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

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

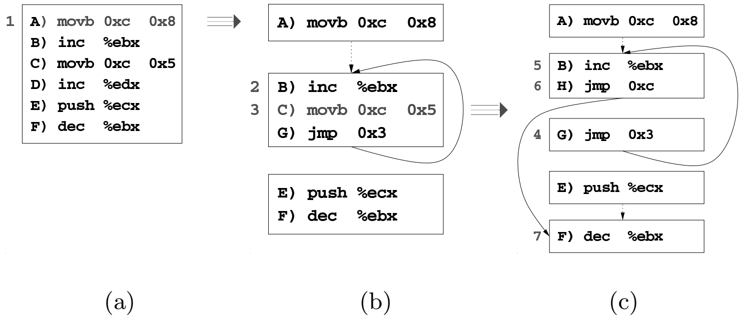
For our example, we introduce a simple and limited instruction set which is loosely based on the 80x86. For the sake of brevity, the addresses and immediates are assumed to be 1 byte. It is summarized below:

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*

As a running example, we have chosen to hide one of the simplest operations. The linear disassembly of the obfuscated version is as follows:

Address	Assembly	Binary
0x0	movb 0xc 0x8	c6 0c 08
0x3	inc %ebx	40 01
0x5	movb 0xc 0x5	c6 0c 05
0x8	inc %edx	40 03
0xa	push %ecx	ff 02
0xc	dec %ebx	48 01

If we would perform traditional CFG (Control Flow Graph) construction on this binary, we would obtain a single basic block as shown in Figure 1(a). If we step through the program however, we can observe that instruction `A` changes instruction `D` into instruction `G`, resulting in a new CFG as shown in part (b). Next instruction B is executed, followed by instruction `C` which changes itself into jump instruction `H` (c). Then, instruction `G` transfers control back to `B` after which `H` and `F` are executed. The only possible trace therefore is `A,B,C,G,B,H,F`. While not apparent at first sight, we can now see that these instructions could be replaced by a single instruction: `inc %ebx`.



(a) (b) (c)

Fig. 1. Traditional CFG construction before execution (a), after the first write instruction A (b), and after the second write instruction C (c)

3 The State-Enhanced Control Flow Graph (SE-CFG)

CFGs have since long been used to discover the hierarchical flow of control and for data-flow analysis to determine global information about the manipulation of data [16]. They have proved to be a very useful representation enabling the analysis and transformation of code. Given the vast amount of research that has gone into the development of analyses on and transformations of this program representation, we are eager to reuse the knowledge resulting from this research.

3.1 A Control Flow Graph for Self-Modifying Code

One of the reasons a CFG is so useful is that it represents a superset of all the possible executions that may occur at run time. As a result, many analyses rely on this representation to reason about every possible behavior of the program. Unfortunately, traditional CFG construction algorithms fail in the presence of self-modifying code. If they are applied on our running example at different moments in time, we obtain the three CFGs shown in Figure 1. However, none of these CFGs allows for both a conservative and accurate analysis of the code.

We can illustrate this by applying unreachable code elimination on these CFGs. This simple analysis removes every basic block that cannot be reached from the entry block. If it is applied on Figure 1(a), then no code will be considered to be unreachable. This is not accurate as, e.g., instruction `E` is unreachable. If we apply it on Figure 1(b), instructions `E` and `F` are considered to be unreachable, while Figure 1(c) would yield `G` and `E`. However, both `F` and `G` are reachable. Therefore in this case, the result is not conservative.

We can however still maintain the formal definition of a CFG: a CFG is a directed graph $G(V, E)$ which consists of a set of vertices V, basic blocks, and a set of edges E, which indicate possible flow of control between basic blocks. A basic block is defined to be a sequence of instructions for which every instruction in a certain position dominates all those in later positions, and no other instruction executes between two instructions in the sequence.

The concept of an edge remains unchanged as well: a directed edge is drawn from basic block a to basic block b if we conservatively assume that control can flow from a to b. The CFG for our running example is given in Figure 2.

