Digital Forensics
Lecture 4 - Reverse Engineering

Deciphering Code Structures

Akbar S. Namin
Texas Tech University
Spring 2018

Reverse Engineering – Deciphering Code Structures

- Goal:
  - Learn about the most common logical and control flow constructs used in high-level languages
  - How they are implemented in assembly languages including loops, and conditional blocks
- It helps to know about the types of typical assembly language sequences usually observed in assembly language code

- An example of a conditional codes
  - "je" will cause a jumpt to "SomePlace" if "EAX" equals to 7

```
cmp
eax, 7
je
SomePlace
```

Reverse Engineering – Deciphering Code Structures

- Control flow and program layout
  - Branches are the most popular tools for implementing logic of the program
  - Most common programming constructs
  - Identifying branches and understanding their meaning and purpose is important in reverse engineering

- Deciphering functions
  - Basic building block in a program
  - On I-32 bits processors function are called using the “CALL” instruction
    - It stores the current instruction pointer in the stack and jumps to the function address
    - It helps to distinguish it from unconditional jumps.

Reverse Engineering – Deciphering Code Structures

- Internal functions
  - They are called from the same binary executable that contains their implementation
  - Compilers generate an internal function call sequence
  - They embed the function's address into the code (easy to detect)
  - E.g., "Call CodeSectionAddress"

Reverse Engineering – Deciphering Code Structures

- Imported functions
  - When a module is making a call into a function implemented in another binary executable
  - During the compilation process the compiler has no idea where the imported function can be found
    - Unable to embed the function's address into the code
  - "Imported Address Table" and "Import Directory" are usually used to implement calling imported functions
    - The Import Directory is used in runtime for resolving the function's name with a matching function in the target executable
    - IAT stores the actual address of the target function
    - The caller then loads the function's pointer from the IAT and calls it

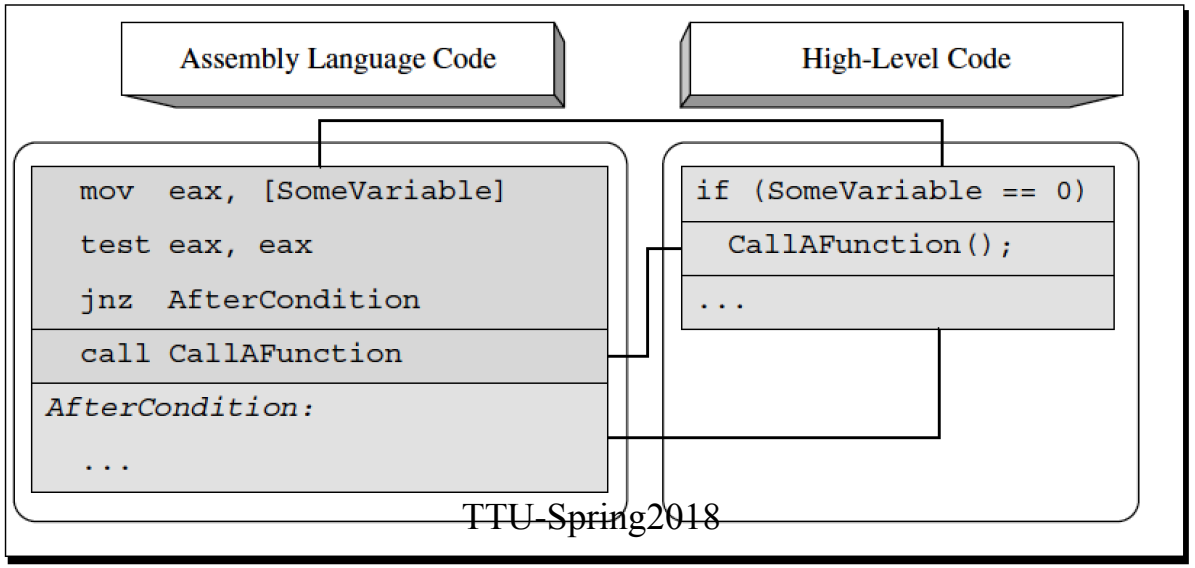
Reverse Engineering – Deciphering Code Structures

- Imported functions
  - An example of a typical imported function call
    - "call DWORD PTR [IAT_Pointer]
    - "DWORD PTR" tells the CPU to jump not to the address of IAT_Pointer but to the address that is pointed to by IAT_Pointer
  - Detecting imported calls is easy because of these types of calls

Reverse Engineering – Deciphering Code Structures

```
if (SomeVariable == 0)
    CallAFunction();
```

