Pure Infinitely Self-Modifying Code is Realizable and Turing-complete

Gregory Morse

Abstract—Although self-modifying code has been shied away from due to its complexity and discouragement due to safety issues, it nevertheless provides for a very unique obfuscation method and a different perspective on the relationship between data and code. The generality of the von Neumann architecture is hardly realized by today’s processor models. A code-only model is shown where every instruction merely modifies other instructions yet achieves the ability to compute and Turing machine operation is easily possible.

Keywords—x86, x86-64, Assembly language, Self-Modifying code, Turing-completeness, Code obfuscation

I. INTRODUCTION

Although the mov instruction alone is enough to prove Turing-completeness, especially in the self-modifying code case which has been discussed in prior research, pure infinitely self-modifying code, where not a single operation is not in some way a modification to the code has not been studied or clearly defined.

Part of the challenge is that to construct such a model, it blurs the line between code and data. In fact everything in the model is code, while code itself simply represents data flowing through the system. To do this, a mapping of code to data and data to code is defined, as a practical matter which can be implemented with pure infinitely self-modifying code. Within self-modifying code, there are several categories of modifications which can be further useful:

- Modifying instructions
- Modifying arguments and operands
- Overlaying and relocating code

In fact there are 2 self-modifying code aspects based on this, one is pure self-modifying code which modifies its own code blueprint or code skeleton and details, while the other is merely reuse of a code region. Reuse of code regions such as via overlays, is a classic technique often due to limited memory, or in modern times due to limited executable space, or for encryption and decompression. But little is practically known of pure self-modifying code. Partly the difficulty of factoring code requires a new binary translation and transformation methods beyond the limitations of the control flow graph (CFG). Pure self-modifying code can be defined as any code which modifies itself while always maintaining validity of its code outside contiguous modification regions.

Infinitely self-modifying code, is code which always modifies code instructions in memory in every statement. It is a unique and novel type of code that could be used in subsequences of code for proof of concept and for efficiency or obfuscation. Infinitely self-modifying code clearly requires pure self-modifying code as such criterion is inherent in the definition, though the opposite does not need to hold.

II. MODEL

Instructions of interest for constantly self-modifying code are only the subset which write to memory as part of their operation. Although in this paper, the x86 and x86-64 processor instruction sets are used for demonstration purposes, the instructions are very general and can be translated to other architectures relatively easily. Of course the expressivity of the instruction set is a determining factor for whether this is possible on a given architecture as the binary argument encoding of instructions will be modified carefully and precisely. This practical choice is supported by the potential obfuscation uses of such self-modifying code.

Modern operating systems implement the processor feature enabling data execution prevention (DEP) which makes rendering self-modifying code and calling it in a data segment, the heap or virtual memory impractical or limited as an approach however operating systems still support manual methods of flagging data as code such as the VirtualAlloc/VirtualProtect with PAGE_EXECUTE flag/VirtualFree pattern on Windows. Thus the static code segment area of an executable must contain the entire space needed in one way or another. If this is a significantly large enough area, overlay techniques can generalize it so that code can be modified in data areas and then swapped in via overlays.

Code segments are generally treated as read and execute only in modern compilers especially for 64-bit code, so care must be taken in linkers to use flags that mark it as read, execute and write if such self-modifying code will be used from the code segment. 64-bit code has almost no 64-bit data in the code except for offsets which can only be loaded into the accumulator register for practical restraints largely to make instructions fall within a 15 byte maximum limit. A far call instruction is the only other possible candidate. Yet this is sufficient to make a complete model also work on 64-bit addressed code.

Author is with Eötvös Loránd University, Hungary (e-mail: gregory.morse@live.com)
TABLE I

<table>
<thead>
<tr>
<th>Type</th>
<th>Instruction names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory movement</td>
<td>movsb, movsw, movsd</td>
</tr>
<tr>
<td>Memory writing</td>
<td>mov</td>
</tr>
<tr>
<td>Boolean and mathematical logic</td>
<td>add, sub, adc, sbb, xor and, or, not, neg, shl, shr</td>
</tr>
<tr>
<td>Transfer</td>
<td>call</td>
</tr>
</tbody>
</table>

TABLE II

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>single byte opcode size</td>
<td>OPSZ1</td>
<td>1</td>
</tr>
<tr>
<td>double byte instruction size</td>
<td>OPSZ2</td>
<td>2</td>
</tr>
<tr>
<td>Inst. with 2 32-bit args</td>
<td>OPDBLARGSZ</td>
<td>OPSZ2 + PTRSZ32 * 2</td>
</tr>
<tr>
<td>Offset to 32-bit arg in memory inst.</td>
<td>IMMOFFS</td>
<td>OPSZ2 + PTRSZ32</td>
</tr>
<tr>
<td>Offset to 32-bit arg in register inst.</td>
<td>REGIMMOFFS</td>
<td>OPSZ2 + 1</td>
</tr>
</tbody>
</table>

Since code cannot be executed contained in registers, no registers are used beyond any temporary specific purpose for instructions which absolutely require them and they are immediately restored to their original values. Even where registers are used, code memory is written simultaneously.