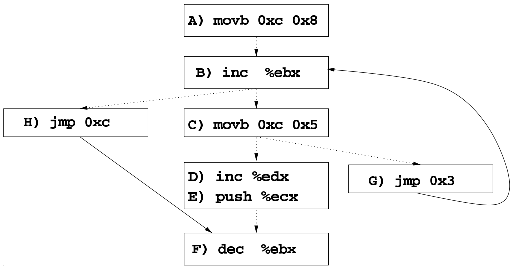


Fig. 2. The CFG of our running example (before optimization)

In essence, this CFG is a superposition of the different CFGs observed at different times. In the middle of Figure 2, we can easily detect the CFG of Figure 1(a). The CFG of Figure 1(b) can also be found: just mask away instruction `D` and `H`. Finally, the CFG of Figure 1(c) can be found by masking instruction `C` and `D`. We will postpone the discussion of the construction of this CFG given the binary representation of the program to Section 4. For now, note that, while this CFG does represent the one possible execution (`A,B,C,G,B,H,F`), it also represents additional executions that will never occur in practice. This will be optimized in Section 5.

3.2 Extension 1: Codebytes

The CFG in Figure 2 satisfies the basic property of a CFG: it represents a superset of all possible executions. As such it can readily be used to reason about a superset of all possible program executions. Unfortunately, this CFG does not yet have the same usability we have come to expect of a CFG.

One of the shortcomings is that it cannot easily be linearized into an executable program. There is no way to go from this CFG to the binary representation of Section 2, simply because it does not contain sufficient information.

For example, there are two fall-through paths out of block `B`. Note that we follow the convention that a dotted arrow represents a fall-through path, meaning that the two connected blocks need to be placed consecutively. Clearly, in a linear representation, only one of these successors can be placed after the increment instruction. Which one should we then choose?

To overcome this and other related problems, we will augment the CFG with a datastructure, called codebytes. This datastructure will allow us to reason about the different states of the program. Furthermore, it will indicate which instructions overlap and what the initial state of the program is.

In practice, there is one codebyte for every byte in the code segment. This codebyte represents the different states the byte can be in. By convention, the first of these states represents the initial state of that byte, i.e. the one that will end up in the binary representation of the program. For every instruction, there is a sequence of states representing its machine code. For our running example, this is illustrated in Figure 3. We can see that instruction A and C occupy three codebytes, while the others occupy two codebytes. A codebyte consists of one or more states. For example, codebyte $0x0$ has one state: $c6$ and codebyte $0x8$ has two states: 40 and $0c$. We can also see that instruction `H` and `C` overlap as they have common codebytes. As the first state of codebyte $0x5$ is that of instruction `C`, and the other states are identical, instruction `C` will be in the binary image of the program, while instruction `H` will not.

Codebytes are not only useful for the representation of the static code section, but also for the representation of code that could be generated in dynamically allocated memory. A region of memory can be dynamically allocated and filled with bytes representing a piece of code which will be executed afterwards. The difference between a codebyte representing a byte in the static code section and a codebyte representing a byte that will be dynamically produced at run time is that it has no initial state because the byte will not end up in the binary representation of the program.

3.3 Extension 2: Codebyte Conditional Edges

We have repeatedly stressed the importance of having a superset of all possible executions. Actually, we are looking for the exact set of all possible executions, not a superset. In practice, it is hard, if not impossible to find a finite representation of all possible executions and no others. The CFG is a compromise in the sense that it is capable of representing all possible executions, at the cost of representing executions that cannot occur in practice. Therefore, analyses on the CFG are conservative, but may be less accurate than optimal because they are safe for executions that can never occur.

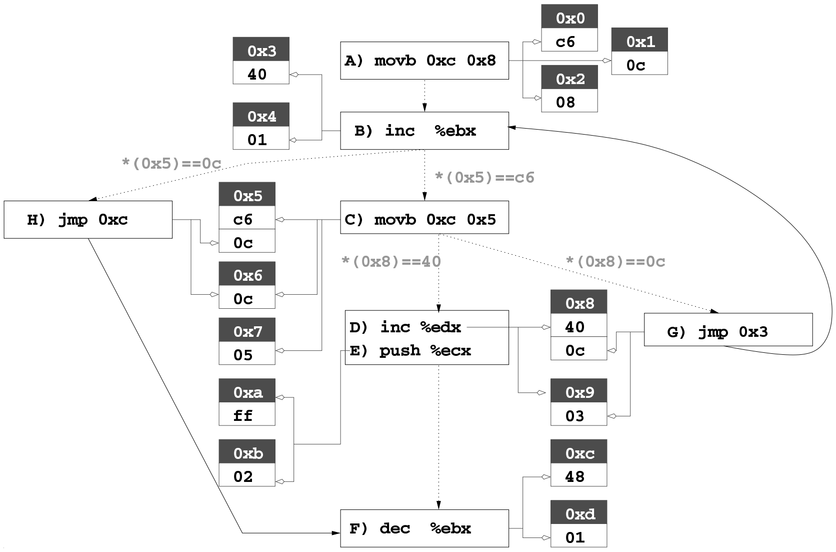


Fig. 3. The SE-CFG of our running example (before optimization)

