The Grand Challenge in Metamorphic Analysis

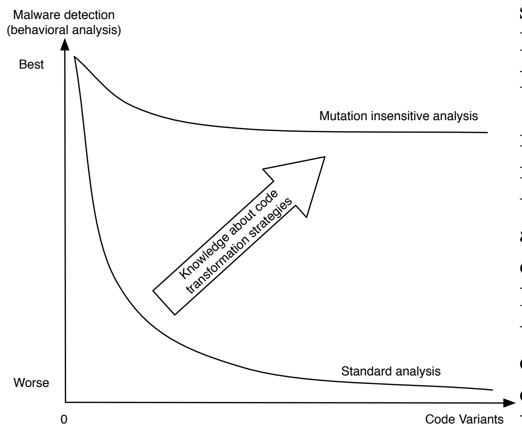
Mila Dalla Preda

Department of Computer Science/INRIA, University of Bologna, Italy.
E-mail: dallapre@cs.unibo.it

Abstract. Malware detection is a crucial aspect of software security. Malware typically recur to a variety of disguise and concealing techniques in order to avoid detection. Metamorphism is the ability of a program to mutate its form yet keeping unchanged its functionality and therefore its danger in case of malware. A major challenge in this field is the development of general automatic/systematic detection techniques that are able to catch the possible variants of a metamorphic malware. We take the position that the key for handling metamorphism relies in a deeper understanding of the semantics of the metamorphic malware. By applying standard formal methods we aim at proving that metamorphic analysis is a special case of program analysis, where the object of computation is code interpreted as a mutational data structure.

1 Metamorphic Malware Analysis

Detecting and neutralizing malware is a major challenge in computer security involving both sophisticated intrusion detection strategies and code manipulation tools and methods. Traditional misuse (or signature-based) malware detectors are syntactic in nature: They use pattern matching to compare the byte sequence comprising the body of the malware against a signature database [23]. Metamorphism emerged in the last decade as an effective strategy to foil misuse malware detectors. Metamorphic malware apply semantics preserving transformations (e.g. code obfuscation techniques) to modify their own code so that one instance of the malware bears very little resemblance to another instance even though semantically their functionality is the same. Thus, a metamorphic malware is a malware equipped with a metamorphic engine that takes the malware, or parts of it, as input and morphs it at run-time to a syntactically different but semantically equivalent variant, in order to foil signature matching. The quantity of metamorphic variants possible for a particular piece of malware makes it impractical to maintain a signature set that is large enough to cover most or all of these variants, making standard signature-based detection ineffective [5]. The reason for this vulnerability to metamorphism lies upon the purely syntactic nature of most exiting and commercial detectors that ignore program functionalities. Following this observation researchers began to develop detection techniques that take into account properties of the malware behavior instead of properties of its syntax. This naturally needs sophisticated program and behavioral analysis techniques, that rely upon known and new formal methods for reasoning about programs that mutate their code during execution.



As far as data/control flow analysis is concerned, program and behavioral analysis are standard in programming languages and system [20]. The situation changes when metamorphism is considered. The main difficulties in applying these techniques to metamorphic code analysis relies upon the fact that code mutation and data/control behavior are interleaved. This makes the first interfering with the second making the analysis impossible or imprecise enough to let malware be indistinguishable from good software.
The reason of this difficulty is twofold: (1) The code of the malware is not fixed, it mutates during execution or may take on extremely different shapes when caught in the wild or executed in an emulator or debugger. An adequate semantics, on which any sound analysis has to be based, has to cope with this aspect of metamorphic code, either by keeping track of code mutations in order to model similarities or being insensitive to these mutations in order to understand malware attacks; (2) It is extremely difficult to isolate the code portions devoted to code mutations, the so called *metamorphic engine*, being this code hardly obfuscated and interleaved into the malware payload. The analysis has therefore to cope with mixed (hybrid) computations involving standard data structures (the payload) as well as the code itself as a data structure.