Noteworthy is that relative to instruction pointer addressing modes are available in 64-bit mode of modern processors, though 32-bit mode would have to use a call/pop type of combination to retrieve the instruction pointer in code that is relocated.

At a minimum, the processor targeted for the maximal application of this technique would require the ability to modify memory contents with immediate values. If full pointer address and immediate values can be encoding in a single instruction, then this provides for compactness and efficiency.

The nature of self-modifying code requires some definitions that have to do with the architecture specific encodings of the instructions and their offsets as well, attempting to largely abstract it through a series of definitions as shown in Tables 2 and 3.

TABLE III

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix length</td>
<td>PFXSZ</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Pointer size</td>
<td>PTRSZ</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Word size name</td>
<td>SZW</td>
<td>dword</td>
<td>qword</td>
</tr>
<tr>
<td>Define raw offset</td>
<td>MOFFS</td>
<td>dxd</td>
<td>dq</td>
</tr>
<tr>
<td>General purpose register</td>
<td>acmltr</td>
<td>eax</td>
<td>rax</td>
</tr>
<tr>
<td>Stack register</td>
<td>gsp</td>
<td>esp</td>
<td>rsp</td>
</tr>
<tr>
<td>Source index</td>
<td>gsi</td>
<td>esi</td>
<td>rsi</td>
</tr>
<tr>
<td>Destination index</td>
<td>gdi</td>
<td>edi</td>
<td>rdi</td>
</tr>
</tbody>
</table>

Anything suffixed with 32, takes the 32-bit form except where explicitly defined.

It has already been shown that only mov instructions is required for Turing-completeness, but only 1 of the 3 addressing modes from those necessary writes to memory and would be applicable to pure infinitely self-modifying code. Reading from memory into a register does not modify code, though it can be done through clever use of exchange instructions. In fact, even control flow can be done in a pure self-modifying manner as well as not having arbitrary memory to clobber for comparisons. It leads to a different model ultimately which is hereby presented.

For example, the can be represented as a data definition (in 32-bit code though generalizable as shown later to 64-bit), or a usage as demonstrated in Listing 1.

