Architecture of a morphological malware detector

Guillaume Bonfante · Matthieu Kaczmarek · Jean-Yves Marion

Received: 20 January 2008 / Revised: 10 July 2008 / Accepted: 17 July 2008 / Published online: 27 September 2008
© Springer-Verlag France 2008

Abstract Most of malware detectors are based on syntactic signatures that identify known malicious programs. Up to now this architecture has been sufficiently efficient to overcome most of malware attacks. Nevertheless, the complexity of malicious codes still increase. As a result the time required to reverse engineer malicious programs and to forge new signatures is increasingly longer. This study proposes an efficient construction of a morphological malware detector, that is a detector which associates syntactic and semantic analysis. It aims at facilitating the task of malware analysts providing some abstraction on the signature representation which is based on control flow graphs. We build an efficient signature matching engine over tree automata techniques. Moreover we describe a generic graph rewriting engine in order to deal with classic mutations techniques. Finally, we provide a preliminary evaluation of the strategy detection carrying out experiments on a malware collection.

1 Introduction

The identification of malicious behavior is a difficult task. Until now, no technologies have been able to automatically prevent the spread of malware. Several approaches have been considered but neither syntactic analysis nor behavioral considerations were really effective. Presently, human analysis of malware seems to be the best strategy, malware detectors based on string signatures remains the most reliable solution. From this point of view, we have tried to easier the task which consists in finding a good signature within a malicious program. Our technique has been inspired from the article [6] where control flow graphs (CFG) are used to detect the different instances of the computer virus MetaPHOR.

Generally speaking, detection strategies based on string signatures uses a database of regular expressions and a string matching engine to scan files and to detect infected ones. Each regular expression of the database is designed to identify a known malicious program. There are at least three difficulties tied to this approach. First, the identification of a malware signature requires a human expert and the time to forge a reliable signature is long compared to the time related to a malware attack. Second, string signature approach can be easily bypassed by obfuscation methods. Among recent works treating this subject, we propose to see for example [4, 7, 14]. Third, as the quantity of malware increases, the ratio of false positives becomes a crucial issue. And removing old malware signatures would open doors for outbreaks of re-engineered malware.

Thus, a current trend in the community is to design the next generation of malware detectors over semantic aspects. [9, 11, 20]. However, most of semantic properties are difficult to decide and even heuristics can be very complex as it is illustrated in the field of computer safety. For those reasons, in [5] we propose to construct a morphological analysis in order to detect malware. The idea is to recognize the shape of a malicious program. Unlike string signature detection, we are not only considering a program as a flat text, but rather as a semantic object, adding a dimension to the analysis. Our approach tries to combine several features: (a) to associate syntactic and semantic analysis, (b) to be efficient and (c) to be as automatic as possible.

Our morphological detector uses a set of CFG which plays the role of a malware signature database. Next, the detection consists in scanning files in order to recognize the shape of a malware. This design is closed to a string signature
based detector and so we think that both approaches may be combined in a near future. Moreover, it is important to notice that this framework makes the signature extraction easier. Indeed, either the extraction is fully automatic when the malware CFG is relevant, or the task of signature makers is facilitated since they can work on an abstract representation of malicious programs.

This detection strategy is close to the ones presented in [9, 6] but we put our strengths to optimize the efficiency of algorithms. For that sake, we use tree automata, a generalization to trees of finite state automata over strings [10]. Intuitively, we transform CFG into trees with pointers in order to represent back edges and cross edges. Then, the collection of malware signatures is a finite set of trees and so a regular tree language. Thanks to the construction of Myhill-Nerode, the minimal automaton gives us a compact and efficient database. Notice that the construction of the database is iterative and it is easy to add the CFG of a newly discovered malicious program.

Another issue of malware detection is the soundness with respect to classic mutation techniques. Here, we detect isomorphic CFG and so several common obfuscations are canonically removed. Moreover, we add a rewriting engine which normalizes CFG in order to have a robust representation of the control flow with respect to mutations. Related works are [6, 8, 20] where data flow of programs is also considered.

The design of the complete chain of process is summarized by Fig. 1.

We also provide large scale experiments, with a collection of 10,156 malicious programs and 2,653 sane programs. Those results are promising, with a completely automatic method for the signature extraction we have obtained a false positive ratio of 0.1%.

This study is organized as follows. First we expose the principles of CFG extraction and normalization. Then, we present a matching engine for CFG that is based on tree automata. Finally we carry out some experiments to validate our method.

2 CFG in x86 languages

Road-map. Since we focus on practical aspects we choose to work on a concrete assembly language. This language is close to the x86 assembly language. We detail how to extract CFG from programs, we underline the difficulties that can be encountered and we outline how they can be overcome with classic methods. Finally, we study the problem of CFG mutations. We propose to normalize the extracted CFG according to rewriting rules in order to remove common mutations.

An x86 assembly language. We present the grammar of the studied programming language. The computation domain is the integers and we use a subset of the commands of the x86 assembly language. The important feature is that we consider the same flow instructions as in x86 architectures, as a result the method that we develop can be directly applied to concrete programs.

Addresses
Offsets
Registers
Expressions
Flow instructions
Sequential instructions
Programs

Next, a program is a sequence of instructions \( p = i_0 ; \ldots ; i_{n-1} \). The address of the instruction \( i_k \) is \( k \). In order to ease the reading and without loss of generality, we suppose that \( i_0 \) is the first instruction to be executed, the address 0 is the so called entry point of the program.

We observe that the control flow of programs is driven by only four kinds of flow instructions. Given an instruction \( i_k \in I^f \), the possible transfers of control are the following.

- If \( i_k \) is an unconditional jump \( \text{jmp} ~ e \). The control is transferred to the address given by the value of the expression \( e \).
- If \( i_k \) is a conditional jump \( \text{jcc} ~ x \). If its associated condition is true, the control is transferred to the address \( k + x \). Otherwise, the control is transferred to the address \( k + 1 \).

Fig. 1 Design of the control flow detector