INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2018, VOL. 64, NO. 2, PP. 123–129
Manuscript received September 29, 2017; revised March, 2018. DOI: 10.24425/119359

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

Gregory Morse

***Abstract*—Although self-modifying code has been shyed 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 inherint 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
SELF-MODIFYING CODE INSTRUCTIONS

Type	Instruction names
Memory movement	movsb, movsw, movsd
Memory writing	mov, xchg
Boolean and mathematical logic	add, sub, adc, sbb, xor and, or, not, neg, shl, shr
Transfer	call

TABLE II
CONSTANTS

Type	Name	Value
single byte opcode size	OPSZ1	1
double byte instruction size	OPSZ2	2
Force 32-bit constant	CNSTFRC32	0x80000000
Inst. with 2 32-bit args	OPDBLARGSZ	OPSZ2 + PTRSZ32 * 2
Offset to 32-bit arg in memory inst.	IMMOFFS	OPSZ2 + PTRSZ32
Offset to 32-bit arg in register inst.	REGIMMOFFS	OPSZ2 + 1

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
32/64 BIT CONSTANTS

Type	Name	32-bit	64-bit
Prefix length	PFXSZ	0	1
Pointer size	PTRSZ	4	8
Word size name	SZW	dword	qword
Define raw offset	MOFFS	dd	dq
General purpose register	acmltr	eax	rax
Stack register	gsp	esp	rsp
Source index	gsi	esi	rsi
Destination index	gdi	edi	rdi

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

mov Rdest, c

```
        mov SZW32 ptr [Rdest + IMMOFFS], c
Rdest:  mov SZW32 ptr [Rdest + IMMOFFS], 0
```

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 + offset], Rsrc

```
Rsrc:   mov SZW32 ptr [Rsrc + IMMOFFS], 0
        xchg acmltr32, [Rsrc + IMMOFFS]
        mov [Rdest + IMMOFFS + offset * (OPDBLARGSZ)
            ], acmltr32
        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
mov acmltr, [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 SZW32 ptr [d + do],
    OFFSET s + so
```

64-bit

```
mov SZW32 ptr [d + do],
     ($-d+do)-($-s+so)
```

Halt:
DOHALT()

```
mov [$], 0
```

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 $>$, $\neq$ cases

```
sub [dataptr], val
if (!onlygt) neg [dataptr]
sbb [maskptr], 0
if (preserve)
       if (!onlygt) neg [dataptr]
       add [dataptr], val
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, OFFSET
    stptr + IMMOFFS)
```

64-bit

```
xchg acmltr, [stptr]
mov [stptr], acmltr
MOFFSMOV(stptr, stptr)
MOFFSMOV(svptr, stptr)
```

JUMP(addr)

32-bit

```
xchg gsp, [stptr +
    IMMOFFS]
call addr
```

64-bit

```
xchg gsp, [stptr]
add [svptr], PTRSZ * 2 +
    PFXSZ + OPSZ1
xchg gsp, [svptr]
call addr
```

JUMPTARGET(addr)

32-bit

```
addr: xchg gsp, [stptr +
     IMMOFFS]
add [stptr + IMMOFFS],
    PTRSZ
xchg acmltr, [stptr +
    IMMOFFS]
mov [stptr + OPSZ2],
    acmltr
xchg acmltr, [stptr +
    IMMOFFS]
```

64-bit

```
addr: xchg gsp, [stptr]
sub [stptr], PFXSZ +
    OPSZ1 + PTRSZ
xchg acmltr, [stptr]
mov [svptr], acmltr
xchg acmltr, [stptr]
```

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, destinit, size)

```
DEFDATA(source, sourceinit)
xchg gsi, [sourceinit]
DEFDATA(dest, destinit)
xchg gdi, [destinit]
MOVSMACRO(size)
xchg gsi, [sourceinit]
sub [sourceinit], size
xchg gdi, [destinit]
sub [destinit], size
```

MOVSMACRO(size)

32-bit

```
size / 4 DUP movsd
if ((size mod 4) & 2)
    movsw
if ((size mod 4) & 1)
    movsb
```

64-bit

```
size / 8 DUP movsq
if ((size mod 8) & 4)
    movsd
if ((size mod 8) & 2)
    movsw
if ((size mod 8) & 1)
    movsb
```

USEWORD(USEOP, source)

```
xchg acmltr, [source]
USEOP
xchg acmltr, [source]
```