Listing 1. Emulating Turing-complete mov instructions with self-modifying code

```plaintext
mov Rdest, c
Rdest: mov SZW32 ptr [Rdest + IMMOFFS], c

mov Rdest, [Rsrc + offset]
Rsrc: mov SZW32 ptr [Rsrc + IMMOFFS], 0
      mov SZW32 ptr [Rsrc + OPDBLARGSZ + IMMOFFS], 0
      xchg acmltr32, [Rsrc + IMMOFFS + offset * (OPDBLARGSZ)]
      mov [Rdest + IMMOFFS], acmltr32
      xchg acmltr32, [Rsrc + IMMOFFS + offset * (OPDBLARGSZ)]
      Rdest: mov SZW32 ptr [Rdest + IMMOFFS], 0

mov Rdest, [Rdest + offset], Rsrc
Rsrc: mov SZW32 ptr [Rsrc + IMMOFFS], 0
      xchg acmltr32, [Rsrc + IMMOFFS]
      mov [Rdest + IMMOFFS + offset * (OPDBLARGSZ)]
      xchg acmltr32, [Rsrc + IMMOFFS]
      Rdest: mov SZW32 ptr [Rdest + IMMOFFS], 0
      mov SZW32 ptr [Rdest + OPDBLARGSZ + IMMOFFS], 0

However, despite the 3 being emulatable using these techniques, the actual data to code transformation, and flow of the application is quite different and a different usage which does not use pure mov instructions, but takes advantage of several other instructions becomes optimal. Listing 2 thus gives the basic defining tools for pure self-modifying code.

Listing 2. Pure self-modifying code definition, basic operations and assignment

MOFFSMOV(labelname, addr) - move full accumulator to memory

if (64) db 48h
db 0A3h ; mov [addr], acmltr
labelname: MOFFS OFFSET addr

DEFADDR(labelname, addr) - data placeholder instruction/effective no-operation

xchg acmltr, [labelname]
mov [labelname], acmltr
MOFFSMOV(labelname, addr)

DEFDATA32(labelname, data) - more efficient 32-bit placeholder

labelname: mov SZW32 ptr [labelname + IMMOFFS], data
DATAOP(op, destptr)
- unary arithmetic operation
  op SZW32 ptr [destptr]

DATAOP(op, destptr, data)
- assign data/binary arithmetic operation
  op SZW32 ptr [destptr], data

OP64(op, destptr, sz)
DATAOP(op, destptr)
if (64) DATAOP(op, destptr + sz)

OP64(op, destptr, sz, data)
DATAOP(op, destptr, data)
if (64) DATAOP(op, destptr + sz, data)

MOVEMODEOFFS(d, do, s, so)
- assign relative offset

32-bit
  mov S8Z32 ptr [d + d], OFSFI s + so

64-bit
  mov S8Z32 ptr [d + d + do], ($-d+do)-($-s+so)

Halt:
DOHALT()

Further operations are needed for the sake of Turing completeness so next comparison is considered. Listing 3 gives a basic comparison example making use of typical assembly language shortcuts which is further developed in the appendix.

Listing 3. Pure self-modifying code comparison instruction

COMPARE(val, dataptr, datamaskptr, onlygt, preserve) - handle >, ≠ cases
  sub [dataptr], val
  if !onlygt neg [dataptr]
  sbb [maskptr], 0
  if (preserve)
    if !onlygt neg [dataptr]
      add [dataptr], val
    fi
  fi

The next consideration is reuse of code namely through control flow constructs as minimally a basic loop construct is needed to emulate any Turing machine. Listing 4 continues listing patterns which give a solution to control flow via self-modifying code whereby novel usage of instruction data for stack storage is realized.

Listing 4. Pure self-modifying code jump instruction

DEFSTACK()

32-bit
  DEFDATA32(stptr, OFFSETstptr)
  mov [stptr + IMMOFFS], acmltr
  MOVSMACRO(size)
  mov [stptr + IMMOFFS], acmltr
  xchg acmltr, [stptr + IMMOFFS]

64-bit
  DEFDATA32(stptr, OFFSETstptr)
  mov [stptr + IMMOFFS], acmltr
  MOVSMACRO(size)
  mov [stptr + IMMOFFS], acmltr
  xchg acmltr, [stptr]

JUMP(addr)

32-bit
  xchg gsp, [stptr + IMMOFFS]
  call addr

64-bit
  xchg gsp, [stptr]
  call addr
  xchg acmltr, [stptr + IMMOFFS]
  mov [stptr + IMMOFFS], acmltr
  xchg acmltr, [stptr + IMMOFFS]

Listing 5 details data transfer of batches of instructions as its not only convenient but useful for certain types of scalable solutions and in this pattern speed can be gained perhaps even with extensions via the rep[z/nz] prefix.

Listing 5. Data transfer

TRANSFERDATA(source, sourceinit, dest, destininit, size)
  DEFDATA(source, sourceinit)
  xchg gsi, [sourceinit]
  DEFDATA(dest, destininit)
  xchg gdi, [destinit]
  MOVSMACRO(size)
  xchg gdi, [destinit]
  mov [destinit], size
  mov [sourceinit], size

Listing 6. Further examples presented to reinforce and strengthen the concept

NLZ(value): Binary Number of Leading Zeros

DOSIGNEDMUL(dataptr, val)
- pure self-modifying code

shl SZW32 ptr [dataptr], NLZ(val)
if ((val = (val >> (NLZ(val) + 1))) != 0)
  USEWORD32(op, source, s, so)
  xchg acmltr, [source]
  USEWORDOP64(op, do, d, data)
  shl S6Z32 ptr [dataptr], NLZ(val)
  while ((val = (val >> (NLZ(val) + 1))) != 0)
    USEWORD32(add [dataptr + mul], 0)
  fi

Finally Fig. 6 gives some ideas of other general operations like multiplication by a constant and transformation from the typical data storage input domain to the data stored as code input domain. In fact the mapping from a normal binary data domain into a code-only data domain is critical to meeting the definition of pure and infinite in the context previously defined.

Listing 6. Further examples presented to reinforce and strengthen the concept

NLZ(value): Binary Number of Leading Zeros

DOSIGNEDMUL(dataptr, val) - pure self-modifying code

shl S6Z32 ptr [dataptr], NLZ(val)
if ((val = (val >> (NLZ(val) + 1))) != 0)
