The Program-Data Separation Issue

Timothy Daly and Mark Pleszkoch

May 11, 2007

1 The Problem

FX currently assumes that programs can be modeled by separate program control state space and program data state space. This appears to be based on a paper by Harlan Mills that shows a method of flowcharting a machine language program using this assumption. The expression of this assumption is that the next instruction can always be statically determined.

This assumption appears to be valid for certain architectures such as the Java Virtual Machine (JVM). The JVM does not allow a running program to change the program control state space. Thus it appears that one can always statically determine the next instruction from the program source.

In the Intel architecture this assumption is not valid. Consider the program fragment:

```
.....
jmp ZFis1      ; (2)
jmp ZFis0      ; (3) <=========
.....
```

The source text appears to clearly indicate that statement (3) can never be reached because statement (2) is an unconditional jump and statement (3) has no label.

Now consider the program fragment:

```
.....
call subr      ; (1) <=========
jmp ZFis1      ; (2)
jmp ZFis0      ; (3)
.....
```

In the Intel semantics the `call` instruction (1) pushes the instruction pointer onto the stack and then unconditionally branches to the subroutine. The instruction pointer has already been incremented so the stack contains the address of the first jump instruction (2).

For most subroutines the expected behavior is that they will eventually do a `ret` instruction which will pop the top of the stack into the instruction pointer register (EIP). This will resume execution at statement (2). Thus, statement (3) is never executed.

1.1 Violating the key assumption on Intel hardware

The Intel chip does not maintain separate program control and program data spaces. These are mixed on the program data stack.

Since the program control state space information contained in the instruction pointer register (EIP) has been "exported" into the program data state space (the stack) we can manipulate this information and dynamically change the program behavior.

The key trick is to add a correct offset to the EIP register value laying on the top of the stack before executing the "ret" instruction. This will cause a new, modified value to be placed in the instruction pointer and start execution at a new location. Virus writers frequently do this.

We can create an example program. The first execution will be a normal, unmodified EIP value which returns to the expected `jmp` instruction at (2). The second execution will be a modified EIP value which returns to the unexpected `jmp` instruction at (3).

2 An Example Program

This program illustrates the tangling between program space and data space. The program does a call which pushes the instruction pointer (EIP) onto the stack. This causes a program state variable, the EIP, to be exposed in data space.

As the main program reads statically then it appears that the second jump can never be reached. and therefore case2 appears to be dead code.

However the `subr` routine will add 4 to the instruction pointer if the ZF is false (ZF=0)

⟨*⟩≡

```
        SECTION .text
        global main
main:   call subr        ; check the ZF
        jmp ZFis1        ; if zero flag is true (ZF=1) go here
        jmp ZFis0        ; if zero flag is false (ZF=0) go here

ZFis1:  add eax,1        ; case 1, the zero flag is true (ZF=1) case
        jmp end

ZFis0:  add eax,2        ; case 2, the zero flag is false (ZF=0) case
        jmp end

        SECTION .text
        global subr
subr:   jne ret1         ; this subr adds 4 to the return addr
        mov ebx,5        ; if ZF is false (ZF=0)
        add ebx,[esp]
        mov [esp],ebx
ret1:   ret              ; otherwise it does nothing

end:   nop
```

3 An Execution Trace

3.1 Compiling the program

First we need to assemble and link the program.

```
nasm -f elf -l jumpjump.lst jumpjump.asm
gcc -g3 -o jumpjump jumpjump.o
```

The bytes in memory are shown in the listing file:

3.2 The listing file

```
 1                                  SECTION .text
 2                                  global main
 3 00000000 E81E000000 main:        call subr
 4 00000005 E905000000              jmp ZFis1
 5 0000000A E90A000000              jmp ZFis0
 6
 7 0000000F 0501000000 ZFis1:       add eax,1
 8 00000014 E918000000              jmp end
 9
10 00000019 0502000000 ZFis0:       add eax,2
11 0000001E E90E000000              jmp end
12
13                                  SECTION .text
14                                  global subr
15 00000023 750B       subr:        jne ret1
16 00000025 BB05000000              mov ebx,5
17 0000002A 031C24                  add ebx,[esp]
18 0000002D 891C24                  mov [esp],ebx
19 00000030 C3         ret1:        ret
20
21 00000031 90         end:       nop
```

4 A gdb session

We will use the GNU debugger `gdb` to step thru this program one instruction at a time, drawing attention to the interesting register values and memory contents. Items of interest are marked by numbers and arrows.

