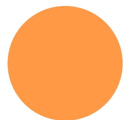
Robert Mireles

CS 6463

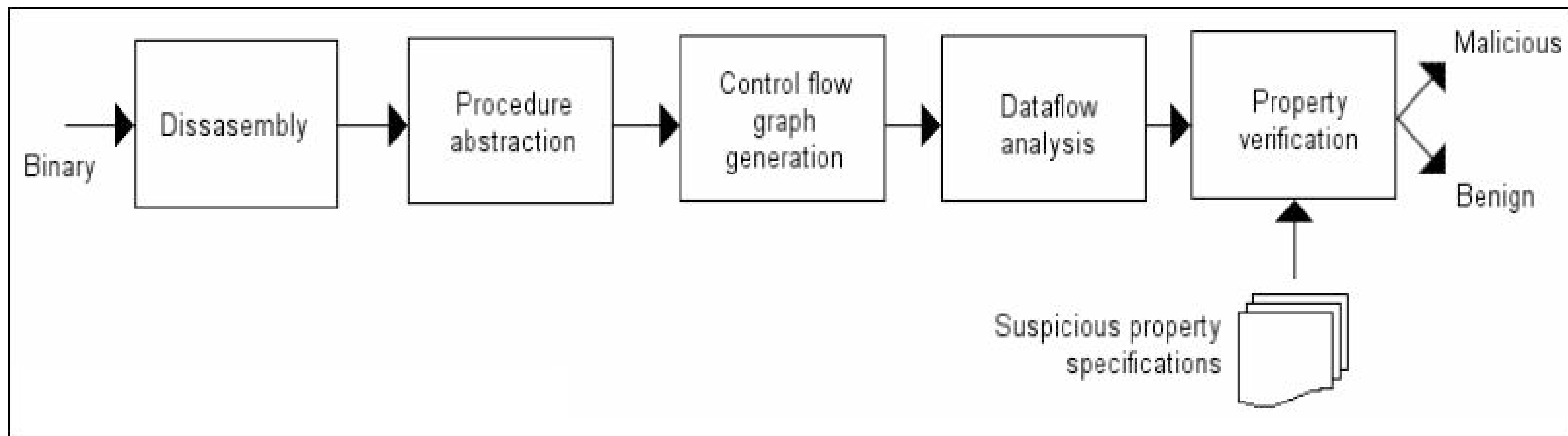
Table Of Contents

- Malware Analysis
- Obfuscating Software
- Normal Function Call
- Obfuscated Function Call
- De-obfuscating an Obfuscated Function Call

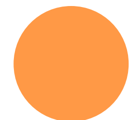


Malware Analysis

- Malware Analysis attempts to detect and prevent malicious code (e.g., viruses, worms, Trojan horses) from executing on the local system.



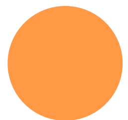
Stages in Static Analysis of Binary



Obfuscating Software

- Software obfuscation is a transformation applied to a binary in order to make it more difficult to understand the executable while preserving the program's functionality.

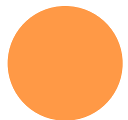
- Primary goal of obfuscation is to increase the effort involved in manually or automatically analyzing a program.
  - Escape detection by automated malware analysis
  - Significantly delay detection by manual analysis



Obfuscating Software

- Function Call Obfuscations attempt to hide the function calls in a program.

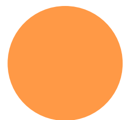
1. Analysis is more difficult if you cannot locate the functions in the program. You lose procedural abstraction.
2. Hide Operating System calls (API calls). Malware analyses rely on OS calls to see what the program is doing.



Normal Function Call

- Assembly Code

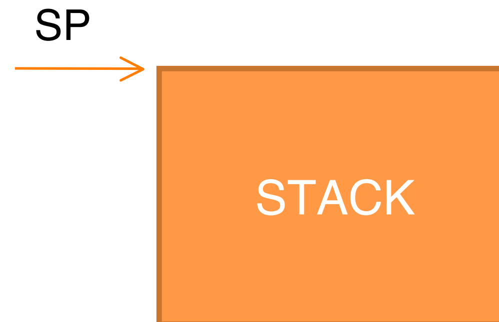
```
addr1:  push   ecx
addr2:  push   eax
addr3:  call   foo
addr4:  add    eax,ebx,eax


foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

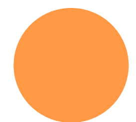


Normal Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  call   foo
addr4:  add    eax,ebx,eax


foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

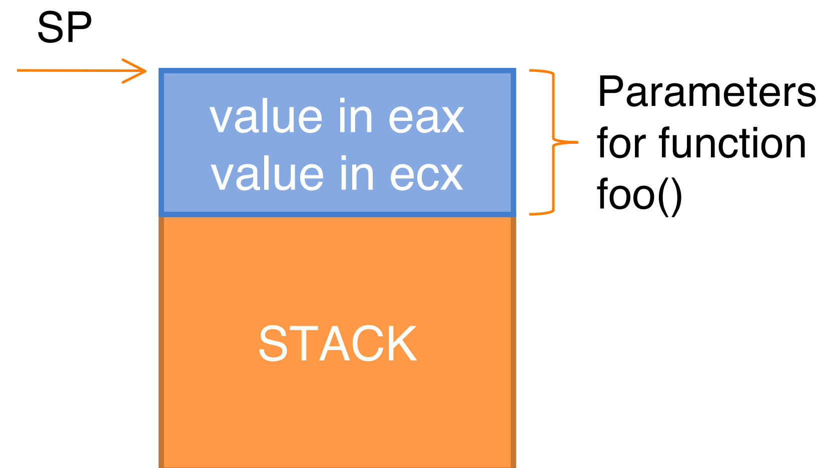




Normal Function Call

- Assembly Code

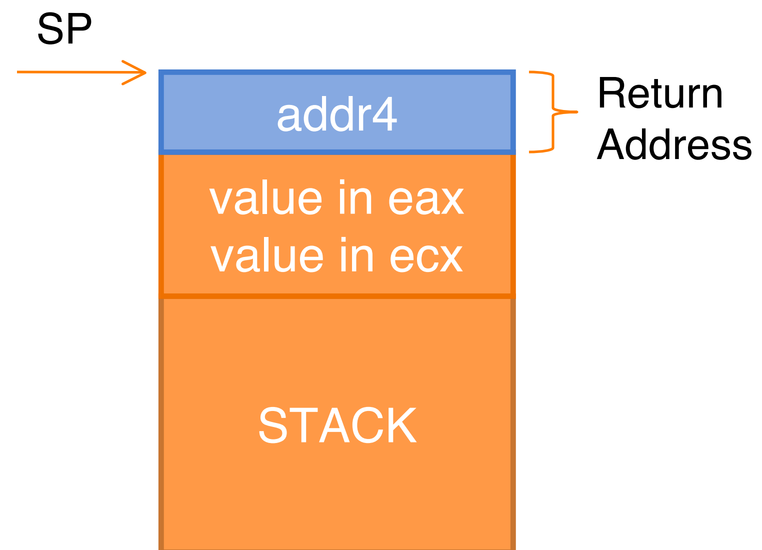
```
addr1:  push   ecx
addr2:  push   eax
addr3:  call   foo
addr4:  add    eax,ebx,eax


foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

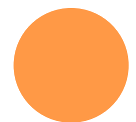


Normal Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  call   foo
addr4:  add    eax,ebx,eax


foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

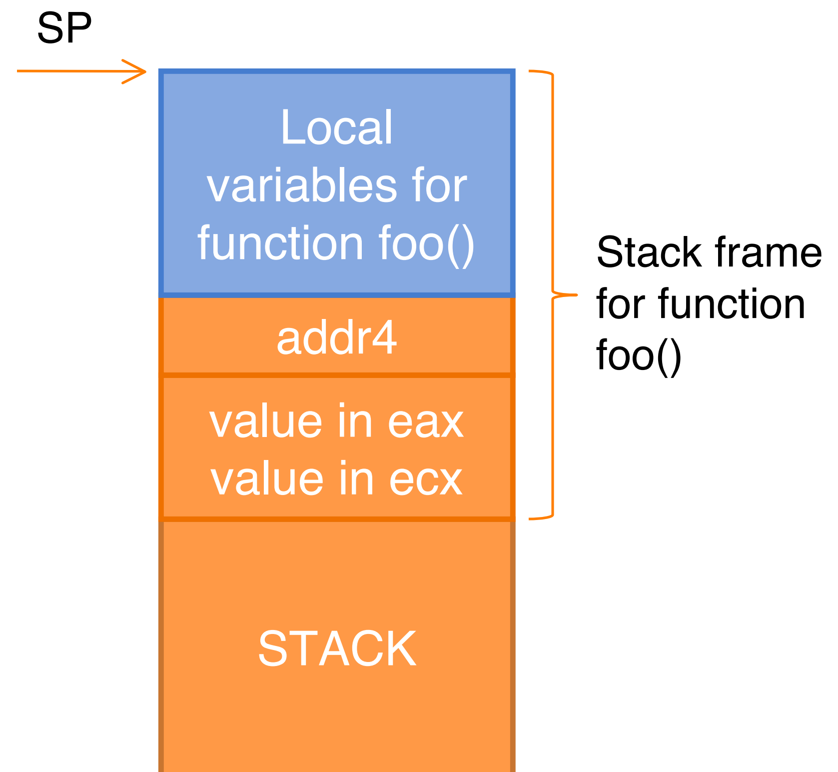




Normal Function Call

- Assembly Code

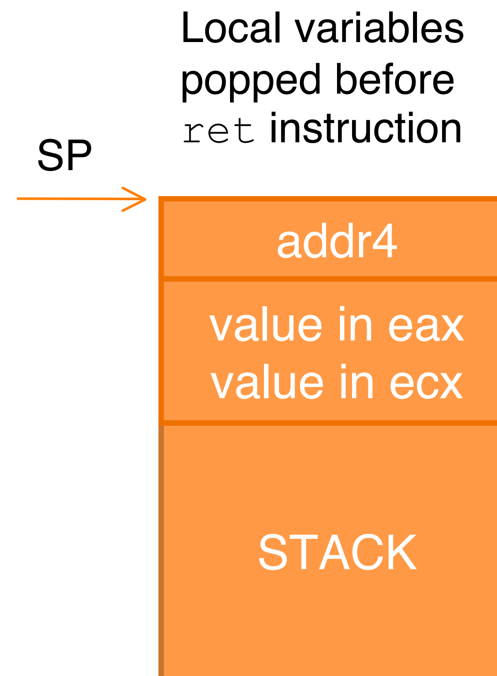
```
addr1:  push   ecx
addr2:  push   eax
addr3:  call   foo
addr4:  add    eax,ebx,eax


foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

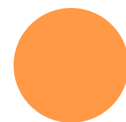


Normal Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  call   foo
addr4:  add    eax,ebx,eax


foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

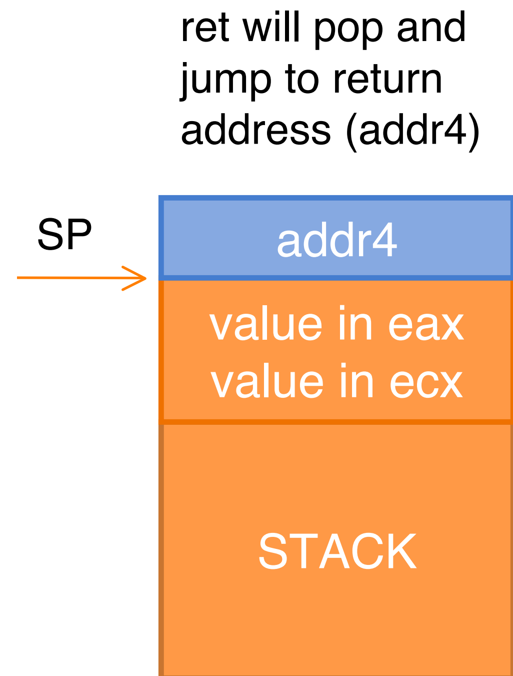


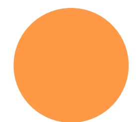


Normal Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  call   foo
addr4:  add    eax,ebx,eax


foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

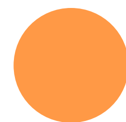




Obfuscated Function Call

- Assembly Code

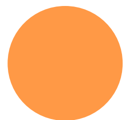
```
addr1:  push   ecx
addr2:  push   eax
addr3:  call   foo
addr4:  add    eax,ebx,eax


foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```



Obfuscated Function Call

- Assembly Code

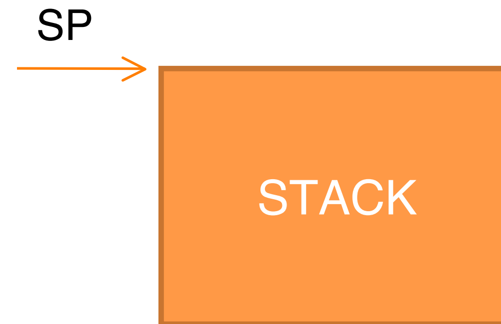
```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr5    // push return address
addr4:  jmp    foo      // manually jump to foo()
addr5:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

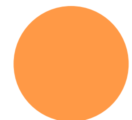


Obfuscated Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr5
Addr4:  jmp    foo
addr5:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

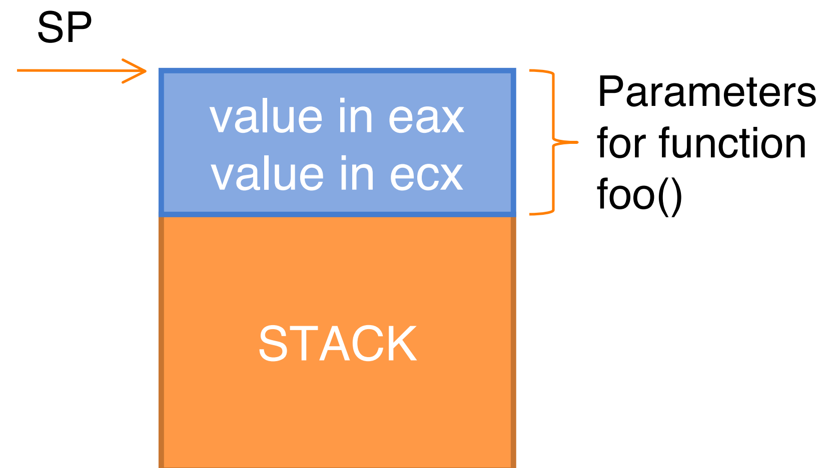


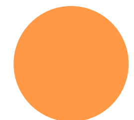


Obfuscated Function Call

- Assembly Code