USEWORDOP64(op, d, do, s, so)

```
USEWORD32(DATAOP(op, d,
    acmltr32), s)
if (64) USEWORD32(DATAOP
   (op, d + do, acmltr32
    ), s + so)
```

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 signed multiply by constant

```
shl SZW32 ptr [dataptr], NLZ(val)
if ((val = (val >> (NLZ(val) + 1))) != 0)
        USEWORD32((mov [dataptr&mul& + IMMOFFS],
            acmltr32), dataptr)
        shl SZW32 ptr [dataptr], NLZ(val)
        while ((val = (val >> (NLZ(val) + 1))) != 0)
                USEWORD32((add [dataptr&mul& +
                    IMMOFFS], acmltr32), dataptr)
                shl SZW32 ptr [dataptr], NLZ(val)
        loop
        dataptr&mul&:
        add SZW32 ptr [dataptr], CNSTFRC32
fi
```

DEFZERO(labelname) - clearing/zeroing instruction

```
labelname: xor [labelname + IMMOFFS], 0
```

code to data (not self-modifying code):
USEWORD((mov [dataloc], acmltr), instruction + IMMOFFS)...

data to code:
TRANSFERDATA(codeblueprint, codedest, codesize)
USEWORD((mov [codedest + IMMOFFS], acmltr), 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 prefered as given in Fig. 1.

itape, ttape 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

TSZ EQU ((PFXSZ + IMMOFFS) * 2 + PFXSZ * 2 + MOVSZ32 + OPSZ1 + PTRSZ) * 2 + (OPDBLARGSZ) * 3
DEFTRANSITION(Index, nextptrval, cursymval, newsymval, directionval, newptrval)

```
DEFADDR(nextptr&Index&, nextptrval)
DEFDATA32(cursym&Index&, cursymval)
DEFDATA32(newsym&Index&, newsymval)
DEFDATA32(direction&Index&, directionval)
DEFADDR(newptr&Index&, newptrval)
```

DEFINPUT(Index, tapeval)

```
DEFDATA32(itape&Index&, tapeval)
```

32-bit optimization: DEFADDR → DEFDATA32
DEFTRANSITION → DEFTRANSITION32

```
TSZ32 EQU (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.

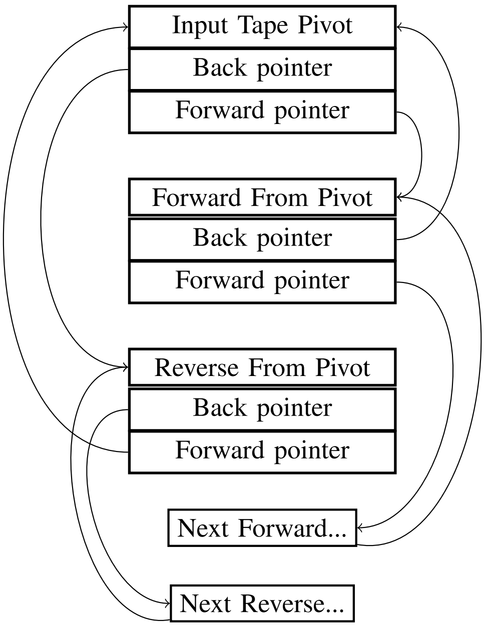

Fig. 1. Generalized Input Tape

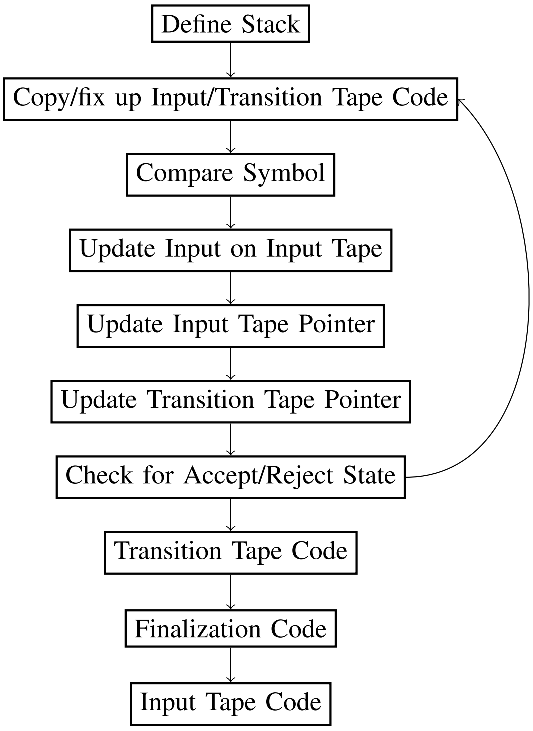
Define Stack
Copy/fix up Input/Transition Tape Code
Compare Symbol
Update Input on Input Tape
Update Input Tape Pointer
Update Transition Tape Pointer
Check for Accept/Reject State
Transition Tape Code
Finalization Code
Input Tape Code

Fig. 2. General algorithm flow

Listing 8. An example implementation