4.1 The normal return

In this first of two examples we will execute the program and select the case that does a normal, unmodified return. That is, the subroutine `subr` will not change the EIP register on the stack.

First, we start the program

```
# gdb jumpjump
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
```

Next, we set a breakpoint to stop execution at the first instruction

```
(gdb) break main
Breakpoint 1 at 0x8048300
```

We can disassemble the main program to see the instructions in memory

```
(gdb) disas main
Dump of assembler code for function main:
0x08048300 <main+0>:        call   0x8048323 <subr>
0x08048305 <main+5>:        jmp    0x804830f <ZFis1>
0x0804830a <main+10>:       jmp    0x8048319 <ZFis0>
End of assembler dump.
```

We also disassemble the subroutine `subr`

```
(gdb) disas subr
Dump of assembler code for function subr:
0x08048323 <subr+0>:        jne    0x8048330 <ret1>
0x08048325 <subr+2>:        mov    $0x5,%ebx
0x0804832a <subr+7>:        add    (%esp,1),%ebx
0x0804832d <subr+10>:       mov    %ebx,(%esp,1)
End of assembler dump.
```

Next we start the program running

```
(gdb) run
Starting program: /mnt/hgfs/vmshare/jumpjump

Breakpoint 1, 0x08048300 in main ()
```

We immediately hit the first breakpoint we set on the first instruction. These are the interesting register values (others are suppressed). The EAX register will eventually be modified after the branch. The EBX regiser is used for arithmetic in the subroutine. The ESP register points to the top of the program data stack. The EIP register points to the next instruction to execute. The EFLAGS register contains the ZF flag (zero flag). We print both the eflags and the decoded ZF value when needed.

```
(gdb) info registers
eax            0x1              1
ebx            0x42130a14       1108544020
esp            0xbffff01c       0xbffff01c
eip            0x8048300        0x8048300          <=========
eflags         0x210246         2163270
```

Here we see that the ZF flag is true by default. We need to change it so that the subroutine will not modify the EIP value on the stack.

```
(gdb) ZF
ZF is true
```

This changes the EFLAGS register, which contains the ZF flag, to 0.

```
(gdb) set $eflags=0x0
```

And we can see that the ZF flag is changed.

```
(gdb) ZF
ZF is false                                        <=========
```

Now we "single instruction step" (si) to execute the `call` instruction.

```
(gdb) si
0x08048323 in subr ()                              <=========
```

We can see that the EIP register has been changed by 0x23 to point to the `jne` instruction, the first instruction in `subr`.

```
(gdb) info register
eax            0x1              1
ebx            0x42130a14       1108544020
esp            0xbffff018       0xbffff018
eip            0x8048323        0x8048323          <=========
eflags         0x210302         2163458
```

And we can see that the program data stack contains the "return address" on the top of the stack. This is the instruction pointer address of the first `jmp` instruction. The "ret" instruction will effectively go to this location.

```
(gdb) x 0xbffff018
0xbffff018:        0x08048305                      <=========
```

Now we execute the `jne` instruction. Since the ZF flag is false the `jne` instruction takes the branch. The next instruction will be a `ret` instruction.

```
(gdb) si
0x08048330 in ret1 ()                              <=========
```

We can see the registers just before the return happens.

```
(gdb) info registers
eax            0x1              1
ebx            0x42130a14       1108544020
esp            0xbffff018       0xbffff018
eip            0x8048330        0x8048330          <=========
eflags         0x210302         2163458
```

And we can see the address on the top of the stack. This is the address of the next instruction to execute after the `ret`. Note that it is the address of the first `jmp` instruction.

```
(gdb) x 0xbffff018
0xbffff018:        0x08048305                       <=========
```

Now we execute the `ret` instruction and we are back in `main`. We are about to execute the first `jmp` instruction.

```
(gdb) si
0x08048305 in main ()                               <=========
```

