Theoretical Computer Science 577 (2015) 74–97

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Unveiling metamorphism by abstract interpretation of code properties

Mila Dalla Preda[a,*], Roberto Giacobazzi[a], Saumya Debray[b]

[a] University of Verona, Italy
[b] University of Arizona, USA

ARTICLE INFO

Article history:
Received 13 November 2012
Received in revised form 10 May 2014
Accepted 17 February 2015
Available online 23 February 2015
Communicated by V. Sassone

Keywords:
Abstract interpretation
Program semantics
Metamorphic malware detection
Self-modifying programs

ABSTRACT

Metamorphic code includes self-modifying semantics-preserving transformations to exploit code diversification. The impact of metamorphism is growing in security and code protection technologies, both for preventing malicious host attacks, e.g., in software diversification for IP and integrity protection, and in malicious software attacks, e.g., in metamorphic malware self-modifying their own code in order to foil detection systems based on signature matching. In this paper we consider the problem of automatically extracting metamorphic signatures from metamorphic code. We introduce a semantics for self-modifying code, later called *phase semantics*, and prove its correctness by showing that it is an abstract interpretation of the standard trace semantics. Phase semantics precisely models the metamorphic code behavior by providing a set of traces of programs which correspond to the possible evolutions of the metamorphic code during execution. We show that metamorphic signatures can be automatically extracted by abstract interpretation of the phase semantics. In particular, we introduce the notion of regular metamorphism, where the invariants of the phase semantics can be modeled as finite state automata representing the code structure of all possible metamorphic change of a metamorphic code, and we provide a static signature extraction algorithm for metamorphic code where metamorphic signatures are approximated in regular metamorphism.

1. Introduction

Detecting and neutralizing computer malware, such as worms, viruses, trojans, and spyware is a major challenge in modern computer security, involving both sophisticated intrusion detection strategies and advanced code manipulation tools and methods. Traditional misuse malware detectors (also known as *signature-based detectors*) are typically syntactic in nature: they use pattern matching to compare the byte sequence comprising the body of the malware against a *signature database* [34]. Malware writers have responded by using a variety of techniques in order to avoid detection: Encryption, oligomorphism with mutational decryption patterns, and polymorphism with different encryption methods for generating an endless sequence of decryption patterns are typical strategies for achieving malware diversification. Metamorphism emerged in the last decade as an effective alternative strategy to foil misuse malware detectors. Metamorphic malware apply semantics-preserving transformations to modify its own code so that one instance of the malware bears very little resemblance to another instance, in a kind of *body-polymorphism* [33], even though semantically their functionality is the

* Corresponding author.
E-mail addresses: mila.dallapreda@univr.it (M. Dalla Preda), roberto.giacobazzi@univr.it (R. Giacobazzi), debray@cs.arizona.edu (S. Debray).

http://dx.doi.org/10.1016/j.tcs.2015.02.024
0304-3975/© 2015 Elsevier B.V. All rights reserved.

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 avoid detection. Some of the basic metamorphic transformations commonly used by malware are semantic-nop/junk insertion, code permutation, register swap and substitution of equivalent sequences of instructions [7,33]. It is worth noting that most of these transformations can be seen as special cases of code substitution [23]. 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 [10]. Existing malware detectors therefore fall back on a variety of heuristic techniques, but these may be prone to false positives (where innocuous files are mistakenly identified as malware) or false negatives (where malware escape detection) at worst. The reason for this vulnerability to metamorphism lies upon the purely syntactic nature of most existing and commercial detectors. The key for identifying metamorphic malware lies, instead, in a deeper understanding of their semantics. Preliminary works in this direction by Dalla Preda et al. [16], Christodorescu et al. [11], and Kinder et al. [27] confirm the potential benefits of a semantics-based approach to malware detection. Still a major drawback of existing semantics-based methods relies upon their need of an *a priori* knowledge of the obfuscations used to implement the metamorphic engine. Because of this, it is always possible for any expert malware writer to develop alternative metamorphic strategies, even by simple modification of existing ones, able to foil any given detection scheme. Indeed, handling metamorphism represents one of the main challenges in modern malware analysis and detection [18].

Contributions We use the term *metamorphic signature* to refer to an abstract program representation that ideally captures all the possible code variants that might be generated during the execution of a metamorphic program. A metamorphic signature is therefore any (possibly decidable) approximation of the properties of code evolution. We propose a different approach to metamorphic malware detection based on the idea that *extracting metamorphic signatures is approximating malware semantics*. Program semantics concerns here the way code changes, i.e., the effect of instructions that modify other instructions. We face the problem of determining how code mutates, by 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 propose 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. Traditional static analysis techniques are not adequate for this purpose, as they typically assume that programs do not change during execution. We therefore define a more general semantics-based behavioral model, called *phase semantics*, that can cope with changes to the program code at run time. 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. The sequence of phases (once disassembled) represents the sequence of possible code mutations, while the sequence of states within a given phase represents the behavior of a particular code variant.

Phase semantics precisely expresses all the possible phases, namely code variants, that can be generated during the execution of a metamorphic code. Phase semantics can then be used as a metamorphic signature for checking whether a program is a metamorphic variant of another one. Indeed, thanks to the precision of phase semantics, we have that a program Q is a metamorphic variant of a program P if and only if Q appears in the phase semantics of P. Unfortunately, due to the possible infinite sequences of code variants that can be present in the phase semantics of P, the above test for metamorphism is undecidable in general. Thus, in order to gain decidability, we need to loose precision and do so by using the well established theory of abstract interpretation [12,13]. Indeed, abstract interpretation is used here to extract the invariant properties of phases, which are properties of the generated program variants. Abstract domains represent here properties of the code shape in phases. We use the domain of finite state automata (FSA) for approximating phases and provide a static semantics of traces of FSA as an abstraction of the phase semantics. We introduce the notion of *regular metamorphism* as a further approximation obtained by abstracting sequences of FSA into a single FSA. This abstraction provides an upper regular language-based approximation of *any* metamorphic behavior of a program and it leads to a decidable test for metamorphism. This is particularly suitable to extract metamorphic signatures for engines implemented themselves as FSA of basic code transformations, which correspond to the way most classical metamorphic generators are implemented [23,31,35]. Our approach is general and language independent, providing an adequate theoretical foundation for the systematic design of algorithms and methods devoted to the extraction of approximate metamorphic signatures from any metamorphic code P. The main advantage of the phase semantics here is in modeling code mutations without isolating the metamorphic engine from the rest of the viral code. The approximation of the phase semantics by abstract interpretation can make decidable whether a given binary matches a metamorphic signature, without knowing any features of the metamorphic engine itself.

Structure of the paper In Section 3 we describe the behavior of a metamorphic program as a graph, later called *program evolution graph*, where each vertex is a standard static representation of programs (e.g., a control flow graph) and whose edges represent possible run-time changes to the code. We then define the phase semantics of a program as the set of all possible paths in the program evolution graph and we prove its correctness by showing that it is a sound abstract interpretation of standard trace semantics. Thus, phase semantics provides a precise description of the history of run-time code modifications, namely the sequences of "code snapshots" that can be generated during execution. Then, in Section 4, we introduce a general method for extracting metamorphic signatures as abstract interpretation of phase semantics. The result