most of the mutations it suffered have been removed through the code normalizer, we expect that some of them remain undiscovered. Therefore, the code comparator must be able to cope with most of these differences, which we observed are normally local to each basic block[4]. As a consequence, the basic control flow structure (as results from normalization) is in general preserved by mutations.

Thus, we decided to represent the malicious code and the alleged host program by their *inter-procedural control flow graphs.* A control flow graph (CFG) is an abstract representation of a procedure: each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block and jumps end a block. Directed edges are used to represent jumps in the control flow. An inter-procedural CFG links together the CFGs of every function of a program.

Under this assumption, the search for malicious code can be formulated as a subgraph isomorphism decision problem: *given two graphs G_1 and G_2, is G_1*

[4] A *basic block* is a sequence of instructions in which every of them always executes before all the subsequent ones.

isomorphic to a subgraph of G_2? Fig. 4 shows the two graphs just mentioned: the first one models the searched malicious code and the second one the program which is going to be analysed in order to verify if it is hosting the malicious code. We briefly recall that sub-graph isomorphism is an NP-complete problem in the general case, but in our particular case, characterized by highly sparse graphs, it turned out to be computable in a very efficient way.

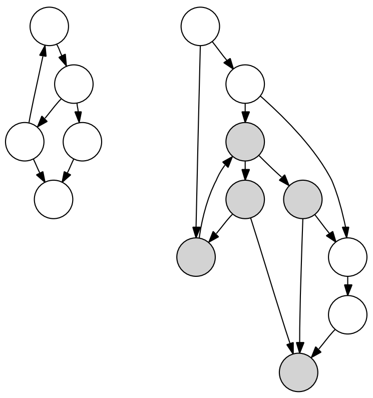

Fig. 4. The graphs representing a malicious code M and a generic normalized program P_N. The nodes highlighted in gray are those of P_N program matching the ones of M.

As comparison through raw inter-procedural control flow graphs is too coarse we decided to augment these graphs *labelling* both nodes and edges: nodes are labelled according to the properties of the instructions belonging to them and edges are labelled according to the type of the flow relations between the nodes they connect. The labelling method we decided to adopt is very similar to the one proposed in [14]. Instructions, similar from the semantic point of view, are grouped together into classes and the label assigned to each node is a number that represents the set of classes in which, instructions of the node, can be grouped. Edges are labelled in the same way: possible flow transitions are grouped into classes according to the type of each transition. Table 1 shows the classes in which we decided to group instructions and flow transitions. Calls to shared library functions are also represented with the same notation: the caller node is connected to the function that is represented with just one node and which is labelled with a hash calculated starting from the function name.

It is important to note that the normalization allows reduce the number of possible classes because assembly instructions are converted into the intermediate representation which explicitly describes each instruction in term of the actions it performs on the CPU. Instructions like `push` and `pop` do not require a dedicated class because they are translated in assignment and integer arithmetic instructions.

Instruction classes	Flow transition 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	

Table 1. Instructions and flow transition classes.

The comparison method gracefully handles malicious code fragments scattered anywhere, no matter where they are located, on condition that it is possible to find out the connections existing among them. That is possible because, for the way in which the problem has been formulated, no information about the physical location of the fragments and about the properties of these locations are considered. For example, a malicious code that, during execution, jumps from the `text` to the `data` segment, and vice versa, or that jumps across different functions is treated as exactly one that would jump to an adjacent memory address. It is a code normalizer duty to unveil the connections existing among the fragments composing the malware.

4 Prototype implementation and experimental results

A prototype has been built in order to verify experimentally our idea both in terms of correctness but also in terms of efficiency. We build our code normalization on top of Boomerang [1], which is an open source decompiler which reconstructs high-level code starting from binary executables. Boomerang allowed us to perform the data-flow and control-flow analysis directly on machine code. We adapted it in order to better handle the set of transformations, previously described, needed for removing the mutations and bring a malware back to its original form. The set of transformations to apply to a code are decided on the basis of the results of control and data flow analysis. The analysis framework we considered is also capable of accommodating the resolution of indirections and to perform jump-table and call-table analysis.

Once the transformations above described are performed on an executable, a labelled control flow of the resulting code is built and it is fed, along with the control flow of a malware, to a sub-graph isomorphism algorithm in order to perform the detection phase. For such a task we referred to the VF2 algorithm contained in the VFlib [13] library.