In order to show that the correct branch is actually taken we will construct a visible value in the EAX register. Note that the register has a value of 1.

```
(gdb) info registers
eax            0x1              1
ebx            0x42130a14       1108544020
esp            0xbffff01c       0xbffff01c
eip            0x8048305        0x8048305          <=========
eflags         0x210302         2163458
```

We execute the `jmp` instruction and are at the `add eax,1` instruction.

```
(gdb) si
0x0804830f in ZFis1 ()
```

This is the register set prior to the `add` instruction.

```
(gdb) info registers
eax            0x1              1                  <=========
ebx            0x42130a14       1108544020
esp            0xbffff01c       0xbffff01c
eip            0x804830f        0x804830f
eflags         0x210302         2163458
```

We execute the `add eax,1` instruction.

```
(gdb) si
0x08048314 in ZFis1 ()
```

And we can see the resulting value in EAX.

```
(gdb) info registers
eax            0x2              2                  <=========
ebx            0x42130a14       1108544020
esp            0xbffff01c       0xbffff01c
eip            0x8048314        0x8048314
eflags         0x210302         2163458
```

We execute the `jmp` to the end of the program and finish.

```
(gdb) si
0x08048331 in end ()
```

4.2 The modified return

In this second of two examples we will execute the program and select the case that does a modified return. That is, the subroutine `subr` will change the EIP register on the stack, causing a return to the second jump instruction rather than the first.

First, we start the program

```
# gdb jumpjump
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
```

Next, we set a breakpoint to stop execution at the first instruction

```
(gdb) break main
Breakpoint 1 at 0x8048300
```

We can disassemble the main program to see the instructions in memory

```
(gdb) disas main
Dump of assembler code for function main:
0x08048300 <main+0>:        call   0x8048323 <subr>
0x08048305 <main+5>:        jmp    0x804830f <ZFis1>
0x0804830a <main+10>:       jmp    0x8048319 <ZFis0>
End of assembler dump.
```

We also disassemble the subroutine `subr`

```
(gdb) disas subr
Dump of assembler code for function subr:
0x08048323 <subr+0>:        jne    0x8048330 <ret1>
0x08048325 <subr+2>:        mov    $0x5,%ebx
0x0804832a <subr+7>:        add    (%esp,1),%ebx
0x0804832d <subr+10>:       mov    %ebx,(%esp,1)
End of assembler dump.
```

Next we start the program running

```
(gdb) run
Starting program: /mnt/hgfs/vmshare/jumpjump

Breakpoint 1, 0x08048300 in main ()
```

We immediately hit the first breakpoint we set on the first instruction These are the interesting register values (others are suppressed). The EAX register will eventually be modified after the branch. The EBX regiser is used for arithmetic in the subroutine. The ESP register points to the top of the program data stack. The EIP register points to the next instruction to execute. The EFLAGS register contains the ZF flag (zero flag). We print both the eflags and the decoded ZF value when needed.

```
(gdb) info registers
eax            0x1              1
ebx            0x42130a14       1108544020
esp            0xbfffe61c       0xbfffe61c
eip            0x8048300        0x8048300          <=========
eflags         0x200246         2097734
```

Here we see that the ZF flag is true by default. We don't need to change it. Thus the subroutine will modify the EIP value on the stack.

```
(gdb) ZF
ZF is true                                         <=========
```

Now we "single instruction step" (si) to execute the `call` instruction.

```
(gdb) si
0x08048323 in subr ()                              <=========
```

We can see that the EIP register has been changed by 0x23 to point to the `jne` instruction, the first instruction in `subr`.

```
(gdb) info register
eax            0x1              1
ebx            0x42130a14       1108544020
esp            0xbfffe618       0xbfffe618
eip            0x8048323        0x8048323          <=========
eflags         0x200346         2097990
```

And we can see that the program data stack contains the "return address" on the top of the stack. This is the instruction pointer address of the first `jmp` instruction. The "ret" instruction will effectively go to this location.

```
(gdb) x 0xbfffe618
0xbfffe618:        0x08048305                      <=========
```

Now we execute the `jne` instruction. Since the ZF flag is true the `jne` instruction does not take the branch. The next instruction will be a `mov` instruction.

```
(gdb) si
0x08048325 in subr ()
```

