Software Exploitation ASLR & Beyond

Georgios (George) Portokalidis

More on ASLR

ASLR and Code

- For ASLR to be applied to code it needs to be position independent
  - Position relative addressing needs to be used to refer to other code or global data
- Libraries → Position Independent Code (PIC)
- Executables → Position Independent (PIE)
- How to address data in PIC or PIE?
  - 32-bit x86 → GETPC code is introduced to get current PC
  - 64-bit x86 → RIP-relative address is available
- Both → relocation metadata is used to patch pointers
  - Display them using `readelf -r elf`

Gradual Adoption

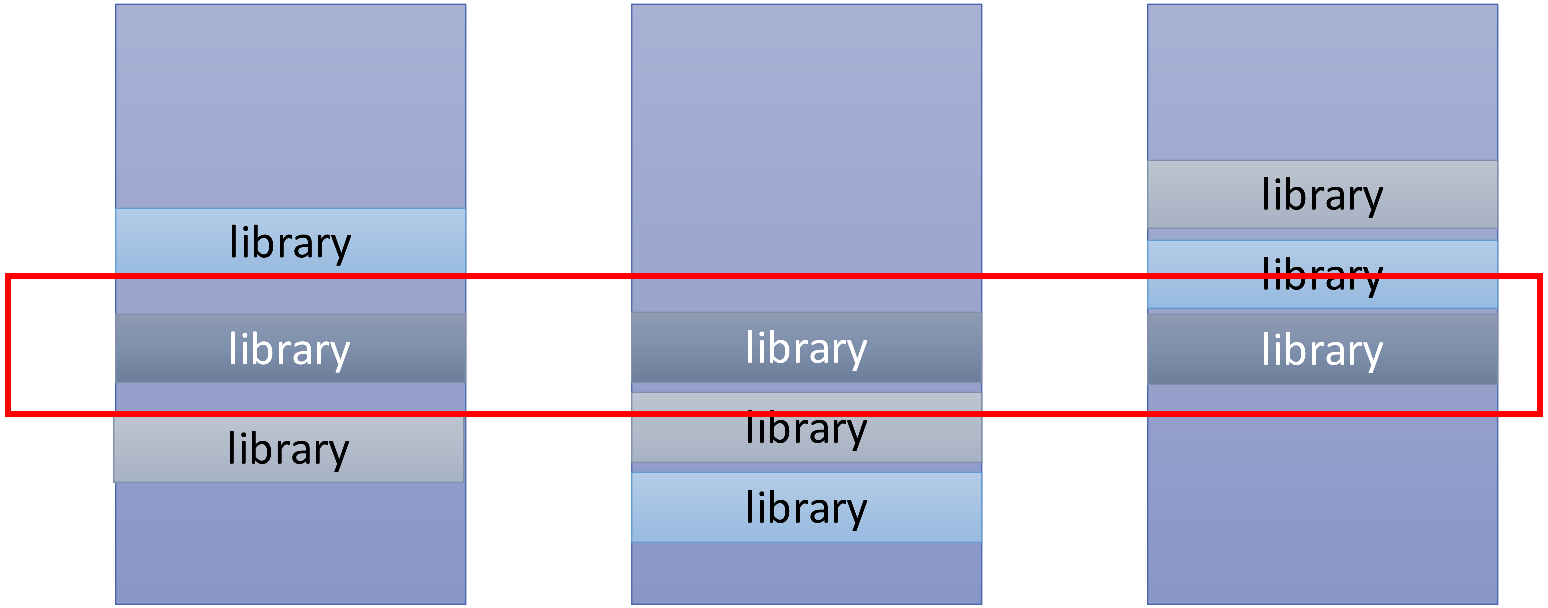
- Libraries were first to become PIC
- Executables followed later

Distribution	Tested Binaries	PIE Enabled	Not PIE
Ubuntu 12.10	646	111 (17.18%)	535
Debian 6	592	61 (10.30%)	531
CentOS 6.3	1340	217 (16.19%)	1123

Percentage of PIE binaries in different Linux distributions

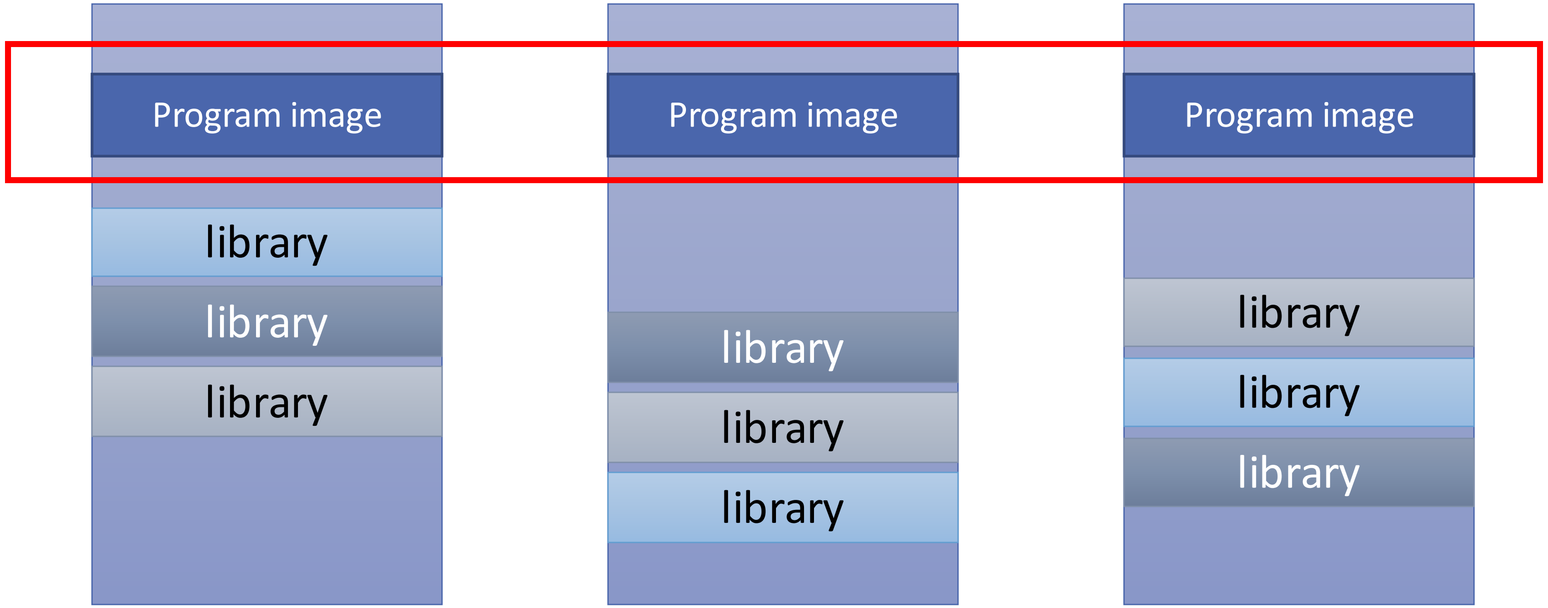
Exploiting the Weakest Link

- A single non-randomized library may be enough



Exploiting the Weakest Link

- Do not forget the program image



Ret2libc in a non-PIE Executables

- Take advantage of the way shared libraries are linked and called

- Key components:
  - Procedure-linkage table (PLT)
  - Global-offset table (GOT)

Calls to Shared Library Functions

- Calls to external libraries are redirect to a stub in the PLT

```
400720:      e8 db fd ff ff          callq  400500 <puts@plt>
40064d:      e8 ce fe ff ff          callq  400520 <printf@plt>
```

