Functional Extraction of Virtual Machine Viruses

Timothy Daly

November 12, 2007

Abstract

One technique used by virus writers is to construct a virtual machine that can execute instructions and write subroutines for that virtual machine that compute the same semantics as the Intel (or other host) instruction set. A virus written in Intel instructions is then translated to the virtual machine and the result is a new virus that executes virtually. Function Extraction can assist in understanding what the program actually does but there are several challenges which we examine in this paper.

1 Function Extraction technology

Function extraction is a technique pioneered by Linger, Mills, and Witt[?] to extract the static mathematical behavior of a program. The fundamental idea involves composing the meaning of each instruction with all previous instructions to derive the full program behavior without execution.

Each machine language statement is rewritten into a conditional concurrent assignment statement (CCA) that captures the semantics in terms of its net change of machine state. CCAs are a static description of the final state (after the machine statement) in terms of the initial state (before the machine statement) under certain conditions:

```
if condition then
   final state = f(initial state)
```

So, for instance, the Intel machine instruction,

```
neg eax
```

has a CCA that looks like:

```
true ==>
 EAX := mod(-EAX,2^32)
 ZF  := (EAX == 0)
 SF  := ((1 <= EAX) and (EAX <= 2^31))
 PF  := is_even_parity(mod(-EAX,2^8))
 CF  := (EAX != 0)
 OF  := (EAX == 2^31)
 AF  := (mod(EAX,2^4) != 0)
```

The ZF (zero), SF (sign), PF (parity), CF (carry), OF (overflow), and AF (adjust) are Intel machine flags. They are all set as side-effects of executing the NEG instruction.

Once the static semantics of each Intel instruction exists we can translate a machine language program into a sequence of these instructions into a sequence of CCAs:

```
(I1 ; I2 ; I3 ; I4 ; I5) => (CCA1 ; CCA2 ; CCA3 ; CCA4 ; CCA5)
```

Then we can compose the sequence of CCAs by applying the function F which takes 2 CCAs and returns a new CCA:

```
program behavior == F(F(F(F(CCA1,CCA2),CCA3),CCA4),CCA5)
```

which is just:

```
t1:=F(CCA1,CCA2)
t2:=F(t1,CCA3)
t3:=F(t2,CCA4)
t4:=F(t3,CCA5)
whole program behavior == t4
```

Notice that the computation does not execute the program but computes the static behavior of the system under all possible conditions.

2 The Virtual Machine problem

The function extraction technology, as well as other techniques, can be used to discover the behavior of a program and, thus, reverse-engineer the virus.

In order to hide the behavior of a virus the programmer can use the idea of a virtual machine. The virtual machine instruction set can have any instructions as long as it is turing complete. Once the virtual machine is defined then a series of subroutines can be written in the new instruction set which mimic exactly the behavior of the intel instruction set. So we would find a virtual machine subroutine that computes the Intel NEG instruction.

A virus written for the Intel instruction set can be automatically "compiled" into a series of virtual subroutines. When the virtual machine "executes" these virtual instructions they compute the same function as the original Intel virus and thus have the same effect. This slows down the execution of the virus but makes it considerably harder to understand what the virus is doing.

In order to understand how we can apply function extraction to this problem we construct our own virtual machine and our own subroutine library that mimics the Intel instructions. For the purpose of this analysis we consider the swap routine as our canonical example and construct a virtual version.

This sequence of instructions effectively swaps the EAX and EBX registers.

```
sub eax,ebx   a-b
add ebx,eax   b+(a-b) = a
sub eax,ebx   (a-b)-(b+(a-b)) = -b
neg eax       -(a-b)-(b+(a-b))= +b
```

3 The CARDIAC computer

CARDIAC (Cardboard Illustrative Aid to Computation) is a teaching tool from Bell Labs[1, 2], designed by David Hagelbarger and Saul Fingerman. It is a piece of cardboard with slides that the user can move to simulate the arithmetic-logic unit of a computer. It has a 100 instruction memory, input, output, and 10 instructions:

```
Opcode Meaning
  0    Input a number into a memory cell
  1    Load a number into the accumulator from a memory cell
  2    Add the contents of a memory cell to the accumulator
  3    Test the sign of the accumulator, branch if negative
  4    Shift the contents of the accumulator left, then right
  5    Output the contents of a memory cell
  6    Store the accumulator into a memory cell
  7    Subtract the contents of a memory cell from the accumulator
  8    Jump to a specified memory cell, write current address to 99
  9    Update the memory address and halt
```