  USEWORD32(op, source, s, so)
  xchg acmltr, [source]
  USEWORDOP64(op, do, d, data)
  shl S6Z32 ptr [dataptr], NLZ(val)
  while ((val = (val >> (NLZ(val) + 1))) != 0)
    USEWORD32(add [dataptr + mul], 0)
  fi

DEFZERO(labelname) - clearing/zeroing instruction

labelname: xor [labelname + IMMOFFS], 0
code to data (not self-modifying code):
\[ \text{USEWORD}(\text{mov [dataloc], acmltr}, \text{instruction} + \text{IMMOFFS})... \]

data to code:
\[ \text{TRANSFERDATA}(\text{codeblueprint, codedest, codesize}) \]
\[ \text{USEWORD}(\text{mov [codedest + IMMOFFS], acmltr}, \text{dataloc})... \]

III. DISCUSSION

To implement a Universal Turing Machine with pure self-modifying code requires the building blocks above and the pseudo-code for it can be done with a transition table which is encoded in a series of 5 DEFDATA statements as shown in Fig. 7.

The input tape would be a series of DEFDATA statements for each slot on the tape. The tape would assumed to have an adequate amount of blank DEFDATA before and after it. A double stack linked list structure could be used or even a double-linked list structure with 2 DEFDATA or 3 DEFDATA statements for each input respectively if such generality is preferred as given in Fig. 1.

itape, tape must be initialized through relocation either via compilation (such as with preprocessor macros in assembly or C boost library) or at runtime. Instruction pointer relative address makes far less relocations in 64-bit code, however 32-bit optimizations in the way the instruction encoding works can significantly enhance the 32-bit versions.

Listing 7. Transition and input tape constants and definitions

\[ \text{TSZ} \text{ EQU} ((\text{PFXSZ} + \text{IMMOFFS}) * 2 + \text{PFXSZ} * 2 + \text{MOVSZ32} + \text{OPSZ1} + \text{PTRSZ}) * 2 + (\text{OPDBLARGSZ}) * 3 \]
\[ \text{DEFTRANSITION}(\text{Index, nextptrval, cursymval, newsymval, directionval, newptrval}) \]
\[ \text{DEFADDR}\text{newptrval} \]
\[ \text{DEFDATA32}(\text{cursymval}) \]
\[ \text{DEFDATA32}(\text{newsymval}) \]
\[ \text{DEFDATA32}(\text{directionval}) \]
\[ \text{DEFDATA32}(\text{itapeval}) \]

32-bit optimization: \[ \text{DEFADDR} \rightarrow \text{DEFDATA32} \]
\[ \text{DEFTRANSITION} \rightarrow \text{DEFTRANSITION32} \]

\[ \text{TSZ32} \text{ EQU} (\text{OPDBLARGSZ}) * 5 \]

Thus the natural flow of a Universal Turing Machine is outlined but the self-modifying code beyond having some formal extra facilitation steps for initialization, finalization and copying, is largely traditional as Fig. 2 shows. Fig. 8 thus gives a full implementation to complete the informal proof.

Fig. 1. Generalized Input Tape

Fig. 2. General algorithm flow
The machine executes every single instruction at least once, remains in an able to execute state after its halting, and it clobbers no registers with an exception of having to make use of the accumulator register which is essential for the 64-bit code. This can be preserved with the prologue/epilogue sequence of Listing 9.

Some 32-bit only optimizations are available, if the DEFDATA is changed to DEFDATA32 in all places including the transition tape code.

Many other building blocks could be constructed including data transfer for variable amounts of data using the repeating prefix “rep” and the counter register for example or arbitrary multiplication using shift left.

Based on the result here, its complexity and difficulty of implementation on current processor architectures, we introduce a concept termed a code blueprint, whereby the skeleton of the code only specifies how the data flows through the system, without concerning the specific instructions or arguments. There is a clear motivation for a new processor design and computer architecture which has RAM memory tightly bound to the NAND-gates which are interconnected in a complex tangled web but the RAM allows the NAND-gates to take on any functional units flowing the data in any manner conceivable while the gates themselves allow that data to be manipulated or left intact. The NAND-gates connect back to the memory and allow the code blueprint to change, so that arithmetic units could be specified anywhere on this type of memory/processor integrated environment. This allows the von Neumann architecture to be properly and fully realized where current limits in efficiency of transistor and memory technology have not allowed for a full utilization yet. It leads to a higher order programming language which no longer concerns itself with byte and word sizes and looks at factoring in terms of data flows rather than strictly control flow which functions and loops largely define as reusable chunks currently.

A code blueprint is thus merely a data flow layout and can be viewed as a data flow graph. The actual operations performed on the data would be part of the data flowing through the system. This is a topic for exploration in future research.

IV. Performance

Several performance issues come up due to the lack of optimization for this type of code on modern processors. The way the caches optimize performance largely assumes lack of self-modifying code. The performance time for several universal turing machines are presented over a simple addition Turing machine and 100000 repeated operations in Table 4.
