Loria
Laboratoire lorrain de recherche en informatique et ses applications

BOA: a control flow graph builder (by Basic blOck Analysis) based on system state prediction

Sylvain CECCHETTO

Loria Carbone team
Third year Ph.D student

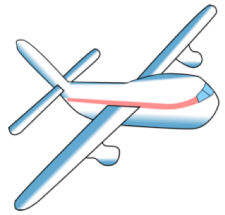
Advised by **Jean-Yves MARION**
and **Guillaume BONFANTE**

Journée protection du code et des données
13 December 2018

Program analysis

- **To do what?**
  - Behaviour analysis
    - Is it malicious? dangerous?

  - Safety — Critical environment
    - Can the program reach an unwanted state? Crash?



  - Reverse
    - Information extraction

- **For who?**
  - Malware analyst
  - Reversers
  - Anti virus
  - …

Loria

Source code vs Binary

- Source code

```
int fib(int n) {
  int first = 0;
  int second = 1;

  int tmp;
  while (n--) {
    tmp = first+second;
    first = second;
    second = tmp;
  }
  return first;
}
```

- High semantic 😃
- Easy to understand 😃
- Not always available ☹️

- Binary

```
55 89 E5 83 EC
10 C7 45 FC 00
00 00 00 C7 45
F8 01 00 00 00
EB 17 8B 55 FC
8B 45 F8 01 D0
89 45 F4 8B 45
F8 89 45 FC 8B
45 F4 89 45 F8
8B 45 08 8D 50
FF 89 55 08 85
C0 75 DC 8B 45
FC C9 C3
```

- Low semantic ☹️
- Hard to understand ☹️
- Available most of the time 😃

Loria

Source code vs Binary

- Source code

```
int fib(int n) {
  int first = 0;
  int second = 1;

  int tmp;
  while (n--) {
    tmp = first+second;
    first = second;
    second = tmp;
  }
  return first;
}
```

- High semantic 😃
- Easy to understand 😃
- Not always available ☹

- Binary

```
55 89 E5 83 EC
10 C7 45 FC 00
00 00 00 C7 45
F8 01 00 00 00
EB 17 8B 55 FC
8B 45 F8 01 D0
89 45 F4 8B 45
F8 89 45 FC 8B
45 F4 89 45 F8
8B 45 08 8D 50
FF 89 55 08 85
C0 75 DC 8B 45
FC C9 C3
```

- Low semantic ☹
- Hard to understand ☹
- Available most of the time 😃

- **Usually we do not have the source code of a malware**

Source code vs Binary

- Source code

```
int fib(int n) {
  int first = 0;
  int second = 1;

  int tmp;
  while (n--) {
    tmp = first+second;
    first = second;
    second = tmp;
  }
  return first;
}
```

- High semantic 😃
- Easy to understand 😃
- Not always available ☹

- Binary

```
55 89 E5 83 EC
10 C7 45 FC 00
00 00 00 C7 45
F8 01 00 00 00
EB 17 8B 55 FC
8B 45 F8 01 D0
89 45 F4 8B 45
F8 89 45 FC 8B
45 F4 89 45 F8
8B 45 08 8D 50
FF 89 55 08 85
C0 75 DC 8B 45
FC C9 C3
```

- Low semantic ☹
- Hard to understand ☹
- Available most of the time 😃

- **Usually we do not have the source code of a malware**

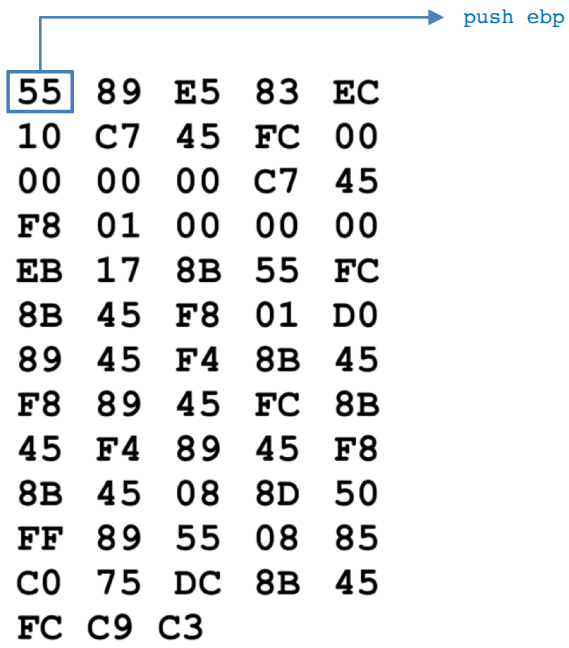
Disassembly

- **Definition:** From a binary file, this task aims to list the program instructions.

```
55  89  E5  83  EC
10  C7  45  FC  00
00  00  00  C7  45
F8  01  00  00  00
EB  17  8B  55  FC
8B  45  F8  01  D0
89  45  F4  8B  45
F8  89  45  FC  8B
45  F4  89  45  F8
8B  45  08  8D  50
FF  89  55  08  85
C0  75  DC  8B  45
FC C9 C3
```

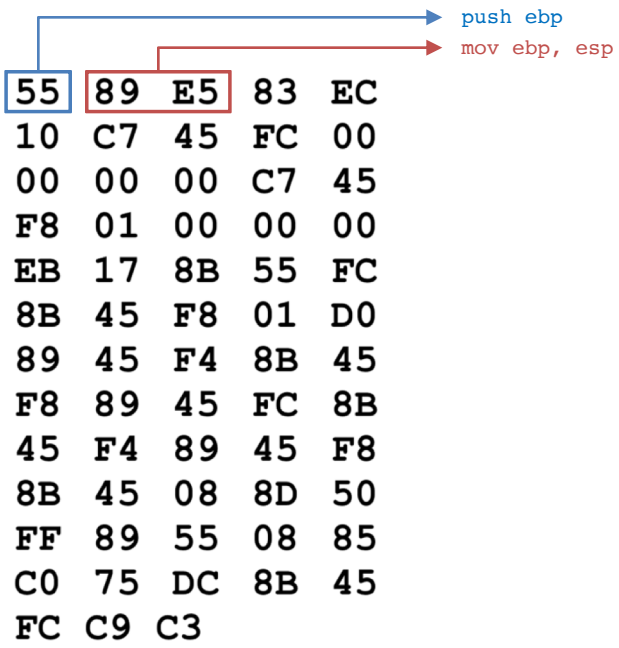
Disassembly

- **Definition:** From a binary file, this task aims to list the program instructions.



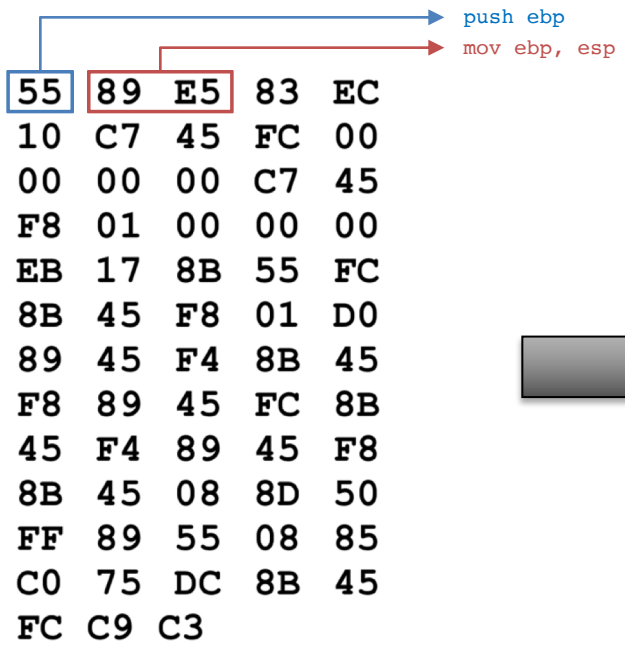
Disassembly

- **Definition:** From a binary file, this task aims to list the program instructions.



Disassembly

- **Definition:** From a binary file, this task aims to list the program instructions.

