Detecting Self-Mutating Malware Using Control-Flow Graph Matching

Danilo Bruschi **Lorenzo Martignoni** Mattia Monga

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
{bruschi,martign,monga}@dico.unimi.it

Conference on Detection of Intrusions and Malware & Vulnerability Assessment – 2006

Outline

Code Obfuscation and Self-mutation
- Strategies adopted to achieve self-mutation and code insertion
- Challenges for the detection

Unveiling malicious code
- Code normalization
- Code comparison

Prototype implementation

Experimental results

Summary and future works

Code obfuscation and self-mutation

- Code obfuscation is a semantic-preserving program transformation that can be used to make a program harder to understand
- Self-mutation is a particular form of code obfuscation, which is performed automatically by the code on itself
- Self-mutation is adopted by malicious code to defeat detectors
- Self-mutation is applied during malicious code replication to generate completely new different instances

Self-mutation

Common transformations adopted to achieve self-mutation:

- Substitution of instructions
- Permutation of instructions
- Garbage insertion
- Substitution of variables
- Control flow alteration

Signature matching becomes useless

Self-mutation

Common transformations adopted to achieve self-mutation:

- Substitution of instructions
- Permutation of instructions
- Garbage insertion
- Substitution of variables
- Control flow alteration

Signature matching becomes useless

Code insertion

Common techniques adopted for malicious code insertion:

- Cavity insertion
- Jump tables manipulation
- Data segment expansion

The malicious code is seamless integrated into the host code

Code insertion

Common techniques adopted for malicious code insertion:

- Cavity insertion
- Jump tables manipulation
- Data segment expansion

The malicious code is seamless integrated into the host code

Challenges for the detection

Conventional detection techniques are likely to fail:

- Pattern matching fails since fragmentation and mutation make hard to find signature patterns
- Emulation would require a complete tracing of analyzed programs as the entry point of the guest is not known; moreover every execution should be traced until the malicious payload is not executed
- Heuristics based on ad-hoc predictable and observable alterations of executables become useless when insertion is performed producing almost no alteration of any of the static properties of the original binary

Theoretical studies (Chess & White) demonstrated that perfect detection of a self-mutating malware is an undecidable problem

Challenges for the detection

Conventional detection techniques are likely to fail:

- Pattern matching fails since fragmentation and mutation make hard to find signature patterns
- Emulation would require a complete tracing of analyzed programs as the entry point of the guest is not known; moreover every execution should be traced until the malicious payload is not executed
- Heuristics based on ad-hoc predictable and observable alterations of executables become useless when insertion is performed producing almost no alteration of any of the static properties of the original binary

Theoretical studies (Chess & White) demonstrated that perfect detection of a self-mutating malware is an undecidable problem

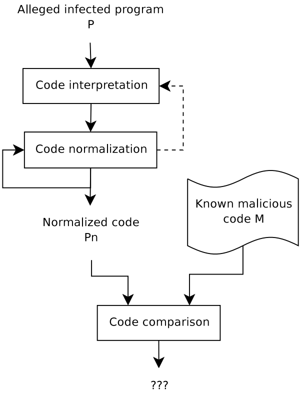
Devised strategy

Code interpretation and normalization

- Given a piece of code P which represents (or contains) an instance of a self-mutating malware we automatically revert all the mutations performed on it
- P is consequently reduced into a form, P_N, which is pretty close to its archetype M and which can be recognized more easily

Code comparison

- Detection is performed by looking for known abstract patterns into the transformed program P_N



Code normalization

Code normalization

A program is transformed into a canonical form which is simpler in term of structure or syntax while preserving the original semantic and that is more suitable for comparison

- Analysis of the transformations adopted to implement self-mutation and experimental observations highlighted some weakness:
  - Transformations led to the generation of useless computations
  - Most transformations are invertible
- Different instances of the same malware can be viewed as under-optimized version of the archetype; the archetype is consequently the normal form of the malicious code
- Code normalization can be performed adopting some of the well known techniques used by compiler to produce compact and efficient code

Code normalization

Some details

- Executable code is disassembled and translated into an intermediate form to explicit the semantic of each machine instruction
- Control-flow analysis and data-flow analysis are performed on the code to collect information that will be used by the next step
- Code transformations aim at:
  - Identify all the instructions that do not contribute to the computation (dead and unreachable code elimination)
  - Rewrite and simplify algebraic expressions in order to statically evaluate most of their sub-expressions (algebraic simplification)
  - Propagate values computed by intermediate instructions to the appropriate use sites (expressions propagation)
  - Analyze and try to evaluate control-flow transition conditions to identify tautologies and to rearrange the control to reduce the number of flow transitions (control-flow normalization)
  - Analyze indirect control flow transitions to discover the smallest set of valid targets and the paths originating (indirections resolution)

Code comparison

Given the normalized program we need to answer the question:

"is the program P_N hosting the malware M?"

- We cannot expect to find a perfect matching of M in P_N even if most of the transformations have been reverted
- The code comparator must be able to cope with some impurities left by normalization (we observed that these impurities are always local to basic blocks)
- The normalized control-flow of the malware is constant

Code comparison

Some details

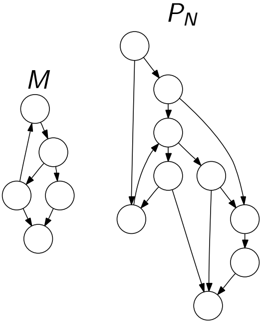
- P_N is represented through its interprocedural-control flow graph (ICFG) and M through its control-flow graph
- The malicious code detection can be formulated as a subgraph isomorphism decision problem: "given two graphs G_1 and G_2, is G_1 isomorphic to a subgraph of G_2?" (G_1 is M and G_2 is P_N)
- The graphs are augmented with labels to achieve the necessary trade-off between precision and abstraction (to handle possible impurities)
- Instructions and flow transitions are partitioned into classes; labels describe the set of classes in which instructions of a basic block can be grouped

Instruction classes
Integer arithmetic
Float arithmetic
Logic
Comparison
Function call
...