```
addr1:  push    ecx
addr2:  push    eax
addr3:  push    addr5
Addr4:  jmp     foo
addr5:  add     eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

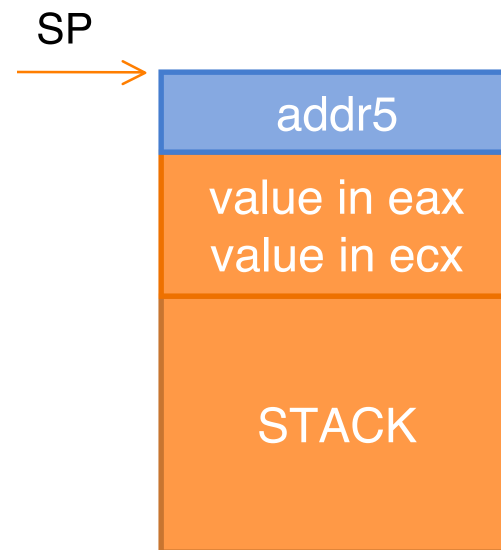


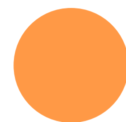


Obfuscated Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr5
addr4:  jmp    foo
addr5:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

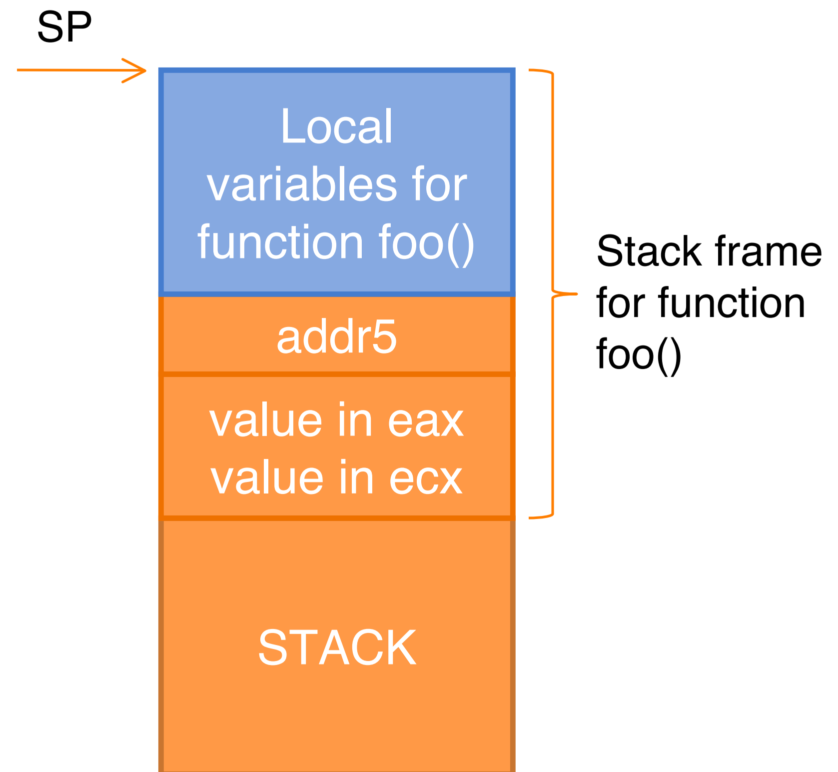


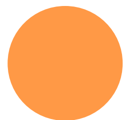


Obfuscated Function Call

- Assembly Code

```
addr1:  push    ecx
addr2:  push    eax
addr3:  push    addr5
addr4:  jmp     foo
addr5:  add     eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

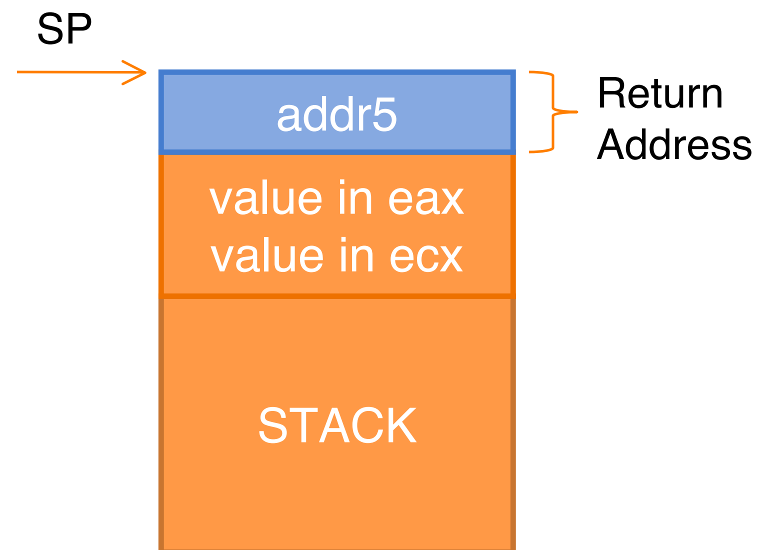


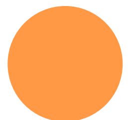


Obfuscated Function Call

- Assembly Code

```
addr1:  push    ecx
addr2:  push    eax
addr3:  push    addr5
addr4:  jmp     foo
addr5:  add     eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

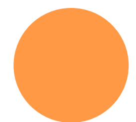




Obfuscated Function Call

- Assembly Code

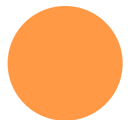
```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr5
addr4:  jmp    foo
addr5:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```



Obfuscated Function Call

- Assembly Code

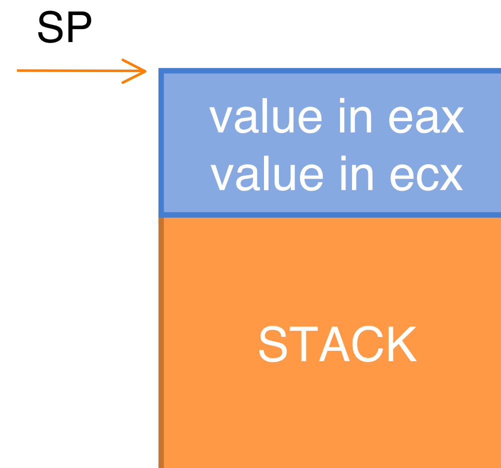
```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr6
addr4:  push   foo
addr5:  ret
addr6:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

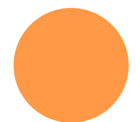


Obfuscated Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr6
addr4:  push   foo
addr5:  ret
addr6:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

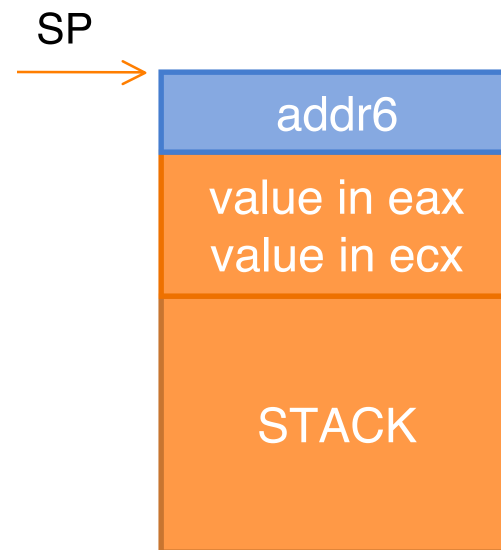


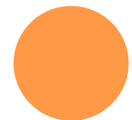


Obfuscated Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr6
addr4:  push   foo
addr5:  ret
addr6:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

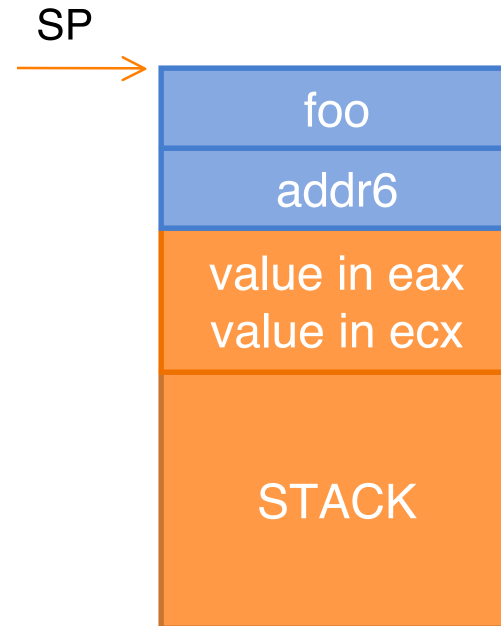


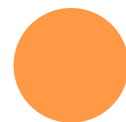


Obfuscated Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr6
addr4:  push   foo
addr5:  ret
addr6:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```



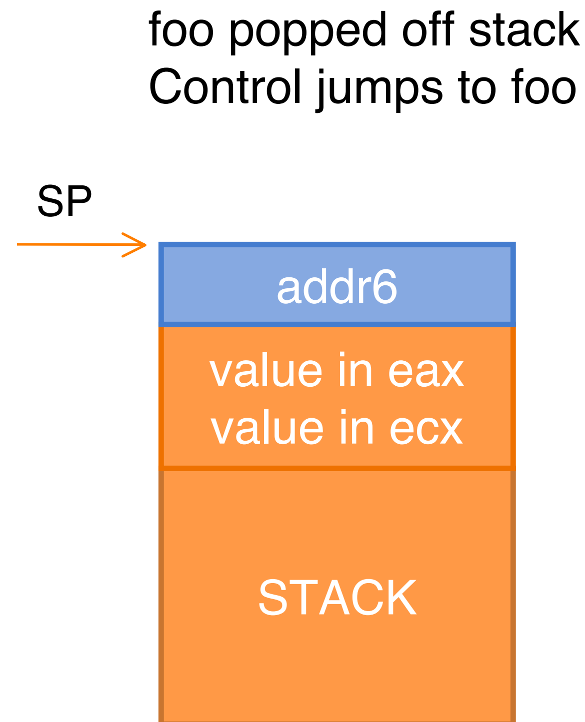


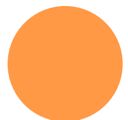
Obfuscated Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr6
addr4:  push   foo
addr5:  ret
addr6:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