4.1 Code normalization evaluation

The effectiveness of code normalization was evaluated in [5] by using the MetaPHOR [2] virus. A big set of virus samples, about 115, was normalized in order to compare the original form with the new one. We observed that the effectiveness of the approach has been confirmed by the fact that all the samples assumed the same shape and that their labelled control flow graphs can be considered isomorphic.

As all possible kind of transformations have been successfully applied during the samples normalization, we believe that the same encouraging results can be obtained when the same approach is used in order to discover the link between the host code and the malicious code entry point because it camouflaged in the same way in which links among malicious code fragments are.

A measure of the time efficiency of this step of the detection process has been performed. It turned out that the time required to normalize small fragments of code composed by few functions, and noticed that the time ranges from 0.2 secs. to 4.4 secs. This data indicates that such a phase will probably be very time consuming with big executables.

4.2 Code comparison evaluation

In order to evaluate the correctness of our approach we performed a set of experimental tests on a huge set of system binary executables. The executables have been picked up from a GNU/Linux distribution. Subsequently they have been processed in order to construct their inter-procedural augmented control flow graphs, from whom the graphs associated to each program function have been generated; duplicated functions have been thrown away[5]. During our preliminary experiments we noticed that small graphs (4 or less) are not suited to describe unambiguously some particular code fragments. For this reason we decided to throw away from our sample set all graphs with 5 or less nodes. Functions, or standalone code fragments, with such a small number of nodes cannot represent a computation that, from the detection point of view, can be considered "typical". Table 2 summarized the characteristics of our sample.

Type	#
Executables	572
Functions (with more then 5 nodes)	25145
Unique functions (with more then 5 nodes)	15429

Table 2. Sample set used during our experiments.

The unique functions (functions found in more then one executable were used only once) identified were used to simulate malicious codes and we look for their

[5] Two functions are considered equivalent if the MD5s, computed on the strings built using the first byte of any machine instruction composing their code, match.

occurrences within the programs of the sample set using our code comparator module. The code comparator reported 55606 matches. In order to evaluate the correctness of such a data we compared it against the results returned by comparing the fingerprints of the considered codes. Note that the fingerprinting method produces almost no false positive, while it can have false negative. It turned out that 96.5% (53635) of the matches found, were confirmed also by the fingerprint method. The two methods instead disagree on the remaining 3.5% (1971) of the samples, for our comparator these were instances of the simulated malicious code while this was not true for the fingerprinting method. As in such a case even the fingerprinting method can be wrong we deepen our analysis, in order to have a better estimate of the false positive ratio of our code comparator. For this reason we randomly chosen, among the subset of the sample on which the two method disagreed, a set E of 50 potentially equivalent pairs of code fragments and inspected them manually. The same was done with a set NE of 50 potentially different pairs of code fragments. The results of our evaluation are reported in Table 3. With the exception made for a few cases, involving rather small graphs, it turned out that our code comparator was correct in determining the equivalence of members of E. Some code fragments required a thorough analysis because of some differences local to the nodes of the graphs (probably the same code compiled in different moment) that, after all, turned out to be equivalent. Other few cases highlighted a bug in the routine that performs labelling[6] and another case involved two enormous graphs there were not possible to compare by hand. With respect to the member in NE all the results of the code comparator were confirmed by the manual inspection.

Positive results	#	%
Equivalent code	35	70
Equivalent code (negligible differences)	9	18
Different code (small number of nodes)	3	6
Unknown	1	2
Bug	2	4

Negative results	#	%
Different code	50	100

Table 3. Manual evaluation of a random subset of the results returned by the code comparator.

Even if sub-graph isomorphism is an NP-complete problem in the case of general graphs, the particular instances of graphs we are dealing with make it well tractable. A generic inter-procedural control flow graph has a huge number of

[6] The prototype was not able to find out the name of two shared library functions, assumed they were unknown and considered them equivalent. The two codes that, apart from the different functions called were equivalent, were erroneously considered equivalent.

nodes but it is highly sparse, in fact the average density we measured (measured as $|E|/|N|^2$, where E is the set of edges and N the set of nodes) was about 0.0088.

In order to verify this assumption we measured the time requested to perform the matching. We decided to distinguish between: (i) average time required to load a graph measured with respect to the number of nodes in the graph and (ii) worst cases time required to perform a complete search within the graph representing host programs under verification (no distinction has been made between positive and negative matches). These measures, collected through a GNU/Linux system with a IA-32 1GHz processor, are reported in Fig. 4. In particular, the data provided shows that the critical phase of the entire procedure is not related to the computation time but instead to the initialization of the requested data structures. A quick glance at the code of the library used to perform the matching highlighted that when a graph is loaded, internal data structures are filled in $O(n^2)$ (n stands for the number of nodes) but we believe that the same goal can be achieved in $O(n)$.