2 Behavioral Approaches to Metamorphism

Nowadays, in the literature we can find a variety of detection algorithms that use standard formal methods and program analysis tools to model the malicious behavior in order to detect malware. Most of these tools and methods are based on the idea that a model of the behavior of a malware may be a valid signature for catching it. This is indeed in the tradition of intrusion detection systems (IDS), where an attack is essentially captured by understanding the attacker behavior in terms of which and how data are manipulated. Christodorescu et al. [6] put forward a very first semantics-aware malware detector that is able to handle some of the metamorphic transformations commonly used by hackers. Singh and Lakhotia specify malicious behaviors through a formula in linear temporal logic (LTL), expressing temporal properties of malware behavior relatively to some state properties, and then use the model checker SPIN to check if this property is satisfied by the control flow graph of a suspicious program [22]. Kinder et al. [15] introduce an extension of the CTL temporal logic, which is able to express some malicious properties that can be used to detect malware through standard model checking algorithms. Christodorescu and Jha [4] describe a malware detection system based on language containment and unification. The malicious code and the possibly infected program are modeled here as automata. In this setting, a program presents a malicious behavior if the intersection between the language of the malware automaton and the one of the program automaton is not empty. Beaucamps et al. [2] approximate the set of possible execution traces of a program with a regular language. They define an abstraction of this regular language with respect to some predefined behavioral patterns that express a certain property of the malware behavior (an invariant of the metamorphic transformations used by the malware). This leads to a description of a program as a regular language of abstract symbols that can be compared to the one of known malware to detect infection. A similar approach has been considered more recently in [1], where a tree automata is derived from system call data-flow dependency graphs, which is insensitive on code mutation. Lo et al. [18] develop a programmable static analysis tool, called MCF (Malicious Code Filter), that uses program slicing and flow analysis to detect malicious code. Lakhotia et al. [17] propose a methodology based on program semantics and abstract interpretation for making context-sensitive analysis of assembly programs even when the call and ret instructions are obfuscated. Jacob et al. [14] propose a model of malware based on the *Join Calculus* and they identify a fragment of the Join calculus where the malware detection problem becomes decidable. All these approaches share a common pattern: They consider a set T of metamorphic transformations commonly used by malware (e.g. variable renaming, code permutation, junk insertion) and then they develop an abstract behavioral model for programs that ideally captures the maliciousness of a program while abstracting form those details that are susceptible to metamorphism, namely that can be changed by the transformations in T (for example symbolic names can be used to handle variable/location renaming). Thus, the design of the abstract model is driven by the considered set T of code transformation. Of course, researchers can recur to any existing tool for the static analysis of programs in order to define the abstract behavioral model of the malware (e.g. model checking, program semantics, abstract interpretation, language theory, data mining). In this context, the process of detecting a malware based on some given behavioral model can be viewed as the process of abstracting its semantics. It is known that abstract interpretation [7, 8] can be used to characterize the obfuscating behavior of any metamorphic transformation in terms of the most concrete semantic property it preserves [11]. Moreover, any abstract behavioral model of programs obtained through static analysis can be expressed as an abstract interpretation of standard trace semantics [7]. This observation lead us to the definition of a general purpose framework based on a formal model of program semantics (trace semantics) and abstract interpretation for proving soundness (no false positives) and completeness (no false negatives) of malware detectors in the presence of metamorphism [10]. This means that the detection strategy and the metamorphic transformation can both be characterized as proper abstractions of program trace semantics. The idea is to use standard trace semantics to describe the concrete behavior of programs and malware, and abstract interpretation to model both the semantic properties preserved by the metamorphic transformation and the behavioral model employed by the detection strategy. Related works that address the analysis of self-modifying code with respect to a different semantics model based on Hoare Logic are the ones of Cai et al. [3] and Myreen [19].