Code comparison

Some details

- P_N is represented through its interprocedural-control flow graph (ICFG) and M through its control-flow graph
- The malicious code detection can be formulated as a subgraph isomorphism decision problem: "given two graphs G_1 and G_2, is G_1 isomorphic to a subgraph of G_2?" (G_1 is M and G_2 is P_N)
- The graphs are augmented with labels to achieve the necessary trade-off between precision and abstraction (to handle possible impurities)
- Instructions and flow transitions are partitioned into classes; labels describe the set of classes in which instructions of a basic block can be grouped

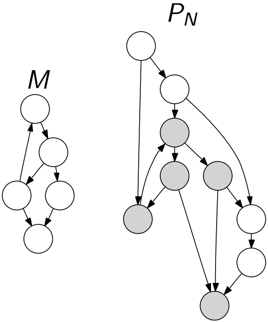


Instruction classes
Integer arithmetic
Float arithmetic
Logic
Comparison
Function call
...

Code comparison

Some details

- P_N is represented through its interprocedural-control flow graph (ICFG) and M through its control-flow graph
- The malicious code detection can be formulated as a subgraph isomorphism decision problem: "given two graphs G_1 and G_2, is G_1 isomorphic to a subgraph of G_2?" (G_1 is M and G_2 is P_N)
- The graphs are augmented with labels to achieve the necessary trade-off between precision and abstraction (to handle possible impurities)
- Instructions and flow transitions are partitioned into classes; labels describe the set of classes in which instructions of a basic block can be grouped

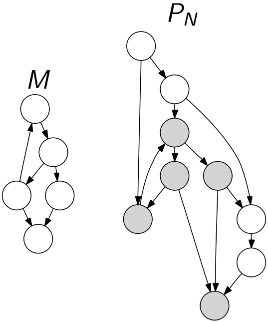


Instruction classes
Integer arithmetic
Float arithmetic
Logic
Comparison
Function call
...

Code comparison

Some details

- P_N is represented through its interprocedural-control flow graph (ICFG) and M through its control-flow graph
- The malicious code detection can be formulated as a subgraph isomorphism decision problem: "given two graphs G_1 and G_2, is G_1 isomorphic to a subgraph of G_2?" (G_1 is M and G_2 is P_N)
- The graphs are augmented with labels to achieve the necessary trade-off between precision and abstraction (to handle possible impurities)
- Instructions and flow transitions are partitioned into classes; labels describe the set of classes in which instructions of a basic block can be grouped



Instruction classes
Integer arithmetic
Float arithmetic
Logic
Comparison
Function call
...

Prototype implementation

- The code normalizer is built on top of BOOMERANG, an open-source decompiler:
  - Translate machine code into the intermediate form through a recursive disassembler
  - Performs data-flow analysis on the intermediate form
  - Performs the normalization steps previously described (some of the transformation have been extended to suit our needs)
  - Able to solve know patterns of indirection
- The prototype receives an executable files and emits its normalized $ICFG_{P_N}$
- The $ICFG_{P_N}$ of the normalized program and the CFG_M of the searched malware are then fed to the VFLIB2 library which is used to identify possible matches
- In case of match the comparison routine returns the set of $ICFG_{P_N}$ nodes that match the ones of the CFG_M

Experimental results

Two independent tests were performed:

1. Evaluation of code normalization effectiveness:
   - Several instances of the same self-mutating malicious code (the virus MetaPHOR) were collected and normalized
   - The normalized control-flow graphs were all isomorphic, they were not before
2. Evaluation of code comparison precision:
   - Different executables were collected and their ICFGs were built
   - Each procedure CFG was used to simulate malicious code and searched inside the ICFGs
   - The results of the subgraph isomorphism detection procedure were compared with the results obtained through code fingerprinting
   - A random set of alleged false-positives and false-negatives were selected and inspected by hand

Experimental results

Some numbers

Type	#
Executables	572
Functions (# nodes > 5)	25145
Unique functions (# nodes > 5)	15429

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

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

Summary

- We proposed a general strategy, based on static analysis, that can be used to pragmatically fight malicious codes that adopt self-mutation to circumvent detectors
- We developed a prototype tool and used it to show that a malware that suffers a cycle of mutations in most cases can be brought back to a canonical shape that is shared among all instances
- We showed that augmented control-flow graphs are well suited to describe a peculiar piece of code and that reliable code identification can be formulated as a subgraph isomorphism decision problem
- Although the subgraph isomorphism is a NP-complete problem, our particular instance seems to be tractable (the graphs we are dealing with are very sparse)

Future works

- Extend our prototype to perform normalization on real world executables and increase the effectiveness of normalization by extending the quality of the analysis performed
- Evaluate algorithms for partial subgraph isomorphism matching and the benefits they could give in our context
- Perform more exhaustive experiments using new malicious code
- Investigate attacks and countermeasures to defeat static analysis

Thank you!