



Memory corruption attacks: Defenses and Challenges

Hyoungshick Kim

Department of Software

College of Software

Sungkyunkwan University

Content

- Input validation
- Detecting overflows with canaries
- Address Space Layout Randomization (ASLR)
- Return Oriented Programming (ROP)
- Control Flow Integrity (CFI)

Input validation

- Consider `strcpy(buffer, argv[1])`

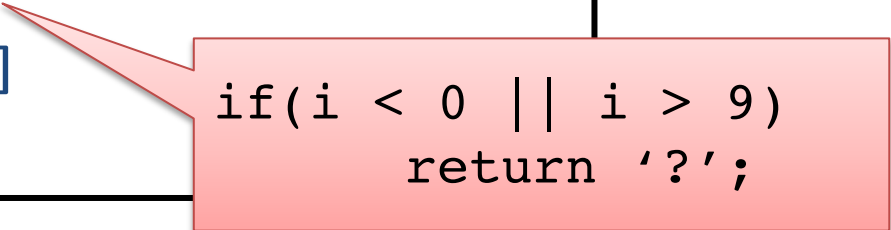
- A buffer overflow occurs if

`len(buffer) < len(argv[1])`

- Software must **validate** the input by checking the length of `argv[1]`

How can we fix this?

```
char digit_to_char(int i) {  
    char convert[] = "0123456789";  
    return convert[i]  
}
```



```
if(i < 0 || i > 9)  
    return '?';
```

Make no assumptions!

- Don't assume the user won't enter more than 512 characters on the command line
- Don't assume that there will always be enough disk space
- Codify and enforce your assumptions

Enforce input validation

- “user input” is where the user (malicious or innocent) can directly impact the program
 - ✓ Always verify the user input
 - ✓ White-listing expected results
- Input can come from a number of places
 - ✓ Text entry
 - ✓ Configuration files
 - ✓ Environment variables
 - ✓ Network

Use safe string functions

- **... for string-oriented functions**

- ✓ strcat -> strlcat
- ✓ strcpy -> strlcpy
- ✓ strncat -> strlcat
- ✓ strncpy -> strlcpy
- ✓ sprintf -> snprintf
- ✓ vsprintf -> vsnprintf
- ✓ gets -> fgets

- **Microsoft versions**

- ✓ strcpy_s, strcat_s, ...

Examples of safe string functions

- Traditional string library routines assume target buffers have sufficient length

```
char str[4];  
char buf[10] = "fine";  
strcpy(str, "hello"); // overflows str  
strcat(buf, "day to you"); // overflows buf
```

- Safe versions check the destination length

```
char str[4];  
char buf[10] = "fine";  
strncpy(str, "hello", sizeof(str)); //fails  
strncat(buf, "day to you", sizeof(buf)); //fails
```

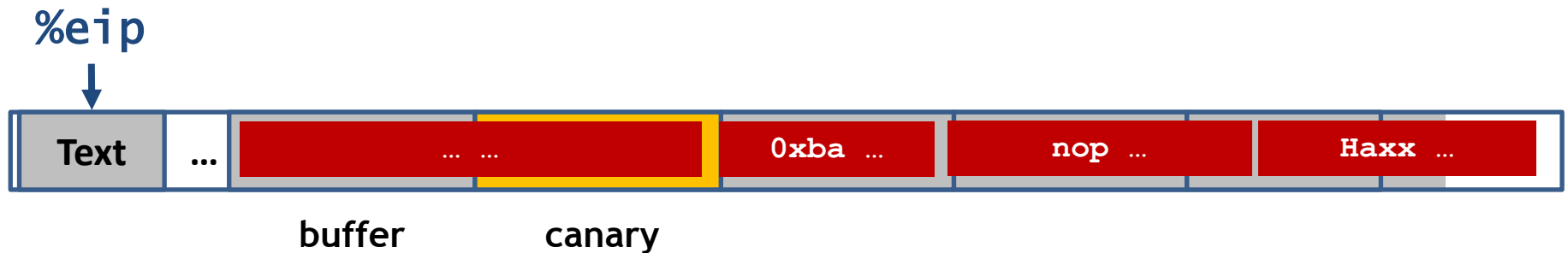

Detecting overflows with canaries

- 19th century coal mine integrity
 - Is the mine safe?
 - Bring in a canary
 - If it dies, abort!