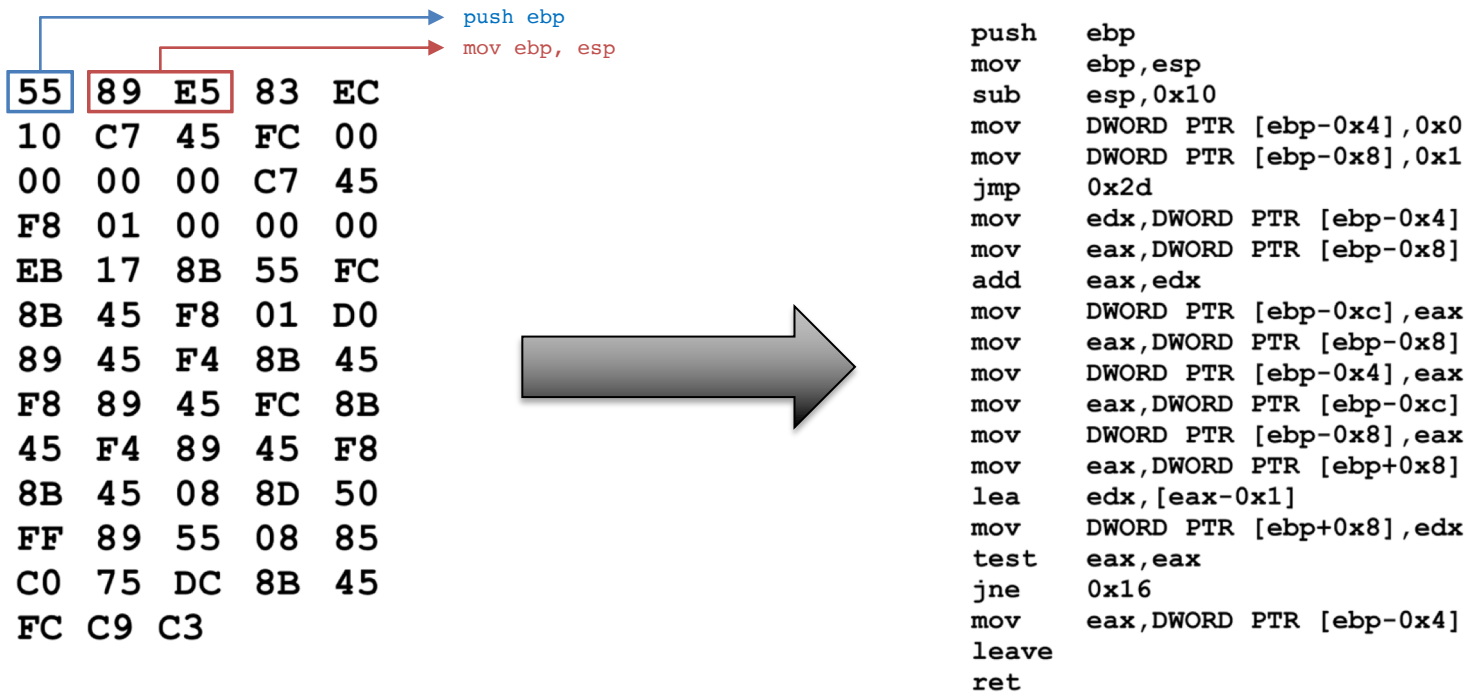



```
push   ebp
mov    ebp,esp
sub    esp,0x10
mov    DWORD PTR [ebp-0x4],0x0
mov    DWORD PTR [ebp-0x8],0x1
jmp    0x2d
mov    edx,DWORD PTR [ebp-0x4]
mov    eax,DWORD PTR [ebp-0x8]
add    eax,edx
mov    DWORD PTR [ebp-0xc],eax
mov    eax,DWORD PTR [ebp-0x8]
mov    DWORD PTR [ebp-0x4],eax
mov    eax,DWORD PTR [ebp-0xc]
mov    DWORD PTR [ebp-0x8],eax
mov    eax,DWORD PTR [ebp+0x8]
lea    edx,[eax-0x1]
mov    DWORD PTR [ebp+0x8],edx
test   eax,eax
jne    0x16
mov    eax,DWORD PTR [ebp-0x4]
leave
ret
```

Loria

Disassembly

- **Definition:** From a binary file, this task aims to list the program instructions.



- **Problem:** A binary file does not only contain code bytes
  - Mixed data
  - Header
  - …

- The disassembly problem is an undecidable problem

Control Flow Graph (CFG)

- **Instruction typology**
  - Sequential (e.g. `mov al, 0x61`)
  - Explicit unconditional jump (e.g. `jmp 0x1005121`)
  - Conditional jump (e.g. `jnz 0x1005324`)
  - Jump depending on the context (e.g. `jmp eax`)

- **Basic block**
  - Ordered sequence of *n* extended instructions (address + instruction)
  - All instructions except the last one must be sequential
  - All instructions except the first and last one must have an unique antecedent and an unique successor

```
0x4011A0:   push   ebp
0x4011A1:   mov    ebp,esp
0x4011A0:   sub    esp,0x10
0x4011A6:   mov    DWORD PTR [ebp-0x4],0x0
0x4011AD:   mov    DWORD PTR [ebp-0x8],0x1
0x4011B4:   jmp    0x2d
0x4011B6:   mov    edx,DWORD PTR [ebp-0x4]
0x4011B9:   mov    eax,DWORD PTR [ebp-0x8]
0x4011BC:   add    eax,edx
0x4011BE:   mov    DWORD PTR [ebp-0xc],eax
0x4011C1:   mov    eax,DWORD PTR [ebp-0x8]
0x4011C4:   mov    DWORD PTR [ebp-0x4],eax
0x4011C7:   mov    eax,DWORD PTR [ebp-0xc]
0x4011CA:   mov    DWORD PTR [ebp-0x8],eax
0x4011CD:   mov    eax,DWORD PTR [ebp+0x8]
0x4011D0:   lea    edx,[eax-0x1]
0x4011D3:   mov    DWORD PTR [ebp+0x8],edx
0x4011D6:   test   eax,eax
0x4011D8:   jne    0x16
0x4011DA:   mov    eax,DWORD PTR [ebp-0x4]
0x4011DD:   leave
0x4011DE:   ret
```

Control Flow Graph (CFG)

- **Instruction typology**
  - Sequential (e.g. `mov al, 0x61`)
  - Explicit unconditional jump (e.g. `jmp 0x1005121`)
  - Conditional jump (e.g. `jnz 0x1005324`)
  - Jump depending on the context (e.g. `jmp eax`)

- **Basic block**
  - Ordered sequence of *n* extended instructions (address + instruction)
  - All instructions except the last one must be sequential
  - All instructions except the first and last one must have an unique antecedent and an unique successor

```
0x4011A0:   push   ebp
0x4011A1:   mov    ebp,esp
0x4011A0:   sub    esp,0x10
0x4011A6:   mov    DWORD PTR [ebp-0x4],0x0
0x4011AD:   mov    DWORD PTR [ebp-0x8],0x1
0x4011B4:   jmp    0x2d
0x4011B6:   mov    edx,DWORD PTR [ebp-0x4]
0x4011B9:   mov    eax,DWORD PTR [ebp-0x8]
0x4011BC:   add    eax,edx
0x4011BE:   mov    DWORD PTR [ebp-0xc],eax
0x4011C1:   mov    eax,DWORD PTR [ebp-0x8]
0x4011C4:   mov    DWORD PTR [ebp-0x4],eax
0x4011C7:   mov    eax,DWORD PTR [ebp-0xc]
0x4011CA:   mov    DWORD PTR [ebp-0x8],eax
0x4011CD:   mov    eax,DWORD PTR [ebp+0x8]
0x4011D0:   lea    edx,[eax-0x1]
0x4011D3:   mov    DWORD PTR [ebp+0x8],edx
0x4011D6:   test   eax,eax
0x4011D8:   jne    0x16
0x4011DA:   mov    eax,DWORD PTR [ebp-0x4]
0x4011DD:   leave
0x4011DE:   ret
```

```
0x4011A0: push ebp
0x4011A1: mov ebp, esp
0x4011A3: sub esp, 0x10
0x4011A6: mov [esp-0x4], 0x0
0x4011AD: mov [esp-0x8], 0x1
0x4011B4: jmp 0x4011CD
```

```
0x4011CD: mov eax, [ebp+0x8]
0x4011D0: lea edx, [eax-0x1]
0x4011D3: [ebp+0x8], edx
0x4011D6: test eax, eax
0x4011D8: jnz 0x4011B6
```

```
0x4011B6: mov edx, [ebp-0x4]
0x4011B9: mov eax, [ebp-0x8]
0x4011BC: add eax, edx
0x4011BE: mov [ebp-0xC], eax
0x4011C1: mov eax, [ebp-0x8]
0x4011C4: mov [ebp-0x4], eax
0x4011C7: mov eax, [ebp-0xC]
0x4011CA: mov [ebp-0x8], eax
```