One of the main limits of all the these formal behavioral approaches to metamorphic malware detection resides in the fact that they all assume to know the metamorphic transformations used by the malware. In fact, the design of the abstract model that specifies the behavior of programs is always driven by the obfuscating transformations used by the metamorphic engine. This makes the analysis mutation insensitive. Of course, a malware writer who has access to the detection algorithm, or who is aware of the set of basic transformations T used for deriving the abstract semantics, can exploit this knowledge in order to design new and ad-hoc obfuscation technique to bypass detection, even by simple modifications of the existing ones. As the malware detection problem is in general undecidable, for any given malware detector it is always possible to design an obfuscation that defeats that detector. We believe that a deeper understanding of the semantics of metamorphic malware, involving both the payload and the metamorphic engine could lead to a more robust detection system that is not based on the knowledge of the metamorphic techniques used by the malware and is mutation insensitive. The idea is to consider the metamorphic malware as a unique program, acting both as a standard program which modifies memory, and as a program modifying the code structure, which is also a data-structure.

3 Semantics-Based Learning Metamorphism

The *grand challenge* in metamorphic malware detection is to make behavioral analysis mutation insensitive. This means catching a signature which is durable and specific for a wide range of mutations of the malware. In [9] we propose a different approach to metamorphic malware detection based on the idea that extracting metamorphic signatures is approximating malware semantics, where the term metamorphic signature refers to any (possibly decidable) approximation of the properties of code evolution. The code is therefore viewed as a mutational data-structure, and approximating its shape consists in approximating the possible mutations of the malware. We face the problem of determining how code mutates, yet catching properties of this mutation, without any a priori knowledge about the implementation of the metamorphic transformations. We use a formal semantics to model the execution behavior of self-modifying code commonly encountered in malware. Using this as the basis, we developed a theoretical model for statically deriving, by abstract interpretation, an abstract specification of all possible code variants that can be generated during the execution of a metamorphic malware. The mixed computations on code and data are represented, and separated, in the so called *phase semantics*. The idea is to partition each possible execution trace of a metamorphic program into phases, each collecting the computations performed by a particular code variant. Thus, the sequence of phases (once disassembled) represents the sequence of possible code mutations. This means that the phase semantics of a program provides a precise description of the evolution of its code during execution. Indeed, phase semantics can be graphically represented as a set of traces of program representations, e.g., program control-flow graphs, such that two programs P ad P' are consecutive along the trace τ if during the execution, the program P can evolve to program P'. The phase semantics is a sound abstract interpretation of standard program trace semantics. The main advantage of the phase semantics is in modeling code mutations without isolating the metamorphic engine from the rest of the viral code. The phase semantics provides here the basis in order to let standard program analysis methods and algorithms to extract invariant properties of code mutations. Decidable approximations of phases allow to extract an approximate semantics of the metamorphic engine, without knowing a priori any features of the metamorphic engine itself, providing the adequate knowledge in order to make behavioral analysis mutation insensitive. The information extracted by approximating the phase semantics is indeed precisely the information which is necessary in behavioral analysis for designing the appropriate abstractions making the analysis mutation insensitive. At the same time, the information extracted from the phase semantics may provide a signature (the *metamorphic signature*) of the possible evolution of the code. Observe that in this setting abstract domains approximating semantics objects represent properties of the code shape in phases, namely the abstractions capture properties of the evolution of the code rather than of the evolution of program states (e.g., memory or stack), as usual in abstract interpretation. Indeed, the design of such abstract domains for the analysis of code properties (rather than semantic properties) where the code is the object of abstraction and the way it is generated is the object of abstract interpretation, represents a new and interesting research field. This is an aspect of a semantics based learning technique acting at the metamorphic engine level, which is unknown. Indeed, abstract phase semantics expresses both the set of possible code variants generated during execution and the mechanisms of generation of such variants. For example, in [9] we introduce the notion of regular metamorphism that approximates phase semantics of a metamorphic malware M with an automata on the language of abstract instructions Q whose recognized language represents all possible (regular) sequence of instructions in the program evolutions of M. In this case the language recognized by the automata Q represents the regular metamorphic signature for the metamorphic malware M, while the automata Q represents the mechanism of generation of the metamorphic variants and therefore it provides a model of the metamorphic engine of M. Other learning strategies can be used. Metamorphic engines can be modeled as grammars or term rewriting systems. In this case existing algorithms for learning grammars, inductive logic programming, and term rewriting systems from positive examples [21, 12, 13, 16] can be used for implementing more expressive abstarct phase semantics. In this case the idea is that positive examples can be derived from the possible code evolutions expressed by the program evolution graph, i.e., the phase semantics, of the metamorphic program, while the metamorphic transformations are modeled as productions, or rewriting rules.