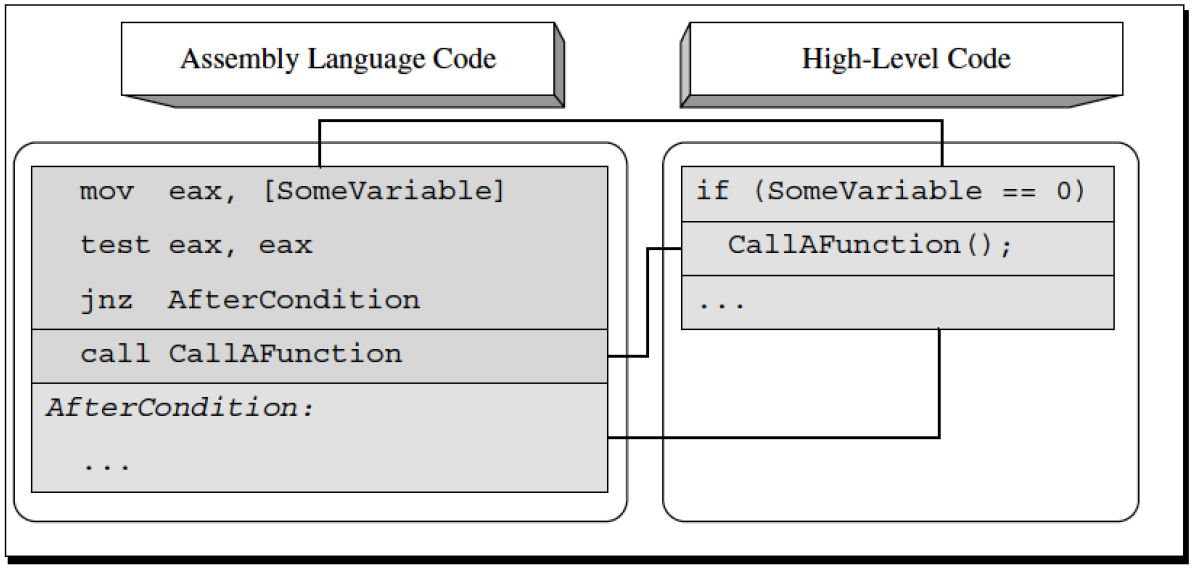
- Single-Branch conditionals
  - Requires a logical check to determine whether "SomeVariable" contains "0" or not
  - Skip the conditional block by performing a conditional jump if "SomeVariable" is nonzero
  - Use a simple TEST to perform a simple zero check for EAX.
    - A bitwise AND operation on EAX and setting flags to reflect the result
    - An effective way : TEST sets the zero flag (ZF) with respect to the result of the bitwise AND operation



Reverse Engineering – Deciphering Code Structures

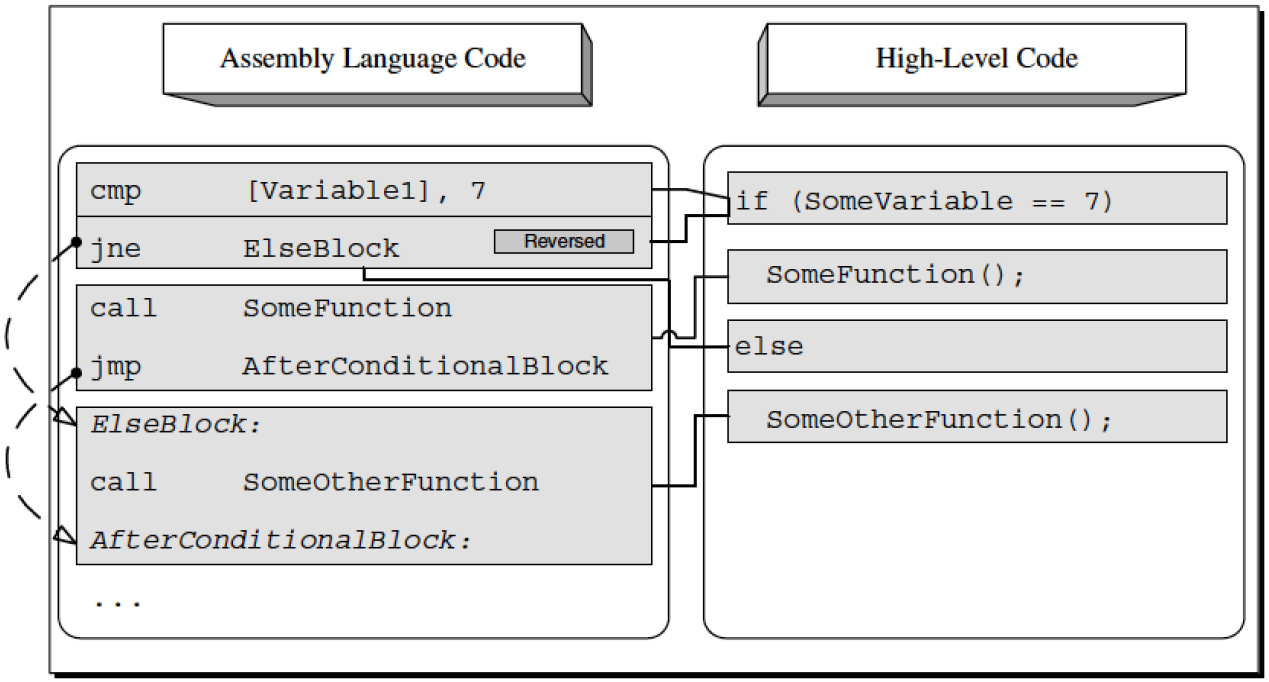
```
if (SomeVariable == 0)
    CallAFunction();
```