A partial solution to this problem consists of transforming the analyses into path-sensitive variants. These analyses are an attempt to not take into account certain unexecutable paths. Clearly, for every block with multiple outgoing paths, only one will be taken at a given point in the execution. For constant code, the chosen path may depend upon a status flag (conditional jump), a value on the stack (return), the value of a register (indirect call or jump), ... However, once the target of the control transfer is known, it is also known which instruction will be executed next. For self-modifying code the target address alone does not determine the next instruction to be executed. The values of the target locations determine the instruction that will be executed as well. To take this into account, we introduce additional conditions on arrows. These conditions can be found on the arrows itself in Figure 3. As instruction `B` is not a control transfer instruction, control will flow to the instruction at the next address: $0x5$. For constant code, this would determine which instruction is executed next: there is at most one instruction at a given address. For self-modifying code, this is not necessarily the case. Depending on the state of the program, instruction `B` can be followed by instruction `C` (*(0x5)==c6) or instruction `H` (*(0x5)==0c).

3.4 Extension 3: Consumption of Codebyte Values

The third, and final extension is designed to model the fact that when an instruction is executed, the bytes representing that instruction are read by the CPU. Therefore, in our model, an instruction uses the codebytes it occupies. This will enable us to treat code as data in data-flow analyses. For example, if we want to apply liveness analysis on a codebyte, we have the traditional uses and definitions of that value: it is read or written by another instruction. For example, codebyte $0x8$ is defined by instruction `A`. On top of that, a codebyte is used when it is part of an instruction, *e.g.*, codebyte $0x8$ is used by instruction `D` and `G`. Note that this information can be deduced from the codebyte structure.

Wrap-up The SE-CFG still contains a CFG and therefore, existing analyses which operate on a CFG can be readily applied to an SE-CFG. Furthermore, code can be treated exactly the same way as data: the initial values of the codebytes are written when the program is loaded, they can be read or written just as any other memory location and are also read when they are executed.

Note that in our model traditional code is just a special case of self-modifying code. The extensions can be omitted for traditional code as: (i) the code can easily be linearized since instructions do not overlap, (ii) the target locations of control transfers can only be in one state, and (iii) the result of data analyses on code are trivial as the code is constant.

Where possible, we will make the same simplifications. For example, we will only add constraints to arrows where necessary and limit them to the smallest number of states to discriminate between different successors.

4 Construction and Linearization of the SE-CFG

In this section, we discuss how an SE-CFG can be constructed from assembly code. Next, it is shown how the SE-CFG representation can be linearized.

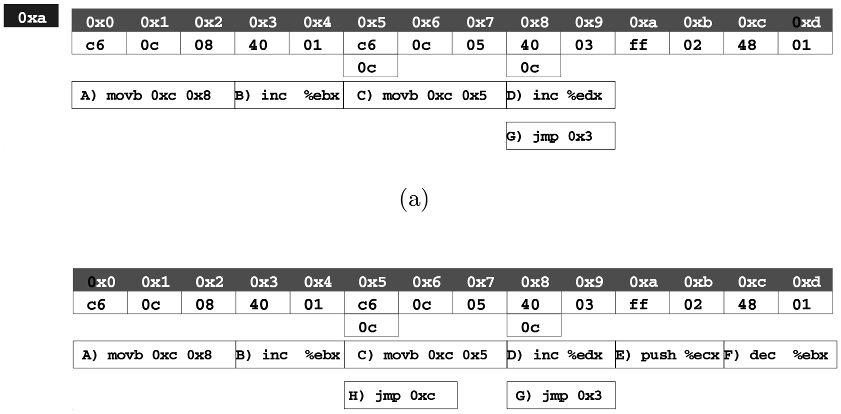
4.1 SE-CFG Construction

Static SE-CFG construction is only possible when we can deduce sufficient information about the code. If we cannot detect the targets of indirect control transfers, we need to assume that they can go to every byte of the program. If we cannot detect information about the write instructions, we need to assume that any instruction can be at any position in the program. This would result in overly conservative assumptions, hindering analyses and transformations.