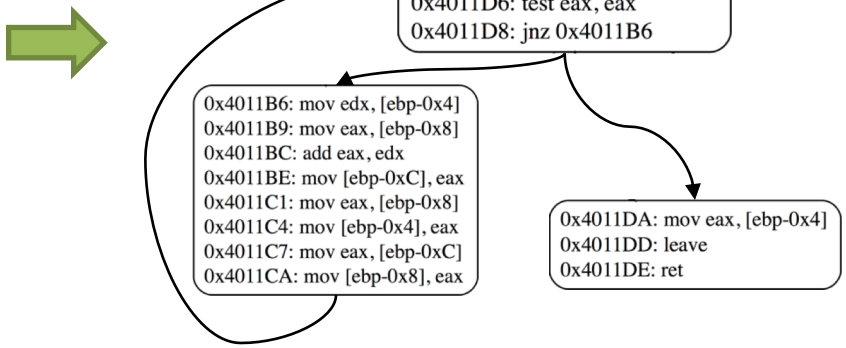
```
0x4011DA: mov eax, [ebp-0x4]
0x4011DD: leave
0x4011DE: ret
```

Control Flow Graph (CFG)

- **Instruction typology**
  - Sequential (e.g. `mov al, 0x61`)
  - Explicit unconditional jump (e.g. `jmp 0x1005121`)
  - Conditional jump (e.g. `jnz 0x1005324`)
  - Jump depending on the context (e.g. `jmp eax`)

- **Basic block**
  - Ordered sequence of *n* extended instructions (address + instruction)
  - All instructions except the last one must be sequential
  - All instructions except the first and last one must have an unique antecedent and an unique successor

```
0x4011A0:   push   ebp
0x4011A1:   mov    ebp,esp
0x4011A0:   sub    esp,0x10
0x4011A6:   mov    DWORD PTR [ebp-0x4],0x0
0x4011AD:   mov    DWORD PTR [ebp-0x8],0x1
0x4011B4:   jmp    0x2d
0x4011B6:   mov    edx,DWORD PTR [ebp-0x4]
0x4011B9:   mov    eax,DWORD PTR [ebp-0x8]
0x4011BC:   add    eax,edx
0x4011BE:   mov    DWORD PTR [ebp-0xc],eax
0x4011C1:   mov    eax,DWORD PTR [ebp-0x8]
0x4011C4:   mov    DWORD PTR [ebp-0x4],eax
0x4011C7:   mov    eax,DWORD PTR [ebp-0xc]
0x4011CA:   mov    DWORD PTR [ebp-0x8],eax
0x4011CD:   mov    eax,DWORD PTR [ebp+0x8]
0x4011D0:   lea    edx,[eax-0x1]
0x4011D3:   mov    DWORD PTR [ebp+0x8],edx
0x4011D6:   test   eax,eax
0x4011D8:   jne    0x16
0x4011DA:   mov    eax,DWORD PTR [ebp-0x4]
0x4011DD:   leave
0x4011DE:   ret
```

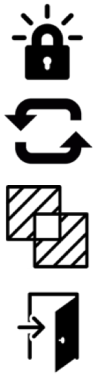


Obfuscations

- **Problem:** Programs are protected against analysis

- **Why?**
  - Intellectual property (paid software, …)
  - Hide functionality, program intelligence (**Malware)**

- **How?**
  - Cryptography
  - Self-modification
  - Code overlapping
  - Anti-analysis tricks



Loria

Problematic and objectives

- **Problematic**
  - Hard to disassemble a binary
  - **Very** hard to disassemble an **obfuscated** binary

- **Objectives**
  - Improve **disassembly** step (completeness and correction)
  - Improve **Control Flow Graph**
  - Detect **statically**
    - Call Stack Tampering
    - Self modification
    - Opaque predicate
    - Corrupted exception handler
  - Give **information** about memory cells and registers values at the entry and the exit of basic blocks

Loria

Our approach

1. Start **recursive disassembly** from the program entry point until we reach the first ”jump depending on the context” instruction

2. Build an **initial Control Flow Graph**

3. Apply **symbolic execution** on each basic block

4. **Propagate formula** between basic block and **resolve dynamic jumps**

5. Re-start **disassembly** from new **dynamic jumps targets** and so on

Loria

Symbolic execution (BinSec)

```
0x10011de   call 0x1001374
```

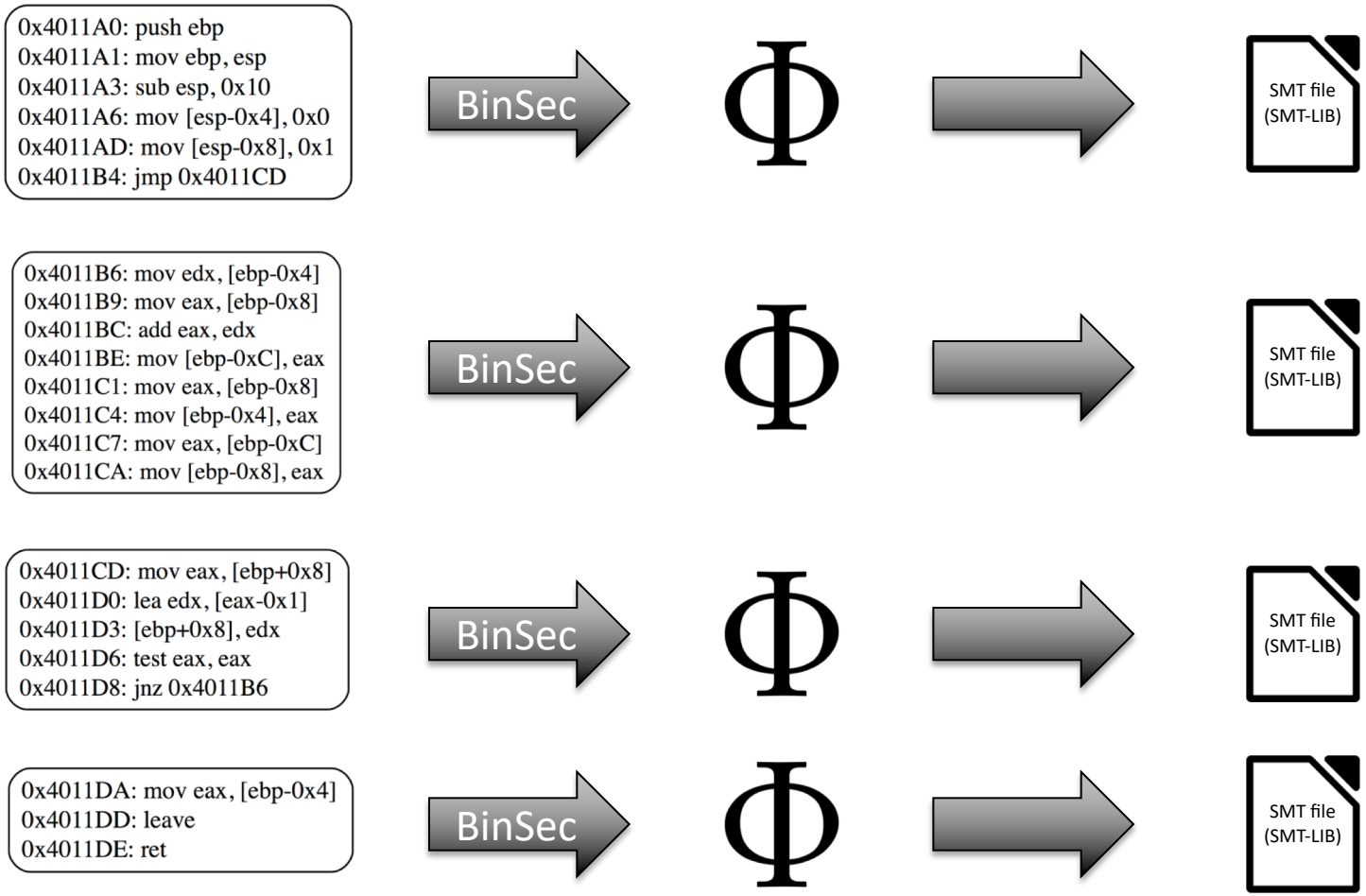
BINSEC*

```
1  (declare-fun esp () (_ BitVec 32))
2  (declare-fun memory () (Array (_ BitVec 32) (_ BitVec 8)))
3
4  ; ------[ # Instruction 0: 0x10011de (call 0x1001374) ]------
5
6
7  ; ------[ * DBA instr 0: esp := (esp(32) - 4(32)) ]------
8
9  (define-fun esp1 () (_ BitVec 32) (bvsub esp (_ bv4 32)))
10
11
12
13 ; ------[ * DBA instr 1: @[esp(32)]L4 := {0x010011e3; 32} ]------
14
15 (define-fun memory1 () (Array (_ BitVec 32) (_ BitVec 8))
16     (store
17         (store
18             (store
19                 (store
20                     memory esp1 (_ bv227 8)
21                 )
22                 (bvadd esp1 (_ bv1 32)) (_ bv17 8)
23             )
24             (bvadd esp1 (_ bv2 32)) (_ bv0 8)
25         )
26         (bvadd esp1 (_ bv3 32)) (_ bv1 8)
27     )
28 )
```