We can construct a virtual machine using the cardiac computer (CVM) which has 10 instructions and a memory of 100 cells. The top level loop of the virtual machine sets up the memory with the required subroutine, fills in the data in the memory array for input, executes the CVM program, and copies the results from the memory array.

4 The Key Idea

Since the function extraction technology already has the CCA semantics of the Intel instructions we can create CVM subroutines that compute CCA semantics. This gives a nice correspondence between virtual machine subroutines and CCAs. Note that this will not be true in general for any virtual machine but we wish to give ourselves every advantage in this problem. Even with this spurious correspondence we will encounter deep problems that admit no obvious solutions.

For the purpose of illustration we will only illustrate the CVM subroutines for the CCA of the NEG instruction. Recall that

```
neg eax
```

has a CCA that looks like:

```
true ==>
 EAX := mod(-EAX,2^32)
 ZF  := (EAX == 0)
 SF  := ((1 <= EAX) and (EAX <= 2^31))
 PF  := is_even_parity(mod(-EAX,2^8))
 CF  := (EAX != 0)
 OF  := (EAX == 2^31)
 AF  := (mod(EAX,2^4) != 0)
```

The idea is to set up a command loop that computes each of the individual lines in the CCA, that is, a call to compute EAX, a call to compute ZF, etc. Thus computing the NEG instruction involves 7 virtual functions. Each function is composed of smaller functions, called "micro-ops", such as 'is_even_parity', 'mod', '<=', etc. We will construct CVM subroutines to compute these micro-ops.

An interesting complication is that the intel machine is fundamentally a binary representation and the CVM is fundamentally a decimal representation. Thus an intel 'byte' has a range of 0..255 and a CVM byte has a range of 0..999. We need to do cell-based modular arithmetic in order to stay within the intel range.

5 The NEG Instruction

The NEG instruction CCA is:

```
OP1 := mod(-OP1,2^NN)
ZF  := (OP1 == 0)
SF  := ((1 <= OP1) and (OP1 <= 2^(NN-1))
PF  := is_even_parity(mod(-OP1,2^8))
CF  := (OP1 != 0)
OF  := (OP1 == 2^(NN-1))
AF  := (mod(OP1,2^4) != 0)
```

Clearly we need the micro-op functions mod, is_even_parity, ==, <=, −, ! =, and exponentiation.

If we look at computing the zero flag (ZF) behavior we see that

```
ZF := (EAX == 0)
```

so we need a CVM micro-op routine '==' which compares 2 32-bit binary encoded integers. For convenience we insert the value of the EAX register into the memory image as a sequence of 4 values which range from [0..255], that is, the decimal value of the byte. The '==' routine simply compares each byte and returns either a false (zero) or true (one). Here is the routine in Cardiac machine language:

6 The ZF flag and the == subroutine

The Intel NEG instruction has a CCA which computes

```
ZF := (EAX == 0)
```

so the == CVM subroutine requires 2 inputs which would be laid out in memory as:

```
input op1 [90..93], op2 [94..97]
output op1==op2? true=>1, false=>0 [98]

algorithm:
  compare each byte for equality
  if any byte differs then return false

  bytes are equal if (a-b)=(b-a)
  if this is not true then one side of
  the equation is negative.

input memory layout:

program: start at address 01

01:190  load op1 byte 1      is op1[1]== op2[1]
02:794  subtract op2 byte 1  a-b
```

```
03:387  jump if negative
04:194  load op2 byte 1      b-a
05:790  subtract op1 byte 1
06:387  jump if negative

07:191  load op1 byte 2      is op1[2]==op2[2]
08:795  subtract op2 byte 2
09:387  jump if negative
10:195  load op2 byte 2
11:791  subtract op1 byte 2
12:387  jump if negative

13:192  load op1 byte 3      is op1[3]==op2[3]
14:796  subtract op2 byte 3
15:387  jump if negative
16:196  load op2 byte 3
17:792  subtract op1 byte 3
18:387  jump if negative

19:193  load op1 byte 4      is op1[4]==op2[4]
20:797  subtract op2 byte 4
21:387  jump if negative
22:197  load op2 byte 4
23:793  subtract op1 byte 4
24:387  jump if negative

25:883  jump to true return

82:001  true                 return true
83:182  load 1==true
84:698  store 1 in result
85:900  halt

86:000  false                return false
87:186  load 0==false
88:698  store 0 in result
89:900  halt

data start at address 90

90 op1 byte 1 (op1[0..7])
91 op1 byte 2 (op1[8..15])
92 op1 byte 3 (op1[16..23])
93 op1 byte 4 (op1[24..31])
94 op2 byte 1 (op2[0..7])
95 op2 byte 2 (op2[8..15])
```

```
96 op2 byte 3 (op2[16..23])
97 op2 byte 4 (op2[24..31])
98 flag result (0 or 1)
```

In order to use this routine we construct a memory array of 100 cells capable of holding 3 decimal digits, roughly 10 bits, or rounding up we need 2 bytes per memory cell. Thus our virtual computation is naturally suited to 16 bit arithmetic and we allocate 16 bits per cell.

The two values to be compared (in this case, the contents of EAX and 0) are written into the memory image directly. EAX is written into cells [90..93] and 0 is written into cells [94..97]. The result of the computation will be either 0 (false) or 1 (true) and that result will be available in cell 98.

7 The Virtual CCA problem

We can construct CCAs for the 10 virtual machine instructions we use the opcodes LOAD (1), TEST (3), STORE (6), SUBTRACT (7), JUMP (8), and HALT (9).

We could write CCAs for each of these virtual instructions. This would allow us to compute the whole program behavior for the '==' CVM subroutine. Thus we could, in theory, show that the above code will compute the '==' function.

There are some problems that arise at this stage.

- First, we would not, in general, know the instruction set of the virtual machine used in the virus and would not have clear semantics for those instructions.
- Second, even if we know the instruction set and we can construct the CCAs there would be 'side-effects' encoded in the CCAs for the virtual machine. For instance, in the above code we compute not only the '==' function but we have various side-effects on memory, both virtual (e.g. cell 98) and real (e.g. the bytes holding cell 98).
- Third, we need to know that we are computing a flag which can only have either a 0 or 1 value so we know enough to throw away the side-effects.

8 The SF flag and the <= subroutine

Similar to the ZF flag computation we can write a CVM micro-op for the '<=' subroutine:

```
input op1 [90..93], op2 [94..97]
output op1<=op2? true=>1, false=>0 [98]

algorithm:
 For each byte in 4..1 do
```

```
  load byte(i) for op2
  subtract byte(i) for op1
  if any byte differs then return false

  bytes are equal if (a-b)=(b-a)
  if this is not true then one side of
  the equation is negative.

input memory layout:

program: start at address 01

01:190  load op2 byte 1      is op2[1] > op1[1]
02:794  subtract op1 byte 1
03:386  jump if negative

04:194  load op2 byte 2      is op2[2] > op1[2]
05:790  subtract op1 byte 2
06:386  jump if negative

07:191  load op2 byte 3      is op2[3] > op1[3]
08:795  subtract op1 byte 3
09:386  jump if negative

10:195  load op2 byte 4      is op3[4] > op1[4]
11:791  subtract op1 byte 4
12:386  jump if negative

13:883  jump to true return

82:001  true                 return true
83:182  load 1==true
84:698  store 1 in result
85:900  halt

86:000  false                return false
87:186  load 0==false
88:698  store 0 in result
89:900  halt

data start at address 90

90 op1 byte 1 (op1[0..7])
91 op1 byte 2 (op1[8..15])
92 op1 byte 3 (op1[16..23])
93 op1 byte 4 (op1[24..31])
```

```
94 op2 byte 1 (op2[0..7])
95 op2 byte 2 (op2[8..15])
96 op2 byte 3 (op2[16..23])
97 op2 byte 4 (op2[24..31])
98 flag result (0 or 1)
```

Computing the SF flag involves 2 calls to the '<=' subroutine with different inputs and the resulting flags need to be ANDed together.