This is the interesting case. Here we are about to do some simple arithmetic to modify the instruction pointer (EIP) value on the top of the stack.

```
(gdb) info registers
eax            0x1              1
ebx            0x42130a14       1108544020          <=========
esp            0xbfffe618       0xbfffe618
eip            0x8048325        0x8048325
eflags         0x200346         2097990
```

We execute the `mov ebx,5` instruction. This loads the length of a normal `jmp` instruction into the EBX register.

```
(gdb) si
0x0804832a in subr ()
```

And you can see that the EBX register contains that length.

```
(gdb) info registers
eax            0x1              1
ebx            0x5              5                   <=========
esp            0xbfffe618       0xbfffe618
eip            0x804832a        0x804832a
eflags         0x200346         2097990
```

Next we add the EIP value on the top of the stack to the EBX register. Note that we did not **POP** the stack, we simply copied the value so the stack pointer (ESP) register is unchanged.

```
(gdb) si
0x0804832d in subr ()
```

Here we have computed the address of the second `jmp` instruction in the `main` program.

```
(gdb) info registers
eax            0x1              1
ebx            0x804830a        134513418           <=========
esp            0xbfffe618       0xbfffe618
eip            0x804832d        0x804832d
eflags         0x200306         2097926
```

And we can see that the address of the first `jmp` instruction is still on top of the stack.

```
(gdb) x 0xbfffe618
0xbfffe618:        0x08048305                        <=========
```

Now we store the address of the second `jmp` instruction into the top of stack location. This will change the location in the main program that gets executed from the first `jmp` instruction to the second `jmp` instruction.

```
(gdb) si
0x08048330 in ret1 ()
```

We can see the registers just before the return happens.

```
(gdb) info registers
eax            0x1              1
ebx            0x804830a        134513418
esp            0xbfffe618       0xbfffe618
eip            0x8048330        0x8048330
eflags         0x200306         2097926
```

And we can see the address on the top of the stack. This is the address of the next instruction to execute after the `ret`. Note that it is the address of the second `jmp` instruction.

```
(gdb) x 0xbfffe618
0xbfffe618:        0x0804830a                    <=========
```

Now we execute the `ret` instruction and we are back in `main`. We are about to execute the second `jmp` instruction.

```
(gdb) si
0x0804830a in main ()                            <!!!!!!!!!
```

In order to show that the correct branch is actually taken we will construct a visible value in the EAX register. Note that the register has a value of 1.

```
(gdb) info registers
eax            0x1              1                <=========
ebx            0x804830a        134513418
esp            0xbfffe61c       0xbfffe61c
eip            0x804830a        0x804830a
eflags         0x200306         2097926
```

We execute the `jmp` instruction and are at the `add eax,2` instruction.

```
(gdb) si
0x08048319 in ZFis0 ()

This is the register set prior to the {\tt add} instruction.
(gdb) info registers
eax            0x1              1                <=========
ebx            0x804830a        134513418
esp            0xbfffe61c       0xbfffe61c
eip            0x8048319        0x8048319
eflags         0x200306         2097926
```

We execute the `add eax,2` instruction.

```
(gdb) si
0x0804831e in ZFis0 ()
```

And we can see the resulting value in EAX.

```
(gdb) info registers
eax            0x3              3                  <=========
ebx            0x804830a        134513418
esp            0xbfffe61c       0xbfffe61c
eip            0x804831e        0x804831e
eflags         0x200306         2097926
```

We execute the `jmp` to the end of the program and finish.

```
(gdb) si
0x08048331 in end ()
```

5 A CCA expansion without using EIP

Subroutine:

```
subr:  jne ret1
       mov ebx, 5
       add ebx, [esp]
       mov [esp], ebx
ret1:  ret
```

Structured Subroutine:

```
       if (E) then
          mov ebx, 5
          add ebx, [esp]
          mov [esp], ebx
       fi
       ret
```