foo popped off stack
Control jumps to foo

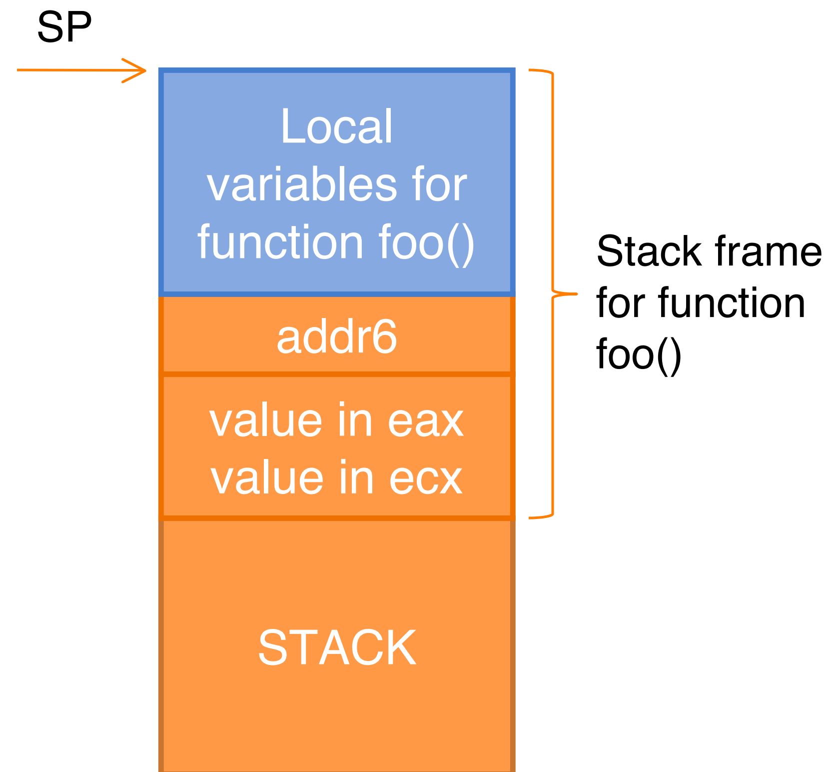


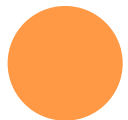


Obfuscated Function Call

- Assembly Code

```
addr1:  push   ecx
addr2:  push   eax
addr3:  push   addr6
addr4:  push   foo
addr5:  ret
addr6:  add    eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```



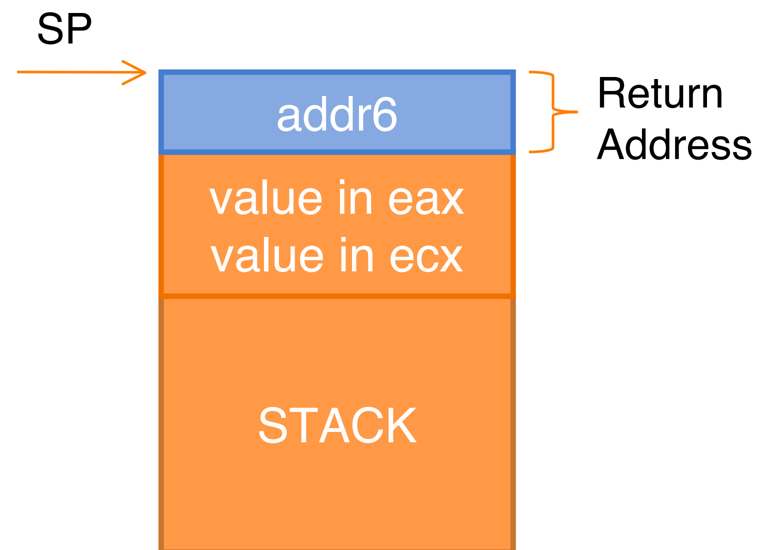


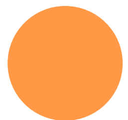
Obfuscated Function Call

- Assembly Code

```
addr1:  push    ecx
addr2:  push    eax
addr3:  push    addr6
addr4:  push    foo
addr5:  ret
addr6:  add     eax,ebx,eax

foo(int a, int b)
foo:    <foo instructions>
...
foo+x:  ret
```

Same stack as when using the `call` instruction





Obfuscated Function Call

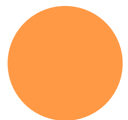
Replace:

```
call    foo
```

With:

```
push    fooRetAddr
push    foo
ret
```

These are semantically equivalent!



Obfuscated Function Call

- Obfuscation can be further enhanced by:

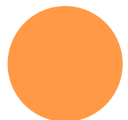
1. Spreading out the three instructions

2. Dynamically calculate function address

   Use Windows API function `GetProcAddress()`

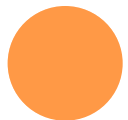
   Replace:
   ```
   push  foo
   ret
   ```

   With:
   ```
   call  GetProcAddress("foo")
   push  eax
   ret
   ```



De-Obfuscating An Obfuscated Function Call

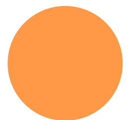
- Abstract Stack
- Abstract Stack Graph



Abstract Stack

- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

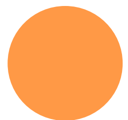


Abstract Stack

- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

- The actual stack keeps track of data in a LIFO order.



Abstract Stack

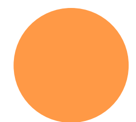
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

Actual Stack

SP

- The actual stack keeps track of data in a LIFO order.

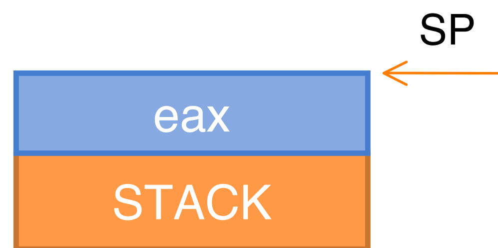


Abstract Stack

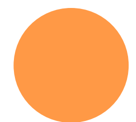
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

Actual Stack



- The actual stack keeps track of data in a LIFO order.

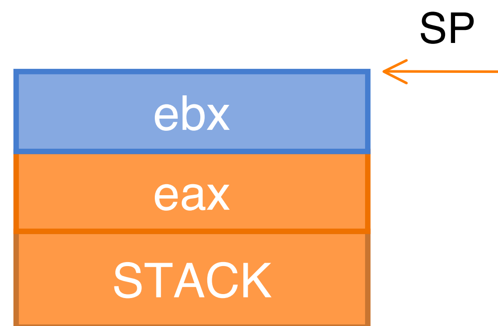


Abstract Stack

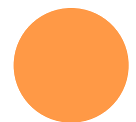
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

Actual Stack



- The actual stack keeps track of data in a LIFO order.

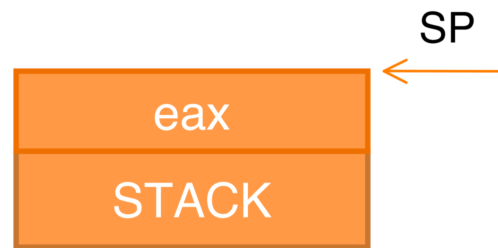


Abstract Stack

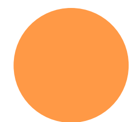
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

Actual Stack



- The actual stack keeps track of data in a LIFO order.

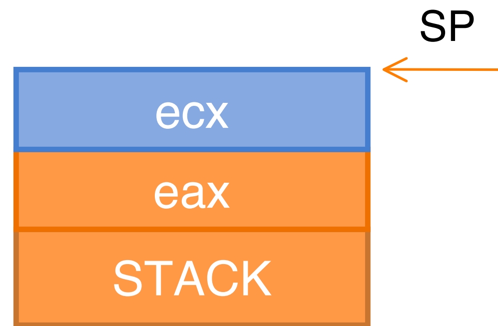


Abstract Stack

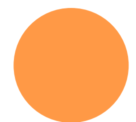
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

Actual Stack



- The actual stack keeps track of data in a LIFO order.

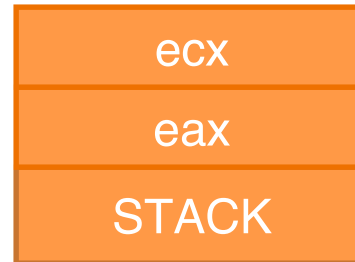


Abstract Stack

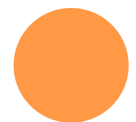
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

Actual Stack



- The actual stack keeps track of data in a LIFO order.
- The abstract stack stores the addresses of the instructions that push and pop values in a LIFO order.

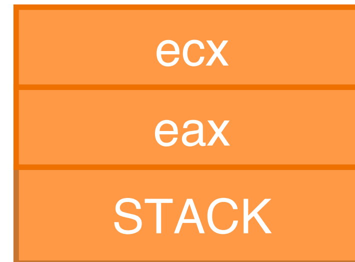


Abstract Stack

- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

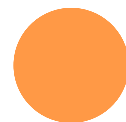
Actual Stack



SP

Abstract Stack

- The actual stack keeps track of data in a LIFO order.
- The abstract stack stores the addresses of the instructions that push and pop values in a LIFO order.

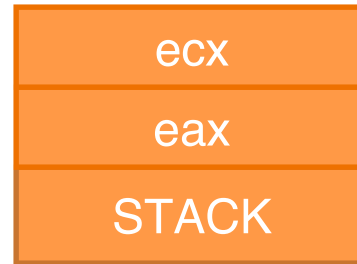


Abstract Stack

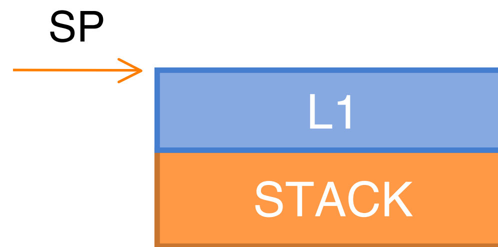
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

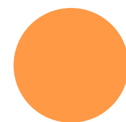
Actual Stack



Abstract Stack



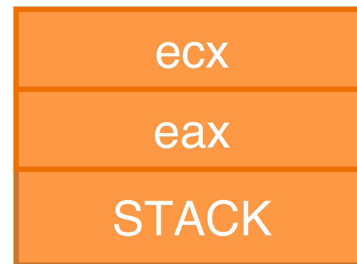
- The actual stack keeps track of data in a LIFO order.
- The abstract stack stores the addresses of the instructions that push and pop values in a LIFO order.

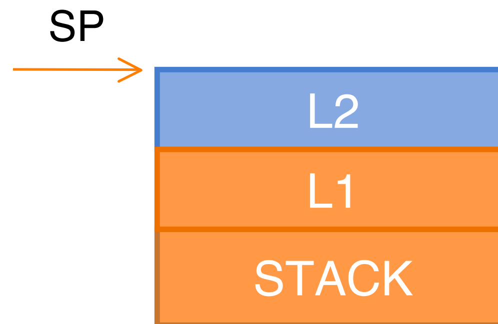


Abstract Stack

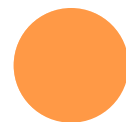
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```





- The actual stack keeps track of data in a LIFO order.
- The abstract stack stores the addresses of the instructions that push and pop values in a LIFO order.

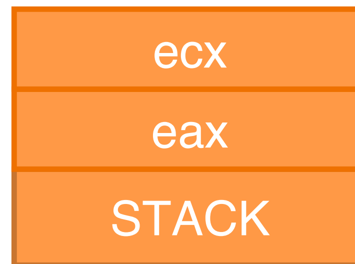


Abstract Stack

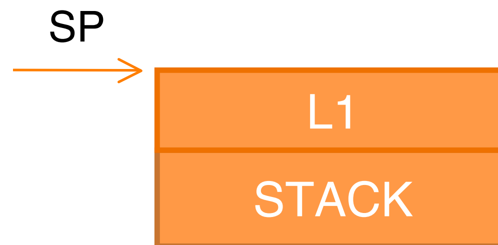
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

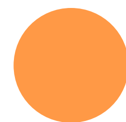
Actual Stack



Abstract Stack



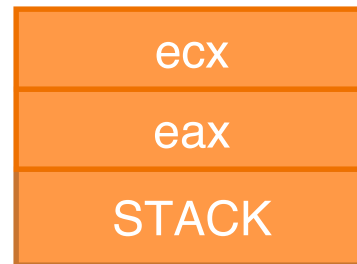
- The actual stack keeps track of data in a LIFO order.
- The abstract stack stores the addresses of the instructions that push and pop values in a LIFO order.

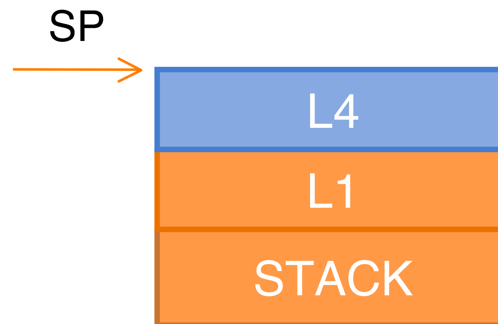


Abstract Stack

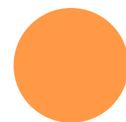
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```





- The actual stack keeps track of data in a LIFO order.
- The abstract stack stores the addresses of the instructions that push and pop values in a LIFO order.

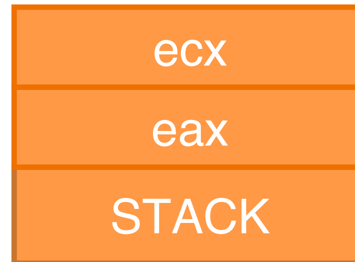


Abstract Stack

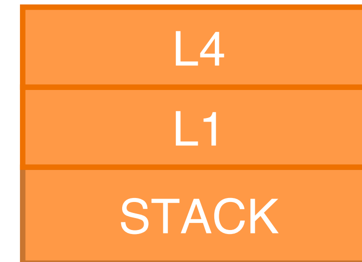
- Sample program