9 The CF flag and the ! = subroutine

Similar to the ZF flag computation we can use the == micro-op and invert the result to compute the '! =' micro-op.

10 The -OP subroutines

Computing the negative of the EAX register is much harder than it would seem. The difficulty lies in the mismatch between the Intel binary representation and the CARDIAC decimal representation.

To compute the negative of a binary number we compute the two's complement of the number. This is done in two stages using 3 micro-op subroutines due to the limitations of the available memory in CARDIAC. The first stage computes the one's complement of the number which involves changing zero bits to one and one bits to zero. We do this on a byte by byte basis using a main program and a subroutine. The main program for the one's complement expects the number to complement in [25..28]. The byte-complement subroutine expects a byte to complement in cell 97 and returns the byte-complement in cell 98. The main program simply calls the subroutine 4 times, once for each input byte. The result is in [25..28]

10.1 The 1's complement subroutine

```
input 32-bit op1 in cells [25..28]
output bit-not(op1) in cells [25..28]
start at 01

Algorithm: the one's complement flips all the bits.


    complement 32 bit word

01:125 load byte 4
02:697 store it in subr
03:829 gosub complement
04:198 load complement result
```

```
05:625 store it

06:126 load byte 3
07:697 store it in subr
08:829 gosub complement
09:198 load complement result
10:626 store it

11:127 load byte 2
12:697 store it in subr
13:829 gosub complement
14:198 load complement result
15:627 store it

16:128 load byte 1
17:697 store it in subr
18:829 gosub complement result
19:198 load complement result
20:628 store it

21:900 halt and return 1's complement

25 byte 4
26 byte 3
27 byte 2
28 byte 1

     complement one byte

29:199 load return address
30:688 store return address

31:197 load byte

32:789 subtract 128 -- is bit 8 set?
33:335 jump if not set
34:839 go test next bit
35:198 load bit-not
36:289 add 128 (complement bit 8)
37:698 store bit-not
38:197 load byte

39:790 subtract 64 -- is bit 7 set?
40:342 jump if not set
41:846 go test next bit
42:198 load bit-not
```

```
43:290 add 64 (complement bit 7)
44:698 store bit-not
45:197 load byte

46:791 subtract 32 -- is bit 6 set?
47:349 jump if not set
48:853 go test next bit
49:198 load bit-not
50:291 add 32 (complement bit 6)
51:698 store bit-not
52:197 load byte

53:792 subtract 16 -- is bit 5 set?
54:356 jump if not set
55:860 go test next bit
56:198 load bit-not
57:292 add 16 (complement bit 5)
58:698 store bit-not
59:197 load byte

60:793 subtract 8 -- is bit 4 set?
61:363 jump if not set
62:867 go test next bit
63:198 load bit-not
64:293 add 8 (complement bit 4)
65:698 store bit-not
66:197 load byte

67:794 subtract 4 -- is bit 3 set?
68:370 jump if not set
69:874 go test next bit
70:198 load bit-not
71:294 add 4 (complement bit 3)
72:698 store bit-not
73:197 load byte

74:795 subtract 2 -- is bit 2 set?
75:377 jump if not set
76:881 go test next bit
77:198 load bit-not
78:295 add 2 (complement bit 2)
79:698 store bit-not
80:197 load byte

81:796 subtract 1 -- is bit 1 set?
82:384 jump if not set
```

```
83:888 jump to return from subr
84:198 load bit-not
85:296 add 1 (complement bit 1)
86:698 store bit-not
87:197 load byte

88:800 return

89 128
90 64
91 32
92 16
93 8
94 4
95 2
96 1

97 byte
98 0 (bit-not byte)
```