- Each PLT entry consists of 3 instructions

```
0000000000400500 <puts@plt>:
  400500:      ff 25 12 0b 20 00       jmpq   *0x200b12(%rip)
  400506:      68 00 00 00 00          pushq  $0x0
  40050b:      e9 e0 ff ff ff          jmpq   4004f0 <.plt>

0000000000400520 <printf@plt>:
  400520:      ff 25 02 0b 20 00       jmpq   *0x200b02(%rip)
  400526:      68 02 00 00 00          pushq  $0x2
  40052b:      e9 c0 ff ff ff          jmpq   4004f0 <.plt>
```

Calls to Shared Library Functions

- Jump using pointer stored in the GOT
- First run → pointer points to PLT itself
- The other two instructions invoke the linker and call the function

```
0000000000600ac0 <_GLOBAL_OFFSET_TABLE_>:
puts@plt+6
...
```

```
0000000000400500 <puts@plt>:
  400500:       ff 25 12 0b 20 00       jmpq   *0x200b12(%rip)
  400506:       68 00 00 00 00          pushq  $0x0
  40050b:       e9 e0 ff ff ff          jmpq   4004f0 <.plt>
```

Calls to Shared Library Functions

- Jump using pointer stored in the GOT
- Second run → pointer in GOT points to actual function

```
0000000000600ac0 <_GLOBAL_OFFSET_TABLE_>:
puts
...
```

```
0000000000400500 <puts@plt>:
  400500:       ff 25 12 0b 20 00       jmpq   *0x200b12(%rip)
  400506:       68 00 00 00 00          pushq  $0x0
  40050b:       e9 e0 ff ff ff          jmpq   4004f0 <.plt>
```

Ret2libc in non-PIE Executables

- Return-to-PLT
  - Does not matter if function has been called, but needs to have a PLT entry

```
0000000000600ac0 <_GLOBAL_OFFSET_TABLE_>:
puts
...
```

```
0000000000400500 <puts@plt>:
  400500:       ff 25 12 0b 20 00       jmpq   *0x200b12(%rip)
  400506:       68 00 00 00 00          pushq  $0x0
  40050b:       e9 e0 ff ff ff          jmpq   4004f0 <.plt>
```

GOT Overwrite

- Pointers in the GOT can be overwritten to hijack control flow

```
0000000000600ac0 <_GLOBAL_OFFSET_TABLE_>:
Your pointer
...
```

```
0000000000400500 <puts@plt>:
  400500:       ff 25 12 0b 20 00       jmpq   *0x200b12(%rip)
  400506:       68 00 00 00 00          pushq  $0x0
  40050b:       e9 e0 ff ff ff          jmpq   4004f0 <.plt>
```

GOT Overwrite Defenses

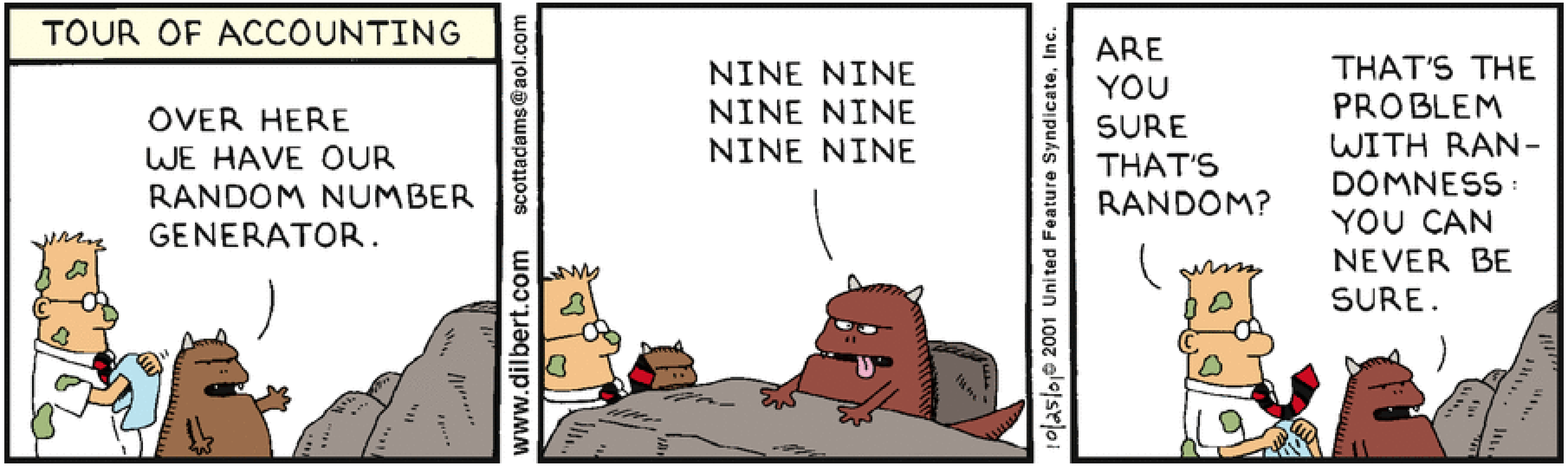
- Full RELRO (RELocation Read-Only)
  - Resolve all shared-library functions at load time (BIND NOW)
  - Move GOT to its own section and mark as read-only after all functions have been resolved

```
Contents of section .got:
puts
printf
...
```

- Partial RELRO
  - GOT is writable but before BSS segment → it cannot be overwritten with a global variable overflow
  - Most common configuration

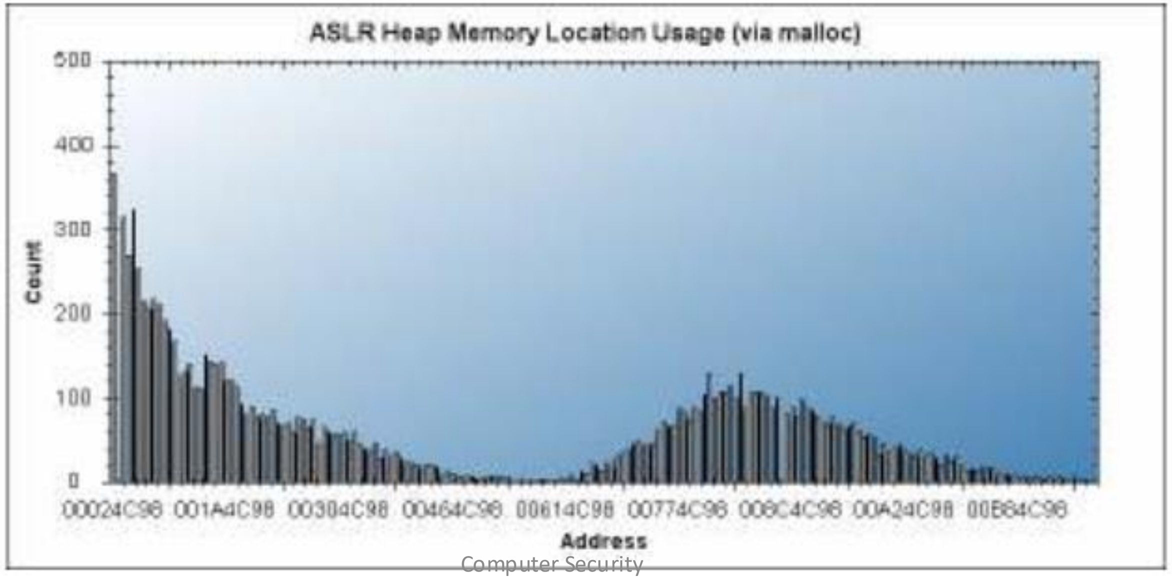
Other ASLR Weaknesses

- Weak random number generators, implementation bugs, lead to predictable segment location