```
L1: push  eax
L2: push  ebx
L3: pop   edx
L4: push  ecx
```

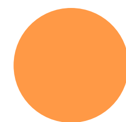
Actual Stack



Abstract Stack



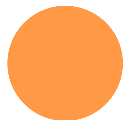
- The actual stack keeps track of data in a LIFO order.
- The abstract stack stores the addresses of the instructions that push and pop values in a LIFO order.



Abstract Stack Graph

- Step 1 – Generate a graph similar to the Control Flow Graph (CFG) but with the following difference, only the first instruction of a basic block can modify the stack pointer.

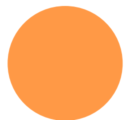
1. For each basic block you can only have a single push, pop, ret, call, or other instruction that modifies the stack pointer.
2. The first instruction of each basic block must be a push, pop, ret, call, or other instruction that modifies the stack pointer.
3. You can have branches into and out of the middle of a basic block.



Abstract Stack Graph

- Sample program

```
E:   // Entry point
B0:  push  eax
B1:  sub   ecx, 1h
B2:  beqz  B8
B3:  push  ebx
B4:  push  ecx
B5:  dec   ecx
B6:  beqz  B3
B7:  jmp   B10
B8:  pop   ebx
B9:  push  esi
B10: pop   edx
B11: beq   B0
B12: call  abc
```

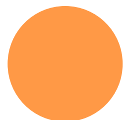


Abstract Stack Graph

- Sample program

```
E:    // Entry point
B0:   push  eax
B1:   sub   ecx, 1h
B2:   beqz  B8
B3:   push  ebx
B4:   push  ecx
B5:   dec   ecx
B6:   beqz  B3
B7:   jmp   B10
B8:   pop   ebx
B9:   push  esi
B10:  pop   edx
B11:  beq   B0
B12:  call  abc
```

```
E: // Entry
```



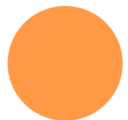
Abstract Stack Graph

- Sample program

```
E:    // Entry point
B0:   push  eax
B1:   sub   ecx, 1h
B2:   beqz  B8
B3:   push  ebx
B4:   push  ecx
B5:   dec   ecx
B6:   beqz  B3
B7:   jmp   B10
B8:   pop   ebx
B9:   push  esi
B10:  pop   edx
B11:  beq   B0
B12:  call  abc
```

```
E: // Entry
```

```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```



Abstract Stack Graph

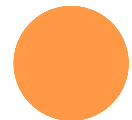
- Sample program

```
E:    // Entry point
B0:   push   eax
B1:   sub    ecx, 1h
B2:   beqz   B8
B3:   push   ebx
B4:   push   ecx
B5:   dec    ecx
B6:   beqz   B3
B7:   jmp    B10
B8:   pop    ebx
B9:   push   esi
B10:  pop    edx
B11:  beq    B0
B12:  call   abc
```

```
E: // Entry
```

```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```

```
B3: push ebx
```



Abstract Stack Graph

- Sample program

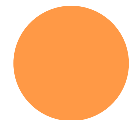
```
E:    // Entry point
B0:   push   eax
B1:   sub    ecx, 1h
B2:   beqz   B8
B3:   push   ebx
B4:   push   ecx
B5:   dec    ecx
B6:   beqz   B3
B7:   jmp    B10
B8:   pop    ebx
B9:   push   esi
B10:  pop    edx
B11:  beq    B0
B12:  call   abc
```

```
E: // Entry
```

```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```

```
B3: push ebx
```

```
B4: push ecx
B5: dec  ecx
B6: beqz B3
B7: jmp  B10
```



Abstract Stack Graph

- Sample program

```
E:    // Entry point
B0:   push   eax
B1:   sub    ecx, 1h
B2:   beqz   B8
B3:   push   ebx
B4:   push   ecx
B5:   dec    ecx
B6:   beqz   B3
B7:   jmp    B10
B8:   pop    ebx
B9:   push   esi
B10:  pop    edx
B11:  beq    B0
B12:  call   abc
```

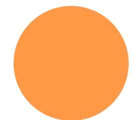
```
E: // Entry
```

```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```

```
B3: push ebx
```

```
B8: pop  ebx
```

```
B4: push ecx
B5: dec  ecx
B6: beqz B3
B7: jmp  B10
```



Abstract Stack Graph

- Sample program

```
E:    // Entry point
B0:   push   eax
B1:   sub    ecx, 1h
B2:   beqz   B8
B3:   push   ebx
B4:   push   ecx
B5:   dec    ecx
B6:   beqz   B3
B7:   jmp    B10
B8:   pop    ebx
B9:   push   esi
B10:  pop    edx
B11:  beq    B0
B12:  call   abc
```

```
E: // Entry
```

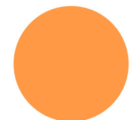
```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```

```
B3: push ebx
```

```
B8: pop  ebx
```

```
B9: push esi
```

```
B4: push ecx
B5: dec  ecx
B6: beqz B3
B7: jmp  B10
```



Abstract Stack Graph

- Sample program

```
E:    // Entry point
B0:   push   eax
B1:   sub    ecx, 1h
B2:   beqz   B8
B3:   push   ebx
B4:   push   ecx
B5:   dec    ecx
B6:   beqz   B3
B7:   jmp    B10
B8:   pop    ebx
B9:   push   esi
B10:  pop    edx
B11:  beq    B0
B12:  call   abc
```

```
E: // Entry
```

```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```

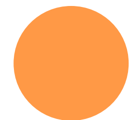
```
B3: push ebx
```

```
B8: pop  ebx
```

```
B4: push ecx
B5: dec  ecx
B6: beqz B3
B7: jmp  B10
```

```
B9: push esi
```

```
B10: pop edx
B11: beq B0
```



Abstract Stack Graph

- Sample program

```
E:    // Entry point
B0:   push   eax
B1:   sub    ecx, 1h
B2:   beqz   B8
B3:   push   ebx
B4:   push   ecx
B5:   dec    ecx
B6:   beqz   B3
B7:   jmp    B10
B8:   pop    ebx
B9:   push   esi
B10:  pop    edx
B11:  beq    B0
B12:  call   abc
```

```
E: // Entry
```

```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```

```
B3: push ebx
```

```
B4: push ecx
B5: dec  ecx
B6: beqz B3
B7: jmp  B10
```

```
B8: pop  ebx
```

```
B9: push esi
```

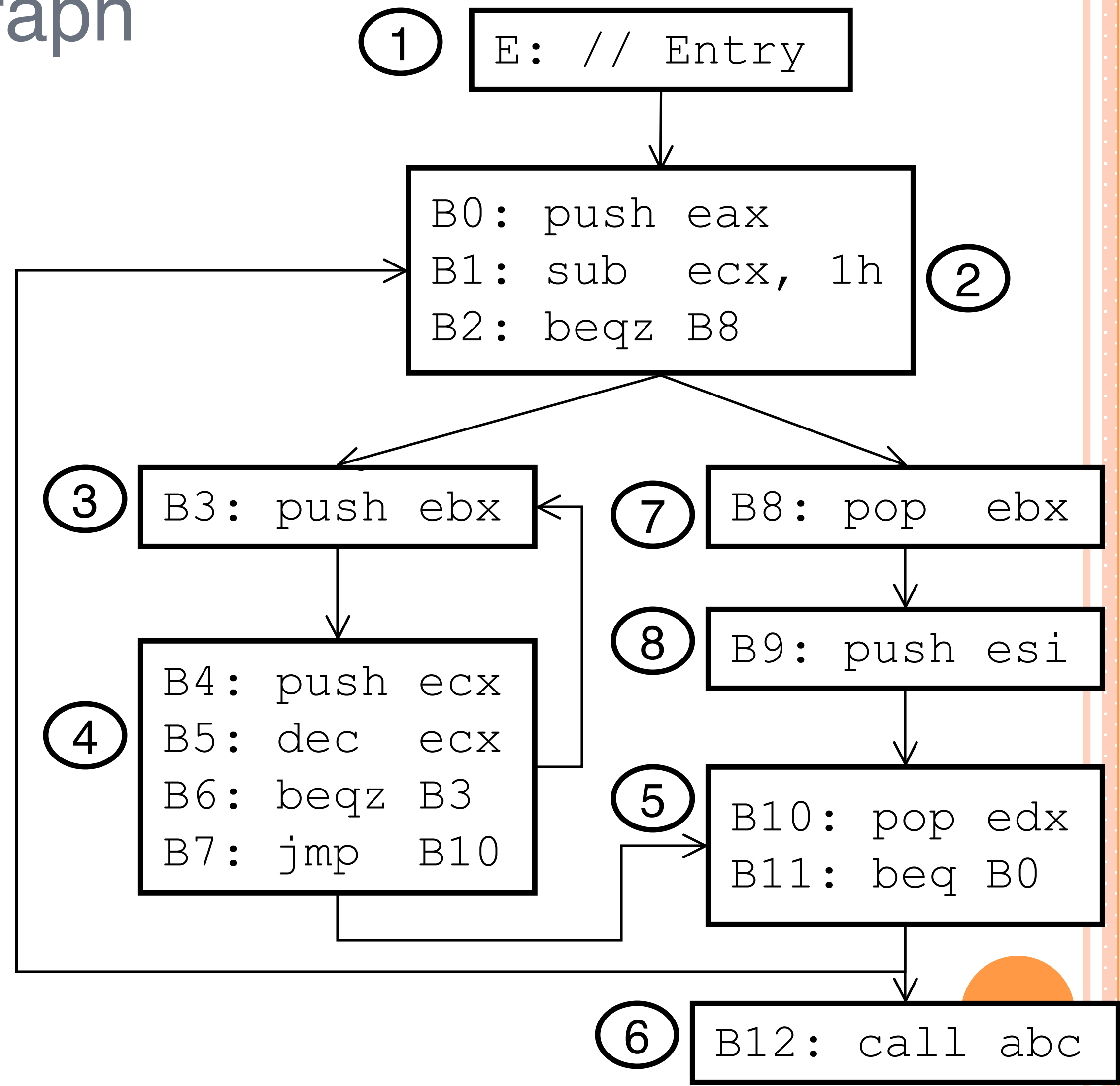
```
B10: pop edx
B11: beq B0
```

```
B12: call abc
```

Abstract Stack Graph

- Sample program

```
E:    // Entry point
B0:   push  eax
B1:   sub   ecx, 1h
B2:   beqz  B8
B3:   push  ebx
B4:   push  ecx
B5:   dec   ecx
B6:   beqz  B3
B7:   jmp   B10
B8:   pop   ebx
B9:   push  esi
B10:  pop   edx
B11:  beq   B0
B12:  call  abc
```



Abstract Stack Graph

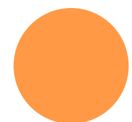
- Step 2 – Generate an Abstract Stack Graph from the basic blocks generated in the modified CFG.
- An Abstract Stack Graph (ASG) is a 3-tuple (N, AE, ASPR) where:

  Let ADDR = set of all addresses in the program.

  N _ ADDR is the set of nodes in the ASG, where an address n _ N if n pushes a value onto the stack.

  AE _ ADDR _ ADDR is the set of edges in the ASG, where an edge (n _ m) in AE instruction at n pushes a value on top of a value pushed at instruction m.

  ASPR _ ADDR _ ADDR captures the set of abstract stack pointers (stack tops) for each statement. (x, n) in ASPR program point x receives top of stack from push at instruction n.



Abstract Stack Graph

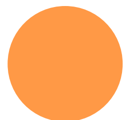
- Step 2 – Generate an Abstract Stack Graph from the basic blocks generated in the modified CFG.
- An Abstract Stack Graph (ASG) is a 3-tuple (N, AE, ASPR) where:

  Let ADDR = set of all addresses in the program.

