A Model for Self-Modifying Code*

Bertrand Anckaert, Matias Madou, and Koen De Bosschere

Ghent University, Electronics and Information Systems Department
Sint-Pietersnieuwstraat 41 9000 Ghent, Belgium
{banckaer,mmadou,kdb}@elis.UGent.be
http://www.elis.UGent.be/paris

Abstract. Self-modifying code is notoriously hard to understand and therefore very well suited to hide program internals. In this paper we introduce a program representation for this type of code: the state-enhanced control flow graph. It is shown how this program representation can be constructed, how it can be linearized into a binary program, and how it can be used to generate, analyze and transform self-modifying code.

1 Introduction

Self-modifying code has a long history of hiding the internals of a program. It was used to hide copy protection instructions in 1980s MS DOS based games. The floppy disk drive access instruction `'int 0x13'` would not appear in the executable program's image but it would be written into the executable's memory image after the program started executing[1]. A number of publications in the academic literature indicate a renewed interest in the application of self-modifying code to prevent undesired reverse engineering [1,10,14].

While hiding the internals of a program can be used to protect the intellectual property contained within or protected by software, it can be applied for less righteous causes as well. Viruses, for example, try to hide their malicious intent through the use of self-modifying code [12].

Self-modifying code is very well suited for these applications as it is generally assumed to be one of the main problems in reverse engineering [3]. Because self-modifying code is so hard to understand, maintain and debug, it is rarely used nowadays. As a result, many analyses and tools make the assumption that code is not self-modifying, i.e., constant. Note that we distinguish self-modifying code from run-time generated code as used in, e.g., a Java Virtual Machine.

This is unfortunate as, in theory, there is nothing unusual about self-modifying code. After all, in the omnipresent model of the stored-program computer, which was anticipated as early as 1937 by Konrad Zuse, instructions and data are held in a single storage structure [22]. Because of this, code can be treated as data and can thus be read and written by the code itself.

* The authors would like to thank the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and the Fund for Scientific Research Flanders (FWO) for their financial support. This research is also partially supported by Ghent University and by the HiPEAC network.

[1] http://en.wikipedia.org/wiki/Self-modifying_code, May 5th 2006.

J. Camenisch et al. (Eds.): IH 2006, LNCS 4437, pp. 232–248, 2007.
© Springer-Verlag Berlin Heidelberg 2007

If we want tools and analyses to work conservatively and accurately on self-modifying code, it is important to have a representation which allows one to easily reason about and transform that type of code. For traditional code, which neither reads nor writes itself, the control flow graph is such a representation. Its main benefit is that it represents a superset of all executions. As such, it allows analyses to reason about every possible run-time behavior of the program. Furthermore, it is well understood how a control flow graph can be constructed, how it can be transformed and how it can be linearized into an executable program. Until now, there was no analogous representation for self-modifying code. Existing approaches are often ad-hoc and usually resort to overly conservative assumptions: a region of self-modifying code is considered to be a black box about which little is known and to which no further changes can be made.

In this paper, we will discuss why the basic concept of the control flow graph is inadequate to deal with self-modifying code and introduce a number of extensions which can overcome this limitation. These extensions are: (i) a datastructure keeps track of the possible states of the program, (ii) an edge can be conditional on the state of the target memory locations, and (iii) an instruction uses the memory locations in which it resides.

We refer to a control flow graph augmented with these extensions as a state-enhanced control flow graph. These extensions ensure that we no longer have to artificially assume that code is constant. In fact, existing data analyses can now readily be applied on code, as desired in the model of the stored-program computer. Furthermore, we will discuss how the state-enhanced control flow graph allows for the transformation of self-modifying code and how it can be linearized into an executable program.

The remainder of this paper is structured as follows: Section 2 introduces the running example. Next, the extensions to the traditional control flow graph are introduced in Section 3. Section 4 provides algorithms to construct a state-enhanced control flow graph from a binary program and vice versa. Example analyses on and transformations of this program representation are the topic of Section 5. An experimental evaluation is given in Section 6. Related work is the topic of Section 7 and conclusions are drawn in Section 8.

2 The Running Example

For our example, we introduce a simple and limited instruction set which is loosely based on the 80x86. For the sake of brevity, the addresses and immediates are assumed to be 1 byte. It is summarized below:

Assembly	Binary	Semantics
`movb` *value to*	`0xc6` *value to*	set byte at address *to* to value *value*
`inc` *reg*	`0x40` *reg*	increment register *reg*
`dec` *reg*	`0x48` *reg*	decrement register *reg*
`push` *reg*	`0xff` *reg*	push register *reg* on the stack
`jmp` *to*	`0x0c` *to*	jump to absolute address *to*