Extracted Subroutine:

```
[ ZF ->
  [ M:=update_32(M,SS,ESP,add_32(5, access_32(M,SS,ESP)))
  : EBX:=add_32(5, access_32(M,SS,ESP))
  : ESP:=add_32(ESP, 4)
  : ZF:=int_equal(add_32(5, access_32(M,SS,ESP)), 0)
  : SF:=is_neg_signed_32(add_32(5, access_32(M,SS,ESP)))
  : PF:=is_even_parity_lowbyte(add_32(5, access_32(M,SS,ESP)))
  : CF:=carry_flag_add_32(5, access_32(M,SS,ESP))
  : OF:=overflow_flag_add_32(5, access_32(M,SS,ESP))
```

```
  : AF:=auxiliary_carry_flag_add(5, access_32(M,SS,ESP))
    goto access_32(
     update_32(M,SS,ESP,add_32(5, access_32(M,SS,ESP))),SS,ESP) ]
| not(ZF) ->
  [ ESP:=add_32(ESP, 4)
    goto access_32(M,SS,ESP) ]
]
do
 [ ZF ->
  [ M:=update_32(M,SS,ESP,add_32(5, access_32(M,SS,ESP)))
  : EBX:=add_32(5, access_32(M,SS,ESP))
  : ZF:=int_equal(add_32(5, access_32(M,SS,ESP)), 0)
  : SF:=is_neg_signed_32(add_32(5, access_32(M,SS,ESP)))
  : PF:=is_even_parity_lowbyte(add_32(5, access_32(M,SS,ESP)))
  : CF:=carry_flag_add_32(5, access_32(M,SS,ESP))
  : OF:=overflow_flag_add_32(5, access_32(M,SS,ESP))
  : AF:=auxiliary_carry_flag_add(5, access_32(M,SS,ESP)) ]
 | not(ZF) ->
  [ IDENTITY ]
 ]
 if (E) then
  [ M:=update_32(M,SS,ESP,add_32(5, access_32(M,SS,ESP)))
  : EBX:=add_32(5, access_32(M,SS,ESP))
  : ZF:=int_equal(add_32(5, access_32(M,SS,ESP)), 0)
  : SF:=is_neg_signed_32(add_32(5, access_32(M,SS,ESP)))
  : PF:=is_even_parity_lowbyte(add_32(5, access_32(M,SS,ESP)))
  : CF:=carry_flag_add_32(5, access_32(M,SS,ESP))
  : OF:=overflow_flag_add_32(5, access_32(M,SS,ESP))
  : AF:=auxiliary_carry_flag_add(5, access_32(M,SS,ESP)) ]
  do
   [ EBX:=5 ]
   mov ebx, 5
    [ EBX:=add_32(EBX, access_32(M,SS,ESP))
   : ZF:=int_equal(add_32(EBX, access_32(M,SS,ESP)), 0)
   : SF:=is_neg_signed_32(add_32(EBX, access_32(M,SS,ESP)))
   : PF:=
      is_even_parity_lowbyte(add_32(EBX, access_32(M,SS,ESP)))
   : CF:=carry_flag_add_32(EBX, access_32(M,SS,ESP))
   : OF:=overflow_flag_add_32(EBX, access_32(M,SS,ESP))
   : AF:=auxiliary_carry_flag_add(EBX, access_32(M,SS,ESP)) ]
   add ebx, [esp]
    [ M:=update_32(M,SS,ESP,EBX) ]
   mov [esp], ebx
  od
 fi
  [ ESP:=add_32(ESP, 4)
```

```
   goto access_32(M,SS,ESP) ]
 ret
od
```

Repository entry for subroutine (after simplification):

```
[ ZF ->
  [ M:=update_32(M,SS,ESP,add_32(5, access_32(M,SS,ESP)))
  : EBX:=add_32(5, access_32(M,SS,ESP))
  : ESP:=add_32(ESP, 4)
  : ZF:=int_equal(add_32(5, access_32(M,SS,ESP)), 0)
  : SF:=is_neg_signed_32(add_32(5, access_32(M,SS,ESP)))
  : PF:=
     is_even_parity_lowbyte(add_32(5, access_32(M,SS,ESP)))
  : CF:=carry_flag_add_32(5, access_32(M,SS,ESP))
  : OF:=overflow_flag_add_32(5, access_32(M,SS,ESP))
  : AF:=auxiliary_carry_flag_add(5, access_32(M,SS,ESP))
    goto add_32(5, access_32(M,SS,ESP)) ]
| not(ZF) ->
  [ ESP:=add_32(ESP, 4)
    goto access_32(M,SS,ESP) ]
]
```