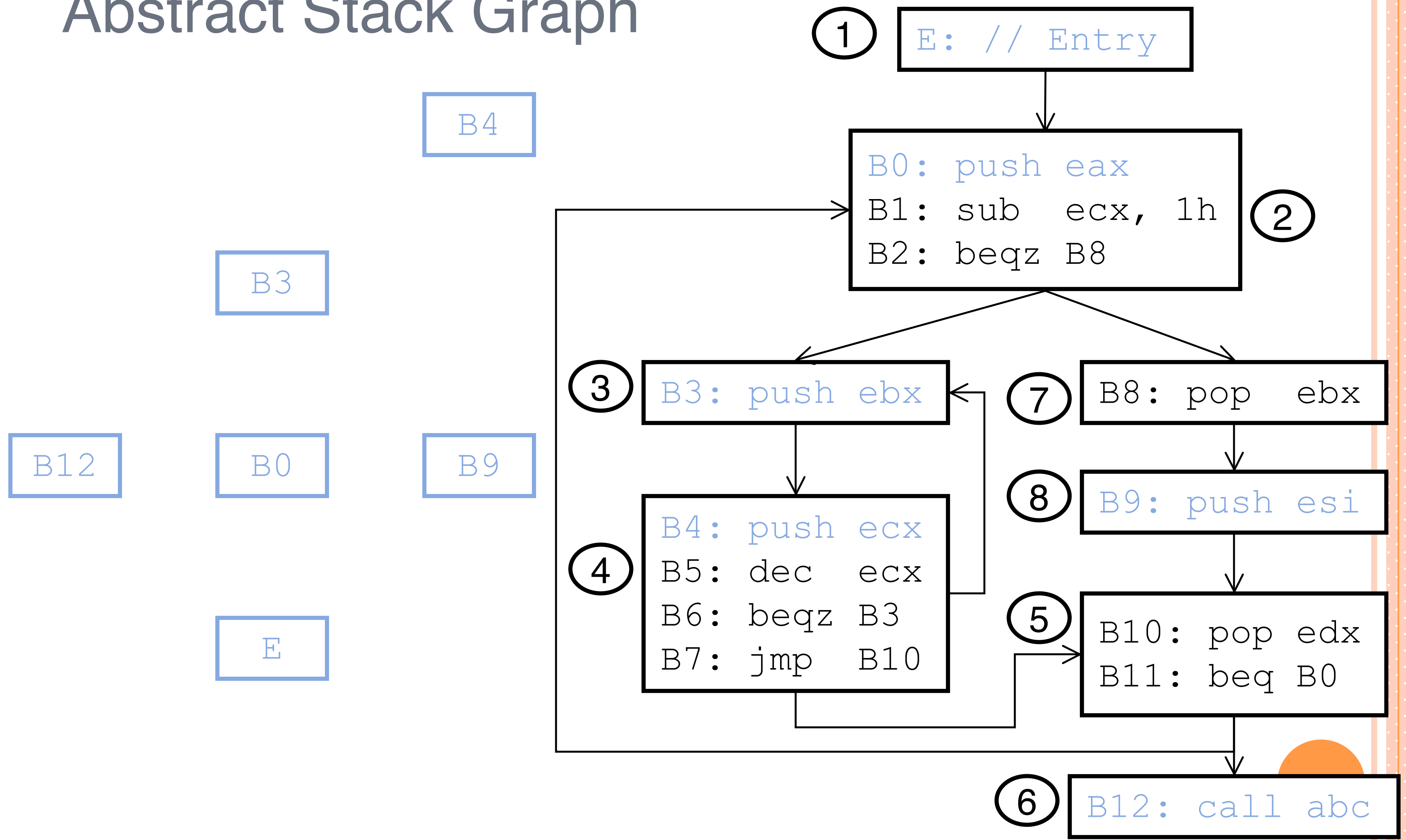
  N _ ADDR is the set of nodes in the ASG, where an address n _ N if n pushes a value onto the stack.

  AE _ ADDR _ ADDR is the set of edges in the ASG, where an edge (n _ m) in AE instruction at n pushes a value on top of a value pushed at instruction m.

  ASPR _ ADDR _ ADDR captures the set of abstract stack pointers (stack tops) for each statement. (x, n) in ASPR program point x receives top of stack from push at instruction n.



Abstract Stack Graph



Abstract Stack Graph

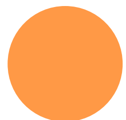
- Step 2 – Generate an Abstract Stack Graph from the basic blocks generated in the modified CFG.
- An Abstract Stack Graph (ASG) is a 3-tuple (N, AE, ASPR) where:

  Let ADDR = set of all addresses in the program.

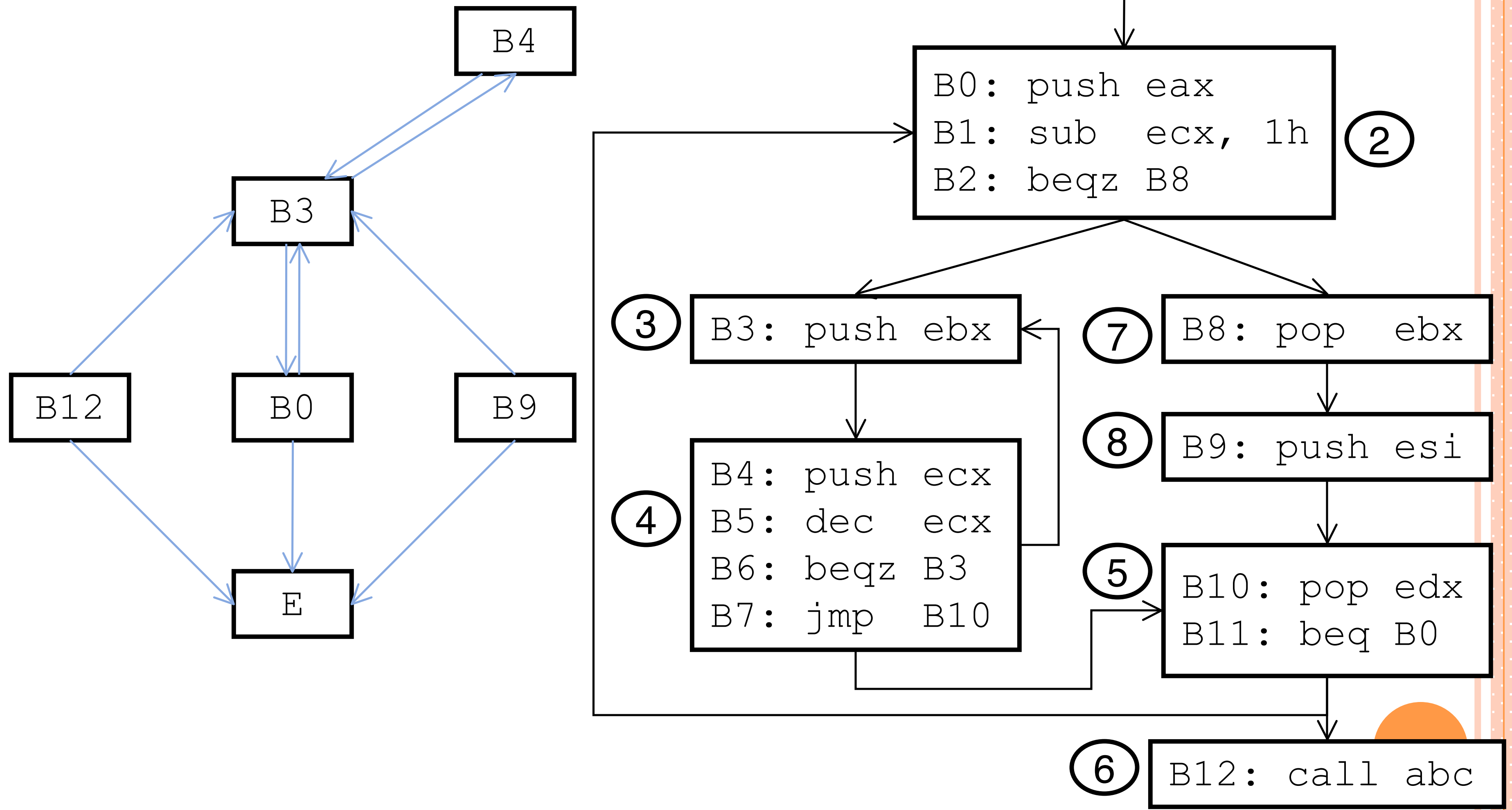
  N _ ADDR is the set of nodes in the ASG, where an address n _ N if n pushes a value onto the stack.

  AE _ ADDR _ ADDR is the set of edges in the ASG, where an edge (n _ m) in AE instruction at n pushes a value on top of a value pushed at instruction m.

  ASPR _ ADDR _ ADDR captures the set of abstract stack pointers (stack tops) for each statement. (x, n) in ASPR program point x receives top of stack from push at instruction n.



Abstract Stack Graph



Abstract Stack Graph

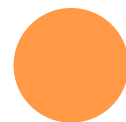
- Step 2 – Generate an Abstract Stack Graph from the basic blocks generated in the modified CFG.
- An Abstract Stack Graph (ASG) is a 3-tuple (N, AE, ASPR) where:

  Let ADDR = set of all addresses in the program.

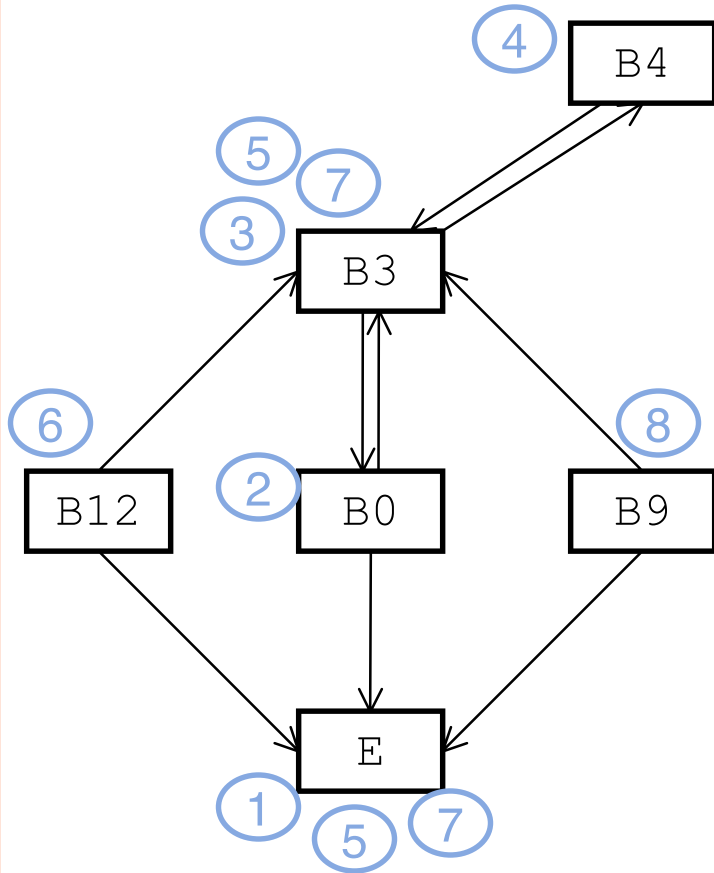
  N _ ADDR is the set of nodes in the ASG, where an address n _ N if n pushes a value onto the stack.

  AE _ ADDR _ ADDR is the set of edges in the ASG, where an edge (n _ m) in AE instruction at n pushes a value on top of a value pushed at instruction m.

  ASPR _ ADDR _ ADDR captures the set of abstract stack pointers (stack tops) for each statement. (x, n) in ASPR means program point x receives top of stack from push at instruction n.