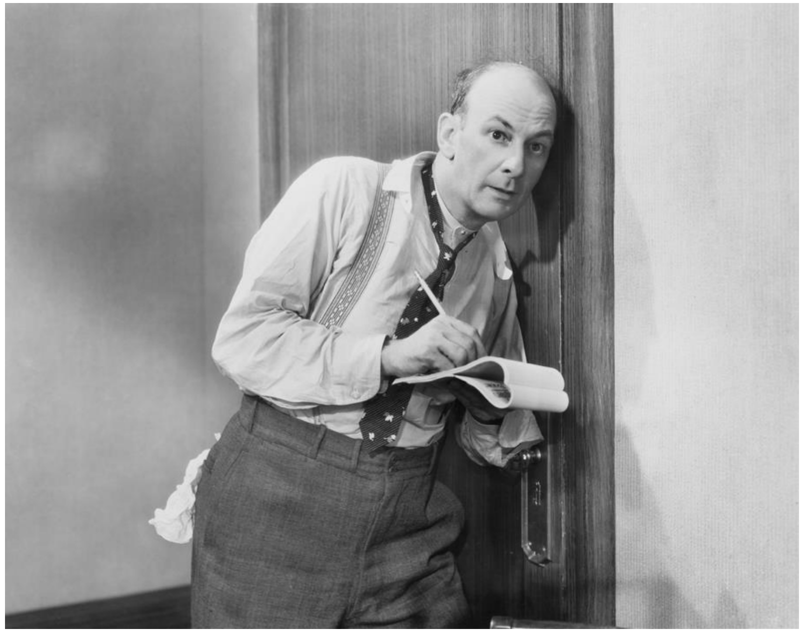
Other ASLR Weaknesses

- Weak random number generators, implementation bugs, lead to predictable segment location
- Biased Selection of Heap Base Address
  - An Analysis of Address Space Layout Randomization on Windows Vista", Ollie Whitehouse, BlackHat 2007



Information Leaks

- An information leak is caused by exploiting a bug that discloses the memory layout and/or contents of a program

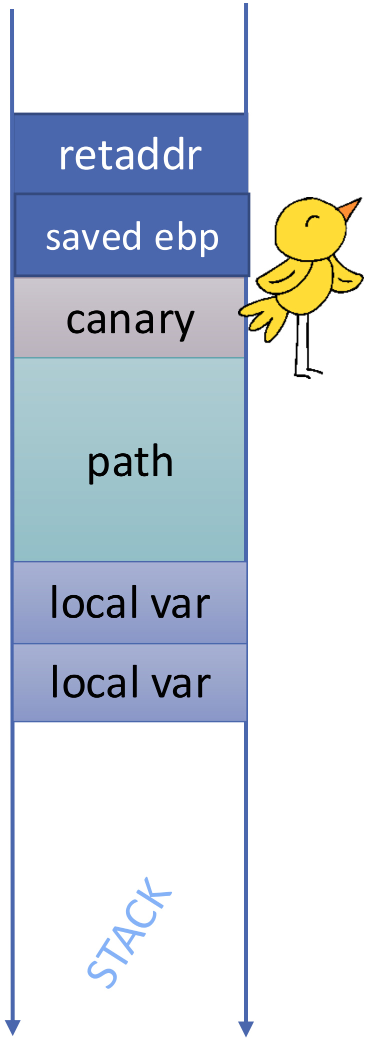


- Main idea:
  - Corrupting (partially) data that affect what or how much is read from memory
  - Receive the output of the read

Leaks Can Occur in the Stack

```
void func(char *filename, int len)
{
      char path[128] = “/tmp/”;

      memcpy(path, filename, len);

      ...
      fprintf(logfl, “Opened %s\n”, path);
      ...
}
```

Omitting or overwriting the terminating ‘\0’ character and reading a string can leak data



Or the Heap

```
void string::copy(string *src)
{
	...
	memcpy(this->data, src->data, src->len);
	...
}

outputfile->copy(userinput);
...
logfl << “user entered ” << userinput << endl;
```

```
class string
{
...
private:
	size_t len;
	char *data;
...
};
```

Or the Heap

```
void string::copy(string *src)
{
      ...
      memcpy(this->data, src->data, src->len);
      ...
}

outputfile->copy(userinput);
...
logfl << “user entered ” << userinput << endl;
```

```
class string
{
...
private:
      size_t len;
      char *data;
...
};
```

Or the Heap

```
void string::copy(string *src)
{
      ...
      memcpy(this->data, src->data, src->len);
      ...
}

outputfile->copy(userinput);
...
logfl << “user entered ” << userinput << endl;
```

```
class strin
{
...
private:
      size_t len;
      char *data;
...
};
```

Control how much data will be read

Control where the data will be read from

Welcome to 2024 - Hugepages!

- Modern HW and OS support 2MB pages
  - Offers performance benefits → less TLB entries accessing more data
- A normal 4KB Page must be 12 bit aligned
- A 2MB Huge Page must be 21 bit aligned.
- Less slots to in the same address range
- ASLRn't: How memory alignment broke library ASLR -- https://zolutal.github.io/aslrnt/
  - ASLR is broken for 32-bit libraries >= 2MB on certain filesystems
  - ASLR entropy on 64-bit libraries of >= 2MB is significantly reduced from 28 bits to 19 bits, on certain filesystems.
    - That 9-bit difference in alignment from 12 to 21

Summary of ASLR Weaknesses

- Memory leaks
  - Combine memory leaks with control-flow hijacking
  - Repeatable arbitrary memory leaks are better
- Insufficient entropy
- Non-randomized binary images
- Hugepages

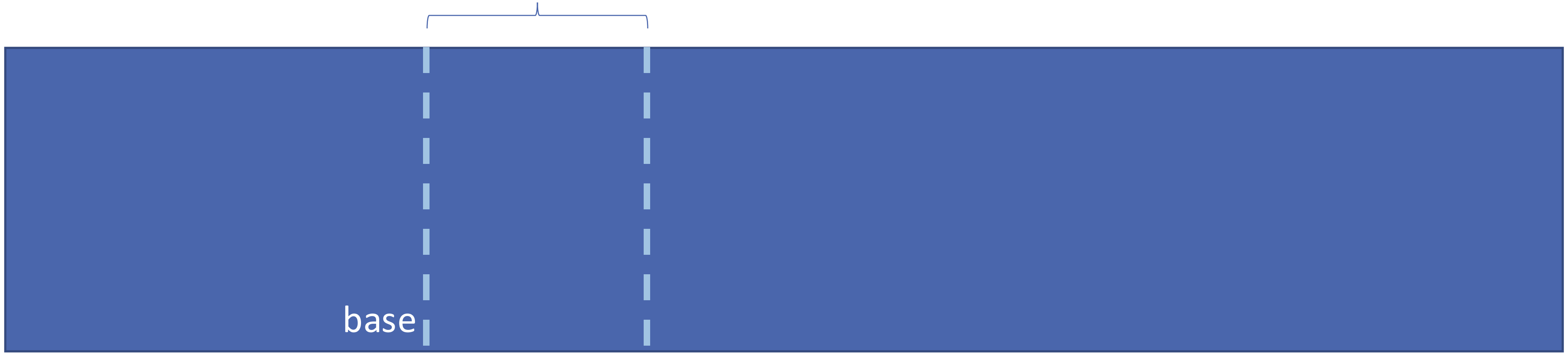
Heap Spraying

A technique that aims to solve the issue of locating the address of an attacker object in the heap