Loria

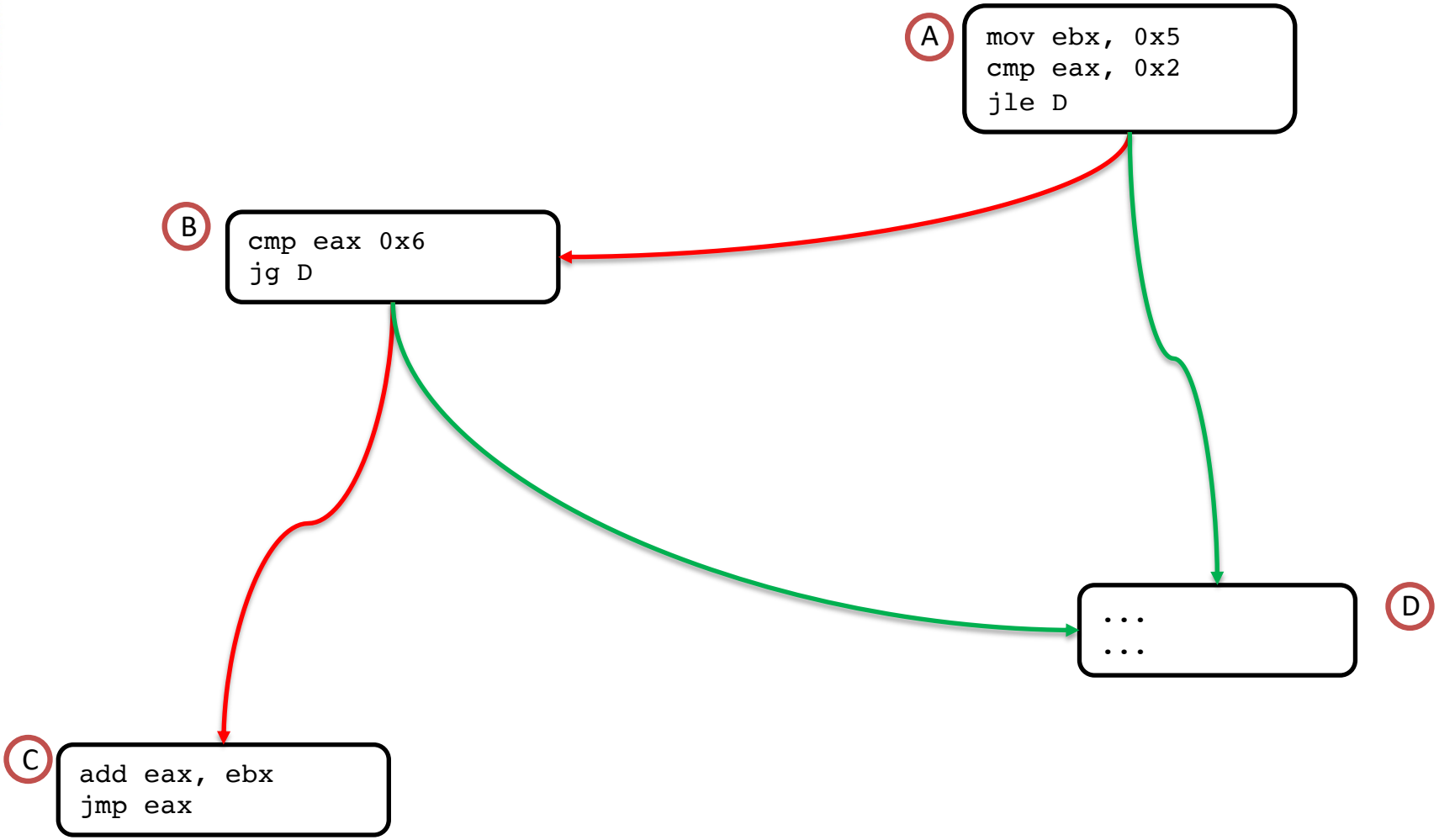
* R. David *et al.*, "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Suita, 2016, pp. 653-656.

Symbolic execution (BinSec)