# nodes	**Average load time (secs.)**	**Worst detection time (secs.)**
0 - 100	0.00	0.00
100 - 1000	0.09	0.00
1000 - 5000	1.40	0.05
5000 - 10000	5.15	0.14
10000 - 15000	11.50	0.32
15000 - 20000	28.38	0.72
20000 - 25000	40.07	0.95
25000 - 50000	215.10	5.85

Table 4. Summary of the measured average load time and of the worst detection time with regards to the number of nodes.

5 Related works

The problem of the detection of mutating malware is not new and the first theoretical studies about the problem [6] have posed serious worryings demonstrating that there could exist malware which no algorithm can detect. Some paper appeared recently in the literature started to pragmatically address the problem of the detection of evolved malicious code with mutating capabilities. Different approaches have been proposed in order to target specific types of malware and specific propagation methods.

The first work which addressed the problem of the detection of obfuscating malware through static analysis was done by Christodorescu and Jha [7] which has been refined in [9]. In their first work annotation of program instructions has been used in order to provide an abstraction from the machine code and to detect common patterns of obfuscation; malicious codes were then searched directly on the annotated code. In their second work deobfuscation through annotation has been replaced by a more sophisticated set of complementary techniques that are used in concomitance to corroborate the results. The techniques adopted provide different levels of resilience to mutation and ranges from the detection of previously known mutated patterns to the proof of instructions sequences equivalence through the use of theorem prover and random execution. Our work shares the same goals but adopts a different strategy which consists in the most complete defection of mutations through normalization.

Polygraph [17] targets polymorphic worms and is capable of automatically determining appropriate signatures. Signatures are generated starting from network streams and consist of multiple disjoint content substrings which are supposed to be shared among different instances; these invariant substrings consists of protocol framing, return addresses, and poorly obfuscated code.

In [14] an algorithm for the detection of unknown polymorphic worms is presented. The algorithm compares the similarity of apparently independent network streams in order to discover if they are carrying the same malicious code. Each network stream is processed in order to identify potential executable code by trying to disassemble each stream and to generate the appropriate control flow graph which is then divided in little sub-graphs in order to fingerprints the stream. Our comparison method share some similarities with the one proposed in the paper: we also represent executables code trough labelled control flow graphs (we work with them in their entirety) but we adopted different comparison strategies and performed normalization because we treated a different type of malicious codes that adopt more sophisticated anti-detection techniques. We believe that the normalization techniques we have proposed can be used to improve the detection power making the system no more susceptible to malware that could adopt more sophisticated mutations.

6 Conclusions and future works

Despite theoretical studies demonstrated that there could exist an undetectable malicious code we have given our contribution in demonstrating that the techniques currently adopted by malicious code writer in order to achieve perfect mutation do not allow to get so close to the theoretical limit.

We analyzed the type of transformations adopted to implement self-mutating malware in order to avoid detection and we convinced ourselves that the only viable way for dealing with such a kind of threat is the construction of detectors which are able to characterize the dynamic behavior of the malware.

We have proposed a pragmatical approach that is able to cope quite well with this treat which is based on (i) the defection of the mutation process and (ii) the analysis of a program in order to verify the presence of the searched malicious code. Mutation process is reverted through code normalization and the problem of detecting malware inside an executable is reduced to the subgraph isomorphism problem: a well known NP-complete problem that nevertheless can be efficiently computed when sparse graphs are concerned.

We believe that experimental results are encouraging and we are working on refining our prototype in order to validate it in more real scenarios.

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. C. Associates. Security advisor center glossary. `http://www3.ca.com/securityadvisor/glossary.aspx`.
5. D. Bruschi, L. Martignoni, and M. Monga. Using code normalization for fighting self-mutating malware. In *To be presented at ISSSE06*.
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. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
11. S. K. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, 2000.
12. P. Ferrie and P. Ször. Zmist opportunities. *Virus Bullettin*, 2001.
13. P. Foggia. The VFLib graph matching library, version 2.0. `http://amalfi.dis.unina.it/graph/db/vflib-2.0/`.
14. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *International Symposium on Recent Advances in Intrusion Detection*, 2005.
15. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM Press.
16. S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
17. J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, 2005.
18. P. Ször and P. Ferrie. Hunting for metamorphic. In *Proceedings of Virus Bulletin Conference*, Sept. 2001.