***We can do the same
for stack integrity***



Detecting overflows with canaries



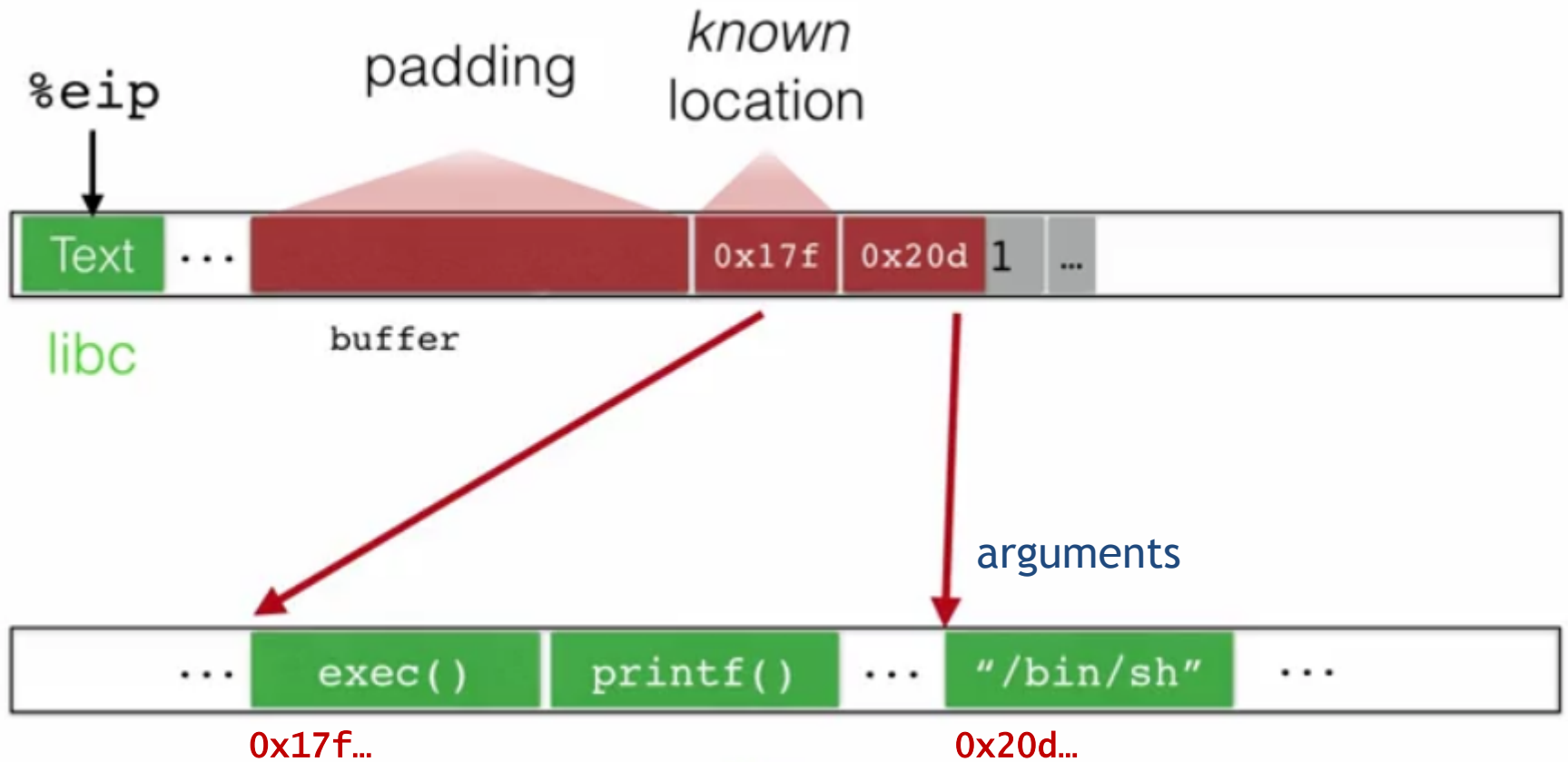
Linker checks the existence of **canary**