Randomized Heap

Heap base address (usually on Linux): [base + random offset]

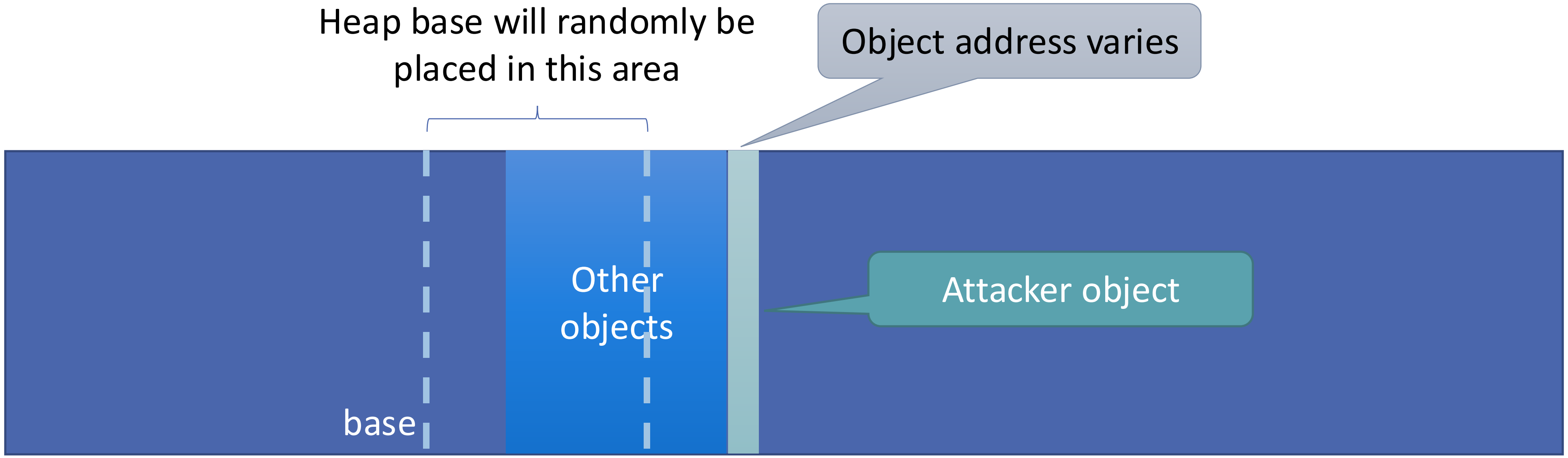
Heap base will randomly be placed in this area



Application address space

Attacker Objects in Randomized Heap

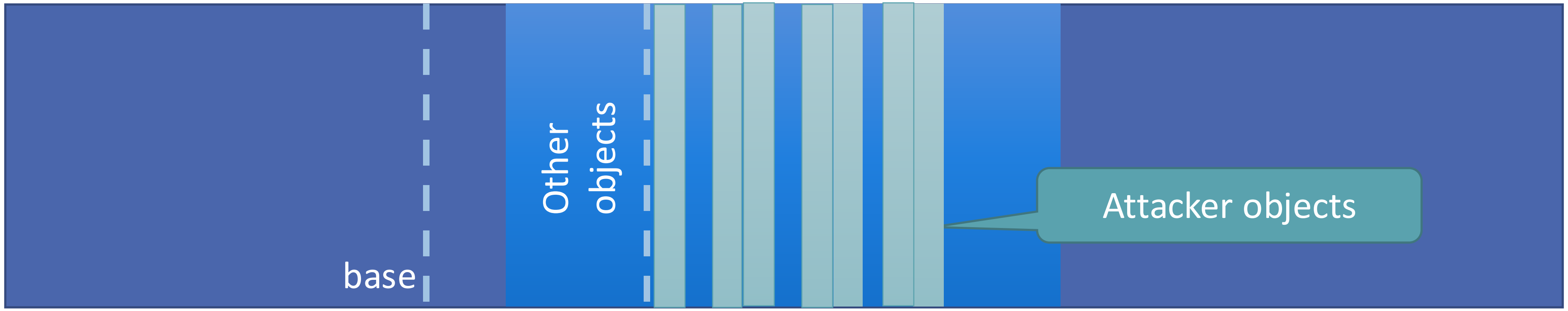
Heap base address (usually on Linux): [base][13 random bits][page offset]



Application address space

Heap Spraying

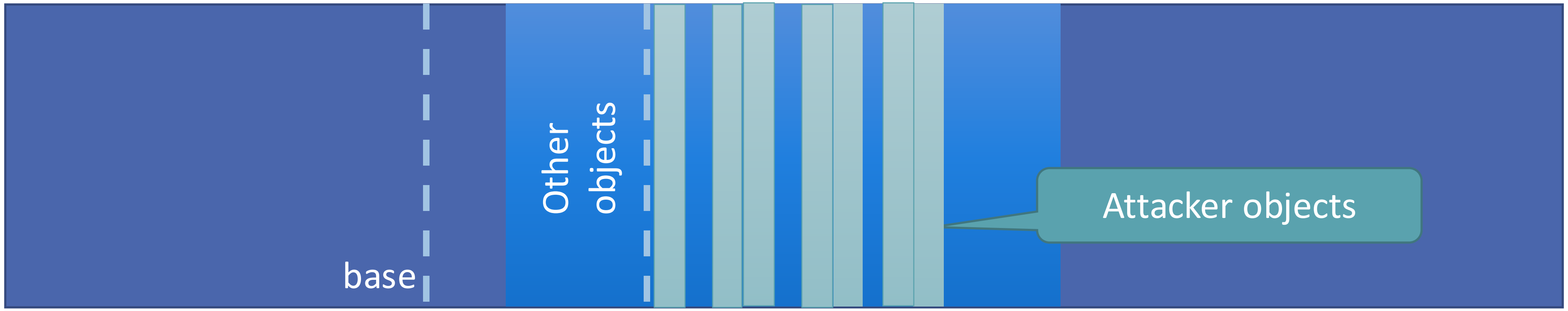
- This technique aims to solve the issue of locating the address of an attacker object in the heap
- Attacker allocate (spray) many copies of their object on the heap → Goal is to statistically increase the chances of one of the objects falling on a constant address



Application address space

Heap Spraying

How is this useful?



