The State-Enhanced Control Flow Graph

Bertrand Anckaert

Promoter(s): Koen De Bosschere

Abstract— **In the omnipresent model of the stored-program computer, both the instructions and data are held in a single storage structure. Therefore, instructions can be read and written as if they were data. In practice however, instructions rarely change during the execution of the program. As a result, it is often assumed that the instructions are constant. Therefore, many tools and analyses fail in the presence of self-modifying code. In this paper, we present an extension to the control flow graph representation, which enables the analysis, optimization and generation of self-modifying code: the state-enhanced control flow graph.**

Keywords—**Self-Modifying Code, Viruses, Obfuscation, State-Enhanced Control Flow Graph**

I. Introduction

THE interest in self-modifying code has decreased over the past decades, as current machine architectures make self-modifying code unnecessary, while processor pipelining and caching schemes make it inefficient. There are however a couple of areas in which self-modifying code still has its merits.

Firstly, self-modifying code can be used to specialize an algorithm at run time. This proves to be a powerful approach in situations where the algorithm is too generic to develop an efficient static algorithm. This is, for example, a popular approach in the domain of real-time graphics [1].

Secondly, self-modifying code is notoriously hard to understand and manipulate. Self-modifying code can therefore be used to protect software against attacks [2]. A copyright protection mechanism could, for example, be protected by changing it into a self-modifying version. This way, an attacker will have a hard time figuring out what happens exactly and hence find it more difficult to crack the protection mechanism.

Finally, the same property of making code harder to understand can be used with malicious intent. Viruses, for example, use self-modifying code to hide their true nature [3].

For these reasons, a powerful model for self-modifying code is necessary to help in the analysis, optimization and generation of self-modifying code. In this paper, we present such a model and briefly illustrate its applications.

II. The State-Enhanced Control Flow Graph

A very useful aid in program analysis and manipulation is the control flow graph $G(V, E)$. In its most simple form, the vertices V represent the instructions of the program, and the edges E represent the possible transitions between the instructions. As such, it is a representation of a superset of all the possible executions of the program. Many analyses therefore rely on this representation of the program to reason about all possible behaviors of the program.

Bertrand Anckaert is with the Electronics and Information Systems Department, Ghent University (UGent), Belgium. This research is funded by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT). E-mail: Bertrand.Anckaert@UGent.be.

Unfortunately, one cannot readily construct a control flow graph for self-modifying code. Furthermore, a control flow graph cannot contain enough information to allow for the manipulation of self-modifying code. The representation we propose consists of the usual control flow graph, extended in a number of ways to overcome these limitations. These extensions are:

- an instructions consumes its own memory locations;
- the target of a control transfer depends upon the state of the target memory locations;
- an additional datastructure to keep track of the possible states of the code.

The additional datastructure, called memState, keeps track of the possible states of every byte in the code. The first state is the initial state, i.e., the state that is in the binary image of the program. As the usual control flow graph is still part of this representation, constant code can be seen as a special case of self-modifying code for which there is only one state per code byte.

III. An Example: a Virus

We will now illustrate the discussed concepts and the applicability of our model. To this end, we introduce a very simple virus. Consider the following code fragment [1].

byte	binary representation	machine instruction	
0:	68 61	push	$0x61
5:	89 e3	mov	%esp,%ebx
7:	b8 0a	mov	$0xa,%eax
c:	cd 80	int	$0x80

This code has the following effect: the first instruction puts the string "a" on the stack, the second puts the address of the string in register `%ebx`, the third instruction sets `%eax` to 10 and the last instruction calls the operating system. The operating system will look at the value of `%eax`; the value 10 indicates that the file refered to by `%ebx` needs to be removed. The net result being that file "a" will be removed.

Using this approach, a virus could delete files on a victim computer. However, this code fragment can easily be analyzed and might be spotted easily. Therefore, the virus could disguise its malicious intent through the use of self-modifying code, as illustrated in the following code fragment:

byte	binary representation	machine instruction	
0:	c6 0a cd	movb	$0xcd,0x9
3:	68 61	push	$0x61
5:	89 e3	mov	%esp,%ebx
7:	b8 0a	mov	$0xa,%eax
9:	0c 80	or	$0x80,%al

If an observer would disassemble this code linearly (as show in the rightmost column), he would obtain a seemingly innocent

[1] Note that we use a slightly modified version of the IA-32 instruction set for the purpose of illustration