```
DEFSTACK()
JUMP(begin)
JUMPTARGET(begin)
TRANSFERDATA(curitapeinit, OFFSET itape, curitapeptr
    , OFFSET curitape, OPDBLARGSZ)
TRANSFERDATA(curttapeinit, OFFSET ttape, curttapeptr
    , OFFSET curtape, TSZ)
MOVEMODEOFFS(curitape, OPSZ2, curinputval, IMMOFFS)
MOVEMODEOFFS(usesymval, OPSZ2, curtinputval, IMMOFFS
    )
MOVEMODEOFFS(usenewsymval, OPSZ2, newinputval,
    IMMOFFS)
MOVEMODEOFFS(usedirval, OPSZ2, inpdir, IMMOFFS)
DEFDATA32(curitape, 0)
DEFADDR(curitapebptr, 0)
DEFADDR(curitapefptr, 0)
DEFADDR(curttape, 0)
DEFDATA32(usesymval, 0)
DEFDATA32(usenewsymval, 0)
DEFDATA32(usedirval, 0)
DEFADDR(usettapeval, 0)
USEWORD((mov [curttapeinit], acmltr), curttape)
USEWORDOP64(mov, newttapeval + IMMOFFS, OPDBLARGSZ,
    usettapeval, PTRSZ32)
USEWORDOP64(mov, itapebptr + IMMOFFS, OPDBLARGSZ,
    curitapebptr, PTRSZ32)
USEWORDOP64(mov, itapefptr + IMMOFFS, OPDBLARGSZ,
    curitapefptr, PTRSZ32)
DATAOP(mov, inputvalmask + IMMOFFS, 0)
USEWORD32((mov [inputupdate + REGIMMOFFS], acmltr32)
    , curinputval)
DEFDATA(0, curtinputval)
curinputval:
COMPARE32(CNSTFRC32, curtinputval + IMMOFFS,
    inputvalmask + IMMOFFS, false, false)
DATAOP(not, inputvalmask + IMMOFFS)
USEWORDOP64(mov, ttapevalmask + IMMOFFS, OPDBLARGSZ,
     inputvalmask, IMMOFFS)
USEWORDOP64(mov, inpdirmask + IMMOFFS, OPDBLARGSZ,
    inputvalmask, IMMOFFS)
newinputval: DATAOP(xor, inputupdate + REGIMMOFFS,
    CNSTFRC32)
inputvalmask: DATAOP(and, inputupdate + REGIMMOFFS,
    CNSTFRC32)
xchg acmltr, [curitapeinit]
inputupdate: DATAOP(xor, acmltr + IMMOFFS, CNSTFRC32
    )
xchg acmltr, [curitapeinit]
if (64) USEWORD32((mov [inpdir + OPDBLARGSZ +
    IMMOFFS], acmltr32), inpdir+IMMOFFS)
USEWORDOP64(mov, inpdircmb + IMMOFFS, OPDBLARGSZ,
    inpdir + IMMOFFS, OPDBLARGSZ)
itapebptr: OP64(and, inpdircmb + IMMOFFS, OPDBLARGSZ
    , CNSTFRC32)
OP64(not, inpdir + IMMOFFS, OPDBLARGSZ)
itapefptr: OP64(and, inpdir + IMMOFFS, OPDBLARGSZ,
    CNSTFRC32)
inpdircmb: OP64(or, inpdir + IMMOFFS, OPDBLARGSZ,
    CNSTFRC32)
USEWORDOP64(xor, inpdir + IMMOFS, OPDBLARGSZ,
    curitapeinit, PTRSZ32)
inpdirmask: OP64(and, inpdir + IMMOFFS, OPDBLARGSZ,
    CNSTFRC32)
inpdir: OP64(xor, curitapeinit, PTRSZ32, CNSTFRC32)
USEWORDOP64(mov, ttapeupdate, PTRSZ32, curttapeinit,
     PTRSZ32)
newttapeval: OP64(xor, ttapeupdate + IMMOFFS,
    OPDBLARGSZ, CNSTFRC32)
ttapevalmask: OP64(and, ttapeupdate + IMMOFFS,
    OPDBLARGSZ, CNSTFRC32)
ttapeupdate: OP64(xor, curttapeinit + IMMOFFS,
    OPDBLARGSZ, CNSTFRC32)
DATAOP(mov, ttapefail + IMMOFFS, 0)
DEFDATA(itapeptr, OFFSET itape)
USEWORD(COMPARE(acmltr, curttapeinit, ttapefail +
    IMMOFFS, false, true), itapeptr)
DATAOP(not, ttapefail + IMMOFFS)
DATAOP(mov, andfail + IMMOFFS, ($ - fail) - ($ -
    begin))
andfail: DATAOP(and, ttapefail + IMMOFFS, CNSTFRC32)
ttapefail: DATAOP(xor, exitcall + OPSZ1, CNSTFRC32)
JUMP(begin, exitcall)
JUMPTARGET(fail)
ttape:
DEFTRANSITION(...)
DATAOP(mov, exitcall + OPSZ1, ($ - fail) - ($ -
    begin))
USEWORD((mov [curitapeinit], acmltr), itapeptr)
DEFDATA(ttapeptrinit, OFFSET ttape)
USEWORD((mov [curttapeinit], acmltr), ttapeptrinit)
DOHALT()
itape:
DEFINPUT(...)
```

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.

Listing 9. Saving the accumulator for data addressing pattern