Application address space

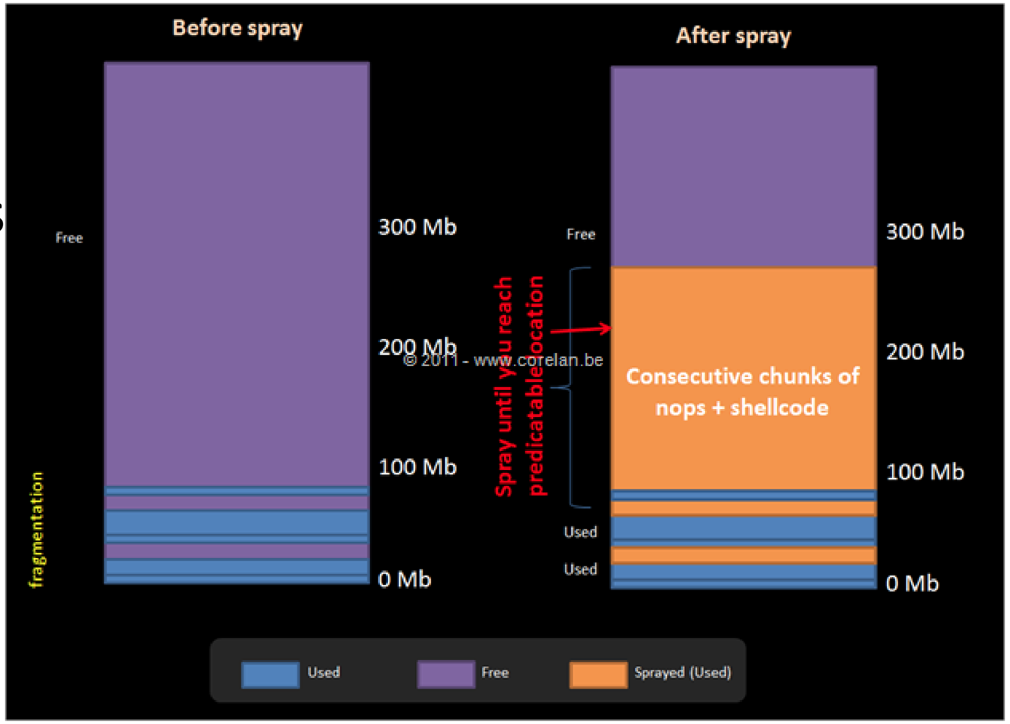
Heap Spraying Against Browsers

- Dynamically expand JS buffer by appending copies of the shellcode
- On the fly generate variables
- Add massive NOP sleds to increase chances of success

https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

Heap Spraying Against Browsers

- Dynamically expand JS buffer by appending copies of the shellcode
- On the fly generate variables
- Add massive NOP sleds to increase chances of success
- The more of the JIT memory used the higher the chances of success



https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

Heap Feng Shui

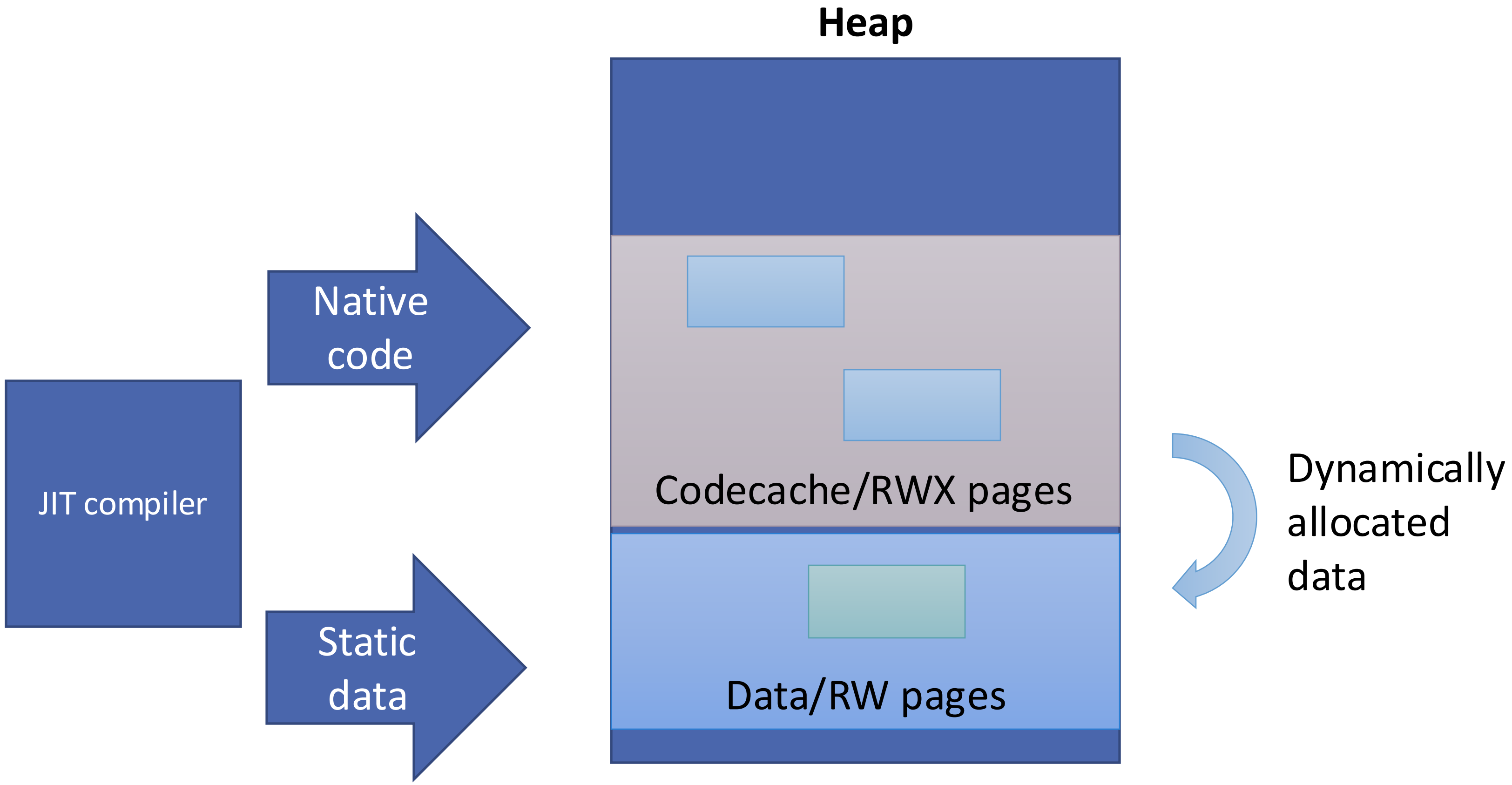
- A terms used to describe techniques that manipulate the heap, so a particular layout is achieved
- For example, to ensure that one object is allocated adjacent to another
- Requires great understanding of heap allocator internals

Summary: Heap Spraying

- May require multiple attempts
- A probabilistic attack against ASLR
- Heap fragmentation is in play
  - May be worse in concurrent systems

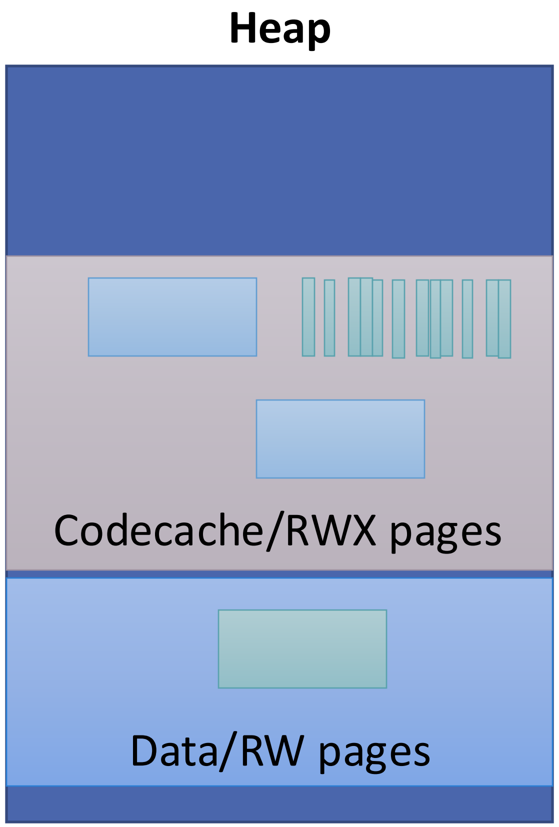
Avoiding Code Injection in Browsers

- Separate code and heap into separate memory areas
- However … it still violates W^X



JIT Spraying

- Constants (char, short, int, etc.) are still allocated in the code cache
- ROP-style gadgets of 4-8 bytes long can be stored in constants
- JIT must be sprayed with many copies to bypass ASLR
  - Unless information leakage is possible