When looking at applications of information hiding, it is likely that attempts will have been made to hide this information. It is nevertheless useful to devise such an algorithm, because there are applications of self-modifying code outside the domain of information hiding which do not actively try to hide such information. Furthermore, reverse engineers often omit the requirement of proved conservativeness and revert to approximate, practically sound information. Finally, it could be used to extend dynamically obtained information over code not covered in observed executions. For programs which have not deliberately been obfuscated, linear disassembly works well. As a result, the disassembly phase can be separated from the flowgraph construction phase. However, when the code is intermixed with data in an unpredictable way, and especially when attempts have been made to thwart linear disassembly [13], it may produce wrong results. Kruegel *et al.*[11] introduce a new method to overcome most of the problems introduced by code obfuscation but the method is not useful when a program contains self-modifying code. To partially solve this problem, disassembly can be combined with the control flow information. Such an approach is recursive traversal. The extended recursive traversal algorithm which deals with self-modifying code is:

```
00: proc main()
01:     for ( addr = code.startAddr; addr $\leq$ code.endAddr; addr++)
02:         codebyte[addr].add(byte at address addr);
03:     while (change)
04:         MarkAllAddressesAsUnvisited();
05:         Recursive(code.entryPoint);
06: proc Recursive(addr)
07:     if (IsMarkedAsVisited(addr)) return;
08:     MarkAsVisited(addr);
09:     for each (Ins) — Ins can start at codebyte[addr]
10:         DisassembleIns(Ins);
11:         for each (v,w) — Ins can write v at codebyte w
12:           codebyte[w].add(v);
13:         for each (target) — control can flow to target after Ins
14:           Recursive(target);
```

Disassembly starts at the only instruction that will certainly be executed as represented in the binary: the entry point (line 5). When multiple instructions can start at a codebyte, all possible instructions are disassembled (line 9, codebyte $0x8$ in Figure 4(a)). When an instruction modifies the code, state(s) are added to the target codebyte(s) (line 11-12). This is illustrated in Figure 4(a): state $0c$ is added to codebyte $0x8$. Next, all possible successors are recursively disassembled (line 13-14). In our example, the main loop (line 3) will be executed three times, as the second instruction at codebyte $0x5$ will be missed in the first run. It will however be added in the second run. In the third run, there will be no further changes. The overall result is shown in Figure 4(b).


0xa
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa 0xb 0xc 0xd
c6 0c 08 40 01 c6 0c 05 40 03 ff 02 48 01
0c
0c
A) movb 0xc 0x8
B) inc %ebx
C) movb 0xc 0x5
D) inc %edx
G) jmp 0x3

(a)

0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa 0xb 0xc 0xd
c6 0c 08 40 01 c6 0c 05 40 03 ff 02 48 01
0c
0c
A) movb 0xc 0x8
B) inc %ebx
C) movb 0xc 0x5
D) inc %edx
E) push %ecx
F) dec %ebx
H) jmp 0xc
G) jmp 0x3

(b)

Fig. 4. Recursive Traversal Disassembly of Self-Modifying Code

Once we have detected the instructions, the SE-CFG construction is straightforward: every instruction `I` is put into a separate basic block $basicblock_I$. If control can flow from instruction `I` to codebyte c, then for every instruction `J` that can start at c, we draw an edge $basicblock_I \rightarrow basicblock_J$. Finally, basic blocks are merged into larger basic blocks where possible. The thus obtained SE-CFG for our running example is given in Figure 3. Note that it still contains instructions that cannot be executed and edges that cannot be followed. It is discussed in Section 5 how these can be pruned.

4.2 SE-CFG Linearization

Traditional CFG linearization consists of concatenating all basic blocks that need to be placed consecutively in chains. The resulting chains can then be ordered arbitrarily, resulting in a list of instructions which can be assembled to obtain the desired program.

When dealing with self-modifying code, we cannot simply concatenate all basic blocks that need to be placed consecutively and write them out. One of the reasons is that this is impossible when dealing with multiple fall-through edges. Instead, we will create chains of codebytes. Two codebytes need to be concatenated if one of the following conditions holds: (i) c and d are successive codebytes belonging to an instruction, (ii) codebyte c is the last codebyte of instruction `I` and codebyte d is the first codebyte of instruction `J` and `I` and `J` are successive instructions in a basic block, and (iii) codebyte c is the last codebyte byte of the last instruction in basic block A and d is the first codebyte of the first instruction in basic block B and A and B need to be placed consecutively because of a fall-through path.