Not the expected value: abort

Write or execute, but not both

- No program segment loaded into memory is both writable and executable
- Employ **non-executable stack** ("No execute" **NX bit**)
- Return-to-Libc attack appeared (to call system function used to execute shell commands)
- Return-oriented programming (ROP) to execute chosen machine instruction sequences called "gadgets" (see <https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf>)

Return-to-Libc



ASLR is used to hide the location of system library

ASLR today

- Address Space Layout Randomization (ASLR)
 - ✓ Prevents an attacker from predicting information needed for correctly changing information flow towards the desirable computation (see “On the effectiveness of Address Space Randomization”)
- Caveats:
 - ✓ Only shifts the offset of memory areas
 - ✓ May not apply to program code, just libraries
 - ✓ Need sufficient randomness, or can brute force
 - 32-bit systems typically offer 16 bits = 65536 possible starting positions; sometimes 20 bits. Shacham demonstrated a brute force attack could defeat such randomness in 216 seconds (on 2004 hardware)
 - 64-bit systems more promising, e.g., 40 bits possible

Return Oriented Programming

- Introduced by Hovav Shacham in 2007
- The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), CCS'07
- Idea: rather than use a single (libc) function to run your shellcode, string together pieces of existing code, called gadgets, to do it instead
- Challenges
 - Find the gadgets you need
 - String them together

Approach

- A gadget is a set of instructions for carrying out a semantic action
 - Gadgets are instruction groups that end with `ret` (mostly)
- ROP attacks
 1. Disassemble code
 2. Identify useful code sequences as gadgets
 3. Assemble gadgets into a desired shellcode

How to find gadgets

- How can we find gadgets to construct an exploit?
 - **Automate a search of the target binary for gadgets** (look for ret instructions, work backwards)
 - Cf. <https://github.com/0vercl0k/rp>
- Are there sufficient gadgets to do anything interesting?
 - Yes: Shacham found that for significant codebases (e.g., libc), gadgets are **Turing complete**
 - Especially true on x86's dense instruction set

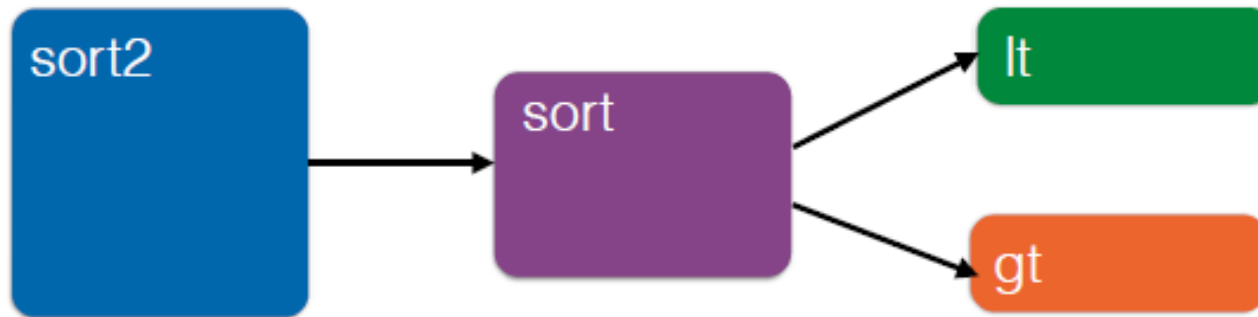
Behavior-based detection

- Stack canaries, non-executable data, and ASLR aim to complicate various steps in a standard attack
 - But they still may not stop it
- Idea: **observe** the program's **behavior** — **is it doing what we expect it to?**
 - If not, might be compromised
 - Challenges
 - How to define “expected behavior”
 - How to detect deviations from expectation efficiently
 - How to avoid compromise of the detector