Call Instruction:

```
   loc:   call subr
   next:
```

Call Instruction Semantics (before repository lookup):

```
  [ M:=update_32(M,SS,add_32(ESP,-4),NEXT)
  : ESP:=add_32(ESP,-4)
    goto SUBR ]
```

Call Instruction Semantics (after composing with repository entry):

```
  // Simplified:
  [ ZF ->
     [ M:=update_32(M,SS,add_32(ESP,-4),add_32(5, NEXT))
     : EBX:=add_32(5, NEXT)
     : ZF:=int_equal(add_32(5, NEXT), 0)
     : SF:=is_neg_signed_32(add_32(5, NEXT))
     : PF:=is_even_parity_lowbyte(add_32(5, NEXT))
     : CF:=carry_flag_add_32(5, NEXT)
     : OF:=overflow_flag_add_32(5, NEXT)
     : AF:=auxiliary_carry_flag_add(5, NEXT)
       goto add_32(5, NEXT) ]
  | not(ZF) ->
     [ IDENTITY
```

```
      goto NEXT ]
  ]
```

In unsimplified form:

```
[ ZF ->
  [ M:=
    update_32(
     update_32(M,SS,add_32(ESP,-4),NEXT),
     SS,
     add_32(ESP,-4),
     add_32(5,
      access_32(
       update_32(M,SS,add_32(ESP,-4),NEXT)
      ,SS,add_32(ESP,-4))))
  : EBX:=
     add_32(5,
      access_32(
       update_32(M,SS,add_32(ESP,-4),NEXT),
       SS,add_32(ESP,-4)))
  : ESP:=add_32(add_32(ESP,-4), 4)
  : ZF:=
     int_equal(
      add_32(5,
       access_32(
        update_32(M,SS,add_32(ESP,-4),NEXT),
        SS,add_32(ESP,-4))), 0)
  : SF:=
    is_neg_signed_32(
     add_32(5,
      access_32(
       update_32(M,SS,add_32(ESP,-4),NEXT),
       SS,add_32(ESP,-4))))
  : PF:=
     is_even_parity_lowbyte(
      add_32(5,
       access_32(
        update_32(M,SS,add_32(ESP,-4),NEXT),
        SS,add_32(ESP,-4))))
  : CF:=
     carry_flag_add_32(5,
      access_32(
       update_32(M,SS,add_32(ESP,-4),NEXT),
       SS,add_32(ESP,-4)))
  : OF:=
     overflow_flag_add_32(5,
      access_32(
```

```
       update_32(M,SS,add_32(ESP,-4),NEXT),
       SS,add_32(ESP,-4)))
  : AF:=
     auxiliary_carry_flag_add(5,
      access_32(
       update_32(M,SS,add_32(ESP,-4),NEXT),
       SS,add_32(ESP,-4)))
    goto add_32(5,
     access_32(
      update_32(M,SS,add_32(ESP,-4),NEXT),
      SS,add_32(ESP,-4))) ]
| not(ZF) ->
  [ ESP:=add_32(add_32(ESP,-4), 4)
    goto
     access_32(
      update_32(M,SS,add_32(ESP,-4),NEXT),
      SS,add_32(ESP,-4)) ]
]
 do
   [ M:=update_32(M,SS,add_32(ESP,-4),NEXT)
   : ESP:=add_32(ESP,-4) ]
    [ ZF ->
     [ M:=update_32(M,SS,ESP,add_32(5, access_32(M,SS,ESP)))
     : EBX:=add_32(5, access_32(M,SS,ESP))
     : ESP:=add_32(ESP, 4)
     : ZF:=int_equal(add_32(5, access_32(M,SS,ESP)), 0)
     : SF:=is_neg_signed_32(add_32(5, access_32(M,SS,ESP)))
     : PF:=
        is_even_parity_lowbyte(add_32(5, access_32(M,SS,ESP)))
     : CF:=carry_flag_add_32(5, access_32(M,SS,ESP))
     : OF:=overflow_flag_add_32(5, access_32(M,SS,ESP))
     : AF:=auxiliary_carry_flag_add(5, access_32(M,SS,ESP))
       goto add_32(5, access_32(M,SS,ESP)) ]
   | not(ZF) ->
     [ ESP:=add_32(ESP, 4)
       goto access_32(M,SS,ESP) ]
   ]
od
```

Main Routine Processing:

Call instruction replaced with simplified composed semantics.