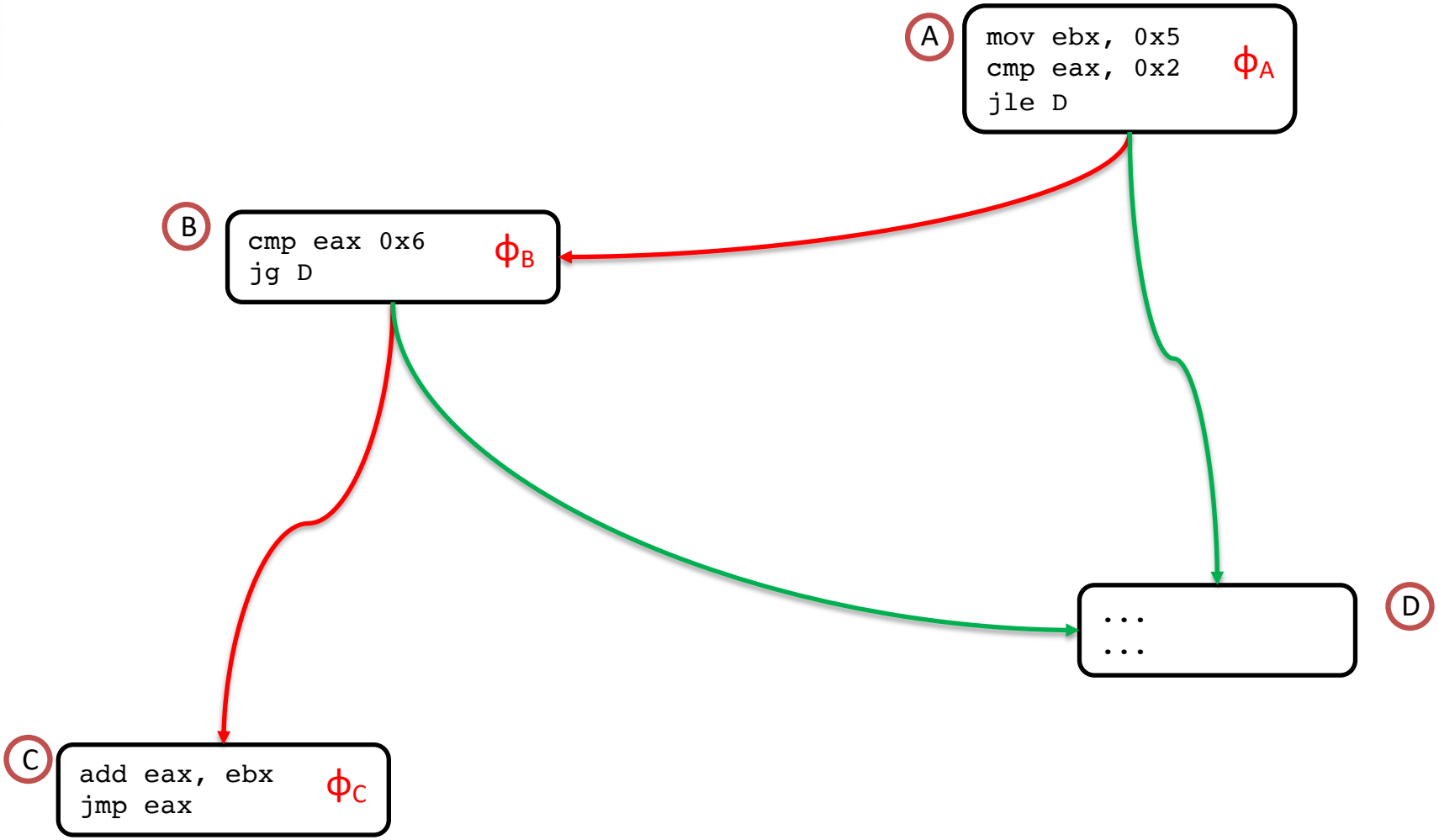


Loria

Dynamic jump example

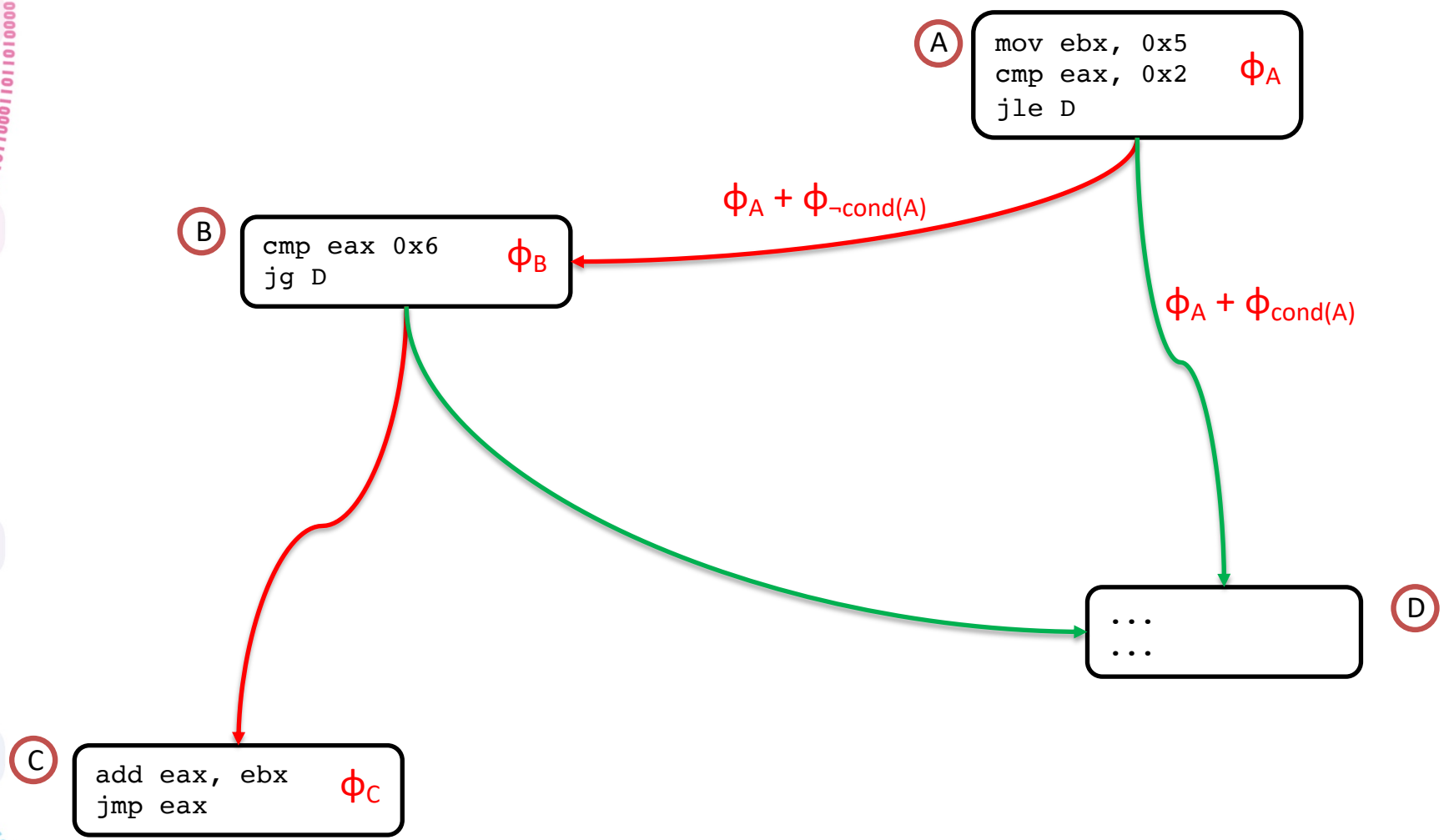


Dynamic jump example



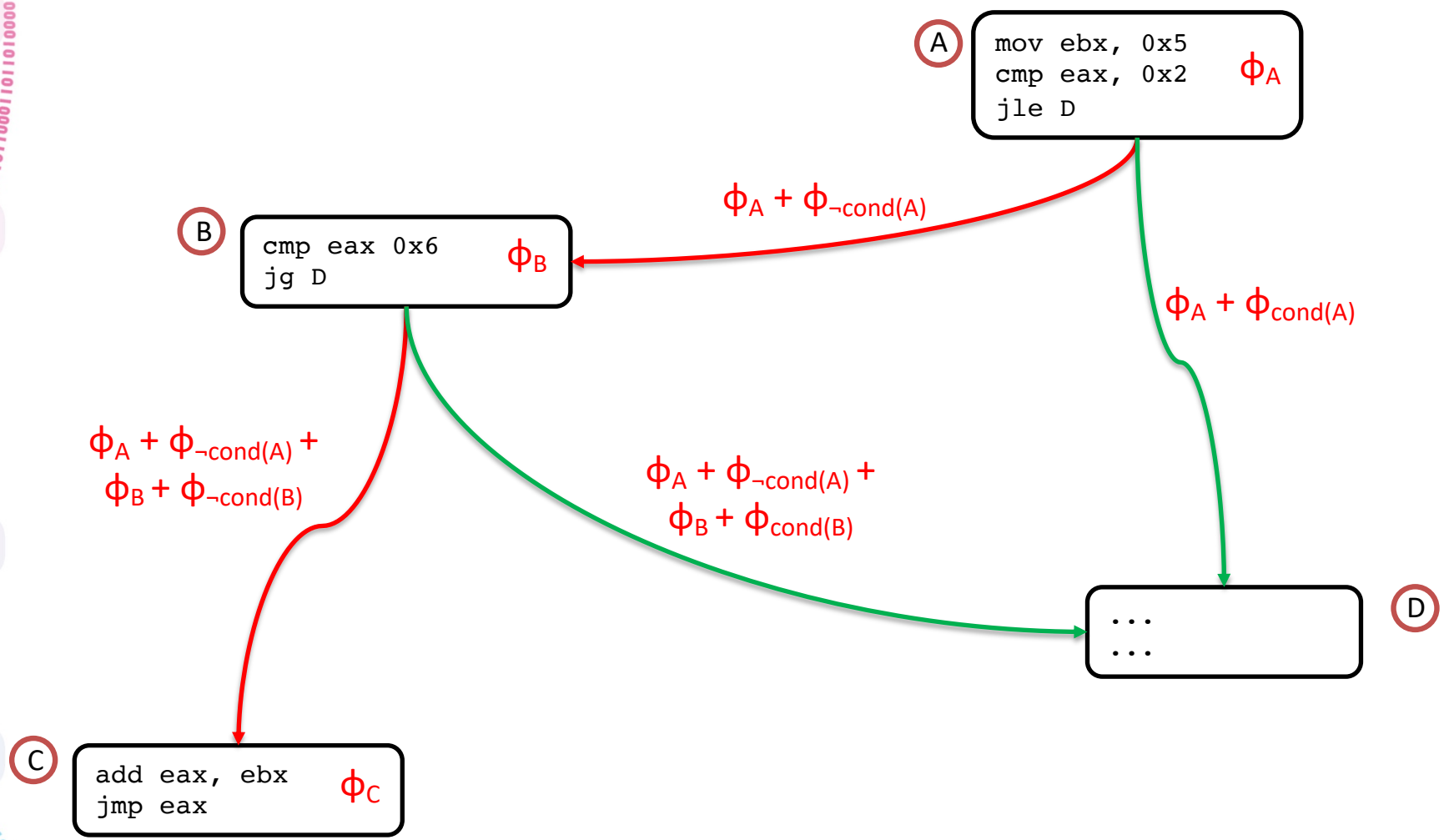
Loria

Dynamic jump example

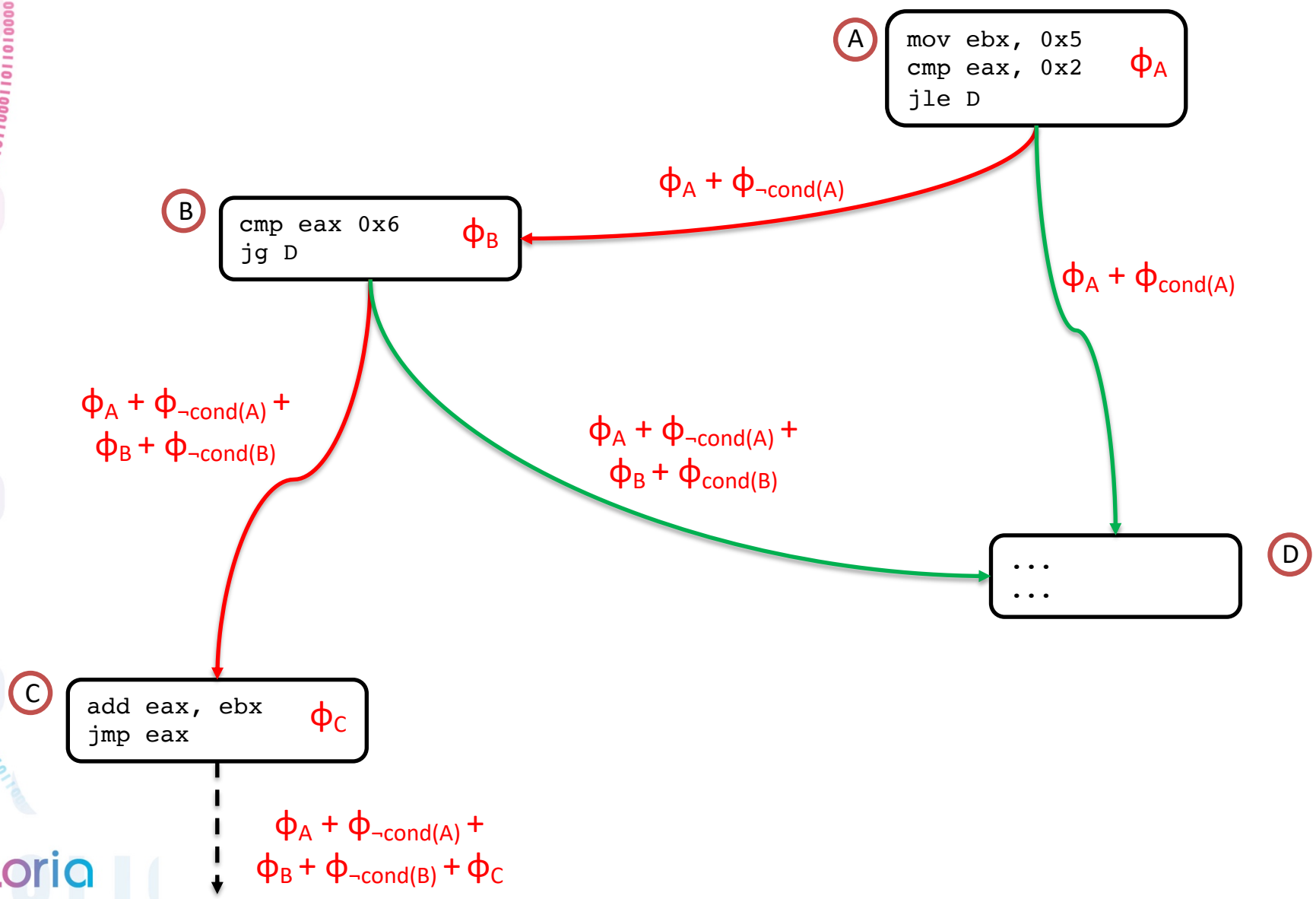


Loria

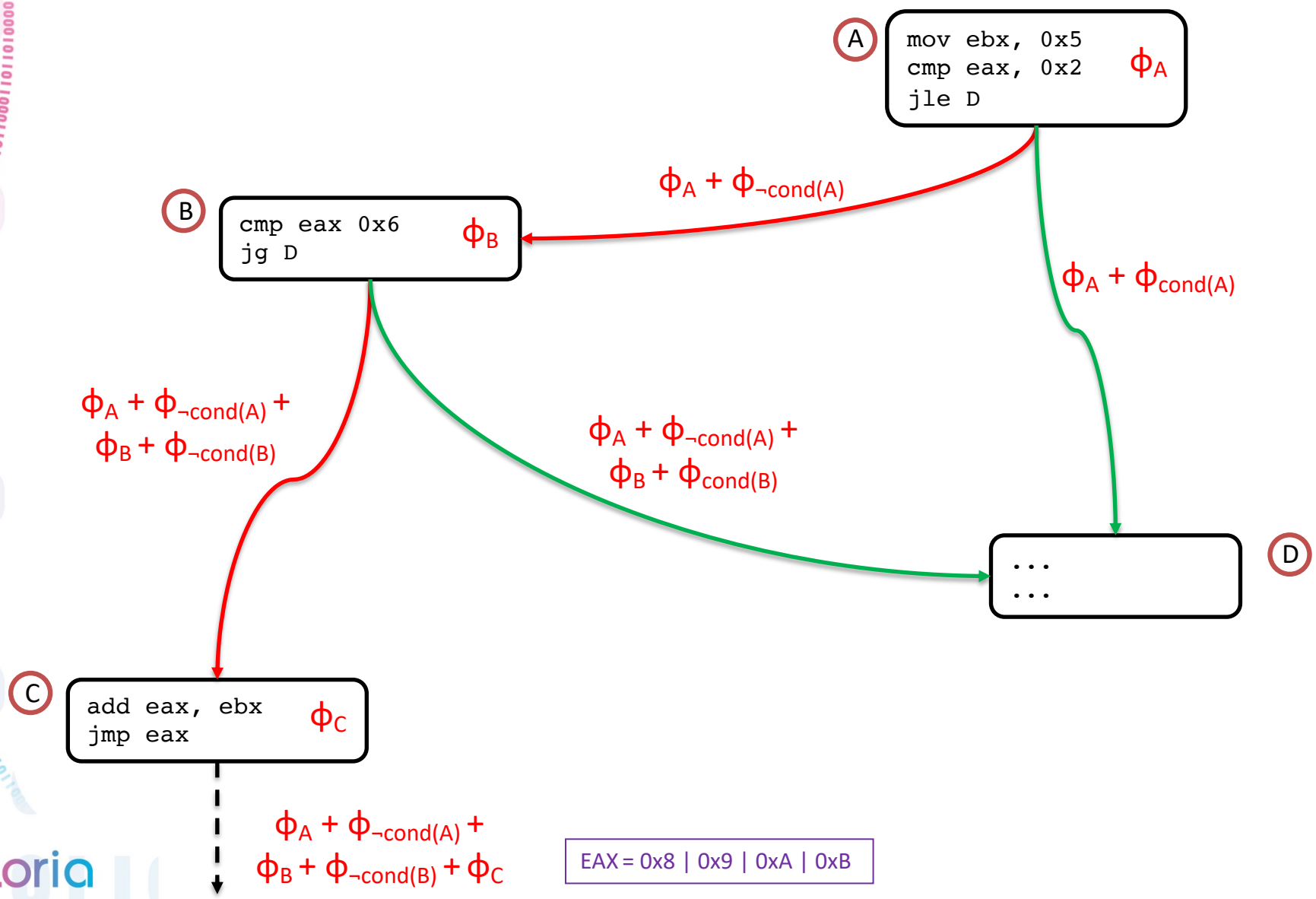
Dynamic jump example



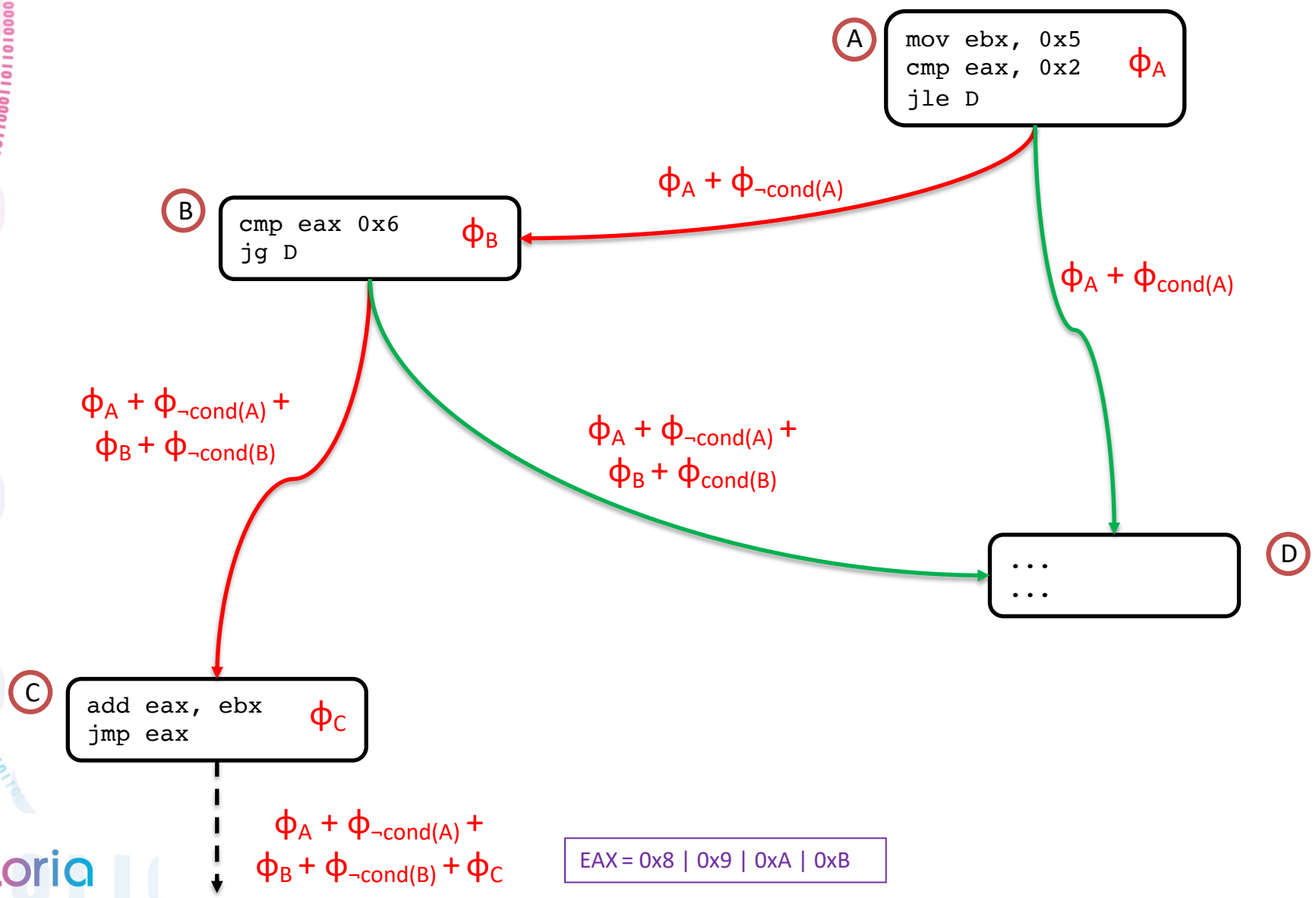
Dynamic jump example



Dynamic jump example

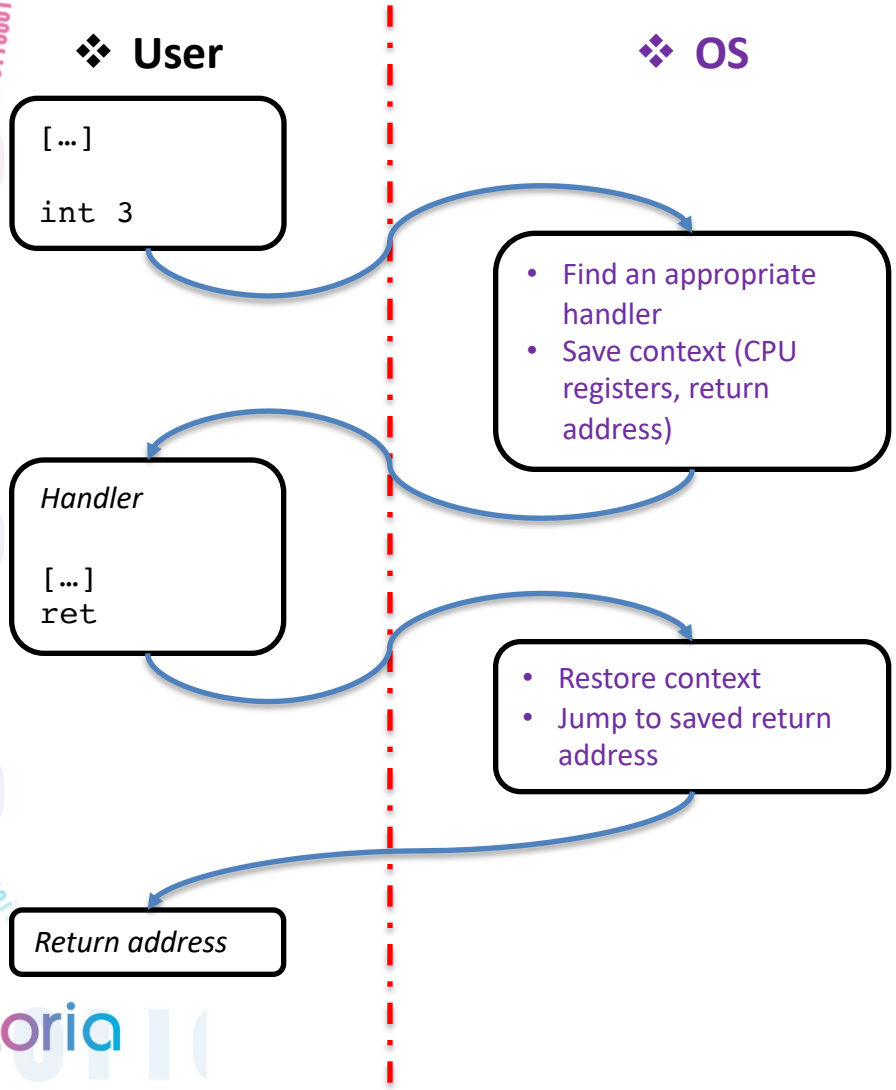


Dynamic jump example



- Jump targets: 0x8 or 0x9 or 0xA or 0xB

Exception handler