- Single-Branch conditionals
  - Note. The condition is reversed.
    - In the source code: the program checks whether "SomeVariable" equals to "zero"
    - The compiled version is reversing the condition: The conditional instruction checks whether "SomeVariable" is nonzero



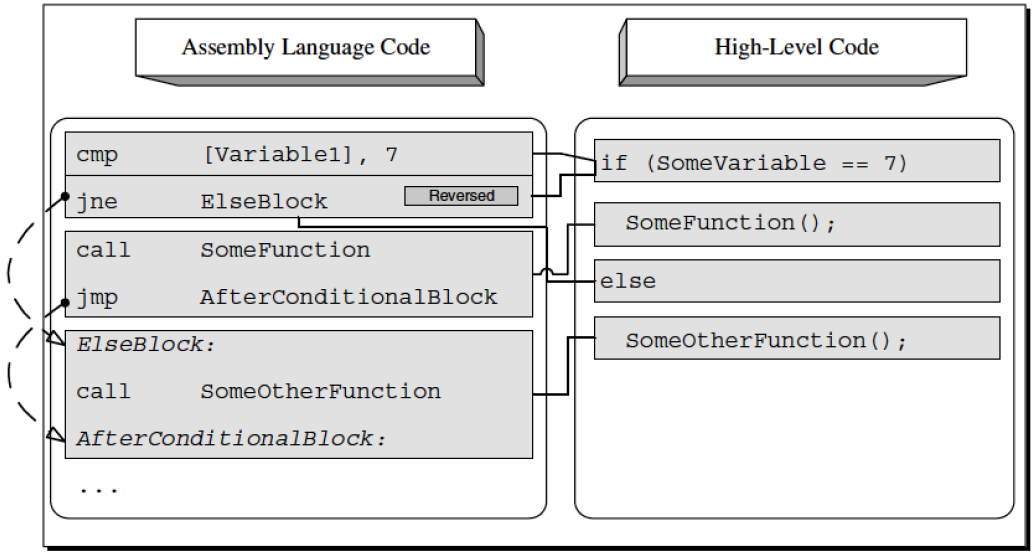
Reverse Engineering – Deciphering Code Structures

- Two-way conditionals
  - Typically implemented in high-level languages using (if-else)