The resulting chains of codebytes can be concatenated in any order into a single chain. At this point, the final layout of the program has been determined, and all relocated values can be computed. Next, the initial states of the codebytes can be written out.

For example, in Figure 3, codebyte $0x0, 0x1$ and $0x2$ need to be concatenated because of condition (i), codebyte $0x9$ and $0xa$ because of condition (ii) and codebyte $0x4$ and $0x5$ because of condition (iii). When all conditions have been evaluated, we obtain a single chain. If we write out the first state of every codebyte in the resulting chain, we obtain the binary code listed in Section 2.

5 Analyses on and Transformations of the SE-CFG

In this section, we will demonstrate the usability of the SE-CFG representation by showing how it can be used for common analyses and transformations. We will illustrate how issues concerning self-modifying code can be mapped onto similar issues encountered with constant code in a number of situations.

Note that once the SE-CFG is constructed, the eventual layout of the code is irrelevant and will be determined by the serialization phase. Therefore, the addresses of codebytes are irrelevant in this phase. However, for the ease of reference, we will retain them in this paper. In practice, addresses are replaced by relocations.

5.1 Constant Propagation

The CFG of Figure 2 satisfies all requirements of a CFG: it is a superset of all possible executions. As this CFG is part of the SE-CFG in Figure 3, analyses which operate on a CFG can be reused without modifications. This includes constant propagation, liveness analysis, ...

Because of the extensions, it is furthermore possible to apply existing data analyses on the code as well. This can be useful when reasoning about self-modifying code. A common question that arises when dealing with self-modifying code is: “What are the possible states of the program at this program point?”. This question can be answered through traditional data analyses on the codebytes, e.g., constant propagation.

If we would perform constant propagation on codebyte $0x8$ on the SE-CFG of Figure 3, we can see that codebyte $0x8$ it is set to 40 when the program is loaded. Subsequently, it is set to $0c$ by instruction `A`. Continuing the analysis, we learn that at program point `C` it can only contain the value $0c$. Therefore, the edge from instruction `C` to instruction `D` is unrealizable, since the condition `*(0x8)==40` can never hold. The edge can therefore be removed.

5.2 Unreachable Code Elimination

Traditionally, unreachable code elimination operates by marking every basic block that is reachable from the entry node of the program. For self-modifying code, the approach is similar. For our running example, this would result in the elimination of basic blocks `D` and `E`. Note that the edge between `C` and `D` is assumed to have been removed by constant propagation.

Similarly, we can remove unreachable codebytes. A codebyte can be removed if it is not part of any reachable basic block and if it is not read by any instruction. This allows us to remove codebyte $0xa$ and $0xb$. While we have removed the `inc %edx`-instruction, its codebytes could not be removed, as they are connected through another instruction. Note that we now have a conservative and accurate unreachable code elimination.

5.3 Liveness Analysis

Another commonly asked question with self-modifying code is as follows: “Can I overwrite a piece of code?”. Again, this is completely identical to the question whether you can overwrite a piece of data. You can overwrite a piece of the program if you can guarantee that the value will not be read later on by the program before it is overwritten. In our model, for self-modifying code, a value is read when (i) it is read by an instruction (standard), (ii) the flow of control can be determined by this value (extension 2), and (iii) the CPU will interpret it as (part of) an instruction (extension 3).

We could, for example, perform liveness analysis on codebyte $0x8$. This shows us that the value 40, which is written when the program is loaded, is a dead value: it is never read before it is written by instruction A. As a result, it can be removed and we could write the second state $0c$ immediately when the program is loaded. In our representation, this means making it the first state of codebyte $0x8$.

Subsequently, an analysis could point out that instruction A has now become an idempotent instruction: it writes a value that was already there. As a result, this instruction can be removed. We have now obtained the SE-CFG of Figure 5.

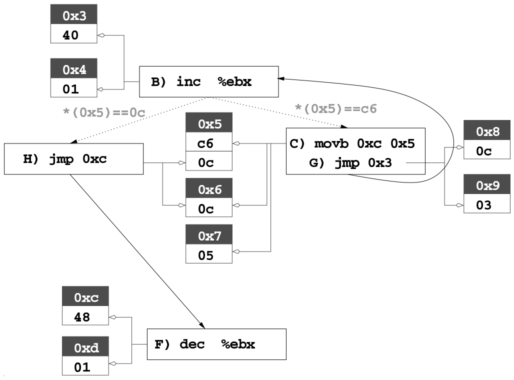


Fig. 5. The SE-CFG after partial optimization, before unrolling

5.4 Loop Unrolling

Subsequently, we could peel of one iteration of the loop to see if this would lead to additional optimizations. This results in the SE-CFG in Figure 6. Note that we had to double write operation C, as we should now write to both the original and the copy of the codebyte in order to be semantically equivalent.