Based on substituted semantics, that call instruction is flowcharted as a one-in, two-out node with functional effects:

- First out address is to add_32(5, NEXT)
- Second out address is to NEXT

6 Adding EIP to the state space is a problem

Sequence of three simple instructions:

```
mov ebx, 5
add ebx, [esp]
mov [esp], ebx
```

Current style CCA's and extraction:

```
[ M:=update_32(M,SS,ESP,add_32(5, access_32(M,SS,ESP)))
: EBX:=add_32(5, access_32(M,SS,ESP))
: ZF:=int_equal(add_32(5, access_32(M,SS,ESP)), 0)
: SF:=is_neg_signed_32(add_32(5, access_32(M,SS,ESP)))
: PF:=
   is_even_parity_lowbyte(add_32(5, access_32(M,SS,ESP)))
: CF:=carry_flag_add_32(5, access_32(M,SS,ESP))
: OF:=overflow_flag_add_32(5, access_32(M,SS,ESP))
: AF:=auxiliary_carry_flag_add(5, access_32(M,SS,ESP)) ]
do
 [ EBX:=5 ]
 mov ebx, 5
  [ EBX:=add_32(EBX, access_32(M,SS,ESP))
 : ZF:=int_equal(add_32(EBX, access_32(M,SS,ESP)), 0)
 : SF:=is_neg_signed_32(add_32(EBX, access_32(M,SS,ESP)))
 : PF:=
    is_even_parity_lowbyte(add_32(EBX, access_32(M,SS,ESP)))
 : CF:=carry_flag_add_32(EBX, access_32(M,SS,ESP))
 : OF:=overflow_flag_add_32(EBX, access_32(M,SS,ESP))
 : AF:=auxiliary_carry_flag_add(EBX, access_32(M,SS,ESP)) ]
 add ebx, [esp]
  [ M:=update_32(M,SS,ESP,EBX) ]
 mov [esp], ebx
od
```

Mathematical justification for correctness:

For any initial state (i.e., any initial values of the state variables), the effect of executing the three-instruction sequence is given by the right-hand side of the extracted function.

Extraction after adding EIP to state space:

```
[ M:=update_32(M,SS,ESP,add_32(5, access_32(M,SS,ESP)))
: EBX:=add_32(5, access_32(M,SS,ESP))
: ZF:=int_equal(add_32(5, access_32(M,SS,ESP)), 0)
: SF:=is_neg_signed_32(add_32(5, access_32(M,SS,ESP)))
: PF:=
   is_even_parity_lowbyte(add_32(5, access_32(M,SS,ESP)))
```

```
: CF:=carry_flag_add_32(5, access_32(M,SS,ESP))
: OF:=overflow_flag_add_32(5, access_32(M,SS,ESP))
: AF:=auxiliary_carry_flag_add(5, access_32(M,SS,ESP))
: EIP:=add_32(EIP,11) ]
do
   [ EBX:=5
   : EIP:=add_32(EIP,5) ]
   mov ebx, 5
    [ EBX:=add_32(EBX, access_32(M,SS,ESP))
   : ZF:=int_equal(add_32(EBX, access_32(M,SS,ESP)), 0)
   : SF:=is_neg_signed_32(add_32(EBX, access_32(M,SS,ESP)))
   : PF:=
      is_even_parity_lowbyte(add_32(EBX, access_32(M,SS,ESP)))
   : CF:=carry_flag_add_32(EBX, access_32(M,SS,ESP))
   : OF:=overflow_flag_add_32(EBX, access_32(M,SS,ESP))
   : AF:=auxiliary_carry_flag_add(EBX, access_32(M,SS,ESP))
   : EIP:=add_32(EIP,3) ]
   add ebx, [esp]
    [ M:=update_32(M,SS,ESP,EBX)
   : EIP:=add_32(EIP,3) ]
   mov [esp], ebx
od
```

Mathematical problem with correctness:

Based on general correctness principles, should be able to plug in any value for initial state, which now includes EIP. However, upon using any initial value for EIP other then the address of the first instruction, this gives the wrong answer, because a completely different sequence of instructions will be executed if EIP is set to a different value.

References

[1] Mills, Harlan