Call graph

```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}
```

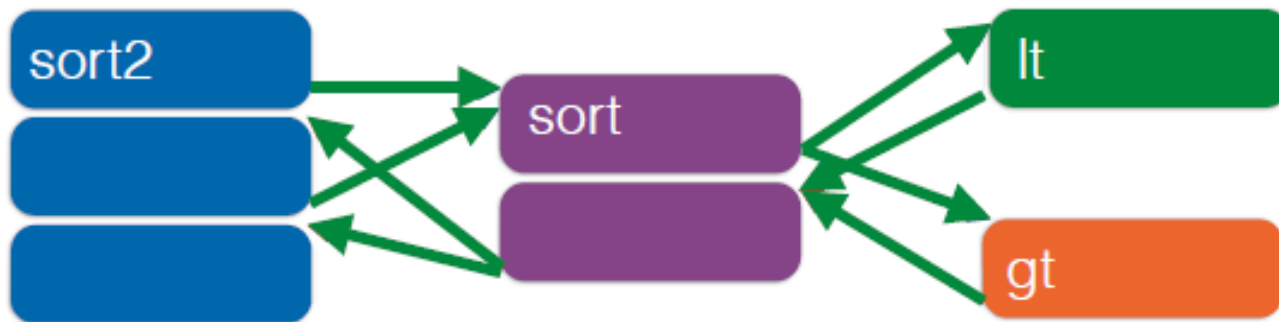


Which functions call other functions

Control flow graph

```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```



Break into **basic blocks**
Distinguish **calls** from **returns**

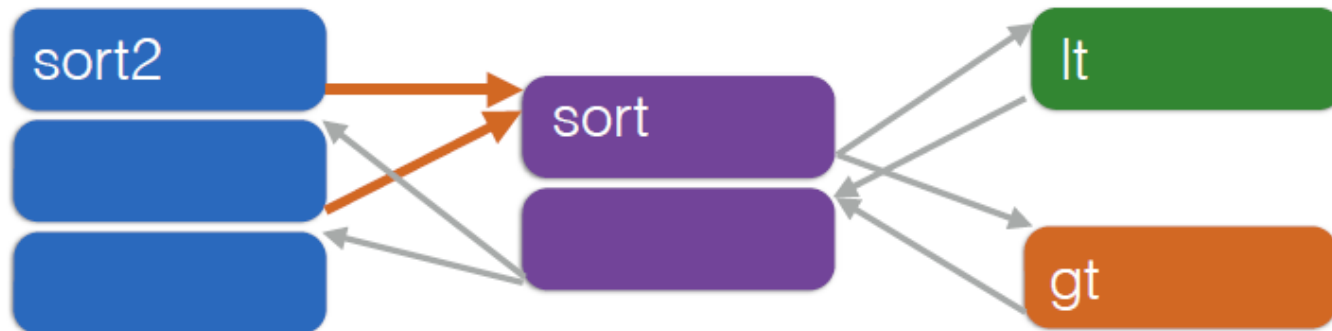
CFI: Compliance with CFG

- **Compute the call/return CFG** in advance
 - During compilation, or from the binary
- **Monitor the control flow** of the program and ensure that it only follows paths allowed by the CFG
- Observation: **Direct calls need not be monitored**
 - Assuming the **code** is **immutable**, the target address cannot be changed
- Therefore: **monitor only indirect calls**
 - jmp, call, ret with **non-constant** targets

Direct transfer