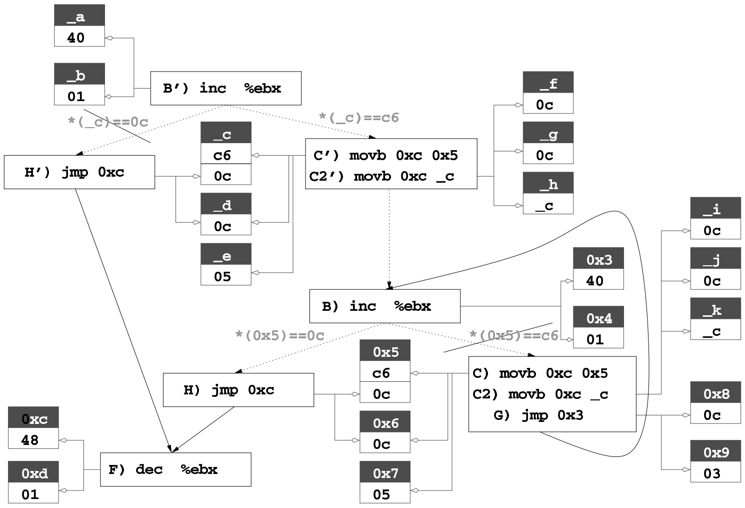


Fig. 6. The SE-CFG after unrolling

5.5 Finishing up

Similar to Section 5.1, we can now find out that the paths `B'`→ `H'` and `B` → `C` are unrealizable. As a result, we no longer have a loop. Instruction `C, C2, G` and `H'` are unreachable. Applying the same optimization as in Section 5.3, we can remove the first state of codebyte `0x5` and instruction `C'`. The value written by `C2'` is never used and thus `C2'` can be removed. Through jump forwarding, we can remove instruction `H`. Finally, given that the decrement instruction performs exactly the opposite of the increment instruction, we now see that the code can be replaced by a single instruction: `inc %ebx`.

6 Evaluation

To evaluate the introduced concepts, we implemented a form of factorization through the use of self-modifying code. The goal however is not to shrink the binary, but to hide program internals. Therefore, we will also perform factorization if the cost is higher than the gain.

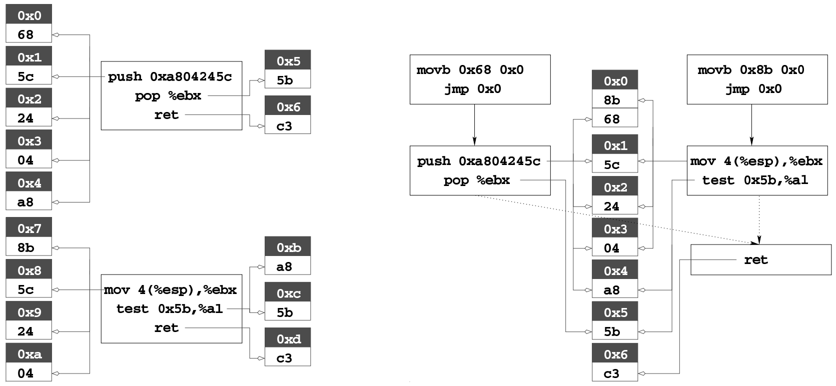
In the first phase, we split the code up in what we call *code snippets*. These code snippets are constructed as follows: if a basic block is not followed by a fall-through edge, the basic block itself makes up a code snippet. If basic block a was followed by a fall-through edge e to basic block b, a new basic block c is created with a single instruction: a jump to b. The target of e is then set to c. The combination of a and c is then called a code snippet.

A code snippet is thus a small piece of code that can be placed independently of the other code. It consists of at most two consecutive basic blocks. If there is a second basic block, this second basic block consists of a single jump instruction. The advantage of code snippets is that they can be transformed and placed independently. The downside is that their construction introduces a large number of jump instructions. This overhead is partially eliminated by performing jump forwarding and basic block merging at the end of the transformation.

Next, we perform what we call code snippet coalescing. Wherever possible with at most one modifier we let two code snippets overlap. Both code snippets are then replaced by at most one modifier and a jump instruction to the coalesced code snippet. On the 80x86, this means that code snippets are merged if they differ in at most 4 consecutive bytes.