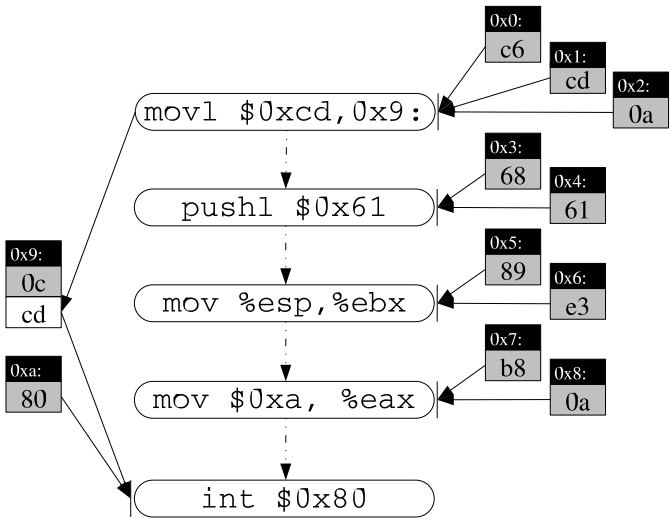

Fig. 1. The internal representation of the virus

code fragment which appears to consist of a number of harmless instructions.

If we look at it more closely, we can see that by the time the last instruction is executed, it has changed from an innocent `or`-instruction to a call to the operating system, since it was overwritten by the first instruction.

IV. Analyzing Self-Modifying Code

When faced with a piece of self-modifying code, traditional techniques such as linear disassembly fail. The reason for this failure is the assumption that code is constant. In order to see the true functionality of the code, we construct the state-enhanced control flow graph. The initialization of this algorithm consists of creating a memState for each byte in the program with one state: the value of the memory location in the binary representation of the program. Then, the following steps are repeated until no changes are made in two successive iterations:

- we start disassembly at the entry point of the program;
- disassembly continues according to recursive traversal;
- whenever a modifier is encountered, one or more states are added to the target memStates;
- when control flows to a certain location, all possible instructions at that location are added to the control flow graph.

In our running example of the disguised virus, this would result in the state-enhanced flow graph of Figure 1, with an extra successor for the `mov $0xa,%eax`-instruction: the `or 0x80,%al`-instruction. As a result, we have obtained a control flow graph which represents a superset of all possible executions.

Next, using constant propagation on the memStates, we are able to detect that memState `0x9:` will never be in state `0xc` at program point `mov $0xa,%eax` and therefore, that instruction cannot be followed by the `or 0x80,%al`-instruction. The latter instruction can thus be removed, resulting in the state-enhanced control flow graph of Figure 1.

V. Manipulating Self-Modifying Code

As in our model, an instruction consumes its own memory locations, the value of a memState can be safely overwritten if and only if it can be shown that the value will not be used anymore in the program. As a result, we can now use existing data analyses to check whether a transformation is allowed on the code.

In the state-enhanced control flow graph of Figure 1, we can easily see that the value `0c`, which is written to byte `0x9:` when the program is loaded, is never used by the program. Therefore, that value is considered dead and can be overwritten with any value, e.g., by the second state `cd` of that memState.

Subsequent analysis on the memState `0x9:` will show us that when the `mov $0xcd, 0x9`-instruction is executed, the memState is already set to the value `cd`. Therefore, this instruction is now considered an idempotent instruction and, since its memStates are not used by any other instruction, it can be removed safely. We have now successfully undone the virus from its disguise. If we would now linearize (see the next section) this representation, we would obtain the virus in its original form.

VI. Generating Self-Modifying Code

In order to generate a self-modifying program, we need to go from the internal representation, the state-enhanced control flow graph, to a linear list of bytes. This is done in a number of steps:
1. all succeeding memStates of every instruction in the flow graph are chained together;
2. for every fall-through edge $a \rightarrow b$, the last memState of instruction a is chained to the first memState of instruction b;
3. the resulting chains are chained together in an arbitrary order;
4. the value of the first state of each memState in the chain is selected.

One can easily verify that when this algorithm is applied to the state-enhanced control flow graph of Figure 1, the disguised virus is obtained.

VII. Concluding Remarks

While the running example has been kept simple deliberately for the purpose of illustration, it does indicate the potential to analyze, manipulate and generate self-modifying code through the use of the state-enhanced control flow graph.

References

[1] Rob Pike, Bart Locanthi, and John Reiser, "Hardware/software tradeoffs for bitmap graphics on the blit," *Software - Practice and Experience*, pp. 131–151, 1985.
[2] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere, "Software protection through dynamic code mutation," in *Proceedings of the 6th Workshop on Information Security Applications*, 2005, pp. 371–385.
[3] "The Leprosy-B virus," 1990, http://familycode.atspace.com/lep.txt.