Abstract Stack Graph



1 E: // Entry

2
B0: push eax
B1: sub ecx, 1h
B2: beqz B8

3 B3: push ebx

4
B4: push ecx
B5: dec ecx
B6: beqz B3
B7: jmp B10

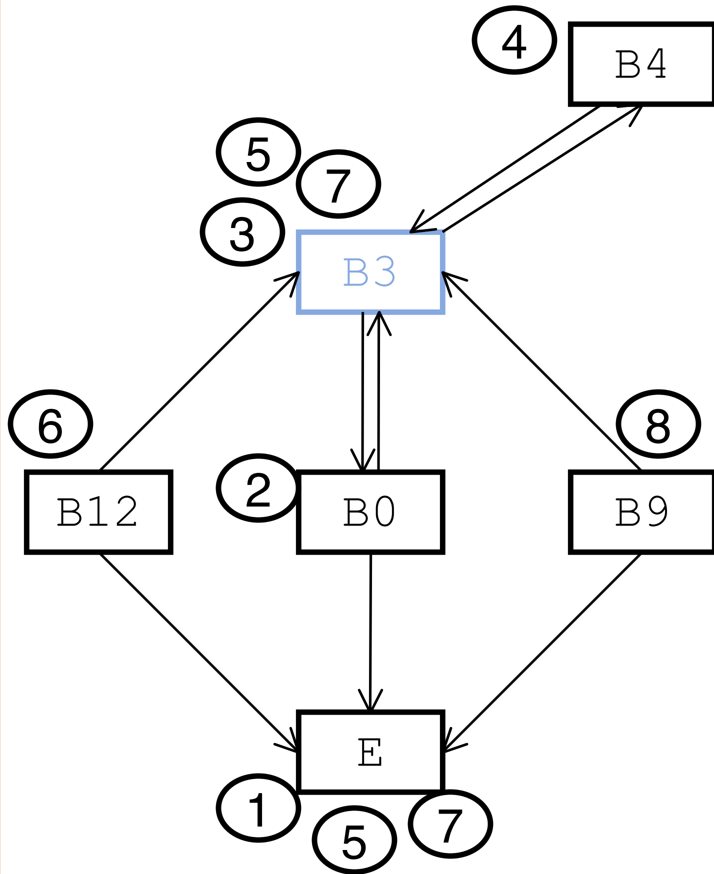
7 B8: pop ebx

8 B9: push esi

5
B10: pop edx
B11: beq B0

6 B12: call abc

Abstract Stack Graph



1
```
E: // Entry
```

2
```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```

3
```
B3: push ebx
```

4
```
B4: push ecx
B5: dec  ecx
B6: beqz B3
B7: jmp  B10
```

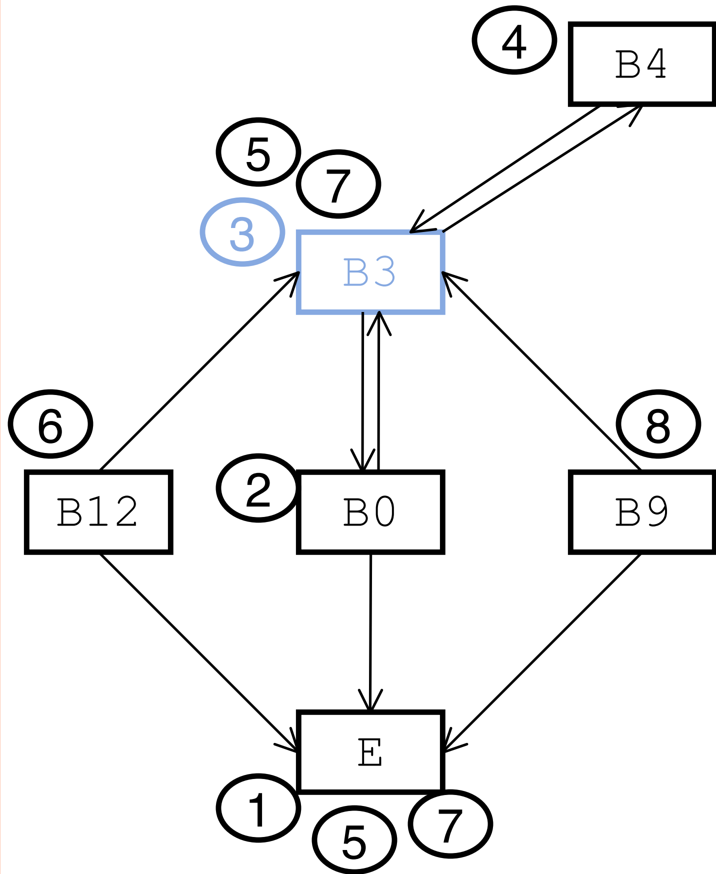
7
```
B8: pop  ebx
```

8
```
B9: push esi
```

5
```
B10: pop edx
B11: beq B0
```

6
```
B12: call abc
```

Abstract Stack Graph



1
```
E: // Entry
```

```
B0: push eax
B1: sub  ecx, 1h
B2: beqz B8
```
2

3
```
B3: push ebx
```

4
```
B4: push ecx
B5: dec  ecx
B6: beqz B3
B7: jmp  B10
```

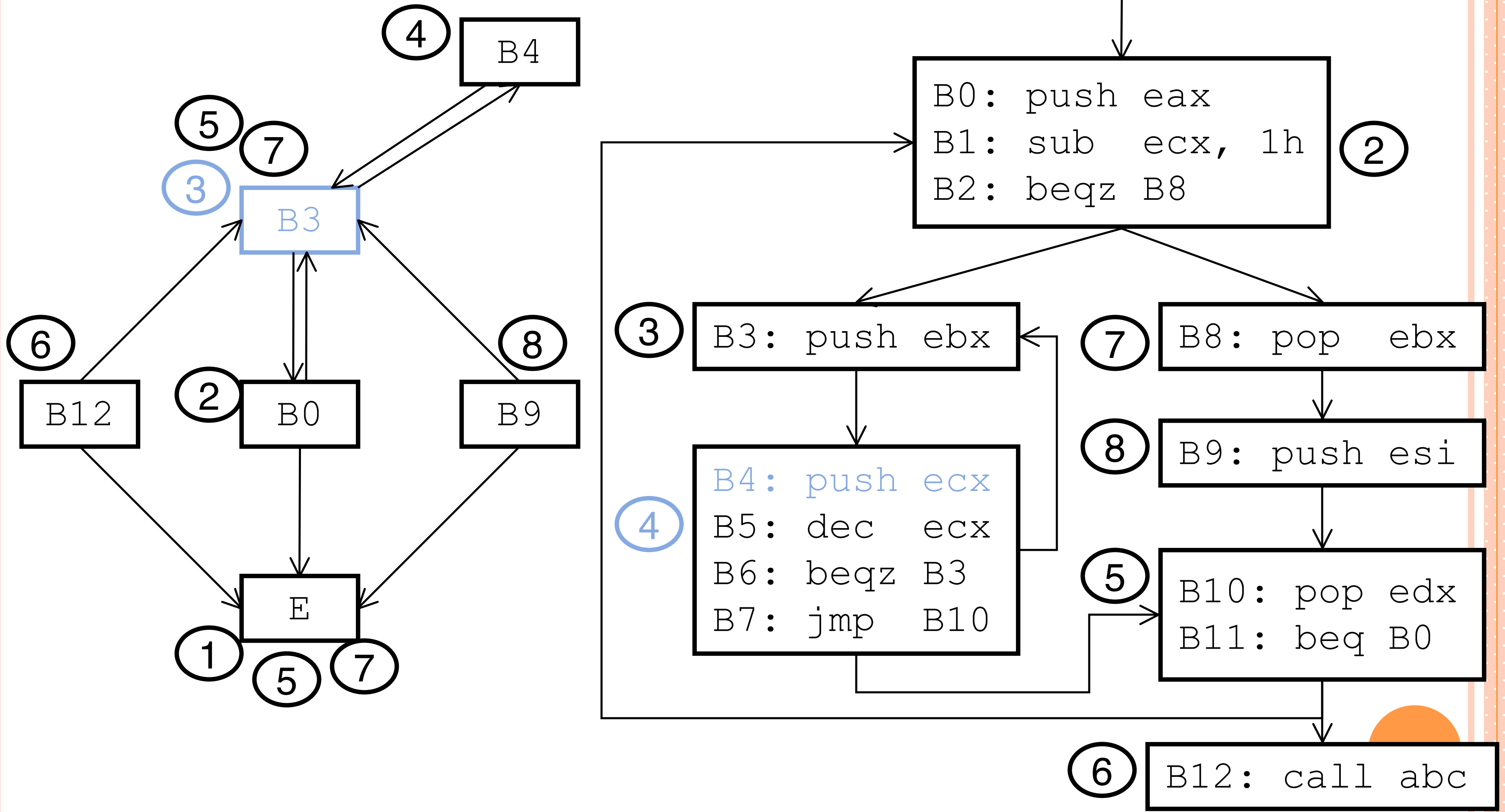
7
```
B8: pop  ebx
```

8
```
B9: push esi
```

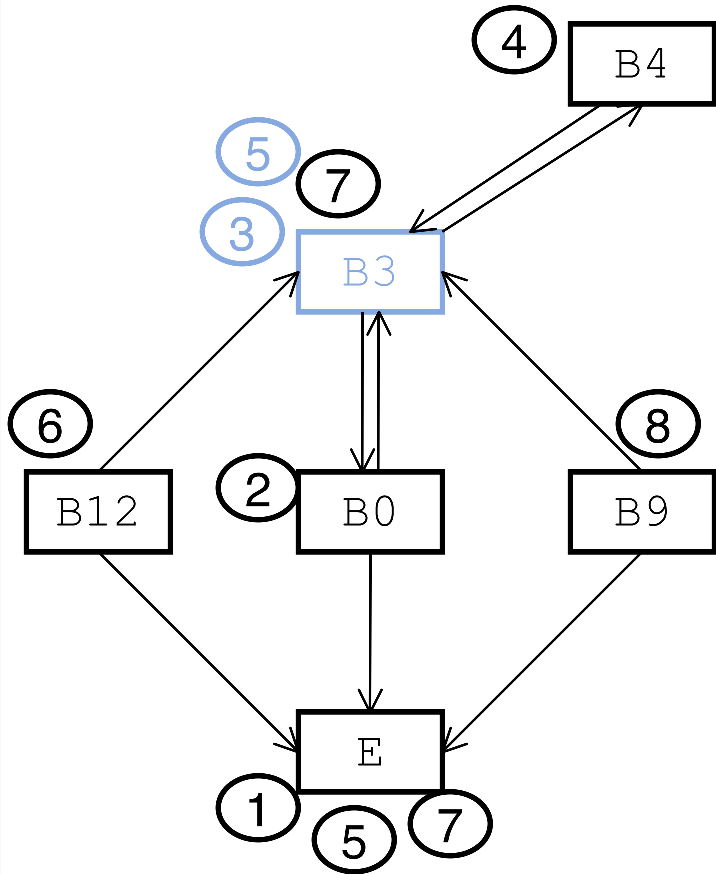
5
```
B10: pop edx
B11: beq B0
```

6
```
B12: call abc
```

Abstract Stack Graph



Abstract Stack Graph



1 E: // Entry

2
B0: push eax
B1: sub ecx, 1h
B2: beqz B8

3 B3: push ebx

4
B4: push ecx
B5: dec ecx
B6: beqz B3
B7: jmp B10

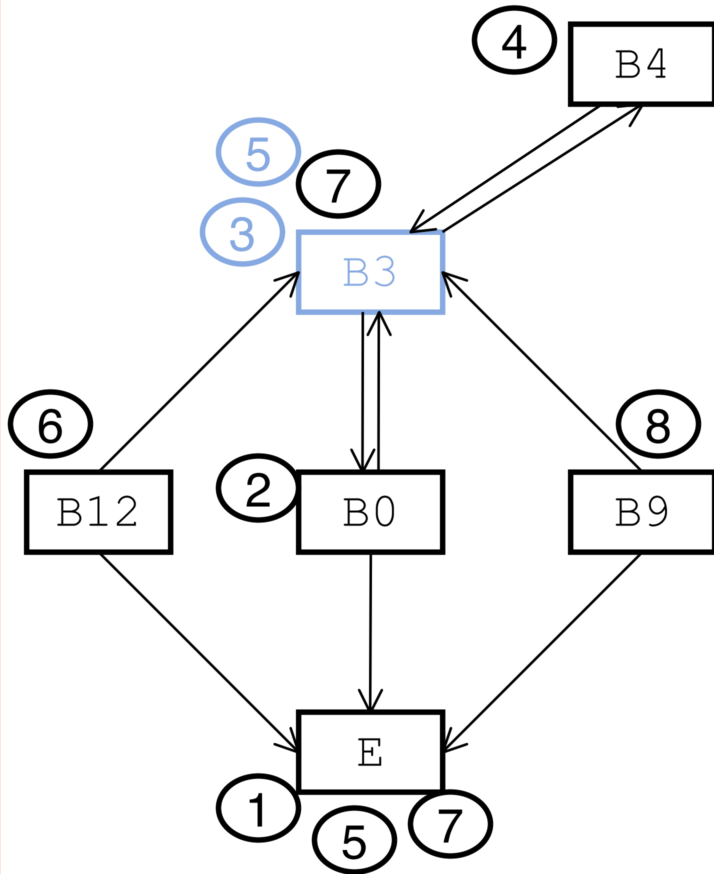
7 B8: pop ebx

8 B9: push esi

5
B10: pop edx
B11: beq B0

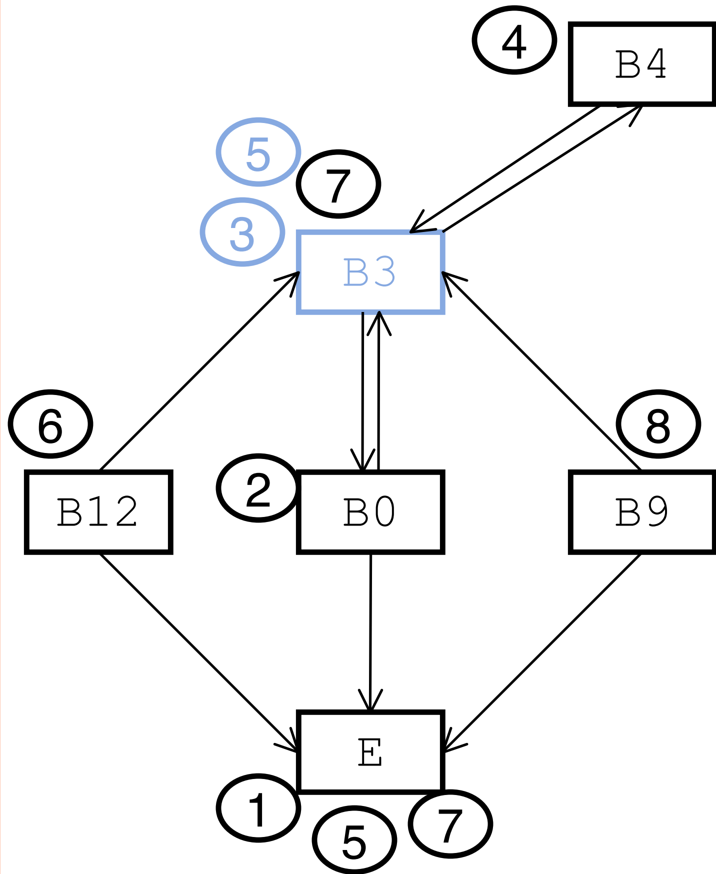
6 B12: call abc

Abstract Stack Graph



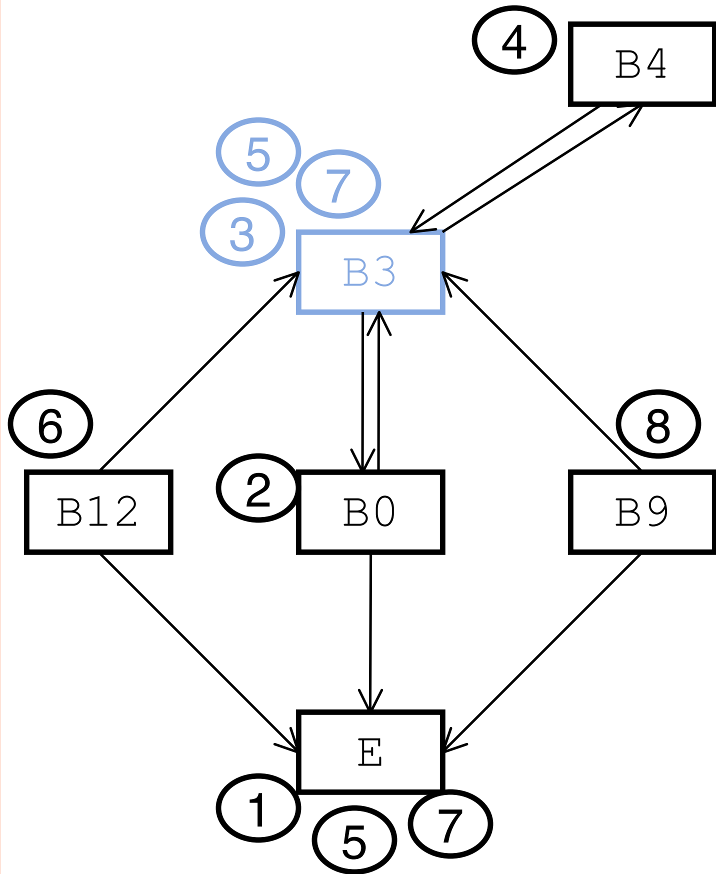