```
mov [acmltrsave], acmltr
;...
xchg acmltr, [acmltrptrsave]
mov [acmltrptrsave], acmltr
xchg acmltr, [acmltrsave]
DEFDATA(acmltrsave, 0)
DEFDATA(acmltrptrsave, OFFSET acmltrsave)
```

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 concievable 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.

It is thus from 30% to 100% slower than the mov-only equivalent, which are all much slower than just general assembly or C implementations. The algorithm however

TABLE IV
PERFORMANCE PROFILING RESULTS

Type	Time for 100,000 executions
x86 self-modifying	1.53 seconds
x86 self-modifying using x64 technique	1.73 seconds
x64 self-modifying	2.51 seconds
x86 non-self modifying C equivalent	0.55 seconds
x64 non-self modifying C equivalent	0.54 seconds
x86 non-self modifying assembly equivalent	0.45 seconds
x64 non-self modifying assembly equivalent	0.46 seconds
x86 non-self modifying mov-only assembly equivalent	0.75 seconds
x64 non-self modifying mov-only assembly equivalent	0.80 seconds

could be optimized even further and is representing a clear proof-of-concept and rough estimate of performance as opposed to an aggressively optimized version which would make it more competitive.

The slowness especially seen on native x64 versions where significant speed increase would be expected as opposed to minor is due to a variety of reasons including the implementation details of the processor: the define data instructions are expensive taking 4 instructions and many more bytes, there are more instructions to execute where 64-bit operands must be operated on via 2 32-bit operations, the code could spill outside the cache pages of the processor, uneven alignment boundaries could be more common, more modifications to the code including more that actually cause temporaral illegal code which breaks certain look-ahead and prediction capabilities and lastly the 32-bit instructions operate slowly in 64-bit mode. The modern processor pipeline optimizations are very complex and a full discussion of self-modifying code is worthy but outside the scope. Needless to say, from an obfuscation perspective, there are still many places where this loss in performance could be acceptable. Optimized 32-bit code is not far from the mov-only UTM.

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.

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 uneffective 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) - <, <=, =, !=, >, >=

```
mov [lbl2+OPSZ1], 0
mov [lbl1+IMMOFFS], 0
if (sign) xor [x], 0x80000000
xchg [x], acmltr32
if (sign) xor [y], 0x80000000
sub [y], acmltr32
if (!gt) neg [y]
```