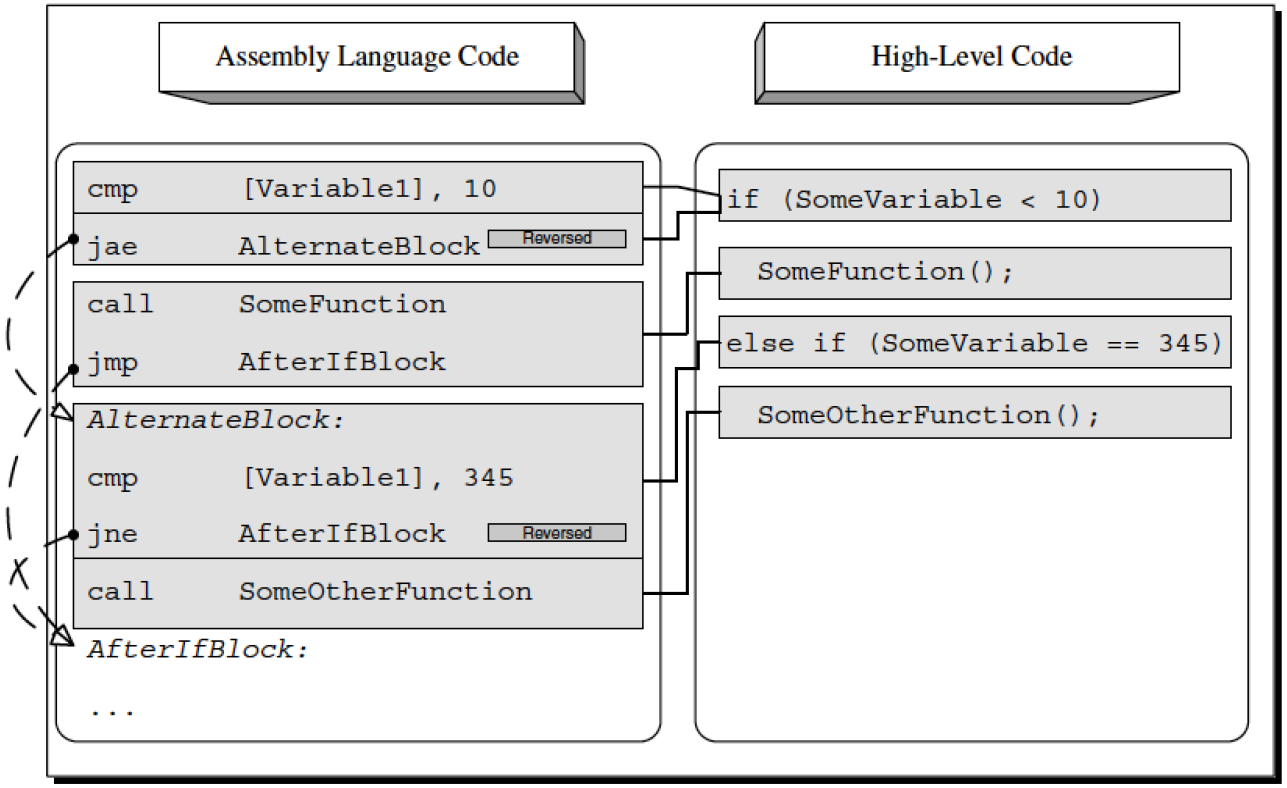
Reverse Engineering – Deciphering Code Structures

- Two-way conditionals
  - How compiler implements two-way conditionals?
    - The conditional branch points to the "else" block and not to the code follows the conditional statement
    - The primary conditional block is placed right after the conditional jump
    - The conditional block always ends with an unconditional jump that essentially skips the "else" block
    - The "else" block is placed at the end of the conditional block, right after that conditional jump.
    - The unconditional "JMP" jumps after the functional call



Reverse Engineering – Deciphering Code Structures

- Two-way conditionals – A more complex case
  - A separate conditional statement is used for each of the two code blocks



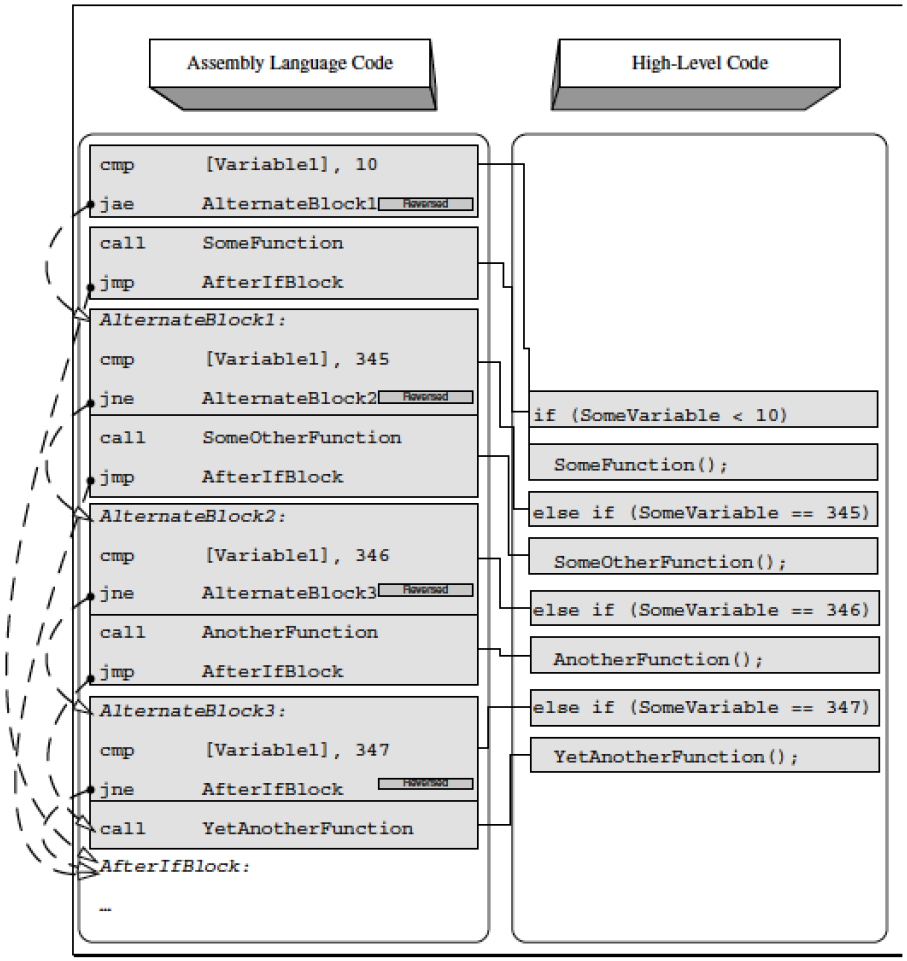
Reverse Engineering – Deciphering Code Structures