```
(1) E: // Entry

(2) B0: push eax
    B1: sub  ecx, 1h
    B2: beqz B8

(3) B3: push ebx

(4) B4: push ecx
    B5: dec  ecx
    B6: beqz B3
    B7: jmp  B10

(7) B8: pop  ebx

(8) B9: push esi

(5) B10: pop edx
    B11: beq B0

(6) B12: call abc
```

Abstract Stack Graph



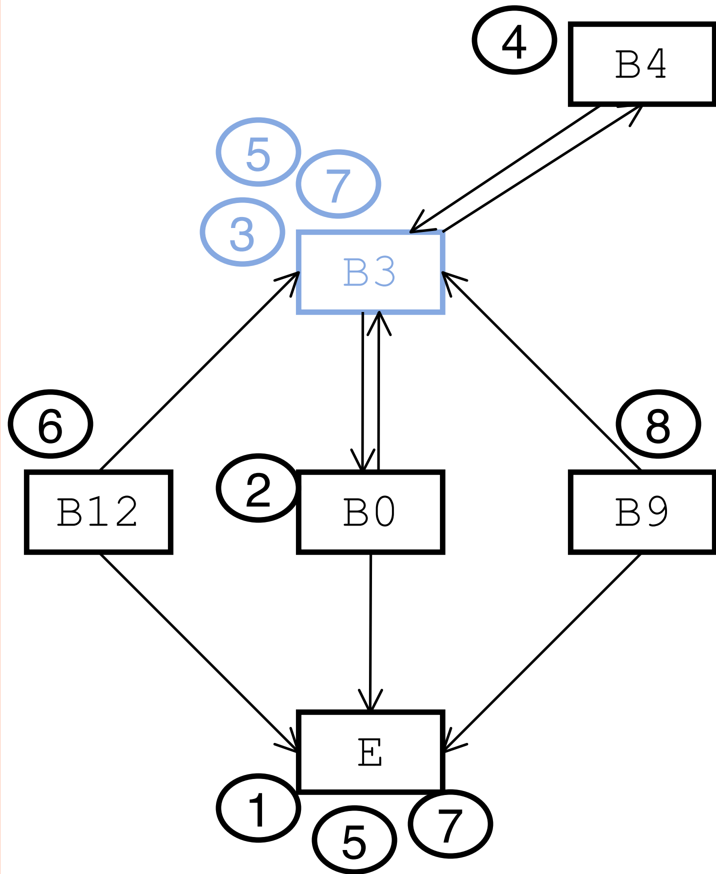
1. E: // Entry
2. B0: push eax
   B1: sub ecx, 1h
   B2: beqz B8
3. B3: push ebx
4. B4: push ecx
   B5: dec ecx
   B6: beqz B3
   B7: jmp B10
7. B8: pop ebx
8. B9: push esi
5. B10: pop edx
   B11: beq B0
6. B12: call abc

Abstract Stack Graph




```
① E: // Entry

B0: push eax
B1: sub  ecx, 1h
B2: beqz B8      ②

③ B3: push ebx

④ B4: push ecx
  B5: dec  ecx
  B6: beqz B3
  B7: jmp  B10

⑦ B8: pop  ebx

⑧ B9: push esi

⑤ B10: pop edx
  B11: beq B0

⑥ B12: call abc
```

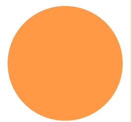
Abstract Stack Graph




```
1  E: // Entry

2  B0: push eax
   B1: sub  ecx, 1h
   B2: beqz B8

3  B3: push ebx

4  B4: push ecx
   B5: dec  ecx
   B6: beqz B3
   B7: jmp  B10

7  B8: pop  ebx

8  B9: push esi

5  B10: pop edx
   B11: beq B0

6  B12: call abc
```

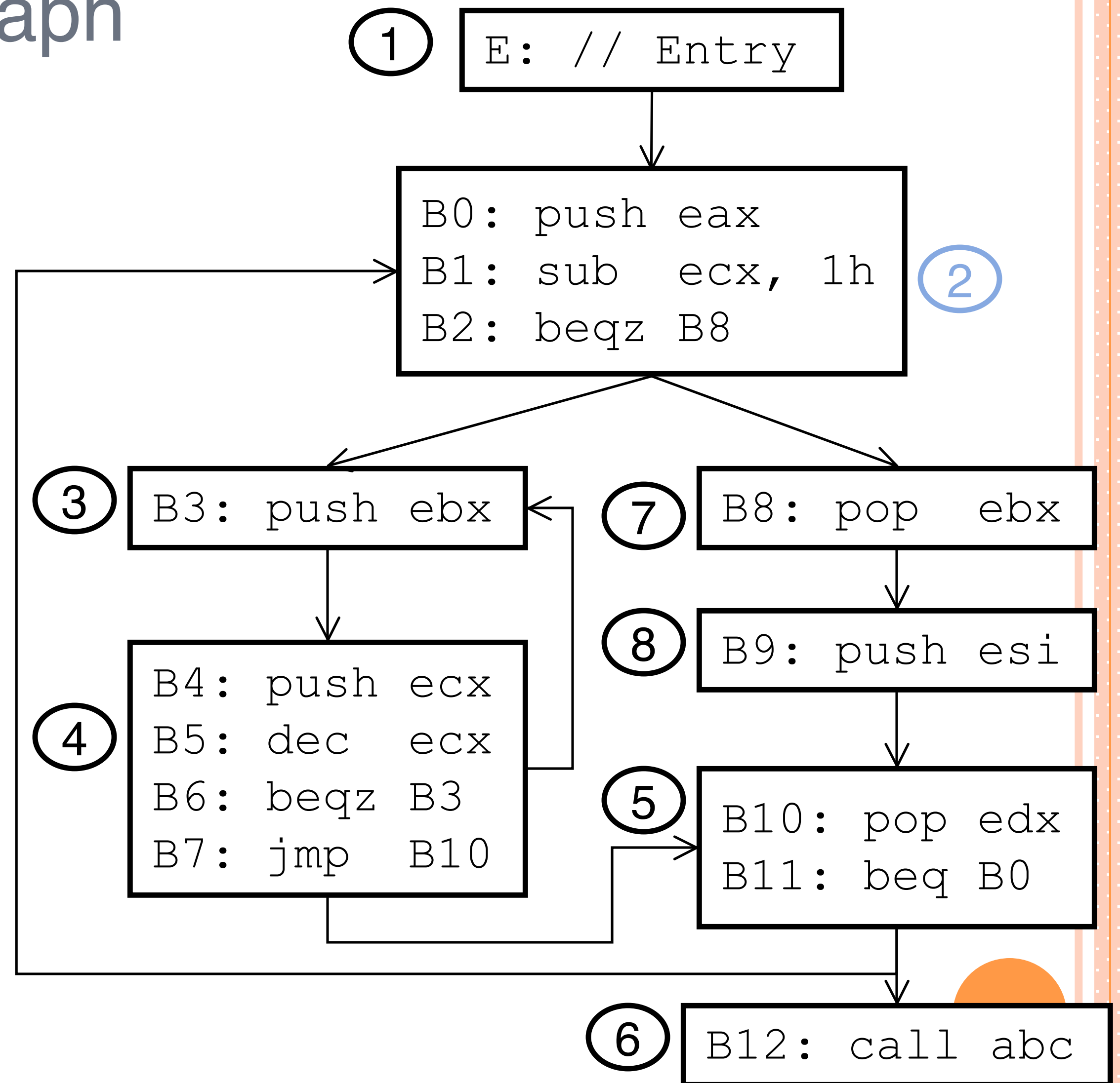
Abstract Stack Graph

- There are an infinite number of abstract stacks.



Abstract Stack Graph

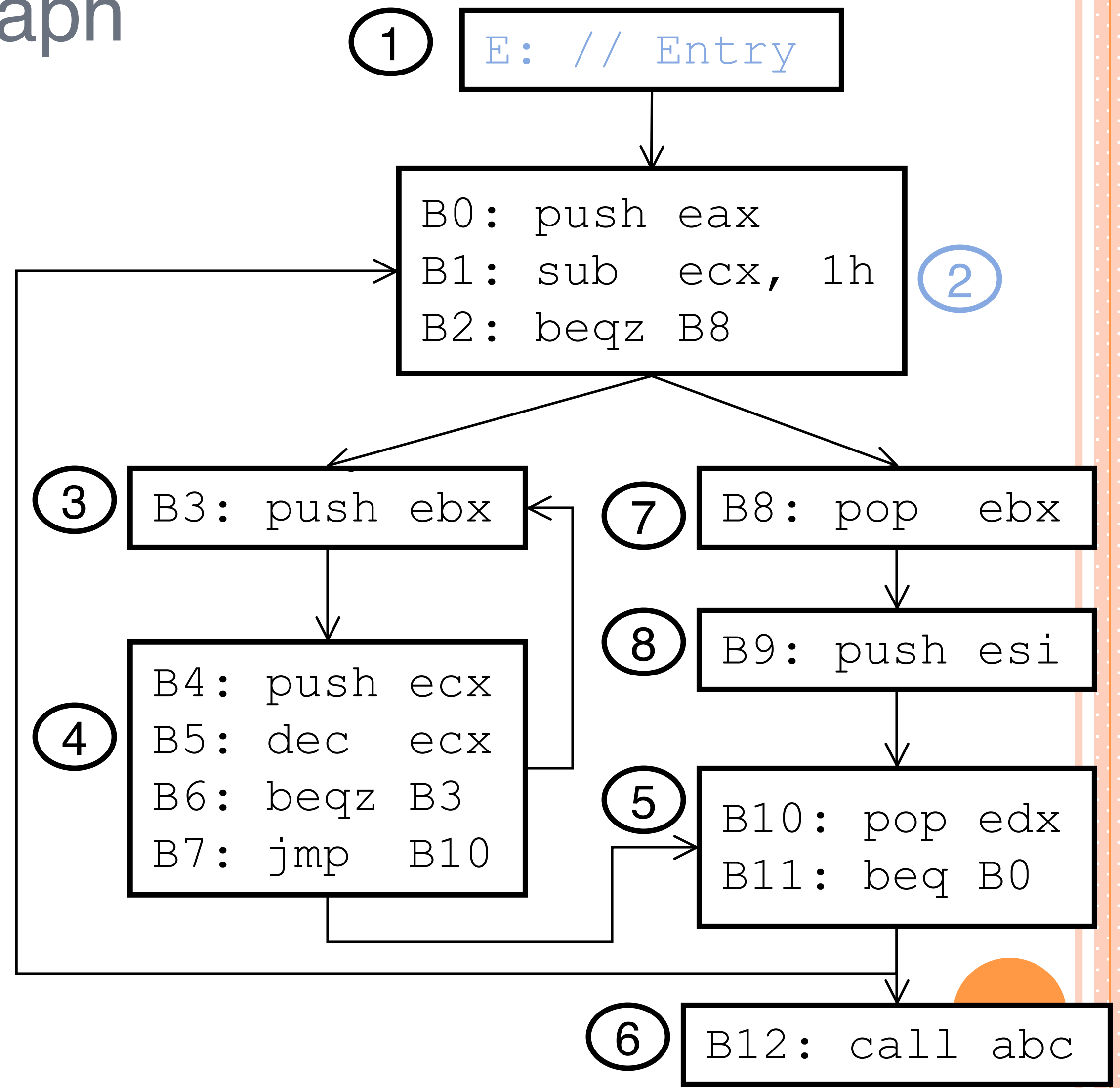
- For BB 2 here are some possible abstract stacks:



Abstract Stack Graph

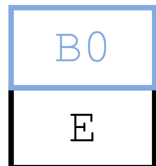
- For BB 2 here are some possible abstract stacks:

E



Abstract Stack Graph

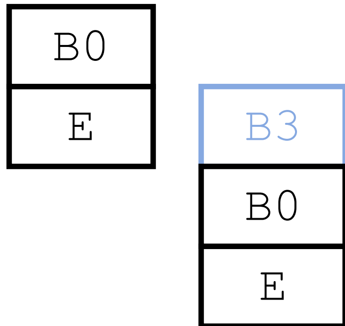
- For BB 2 here are some possible abstract stacks:




```
1  E: // Entry
2  B0: push eax
   B1: sub  ecx, 1h
   B2: beqz B8
3  B3: push ebx
4  B4: push ecx
   B5: dec  ecx
   B6: beqz B3
   B7: jmp  B10
7  B8: pop  ebx
8  B9: push esi
5  B10: pop edx
   B11: beq B0
6  B12: call abc
```

Abstract Stack Graph

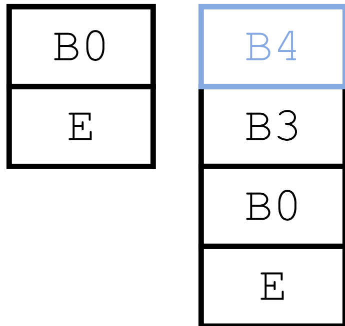
- For BB 2 here are some possible abstract stacks:




```
(1) E: // Entry

(2) B0: push eax
    B1: sub  ecx, 1h
    B2: beqz B8

(3) B3: push ebx

(4) B4: push ecx
    B5: dec  ecx
    B6: beqz B3
    B7: jmp  B10

(7) B8: pop  ebx

(8) B9: push esi

(5) B10: pop edx
    B11: beq B0

(6) B12: call abc
```

Abstract Stack Graph

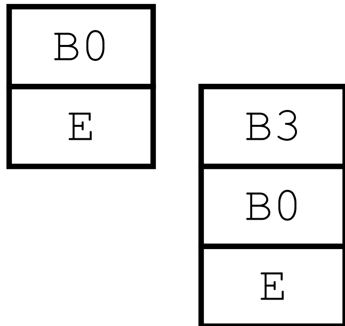
- For BB 2 here are some possible abstract stacks:




```
(1) E: // Entry

(2) B0: push eax
    B1: sub  ecx, 1h
    B2: beqz B8

(3) B3: push ebx

(4) B4: push ecx
    B5: dec  ecx
    B6: beqz B3
    B7: jmp  B10

(7) B8: pop  ebx

(8) B9: push esi

(5) B10: pop edx
    B11: beq B0

(6) B12: call abc
```

Abstract Stack Graph

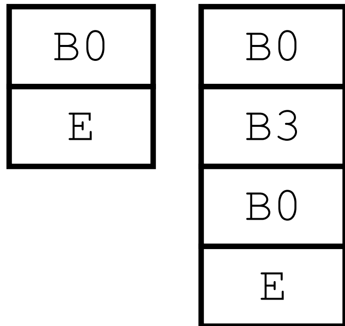
- For BB 2 here are some possible abstract stacks:




```
1  E: // Entry

2  B0: push eax
   B1: sub  ecx, 1h
   B2: beqz B8

3  B3: push ebx

4  B4: push ecx
   B5: dec  ecx
   B6: beqz B3
   B7: jmp  B10

7  B8: pop  ebx

8  B9: push esi

5  B10: pop edx
   B11: beq B0

6  B12: call abc
```

Abstract Stack Graph

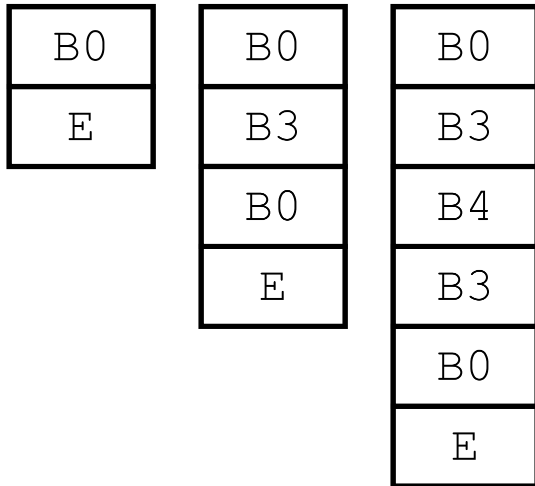
- For BB 2 here are some possible abstract stacks:




```
1  E: // Entry

   B0: push eax
   B1: sub  ecx, 1h     2
   B2: beqz B8

3  B3: push ebx         7  B8: pop  ebx

                        8  B9: push esi
   B4: push ecx
4  B5: dec  ecx
   B6: beqz B3          5  B10: pop edx
   B7: jmp  B10            B11: beq B0

                        6  B12: call abc
```

Abstract Stack Graph

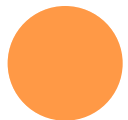
- For BB 2 here are some possible abstract stacks:




```
1  E: // Entry

2  B0: push eax
   B1: sub  ecx, 1h
   B2: beqz B8

3  B3: push ebx

4  B4: push ecx
   B5: dec  ecx
   B6: beqz B3
   B7: jmp  B10

7  B8: pop  ebx

8  B9: push esi

5  B10: pop edx
   B11: beq B0

6  B12: call abc
```

Abstract Stack Graph

- The Goal: For each instruction that pushes a value onto the stack, we want to find every basic block where it may be possible for this value to be on top of the stack.
- May Analysis

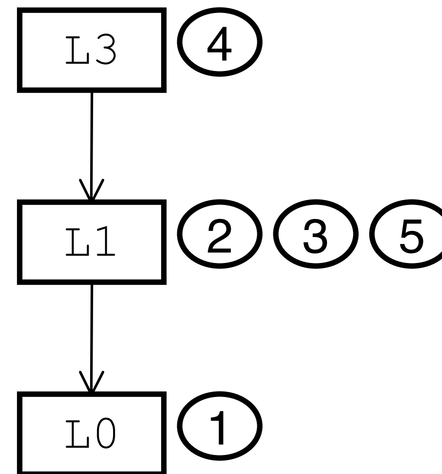


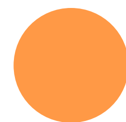
Using The ASG To Detect Obfuscated Calls

- Sample program

```
L0: push  ecx       -> 1
L1: push  L4        -> 2
L2: lea   eax, foo  -> 3
L3: push  eax       -> 4
L4: ret             -> 5
L5: ... // return addr
...
L9: foo() entry     -> 6
```

- Abstract Stack Graph





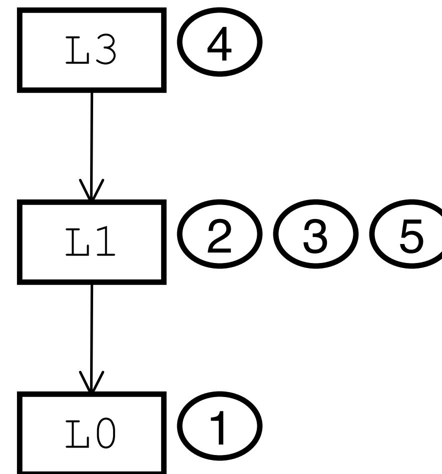
Using The ASG To Detect Obfuscated Calls

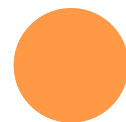
- Sample program

```
L0: push  ecx        -> 1
L1: push  L4         -> 2
L2: lea   eax, foo -> 3
L3: push  eax        -> 4
L4: ret              -> 5
L5: ... // return addr
...
L9: foo() entry     -> 6
```

Return instruction at instruction #5

- Abstract Stack Graph



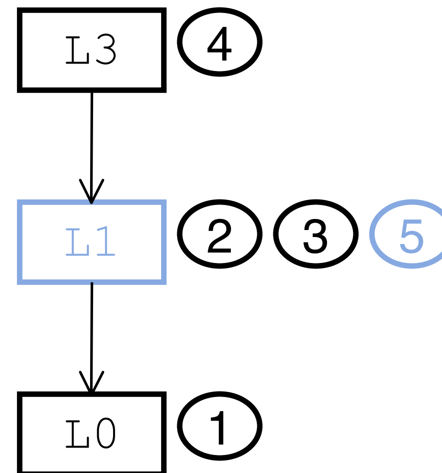


Using The ASG To Detect Obfuscated Calls

- Sample program

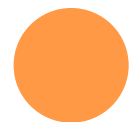
```
L0: push  ecx        -> 1
L1: push  L4         -> 2
L2: lea   eax, foo -> 3
L3: push  eax        -> 4
L4: ret              -> 5
L5: ... // return addr
...
L9: foo() entry      -> 6
```

- Abstract Stack Graph



Return instruction at instruction #5

At instruction #5, the top of stack was created at L0

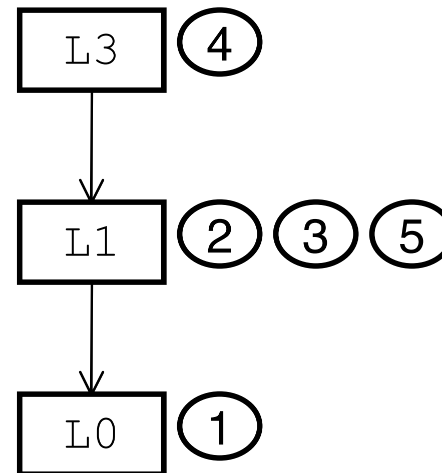


Using The ASG To Detect Obfuscated Calls

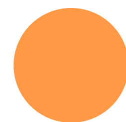
- Sample program

```
L0: push  ecx       -> 1
L1: push  L4        -> 2
L2: lea   eax, foo  -> 3
L3: push  eax       -> 4
L4: ret             -> 5
L5: ... // return addr
...
L9: foo() entry     -> 6
```

- Abstract Stack Graph



Return instruction at instruction #5
At instruction #5, the top of stack was created at L1
Since L0 is not a call instruction we have detected an obfuscated call.
Call instructions are supposed to push the return address.

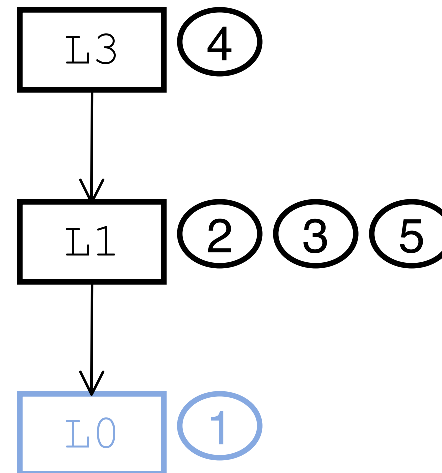


Using The ASG To Detect Obfuscated Parameters

- Sample program

```
L0: push  ecx       -> 1
L1: push  L4        -> 2
L2: lea   eax, foo  -> 3
L3: push  eax       -> 4
L4: ret             -> 5
L5: ... // return addr
...
L9: foo() entry     -> 6
```

- Abstract Stack Graph



Parameters are often obfuscated by placing instructions in between the parameter pushes and the function call. The ASG shows us which instruction pushed each parameter.