```
sbb [lbl1+IMMOFFS], 0
if (!gt) neg [y]
add [y], acmltr32
if (sign) xor [y], 0x80000000
xchg [x], acmltr32
if (sign) xor [x], 0x80000000
if (!not) not [lbl1+IMMOFFS]
mov [lbl0+IMMOFFS], x-lbl2-OPSZ1-PTRSZ
lbl0: and [lbl1+IMMOFFS], 0x80000000
lbl1: xor [lbl2+OPSZ1], 0x80000000
xchg [stptr + IMMOFFS], gsp
lbl2: call [lbl2+OPSZ1+PTRSZ]
```

Listing 11. 32-bit Long Multiplication

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

```
mov [lbl10+IMMOFFS], 0
xchg [x], acmltr
mov [lbl7+IMMOFFS], acmltr
xchg [x], acmltr
xchg [y], acmltr
mov [lbl8+IMMOFFS], acmltr
xchg [y], acmltr
xchg [stptr+IMMOFFS], gsp
call [lbl0]
JUMPTARGET(lbl0)
mov [lbl2+IMMOFFS], 0
neg [lbl8+IMMOFFS]
sbb [lbl2+IMMOFFS], 0
neg [lbl8+IMMOFFS]
not [lbl2+IMMOFFS]
mov [lbl1], ($-lbl9)-($-lbl3-OPSZ1-PTRSZ)
lbl1: and [lbl2+IMMOFFS], 0x80000000
lbl2: xor [lbl3+OPSZ1], 0x80000000
xchg [stptr+IMMOFFS], gsp
lbl3: call [lbl3+OPSZ1+PTRSZ]
JUMPTARGET()
mov [lbl4+IMMOFFS], 1
xchg [lbl8+IMMOFFS], acmltr
and [lbl4+IMMOFFS], acmltr
xchg [lbl8+IMMOFFS], acmltr
xchg [lbl7+IMMOFFS], acmltr
mov [lbl6+IMMOFFS], acmltr
xchg [lbl7+IMMOFFS], acmltr
mov [lbl5+IMMOFFS], 0
lbl4: mov [lbl4+IMMOFFS], 0x80000000
neg [lbl4+IMMOFFS]
sbb [lbl5+IMMOFFS], 0
lbl5: and [lbl6+IMMOFFS], 0x80000000
lbl6: add [lbl10+IMMOFFS], 0x80000000
lbl7: mov [lbl7+IMMOFFS], 0x80000000
shl [lbl7+IMMOFFS], 1
lbl8: mov [lbl8+IMMOFFS], 0x80000000
shr [lbl8+IMMOFFS], 1
xchg [stptr+IMMOFFS], gsp
call [lbl0]
JUMPTARGET()
mov [lbl3+OPSZ1], 0
lbl10: mov [result], 0x80000000
```

Listing 12. 32-bit Long Division

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