- Multiple-alternative conditionals
  - Maybe implemented using "switch"
  - The compiler adds additional

"alternate blocks" that consists of:

1. One or more logical checks
2. The actual conditional code block
3. The final JMP that skips to the end of

The entire block.



Reverse Engineering – Deciphering Code Structures

- Compound conditions with logical operator &&
  - In assembly language:

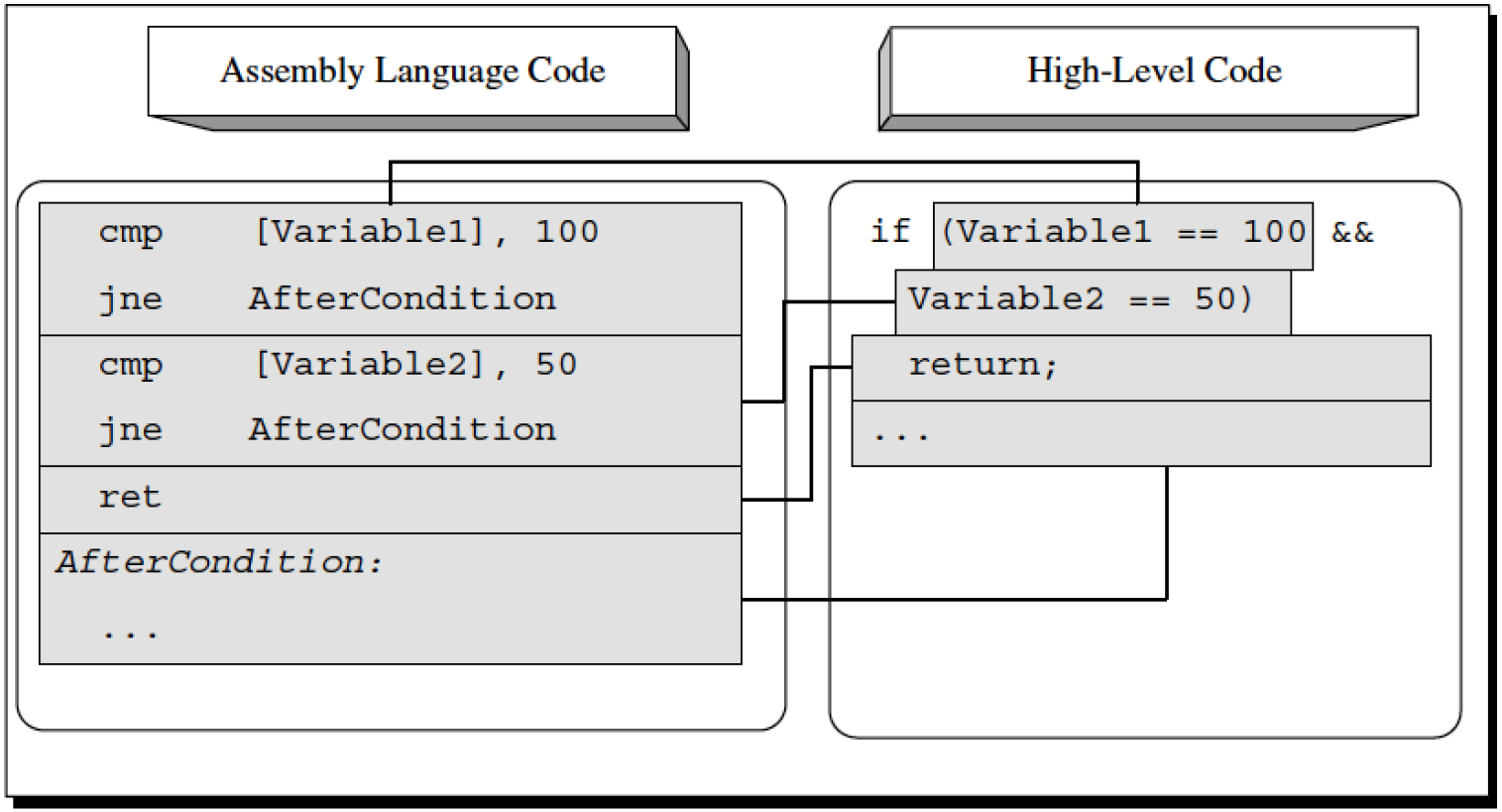
```
cmp     [Variable1], 100
jne     AfterCondition
cmp     [Variable2], 50
jne     AfterCondition
ret
AfterCondition:
...
```