References

1. D. Babic, D. Reynaud, D. Song *Malware Analysis with Tree Automata Inference.* Proc. CAV 2011, LNCS 6806, pp. 116-131, 2011
2. P. Beaucamps, I. Gnaedig and J. Y. Marion. *Behavior Abstraction in Malware Analysis.* RV'10, LNCS 6418, pp. 168-182, 2010.
3. H. Cai, Z. Shao and A. Vaynberg. *Certified self-modifying code.* ACM PLDI, pp. 66-77, 2007.
4. M. Christodorescu and S. Jha. *Static analysis of executables to detect malicious patterns.* USENIX Security Symp. USENIX Association, pp. 169-186. 2003.
5. M. Christodorescu and S. Jha. *Testing malware detectors.* ISSTA'04, pp. 34-44. 2004.
6. M. Christodorescu, S. Jha, S. A. Seshia, D. Song and R. E. Bryant. *Semantics-aware malware detection.* Proc. of the IEEE Security and Privacy. pp. 32-46. 2005
7. P. Cousot and R. Cousot. *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints.* ACM POPL, pp. 238-252. 1977.
8. P. Cousot and R. Cousot. *Systematic design of program analysis frameworks.* ACM POPL, pp. 269-282. 1979.
9. M. Dalla Preda, R. Giacobazzi, S. Debray, K. Coogan and G. Townsedn. *Modelling Metamorphism by Abstract Interpretation.* SAS, LNCS 6337, pp. 218-235. 2010.
10. M. Dalla Preda, M. Christodorescu, S. Jha and S. Debray. *A semantics-based approach to malware detection.* ACM POPL, pp. 377-388. 2007.
11. M. Dalla Preda and R. Giacobazzi. *Semantics-based Code Obfuscation by Abstract Interpretation.* J. of Computer Security, 17(6):855-908, 2009.
12. C. de la Higuera. *Grammatical Inference Learning Automata and Grammars.* Cambridge University Press, 2010.
13. R. Eyraud, C. de la Higuera and J. C. Janodet. *LARS: A Learning Algorithm for Rewriting Systems* Machine Learning, 66(1):7-31, 2007.
14. G. Jacob, E. Filiol and H. Debar. *Formalization of Viruses and Malware Through Process Algebras*, ARES'10, pp. 597-602, IEEE Computer Society, 2010.
15. J. Kinder, S. Katzenbeisser, C. Schallart and H. Veith. *Detecting malicious code by model checking.* Proc. of the 2nd DIMVA, LNCS 3548, pp. 174 - 187. 2005.
16. M. R. K. Krishna Rao. *Some classes of term rewriting systems inferable from positive data.* Theoretical Computer Science, 397(1-3):129–149, 2008
17. A. Lakhotia, D. R. Boccardo, A. Singh and A. Manacero. *Context-sensitive analysis of obfuscated x86 executables.* Proc. of ACM PEPM 2010, pp. 131-140. 2010.
18. R. W. Lo, K. N, Levitt and R. A. Olsson. MCF: *A malicious code filter.* Computers & Security 14:541-566. 1995.
19. M. O. Myreen. *Verified just-in-time compiler on x86.* Proc. of the 37th ACM POPL 2010, pp. 107-118, 2010.
20. F. Nielson, H. Nielson and C. Hankin, "Principles of Program Analysis", 2004.
21. G. Plotkin. *A note on inductive generalization.* Machine Intell., 5:153-163, 1970.
22. P. Singh and A. Lakhotia. *Static verification of worm and virus behaviour in binary executables using model checking.* Proc. of the 4th IEEE Information Assurance Workshop. IEEE Computer Society, Los Alamitos, CA, USA.
23. P. Ször. *The Art of Computer Virus Research and Defense.* Addison-Wesley Professional, 2005.