As an example, consider the two code snippets in Figure 7(a). While these two code fragments seem to have little in common, their binary representation differs in only one byte. Therefore, they are eligible for code snippet coalescing. The result is shown in Figure 7(b). The codebytes of the modifier and jump instructions are not shown to save space. In this example, subsequent branch forwarding will eliminate one of the jumps. (Note that this example uses the actual 80x86 instruction set.)

Intuitively, this makes the binary harder to understand for a number of reasons. Firstly, as overlapping code snippets are used within multiple contexts, the number of interpretations of that code snippet increases. It also becomes



(a) Snippets to coalesce

(b) Coalesced snippets

Fig. 7. Example of coalescing code snippets

more difficult to distinguish functions as there boundaries have been blurred. And most importantly, the common difficulties encountered for self-modifying code have been introduced: the code is not constant and therefore, the static binary does not contain all the instructions that will be executed. Furthermore, multiple instructions will be executed at the same address, so there is no longer a one to one mapping between addresses and instructions.

To further obfuscate the program, we have added an additional transformation, called code snippet splitting. Whenever possible, two different versions are created for code snippets that were not yet protected by the previous transformation. This is often possible because of the redundancy of machine code and especially the 80x86 instruction set. Using an opaque predicate of the type $P^?$ [5] one of both versions is chosen at run time. Next, we merge both versions using code snippet coalescing.

The measurements have been performed a Linux system on a 2.8GHz Pentium IV on 10 C programs of the SPECint 2000 benchmark suite. The benchmarks have been compiled with gcc 3.3.2 and statically linked against uclibc 0.27. The library code has been obfuscated as well. We strongly recommend obfuscating library code as well as it otherwise serves as reference points about which the attacker knows everything and he can then continue to fill in the missing pieces in between two library calls, which allows him to focus on much smaller pieces of code. Furthermore, in the case of data obfuscation, escaping values would need to be turned back into the correct format before every library call, which would severely limit the scope of these obfuscating transformations.

As can be seen in the first row of Table 1, the small granularity of the code snippets and the relatively large overhead of the modifiers (7 byte for a one-byte modifier and 10 byte for a four-byte modifier) can cause a considerable increase in the code size of the program. The impact on the execution speed can be even higher. When all basic blocks are eligible for transformation, the slowdown is unacceptable for most real life applications. Therefore, we excluded hot code (based upon profile information collected from the train input sets) from consideration. The resulting slowdown on the reference input sets is given in the second row of Table 1. The third row indicates the percentage of the total number of original code snippets that is protected by code snippet coalescing. The fourth row the percentage that is protected by code snippet splitting.

Benchmarks	bzip2	crafty	gap	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
code bloat (%)	114.51	100.95	111.8	123.17	121.38	151.63	106.63	100.19	142.01	102.2
slowdown (%)	27.47	128.82	71.47	15.71	0.5	116.37	300	36.38	274.42	21.16
coalescing (%)	22.98	18.14	26.18	22.66	21.79	22.02	27.67	19.6	22.07	21.55
splitting (%)	23.06	26.04	21.24	24.06	22.35	29.2	19.2	23.41	31.62	21.34

Table 1. Increase in code size and execution speed; percentage of coalesced code snippets and split-coalesced code snippets

We have attached a dynamic instrumentation framework [15] to the resulting programs. When no modifications were made to the program other than to keep the program under control and to keep the internal datastructures consistent with the code, we experienced a slowdown of a factor 150 to 200. The bulk of this slowdown is due to the monitoring of the write instructions. These results show that the cost of self-modifying code is fairly high and that it is best avoided in code which will be frequently executed. On the other hand, the slowdown experienced by an attacker, who, e.g., wants to modify the program on the fly, can be much higher.

The concepts described in this paper have been integrated into a link-time binary rewriter: Diablo. It can be downloaded from http://www.elis.ugent.be/diablo.

7 Related Work

Some of the work on self-modifying code is situated in the domain of viruses, and therefore, not well documented. Because pattern matching is a common technique to detect viruses, some viruses contain an encrypted static image of the virus and code to decrypt it at run time [12]. As different keys are used in different generations, they can have many different static forms. This is a specific type of self-modifying code, which we call self-decryption.