  - The equivalent high-level language

```
if (Variable1 == 100 && Variable2 == 50)
                                    return;
```

Reverse Engineering – Deciphering Code Structures

- Compound conditions with logical operator &&



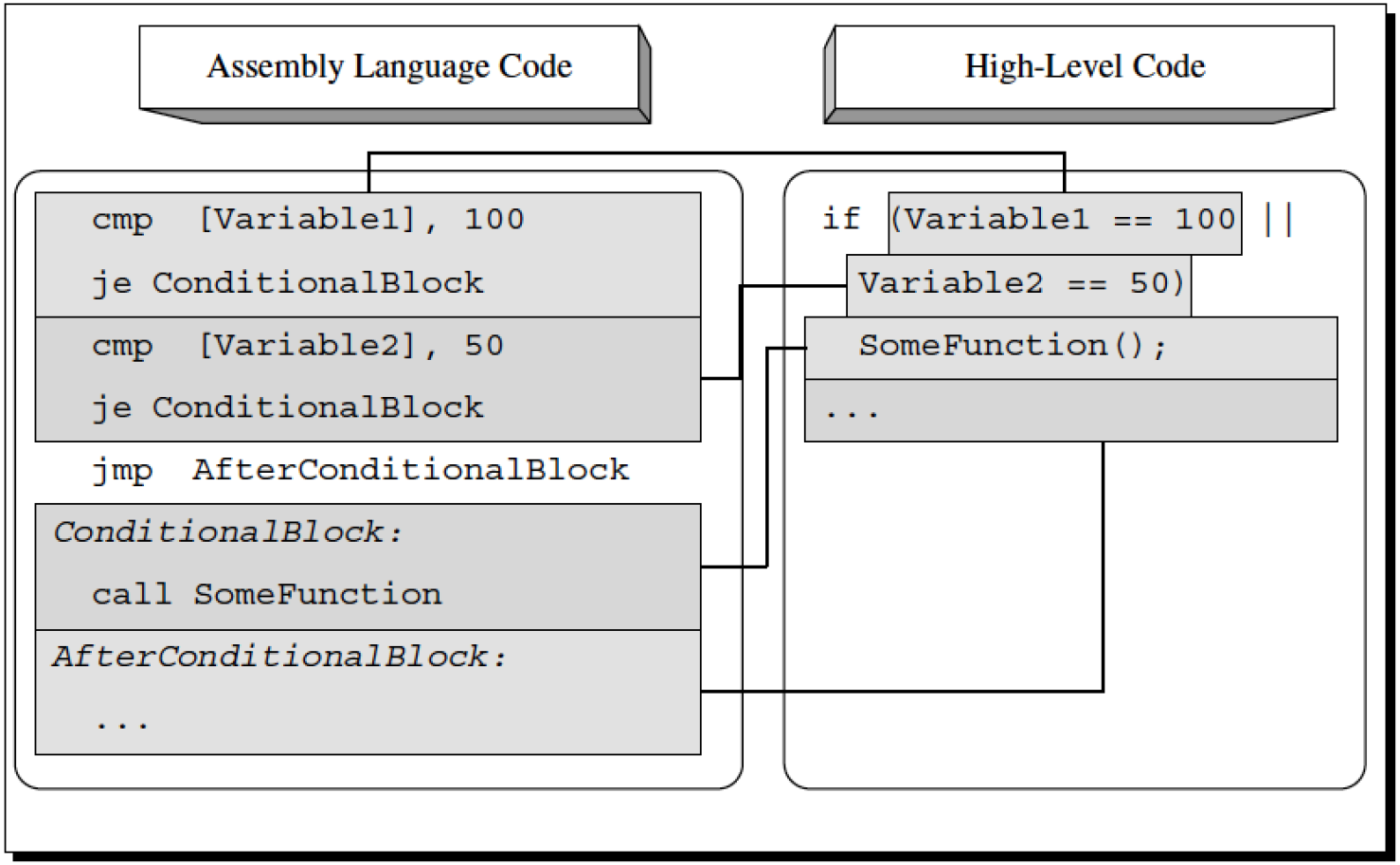
Reverse Engineering – Deciphering Code Structures