10.2 The two's complement

Having computed the one's complement we can call this subroutine which adds 1 to a number, maintaining the binary notation rather than the decimal notation arithmetic. The two's complement is formed by adding 1 to the one's complement. So bit patterns represent the decimals as (e.g. in 4 bits):

```
1100= -4
1101= -3
1110= -2
1111= -1
0000=  0
0001=  1
0010=  2
0011=  3
```

10.3 The X+1 subroutine

```
input: 32 bit binary integer in [95..98]
output: 32 bit binary integer + 1 in [95..98]
start at 01

01:198 load byte 1
02:200 add 1 (location 00 hardwired to 001)
03:698 store result
```

```
04:794 byte-256
05:325 neg => no carry; jump to return

06:698 0; carry 1... store the 0
07:197 load byte 2
08:200 add 1 (carry from last byte)
09:697 store result
10:794 byte-256
11:325 neg => no carry; jump to return

12:697 0; carry 1... store the 0
13:196 load byte 3
14:200 add 1 (carry from last byte)
15:696 store result
16:794 byte-256
17:325 neg => no carry; jump to return

18:695 0; carry 1... store the 0
19:195 load byte 4
20:200 add 1 (carry from last byte)
21:695 store result
22:794 byte-256
23:325 neg => no carry; jump to return

24:695 store zero, ignore carry
25:900 halt and return

94:256
95 byte 4
96 byte 3
97 byte 2
98 byte 1
```

11 More Fundamental Problems

We can continue to construct the CVM subroutines that compute the micro-ops needed for each CCA. Lets suppose we did this for the 4 instructions in the swap example:

```
sub eax,ebx   a-b
add ebx,eax   b+(a-b) = a
sub eax,ebx   (a-b)-(b+(a-b)) = -b
neg eax       -(a-b)-(b+(a-b))= +b
```

What difficulties arise in trying to analyze the behavior of the virtual machine computation?

We mentioned three difficulties above. More hurdles await us.

Suppose we compute the whole program behavior for each micro-op, that is, we can know the complete behavior of each CVM subroutine. Thus we know what the virtual machine is doing for each micro-op.

From that assumption we next need to understand what the top level loop, the arithmetic-logic unit of the CVM is doing. This loop is basically computing the CCA behavior (EAX and the six flags) for each Intel instruction. We need to extract several independent pieces of information from this loop.

- First we need the information about how the micro-ops are composed in order to compute the behavior (e.g. the SF flag requires 3 micro-ops). This tells us the final results of one of the flags.

- Second, we need to partition the computation so that we can group the loop computation into computing the CCA for NEG versus the CCA for SUB or ADD. We need to know when one ends and the other begins.

- Third, we assume there is no optimization performed (e.g. since ADD will step on the flags of SUB we could skip the flag computation of SUB). Any optimization steps will reduce the work of the virtual machine but will make it harder to find the boundary between Intel instructions.

- Fourth, we need to understand the loop itself. This loop is the arithmetic logic unit and the control structure of the virtual machine. We need to know when virtual machine conditions are set that are used for branching (e.g. we need to understand the TEST (3) opcode).

Assuming we leap these low level hurdles we arrive at a set of CCAs extracted from the virtual machine loop. This sequence of CCAs is a sequence of virtual Intel instructions. How do we decide which Intel instruction we have? Well, here we meet the 'fuzzy matching' problem, thus,

- Fifth, we have a set of CCAs and we need to know what Intel instruction they represent. Assuming we know the CCAs of every Intel instruction we can match this unknown set against all possible sets. However, we need to match things like

  ```
  ZF := EAX == 0
  ```

  versus

  ```
  ZF := 0 == EAX
  ```

  in the trivial case. It gets harder as we go along, getting Tarski-like inequalities that need to be matched. Ultimately this involves the problem of showing that one set of equations and inequalities 'cover' another set. In the limited cases we face we might be able to solve the problem.

Again, assuming we leap the fuzzy match hurdle we now have a set of CCAs that represent the original Intel virus extracted from the virtual environment. From here we return to the original function extraction research problem already in progress.

References

[1] http://en.wikipedia.org/wiki/Cardiac_-_cardboard_illustrative_aid_to_computation

[2] http://kylewiki.mine.nu/wiki/Cardiac_Computer