Viruses which do not change during the execution of the virus, but which change in every new generation [20, 19] are often referred to as self-modifying as well. We do not consider them to be self-modifying, however. Instead, we refer to this technique as mutation. An appraoch that could be used to detect viruses which change in every generation is proposed by Chistodorescu and Jha[2].

Protecting a program from being inspected trough the use of self-modifying code is also possible. When the architecture requires explicit cache flushing, a debugger could be fooled if it flushes the cache too early: it will execute the new instruction while the real execution will execute the old instruction untill a cache flush is forced. Vice versa, when cache flushing is done automatically as blocks of code are executed in an instrumentator, anti-debugging could be modifying the next instruction. The instrumentator will execute the old instruction while the real execution will execute the new instruction.

A technique similar to self-decryption can be used for program compaction. In this approach, described by Debray and Evans [6], infrequently executed portions of the code are compressed in the static image of the program and decompressed at run time when needed. This technique could be called self-extraction.

One of the earliest publications in academic literature on tamper-resistant software in general and self-modifying code in particular is due to Aucsmith [1]. The core of the discussed approach consists of integrity verification kernels, which are self-modifying, self-decrypting and installation unique and which verify each other and critical operations of the program.

Kanzaki *et al.* [10] scramble instructions in the static image of the program and restore them at run time. This restoration process is done through modifier instructions that are put along every possible execution path leading to the scrambled instructions. Once the restored instructions are executed, they are scrambled again. As only one instruction can be executed at a given memory location, there is still a one to one mapping between instructions and addresses.

Madou *et al.* [14] introduce a coarse-grained form of self-modifying code. Functions which are not frequently in the same working set are assigned to the same position in the binary. At this position, a template function is placed which contains the common pieces of both functions. Descriptions of the changes that need to be made to the template to obtain the original functions are stored in the binary image as well. At run time, a code editing engine uses these descriptions to create the desired function. As a result the one to one mapping between instructions and addresses is lost.

Dux *et al.* [8] discuss a time-based visualization of self-modifying code, the concept of which can be compared to that of Figure 1. While this visualization can clearly facilitate the understanding of self-modifying code, it does not represent a superset of all possible executions at any time. To the best of our knowledge, existing approaches use specific algorithms and do not use a generally usable representation as the one discussed in this paper.

Other research involves the use of self-modifying code for optimization [18] and the treatment of self-modifying code in dynamic binary translators like Crusoe [7] and Daisy [9].

There is a considerable body of work on code obfuscation in particular and code protection in general that focuses on techniques other than self-modifying code. We refer to other papers for an overview [4, 17, 21].

8 Conclusion

In this paper we have introduced a novel program representation for self-modifying code. We have shown how it enables the generation, accurate and conservative analysis, and transformation of self-modifying code. The evaluation illustrates that self-modifying code can significantly increase the effort an attacker needs to make, but that it should be avoided in frequently executed code.

References

1. D. Aucsmith. Tamper resistant software: an implementation. *Information Hiding, LNCS*, 1174:317–333, 1996.
2. Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186. USENIX Association, 2003.
3. C. Cifuentes and K. Gough. Decompilation of binary programs. *Software - Practice & Experience*, 25(7):811–829, 1995.
4. C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.
5. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, 1998.
6. S. Debray and W. Evans. Profile-guided code compression. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*, 2002.
7. J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges, 2003.
8. B. Dux, A. Iyer, S. Debray, D. Forrester, and S. Kobourov. Visualizing the behavior of dynamically modifiable code. In *Proc. of the 13th International Workshop on Program Comprehension*, pages 337–340, 2005.
9. K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.
10. Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *Proc. of the 27th Annual International Computer Software and Applications Conference*, pages 170–181, 2003.
11. C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. of the 13the USENIX Security Symposium*, 2004.
12. The Leprosy-B virus, 1990. http://familycode.atspace.com/lep.txt.
13. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS)*, pages 290–299, 2003.
14. M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. *Information Security Applications, LNCS*, 3786:194–206, 2005.
15. J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, 2002.
16. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publischers, Inc., 1997.
17. G. Naumovich and N. Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.
18. R. Pike, B. Locanthi, and J. Reiser. Hardware/software tradeoffs for bitmap graphics on the blit. *Software - Practice & Experience*, 15(2):131–151, 1985.
19. P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.
20. P. Szor and P. Ferrie. Hunting for metamorphic, 2001.
21. P.C. van Oorschot. Revisiting software protection. *Information Security, LNCS*, 2851:1–13, 2003.
22. Konrad Zuse. Einführung in die allgemeine dyadik, 1937.