- Compound conditions with logical operator ||
  - The assembly language version

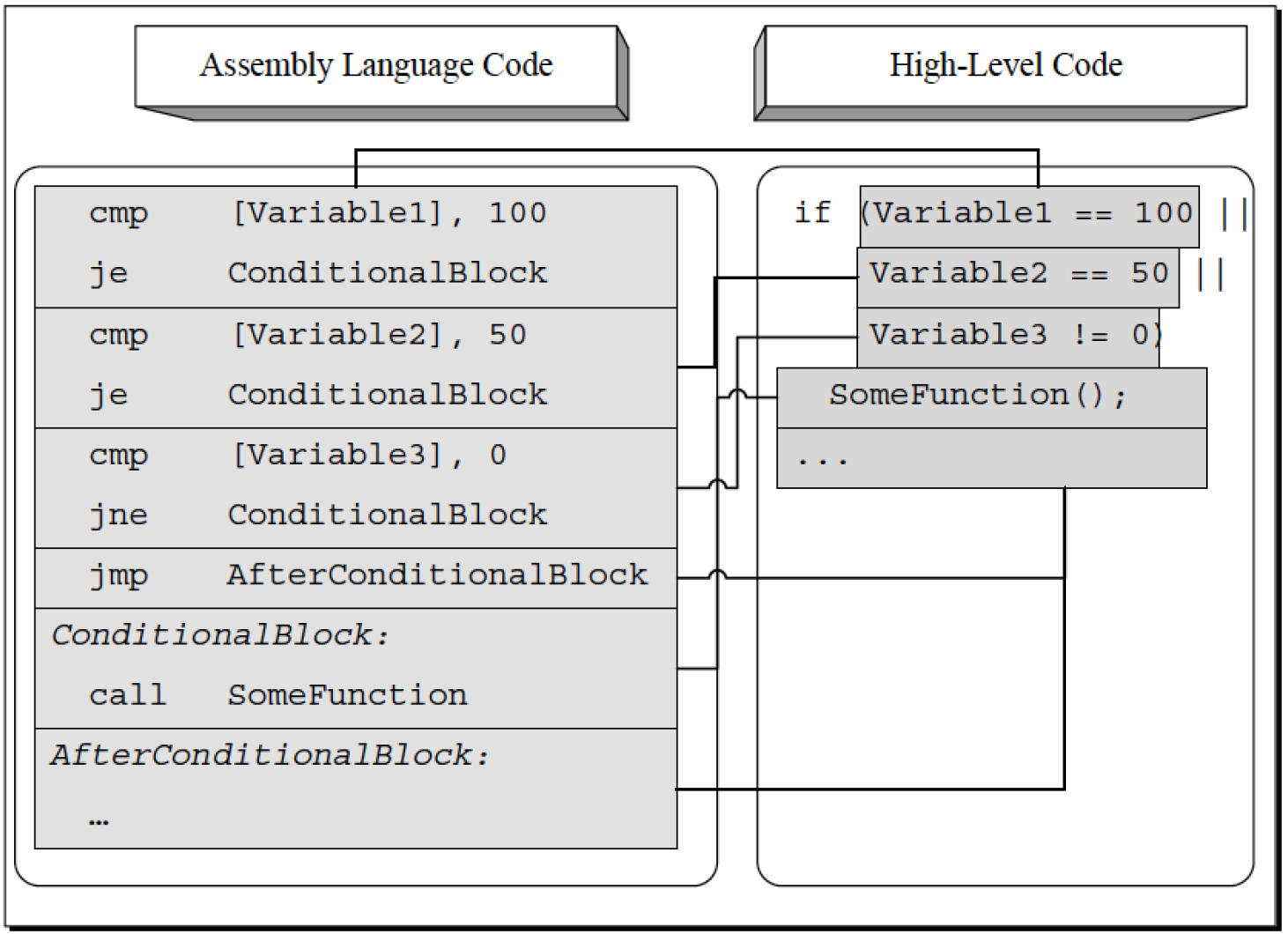
```
cmp                         [Variable1], 100
je                          ConditionalBlock
cmp                         [Variable2], 50
je                          ConditionalBlock
jmp                         AfterConditionalBlock
ConditionalBlock:
call     SomeFunction
AfterConditionalBlock:
...
```