```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}
```

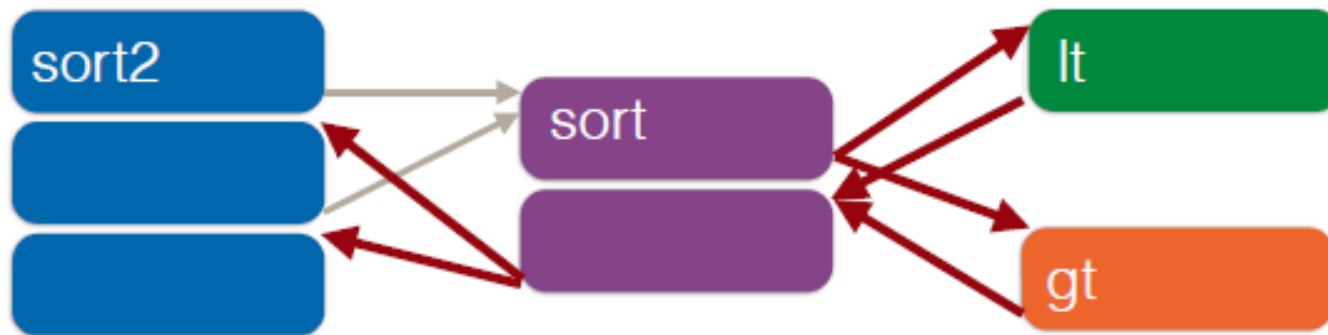


Direct calls (always the same target)

Indirect transfer