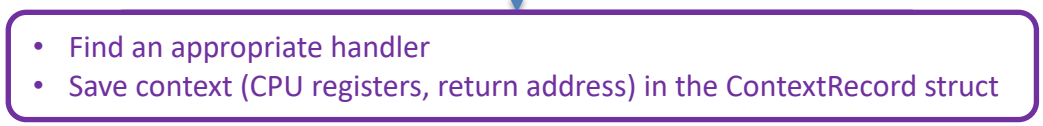
- **Possible corruptions**
  - Context registers tampering
  - Return address tampering
  - Bypass context restoration
- **Detect corruptions**
  - Is the saved context read and/or modified?
  - Is the saved return address modified?
  - Is the handler give back the control to the OS?

Corrupted exception handler example

```
A: mov ecx, 0x3
B: mov eax, 0x4
C: int 3
```

Corrupted exception handler example

```
A: mov ecx, 0x3
B: mov eax, 0x4
C: int 3
```



- Find an appropriate handler
- Save context (CPU registers, return address) in the ContextRecord struct

Loria

Corrupted exception handler example

```
A: mov ecx, 0x3
B: mov eax, 0x4
C: int 3
```

- Find an appropriate handler
- Save context (CPU registers, return address) in the ContextRecord struct

```
// Handler entry

X: mov eax, [esp + 0x0C]      // eax = *ContextRecord
Y: mov [eax + 0xB8], P        // ContextRecord[eip] = P
Z: ret                        // give back the control to the OS

// Handler exit
```

Loria

Corrupted exception handler example

```
A: mov ecx, 0x3
B: mov eax, 0x4
C: int 3
```