Reverse Engineering – Deciphering Code Structures

- Compound conditions with logical operator ||
  -

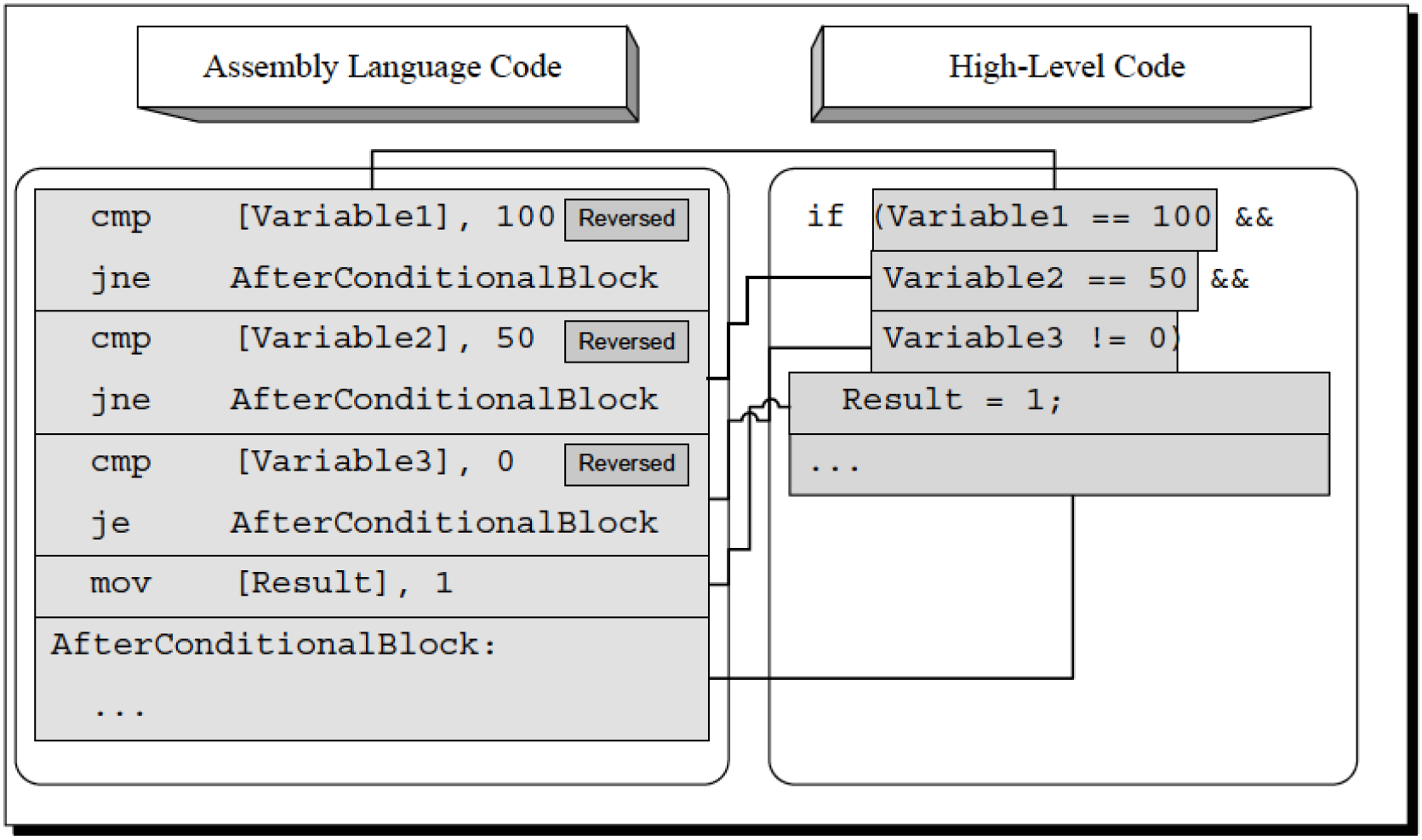


Reverse Engineering – Deciphering Code Structures

- Compound conditions with three conditions combined using OR
  - 

Reverse Engineering – Deciphering Code Structures

- Compound conditions with three conditions combined using AND



Reverse Engineering – Deciphering Code Structures

- Loops
  - A chunk of conditional code that is repeatedly executed until the condition is no longer satisfied
- Pretested loops
  - To present: a pretest loop the assembly language code must contain two jump instructions:
    - A conditional branch instruction in the beginning
    - An unconditional jump at the end that jumps back to the beginning of the loop

```
c = 0;
while (c < 1000)
{
    array[c] = c;
    c++;
}
```

```
mov         ecx, DWORD PTR [array]
xor         eax, eax
LoopStart:
mov         DWORD PTR [ecx+eax*4], eax
add         eax, 1
cmp         eax, 1000
jl          LoopStart
```

References

- Appendix A – Reversing the Secrets of Reverse Engineering (Eldad Eilam)