```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}
```



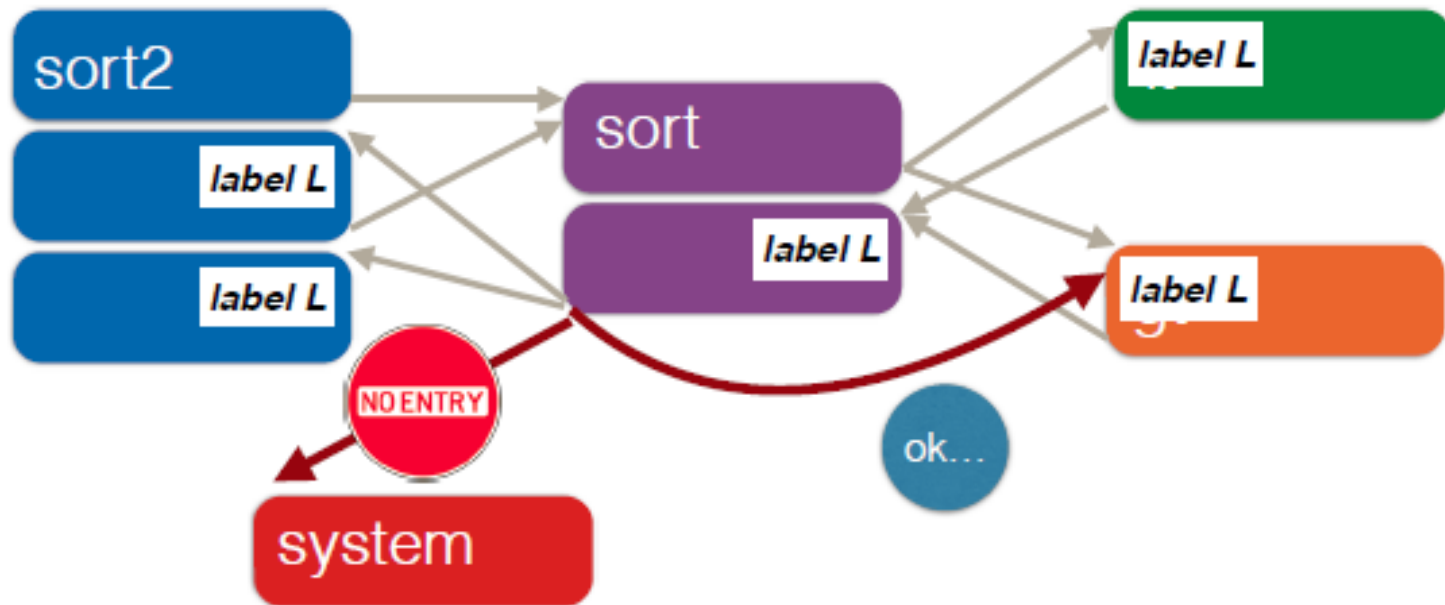
Indirect transfer (call via register, or `ret`)

In this case, the address can be changed.

In-line monitor

- Implement the monitor in-line, as a **program transformation**
- Insert a **label just before the target address** of an indirect transfer
- Insert **code to check the label of the target** at each indirect transfer
 - Abort if the label does not match
- The **labels are determined by the CFG**

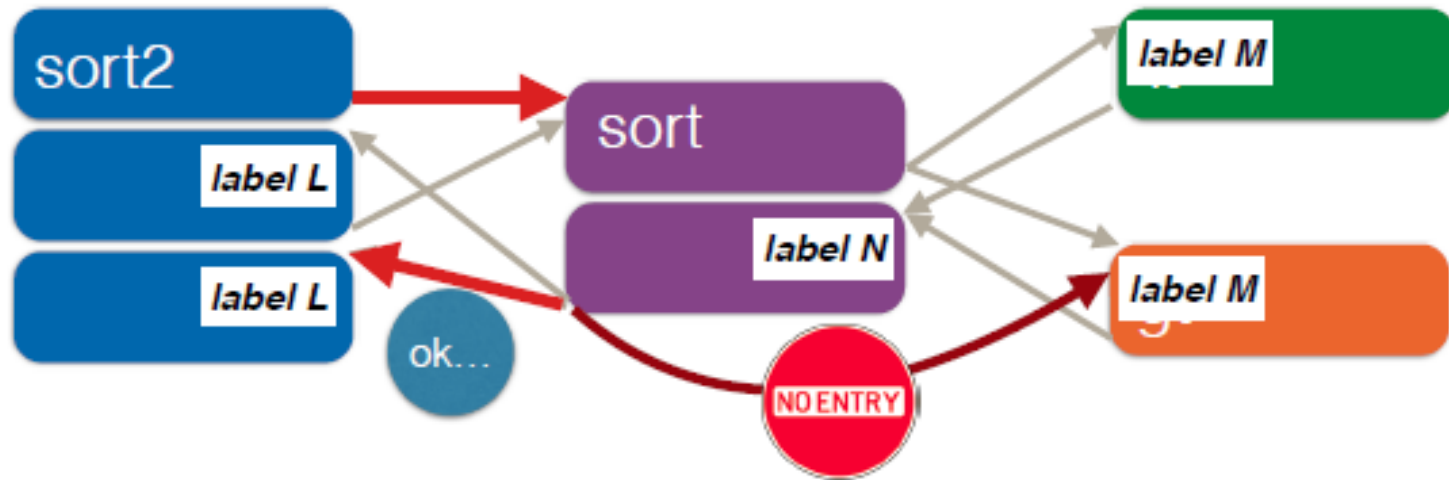
Simplest labeling



Use the same label at all targets

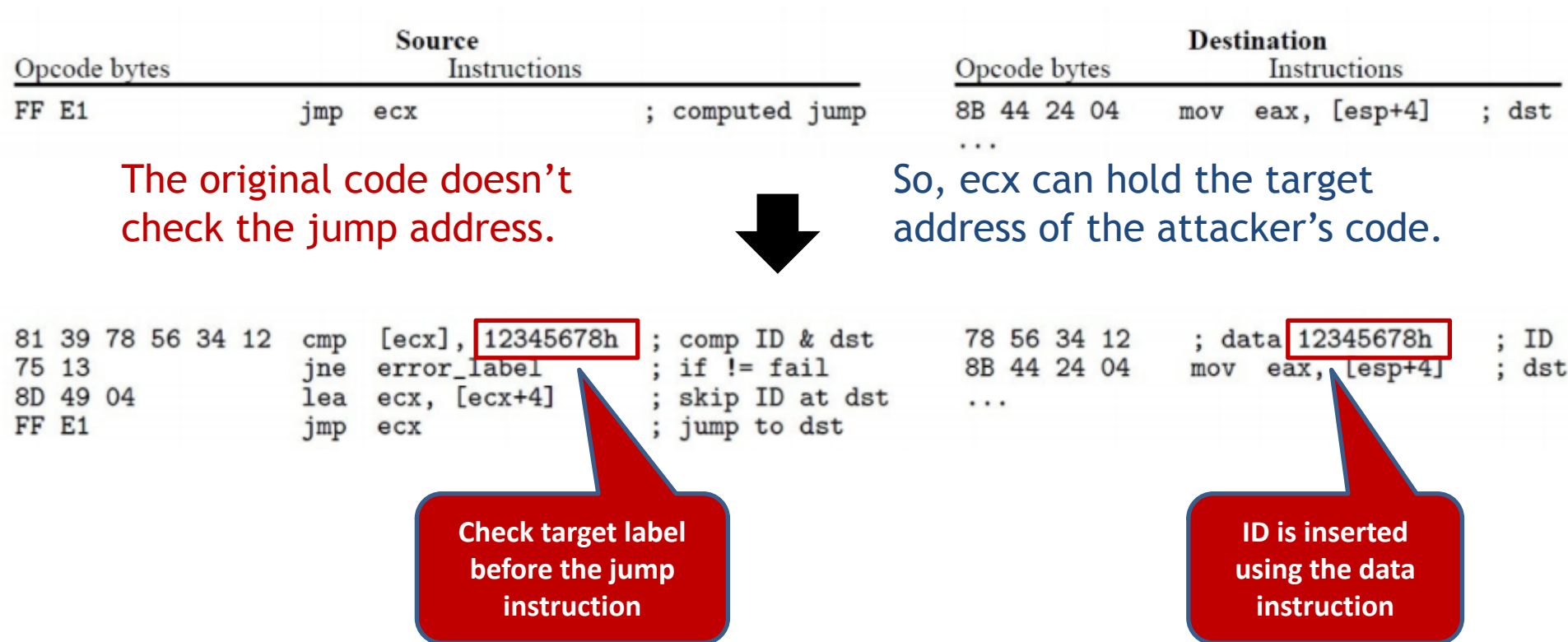
However, a wrong block with the same label can be improperly called.

Detailed labeling



- We use different labels for sort2 and lt/gt blocks
- More fine-grained CFI is possible
- Still permits call from sort to return to lt instead of gt

How to implement CFI



If this checks fails, it jumps to the error label.

Efficiency of CFI

- Classic CFI (2005) imposes 16% overhead on average, 45% in the worst case
 - Works on arbitrary executables
 - Not modular (no dynamically linked libraries)
- Modular CFI (2014) imposes 5% overhead on average, 12% in the worst case
 - C only (part of LLVM)
 - Modular, with separate compilation
 - <http://www.cse.psu.edu/~gxt29//projects/mcfi.html>

Can we defeat CFI?

- Inject code that has a legal label
 - Won't work because we assume non-executable data
- Modify code labels to allow the desired control flow
 - Won't work because the code is immutable

CFI assurances

- CFI defeats control flow-modifying attacks
 - Remote code injection, ROP/return-to-libc, etc.
- But CFI cannot prevent **manipulation of control-flow** that is **allowed by the labels/graph**
 - Called **mimicry attacks**
 - The simple, single-label CFG is susceptible to these
- **Nor data leaks or corruptions**
 - Heartbleed would not be prevented
 - Nor the authenticated overflow
 - Control modification is allowed by graph

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, str);
    if(authenticated) { ...
}
```

Corrupting data

- Attackers can still **overflow data** to
 - **modify a secret key** to be one known to the attacker, to be able to decrypt future intercepted messages
 - **modify state variables** to bypass authorization checks
 - **modify interpreted strings** used as part of commands

Questions?