JIT Spraying Defenses

- Constant blinding aims to "encrypt" large constants placed in the JIT
- Constants are XORed with a random value before being stored in memory
- The JIT emits code to unblind them before the program uses them
  - XOR the blinded value loaded on a register with the random value before use

Format String Attacks

Format String Bugs

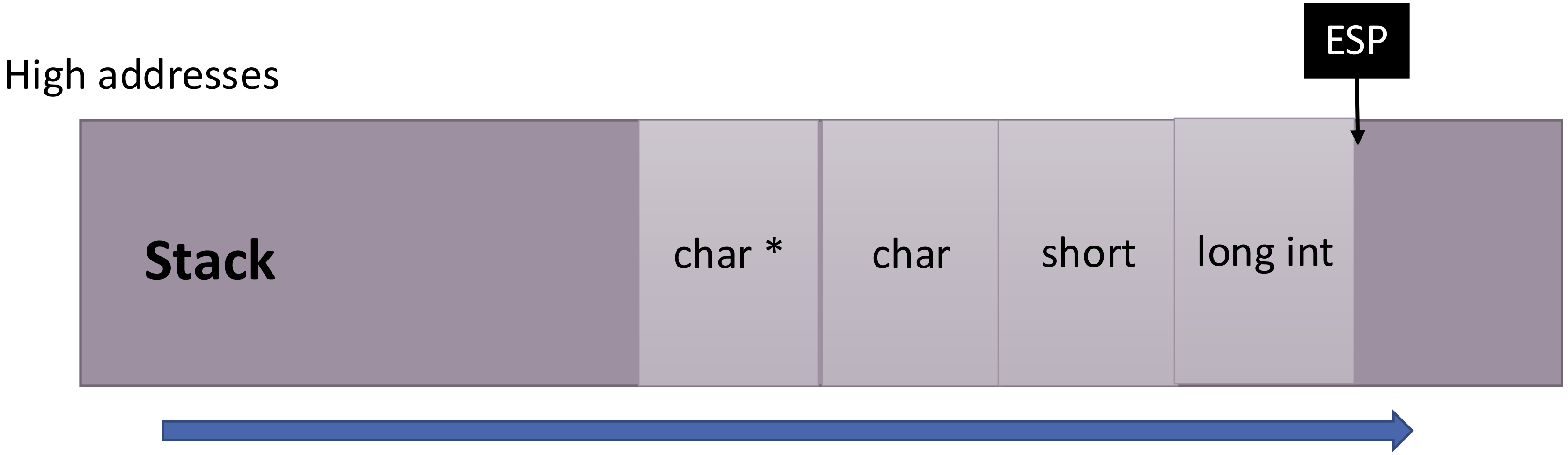
- Exploits functions formatted output functions like printf
- int **printf**(const char * restrict format, ...);
- printf is a variadic function → a function which accepts a variable number of arguments
- Follows calling conventions
  - cdecl – summary: all arguments are passed in the stack
  - System V AMD64 ABI – summary: RDI, RSI, RDX, RCX, R8, R9, and then the stack

Format String Bugs

- The functions consumes arguments based on the value of the format string
  - int **printf**(const char * restrict **format**, ...);
- Example: format string “%d %u %s” will consume three arguments

Argument Types and Number Based on Format String (32-bit)

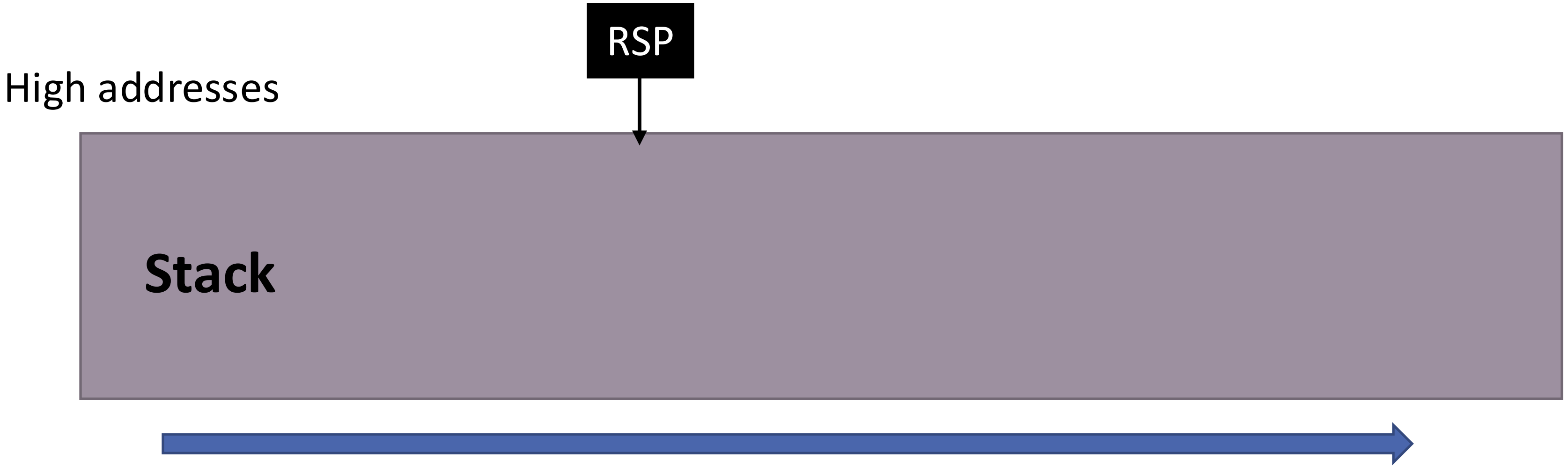
- printf(“%ld %h %c %s”, long_integer, short, character, string);
- Arguments are pushed to the stack!
- printf reads stack arguments based on the format string



Argument Types and Number Based on Format String (64-bit)

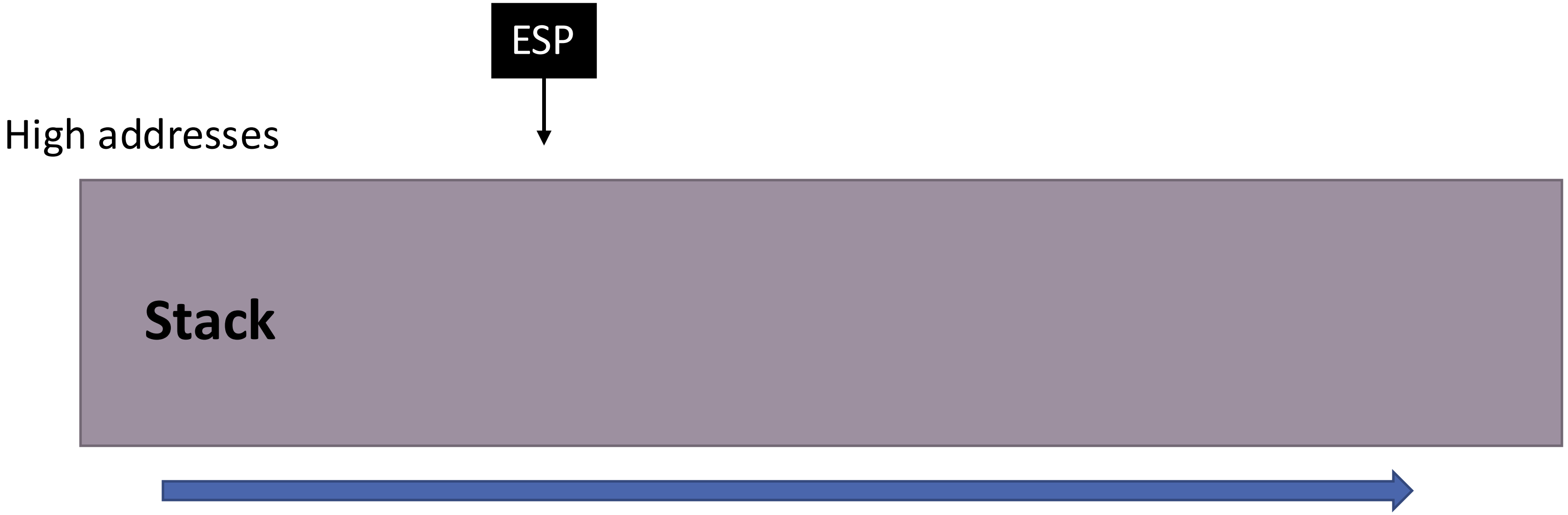
- printf(“%ld %h %c %s”, long_integer, short, character, string);
- Arguments are pushed to the stack!
- printf reads stack arguments based on the format string