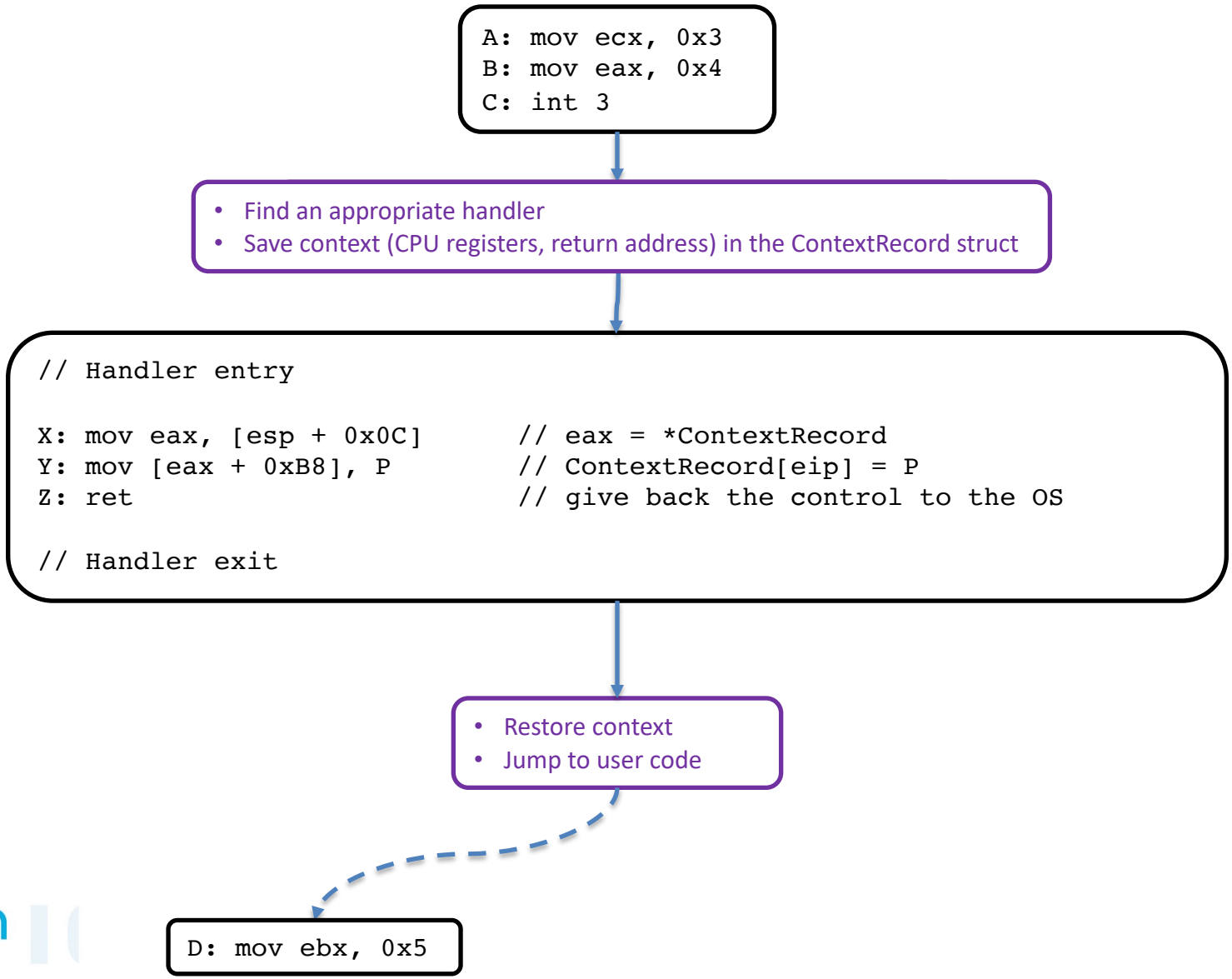
- Find an appropriate handler
- Save context (CPU registers, return address) in the ContextRecord struct

```
// Handler entry

X: mov eax, [esp + 0x0C]      // eax = *ContextRecord
Y: mov [eax + 0xB8], P        // ContextRecord[eip] = P
Z: ret                        // give back the control to the OS

// Handler exit
```

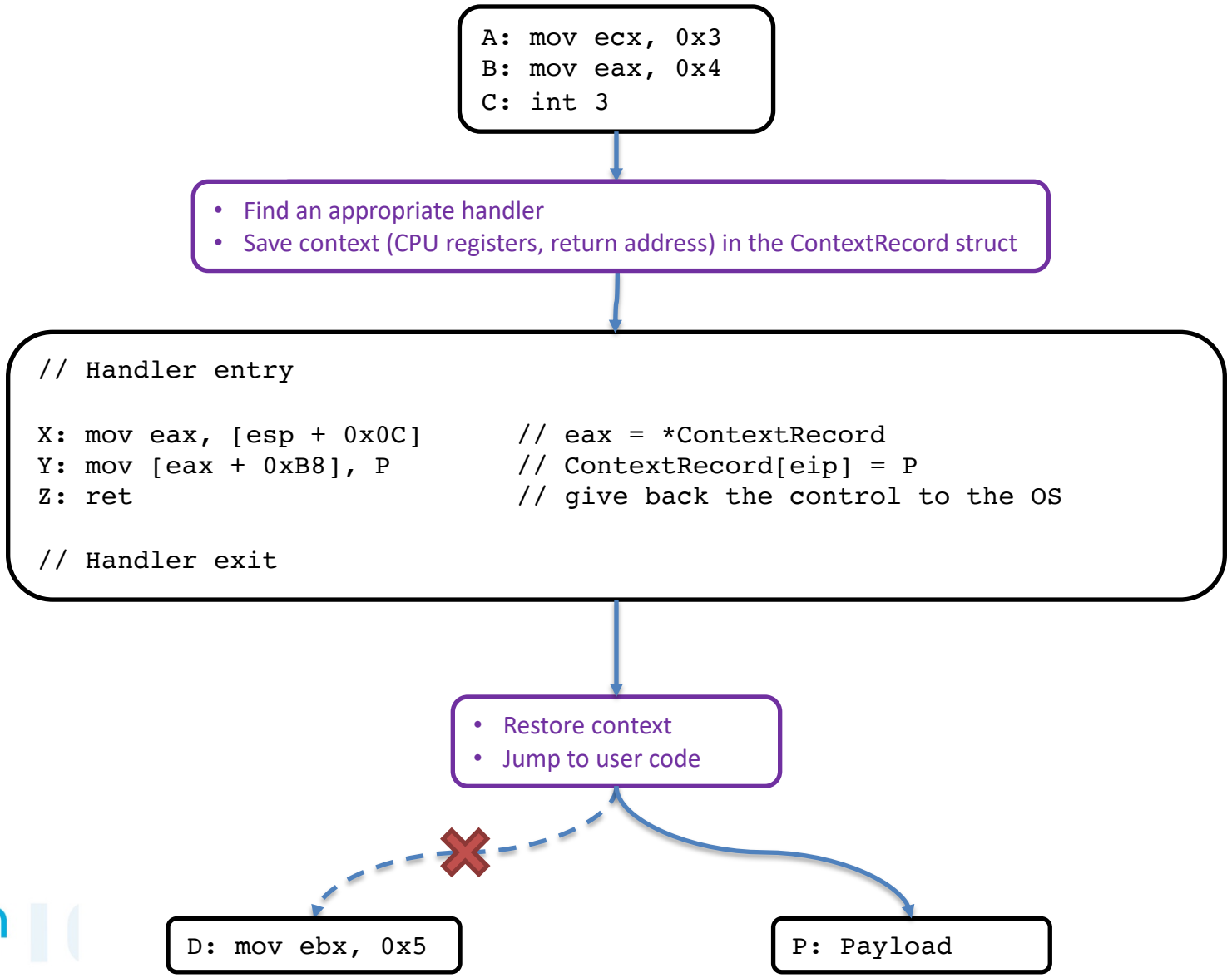
- Restore context
- Jump to user code

Loria

Corrupted exception handler example



Corrupted exception handler example



Others objectives

- Detect **Call Stack Tampering**

  1. Keep in mind each `call` seen during propagation step
  2. Find `ret` target(s) (as for dynamic jumps)
  3. Compare `ret` target(s) addresses with return address pushed by the last `call` seen

- Detect **Self modification**

  1. Keep in mind each disassembly address along with its opcode (tagged as *code*)
  2. Monitor each instruction who write at an address tagged as *code* and replace the tag by *tampered code*
  3. Raise a flag if a *tampered code* address is executed

- Detect **Opaque Predicate**

  - For each basic block ended with a condition jump instruction use the invariant information in order to conclude about the boolean condition

Loria

Conclusion: Work in progress

- Benchmarks
  - Non obfuscated C programs
  - Windows commercial packers
  - Crackme obfuscated with Tigress
- Only working on ELF and PE binaries
- ARM support?
- Need to improve library calls

Loria

Thank you