The code obfuscation techniques will prevent state of the art tools such as IDA Pro and its accompanying Hex-Rays Decompiler from having anything but compounded false assumptions and confusion while trying to analyze and display such code. Manual review of such code is so laborious that automated refactoring tools would be necessary to understand the code unless of course a known algorithm is implemented which can have its parts extracted using black box or grey box techniques as this type of code obfuscation is only effective against white box attacks. Attempts to deploy this via an emulator in the CHES 2017 Capture the Flag Challenge, WhibOx seeking to hide an AES key were thus ineffective as software tracing and differential computational and tracing analysis make this obfuscation a mere ineffective wrapper layer without directly dealing with these issues and research on improvement in these scenarios is ongoing.

More interesting is the future of self-modifying code in the context of a processor and memory architecture whose pipeline is optimized for it. The von Neumann architecture could be realized in its proper generality and more efficient, more parallel, more specialized computing could be done on a very generalized hardware.

APPENDIX

Example compiler assembly emitter algorithm not completely done with previous macros for generality to allow the reader to see and go through that process as an exercise is hereby provided. There are 3 examples, namely for general forms of 32-bit condition branching is contained in Listing 10, long multiplication (not most efficient algorithm for 32 or 64-bit while Karatsuba combined with table look up among others are known) is found in Listing 11 and long division where complexity of general implementation can be gleaned is provided in Listing 12. Of course multiplication by a constant algorithm previously is far more efficient when possible, and division by a constant could be done using precomputed multiplication magic numbers for efficiency though algorithmic optimizations in terms of processor implementation and compiler optimizations are largely beyond the scope of this paper but something to consider when making real world implementations. Floating point emulation units can be compiled using the above techniques to handle all FPU arithmetic and has been done in the assembly emitter of an actual compiler LCC but these details are not enumerated here for sake of brevity though the project will be available as open source in the future. The complexity of debugging, troubleshooting, writing and maintaining such code can thus be considered and appreciated, and the necessity of using a compiler or obfuscation tool is made apparent.

Listing 10. 32-bit Conditional branching

DoBranch(x, y, not, gt, sign) - <, <=, =, !=, >, >=

<table>
<thead>
<tr>
<th>Type</th>
<th>Time for 100,000 executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 self-modifying</td>
<td>1.53 seconds</td>
</tr>
<tr>
<td>x86 self-modifying using x64 technique</td>
<td>1.73 seconds</td>
</tr>
<tr>
<td>x64 self-modifying</td>
<td>2.51 seconds</td>
</tr>
<tr>
<td>x64 non-self modifying C equivalent</td>
<td>0.55 seconds</td>
</tr>
<tr>
<td>x64 non-self modifying C equivalent</td>
<td>0.54 seconds</td>
</tr>
<tr>
<td>x64 non-self modifying assembly equivalent</td>
<td>0.45 seconds</td>
</tr>
<tr>
<td>x64 non-self modifying assembly equivalent</td>
<td>0.46 seconds</td>
</tr>
<tr>
<td>x86 non-self modifying equivalent</td>
<td>0.75 seconds</td>
</tr>
<tr>
<td>x64 non-self modifying mov-only assembly equivalent</td>
<td>0.80 seconds</td>
</tr>
</tbody>
</table>

Whereas there are generally more predictable and limited options in a mov-only environment, pure infinitely self-modifying code provides a whole variety of different optimizations and options for doing the same operations and is much more difficult to disassemble or reverse engineer. A tool to undo the obfuscation would be very difficult if optimization and option variety are used in the process especially if the analysis were based on clock cycles or input dependent.

V. Conclusion

Code obfuscation using the techniques outlined would be significantly harder to undo while the movfuscator project now has parallel demovfuscator deobfuscation tools developed, this technique allows one to use several methods to do the same task, along with maintaining symbols and pointer offsets to make the code specific to each situation as the interlinks between the instructions make the possibility for arbitrary or unique optimizations and strategies.
PURE INFINITELY SELF-MODIFYING CODE IS REALIZABLE AND TURING-COMPLETE

Listing 11. 32-bit Long Multiplication

DoMult(x, y, result) - 32-bit * 32-bit - 32-bit needs no sign considerations

Listing 12. 32-bit Long Division

DoDivision(x, y, result, mod, sign)

ACKNOWLEDGMENT

The author would like to thank Norbert Tihanyi for interesting presentations which lead to the idea explored in this paper.

REFERENCES