First arguments passed in registers: RDI → “%ld %h %c %s”, RSI → long_integer, RDX →short, RCX→character, R8 → string, R9



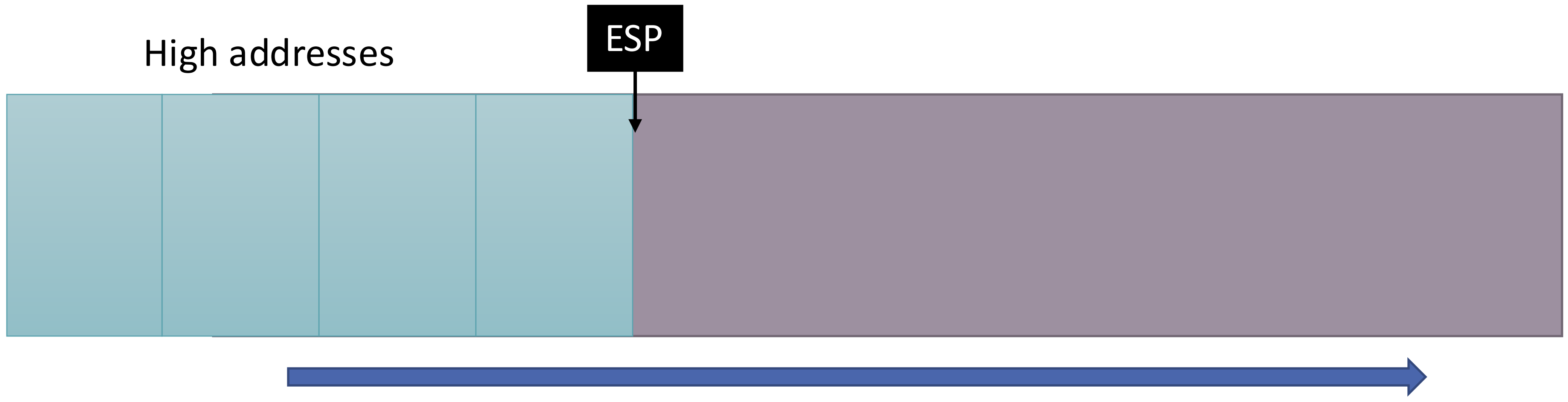
Not Enough Arguments

- **printf("%ld %h %c %s");**
- What happens if there is a mismatch between format string and actual arguments?



Not Enough Arguments

- **printf("%ld %h %c %s");**
- What happens if there is a mismatch between format string and actual arguments?
- The program will still access the data

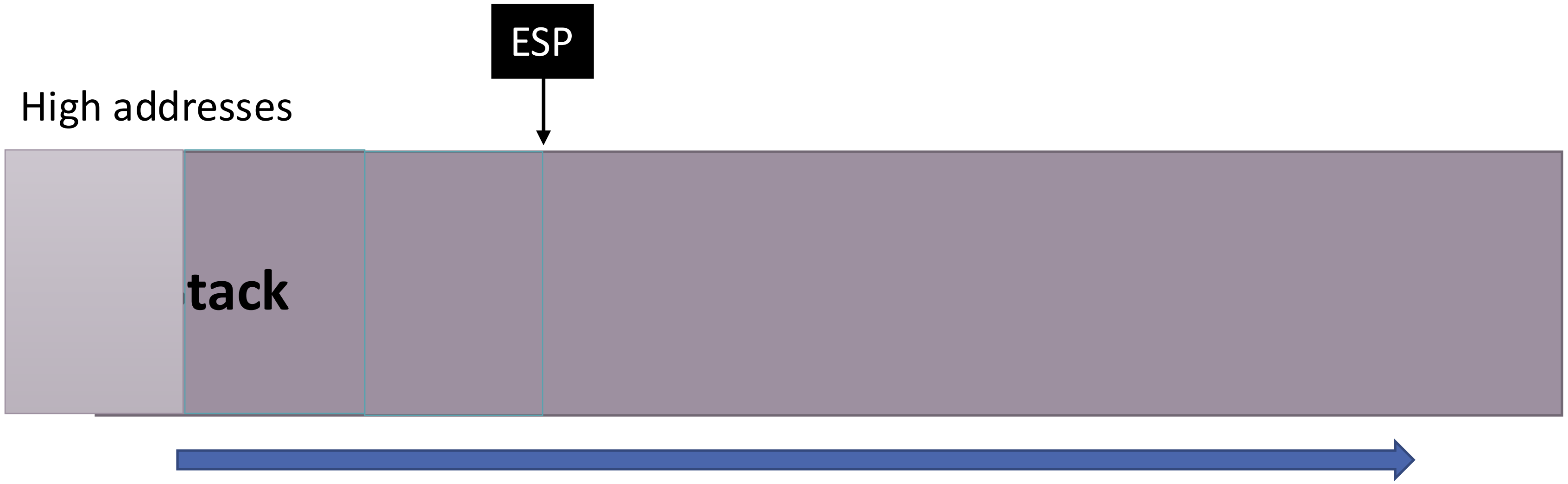


Format String Bugs

- The functions consumes arguments based on the value of the format string
  - int **printf**(const char * restrict **format**, ...);
- Example: format string “%d %u %s” will consume three arguments
- **Occur when untrusted (user) input is used as a format string, leading to a mismatch in the number of arguments passed by the program**
  - Example:
    - Vulnerability: fmt is a string controlled by the user `printf(fmt)`
    - Not a vulnerability: fmt is a constant string or not controlled by the user `printf(“%d %p”, var1, var2)`

Reading Memory Using printf

- Direct parameter access
- Reading the 3rd number from the ESP
  - “%x %x %x” → Access 3 arguments
  - “%3$x” → Access the 3rd argument directly



Writing Memory Using printf

- **%n** can be used to store the number of written characters into an integer pointer

```
int n;
long li = 100;
printf("%ld\n%n", li, &n);
```