```
if (sign)
        mov [lbl0+IMMOFFS], 0x80000000
        xchg [x], acmltr
        and [lbl0+IMMOFFS], acmltr
        xchg [x], acmltr
        if (!mod)
                mov [lbl1+IMMOFFS], 0x80000000
                xchg [y], acmltr
                and [lbl1+IMMOFFS], acmltr
                xchg [y], acmltr
                lbl0: xor [lbl1+IMMOFFS], 0x80000000
        shr [(mod ? lbl0 : lbl1)+IMMOFFS], 31
        neg [(mod ? lbl0 : lbl1)+IMMOFFS]
        (mod ? lbl0 : lbl1): mov [lbl22+IMMOFFS], 0
            x80000000
        xchg [x], acmltr
        mov [lbl20+IMMOFFS], acmltr
        mov [lbl3+IMMOFFS], acmltr
        shl [lbl3+IMMOFFS], 1
        xchg [x], acmltr
        mov [lbl2+IMMOFFS], 0
        sub [lbl20+IMMOFFS], 0x80000000
        sbb [lbl2+IMMOFFS], 0
        add [lbl20+IMMOFFS], 0x80000000
        not [lbl2+IMMOFFS]
        lbl2: and [lbl3+IMMOFFS], 0x80000000
        lbl3: sub [lbl20+IMMOFFS], 0x80000000
        xchg [y], acmltr
        mov [lbl11+IMMOFFS], acmltr
        mov [lbl5+IMMOFFS], acmltr
        shl [lbl5+IMMOFFS], 1
        xchg [y], acmltr
        mov [lbl4+IMMOFFS], 0
        sub [lbl13+IMMOFFS], 0x80000000
        sbb [lbl4+IMMOFFS], 0
        add [lbl13+IMMOFFS], 0x80000000
        not [lbl4+IMMOFFS]
        lbl4: and [lbl5+IMMOFFS], 0x80000000
        lbl5: sub [lbl13+IMMOFFS], 0x80000000
        xchg [lbl13+IMMOFFS], acmltr
        mov [lbl14+IMMOFFS], acmltr
        xchg [lbl13+IMMOFFS], acmltr
else
        xchg [x], acmltr
        mov [lbl20+IMMOFFS], acmltr
        xchg [x], acmltr
        xchg [y], acmltr
        mov [lbl13+IMMOFFS], acmltr
        mov [lbl14+IMMOFFS], acmltr
        xchg [y], acmltr
mov [lbl21+IMMOFFS], 0
mov [lbl6+IMMOFSS], 32
lbl6: mov [lbl6+IMMOFFS], 0x80000000
xchg [stptr+IMMOFFS], gsp
call [lbl7]
JUMPTARGET(lbl7)
mov [lbl9+IMMOFFS], 0
neg [lbl6+IMMOFFS]
sbb [lbl9+IMMOFFS], 0
neg [lbl6+IMMOFFS]
not [lbl9+IMMOFFS]
mov [lbl8+IMMOFFS], ($-lbl19)-($-lbl10-OPSZ1-PTRSZ)
lbl8: and [lbl9+IMMOFFS], 0x80000000
lbl9: xor [lbl10+OPSZ1], 0x80000000
xchg [stptr+IMMOFFS], gsp
lbl10: call [lbl10+OPSZ1+PTRSZ]
JUMPTARGET(lbl10)
xchg [lbl20+IMMOFFS], acmltr
mov [lbl11+IMMOFFS], acmltr
mov [lbl12+IMMOFFS], acmltr
xchg [lbl20+IMMOFFS], acmltr
shl [lbl20+IMMOFFS], 1
shl [lbl21+IMMOFFS], 1
mov [lbl18+IMMOFFS], 1
lbl11: sub [lbl20+IMMOFFS], 0x80000000
adc [lbl21+IMMOFFS], 0
lbl12: add [lbl20+IMMOFFS], 0x80000000
lbl13: mov [lbl17+IMMOFFS], 0x80000000
mov [lbl15+IMMOFFS], 0
lbl14: sub [lbl21+IMMOFFS], 0x80000000
sbb [lbl15+IMMOFFS], 0
lbl15: and [lbl17+IMMOFFS], 0x80000000
xchg [lbl15+IMMOFFS], acmltr
mov [lbl16+IMMOFFS], acmltr
xchg [lbl15+IMMOFFS], acmltr
not [lbl16+IMMOFFS]
lbl16: and [lbl18+IMMOFFS], 0x80000000
lbl17: add [lbl21+IMMOFFS], 0x80000000
lbl18: add [lbl20+IMMOFFS], 0x80000000
dec [lbl6+IMMOFFS]
xchg [stptr+IMMOFFS], gsp
call [lbl7]
JUMPTARGET(lbl19)
mov [lbl10+IMMOFFS], 0
if (mod)
        lbl20: mov [lbl20+IMMOFFS], 0x80000000
        lbl21: mov [result], 0x80000000
else
        lbl20: mov [result], 0x80000000
        lbl21: mov [lbl21+IMMOFFS], 0x80000000
if (sign)
        xchg [result], acmltr
        mov [lbl23+IMMOFFS], acmltr
        shl [lbl23+IMMOFFS], 1
        xchg [result], acmltr
        lbl22: and [lbl23+IMMOFFS], 0x80000000
        lbl23: sub [result], 0x80000000
```

Acknowledgment

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

References

[1] Dolan, S. *mov is Turing-complete*. Computer Laboratory, University of Cambridge. Technical report (2013)

[2] *The M/O/Vfuscator* (2015). https://github.com/xoreaxeaxeax/movfuscator

[3] Anckaert B., Madou M., De Bosschere K. (2007) *A Model for Self-Modifying Code*. In: Camenisch J.L., Collberg C.S., Johnson N.F., Sallee P. (eds) Information Hiding. IH 2006. Lecture Notes in Computer Science, vol 4437. Springer, Berlin, Heidelberg

[4] Hongxu Cai , Zhong Shao , Alexander Vaynberg, *Certified self-modifying code*, ACM SIGPLAN Notices, v.42 n.6, June 2007