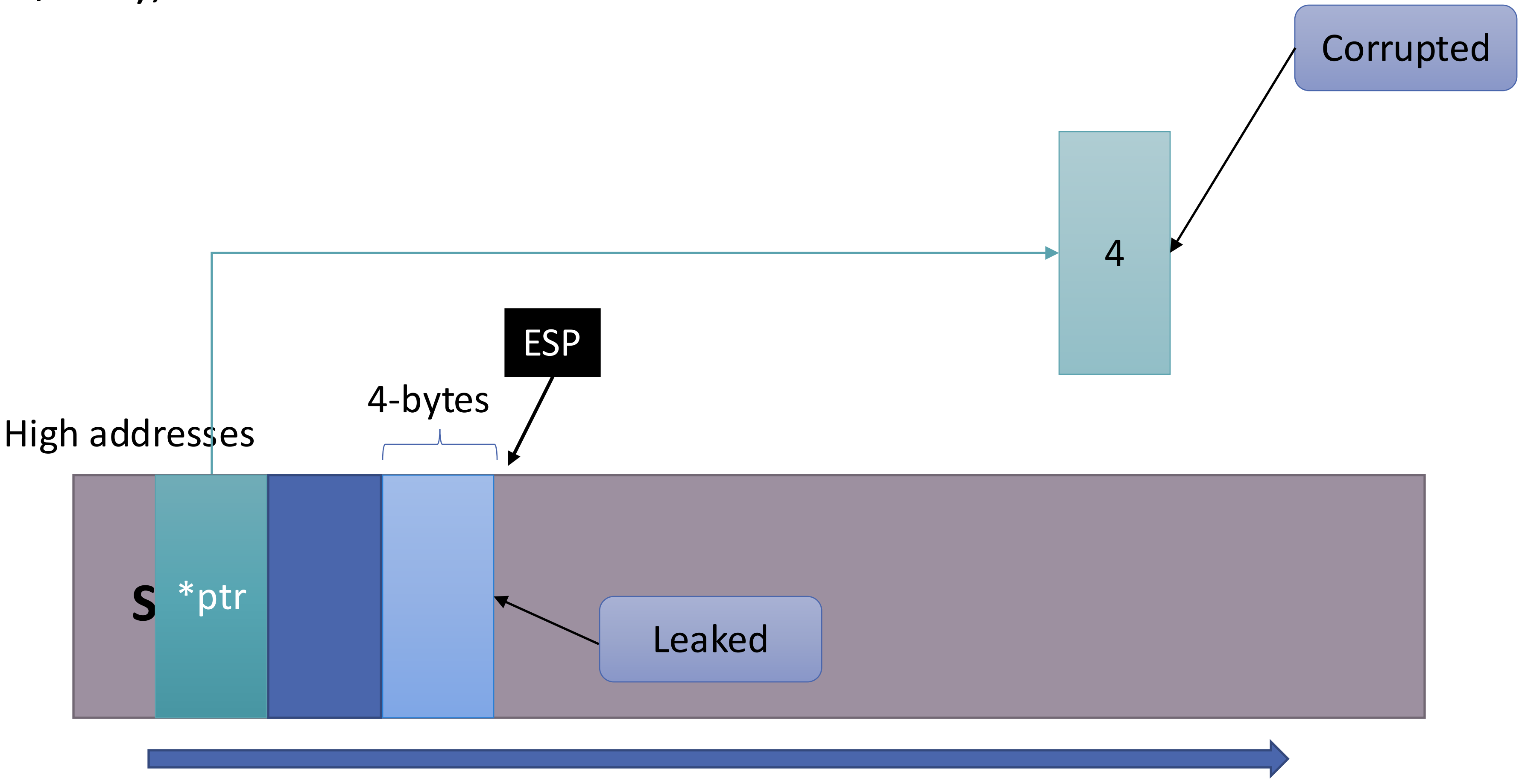
Writing Memory Using printf

- **%n** can be used to store the number of written characters into an integer pointer

```
int n;
long li = 100;
printf("%ld%n", li, &n);

n = 3
```

Example

- printf("%ld%$3n");



Writing Arbitrary Numbers

- Length modifier (+ zero padding)

- int n;
- long li = 23;
- printf(“%0128ld%n\n”, li, &n);

n = 128

```
000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
00000000000023
```

- **It is easy to write a large number of characters!**

Writing Arbitrary Numbers

- Width modifier (+ zero padding)

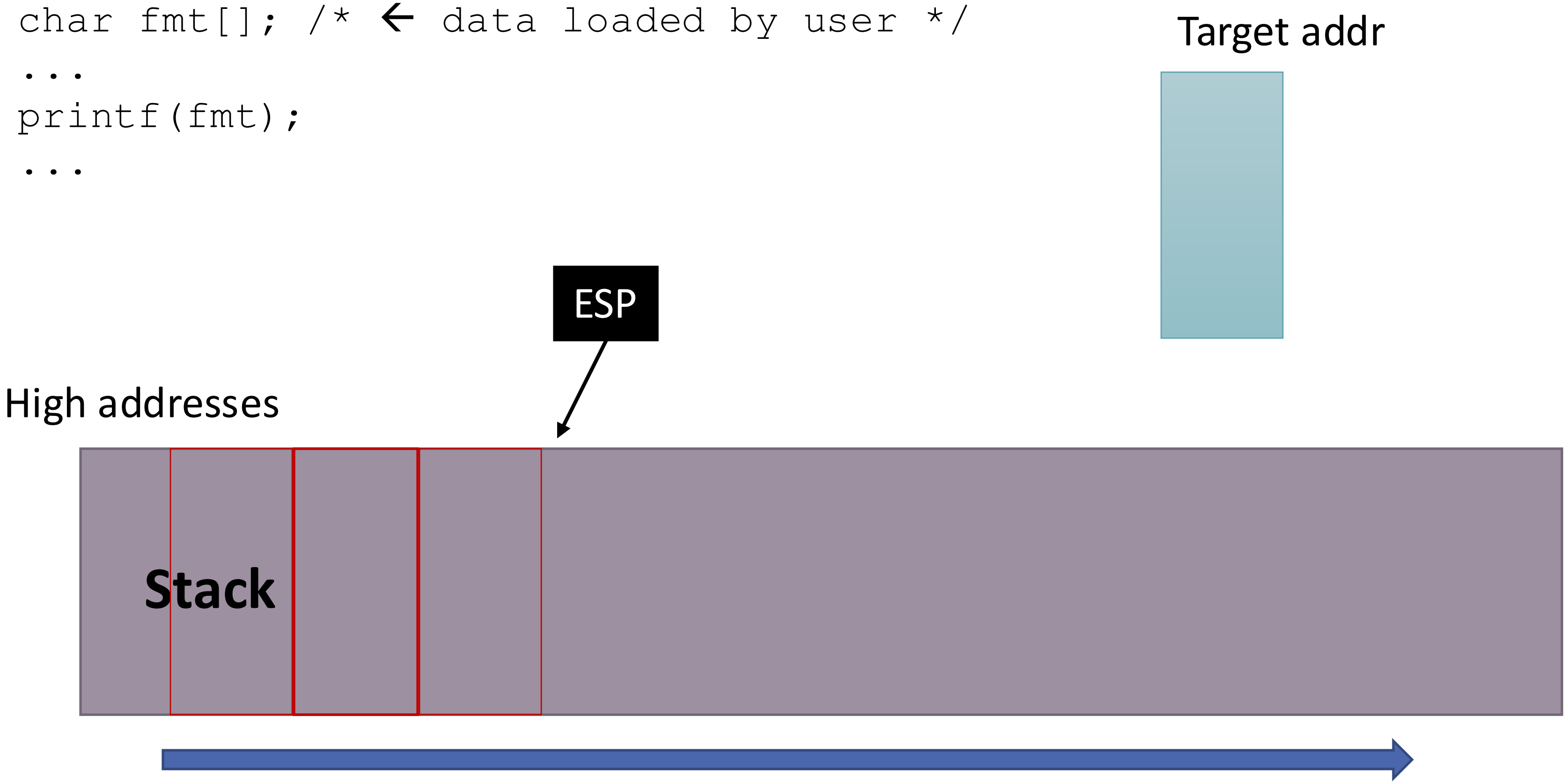
- int n;
- long li = 23;
- int w = 128;
- printf(“%0*ld%n\n”, w, li, &n);

n = 128

```
000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
00000000000023
```

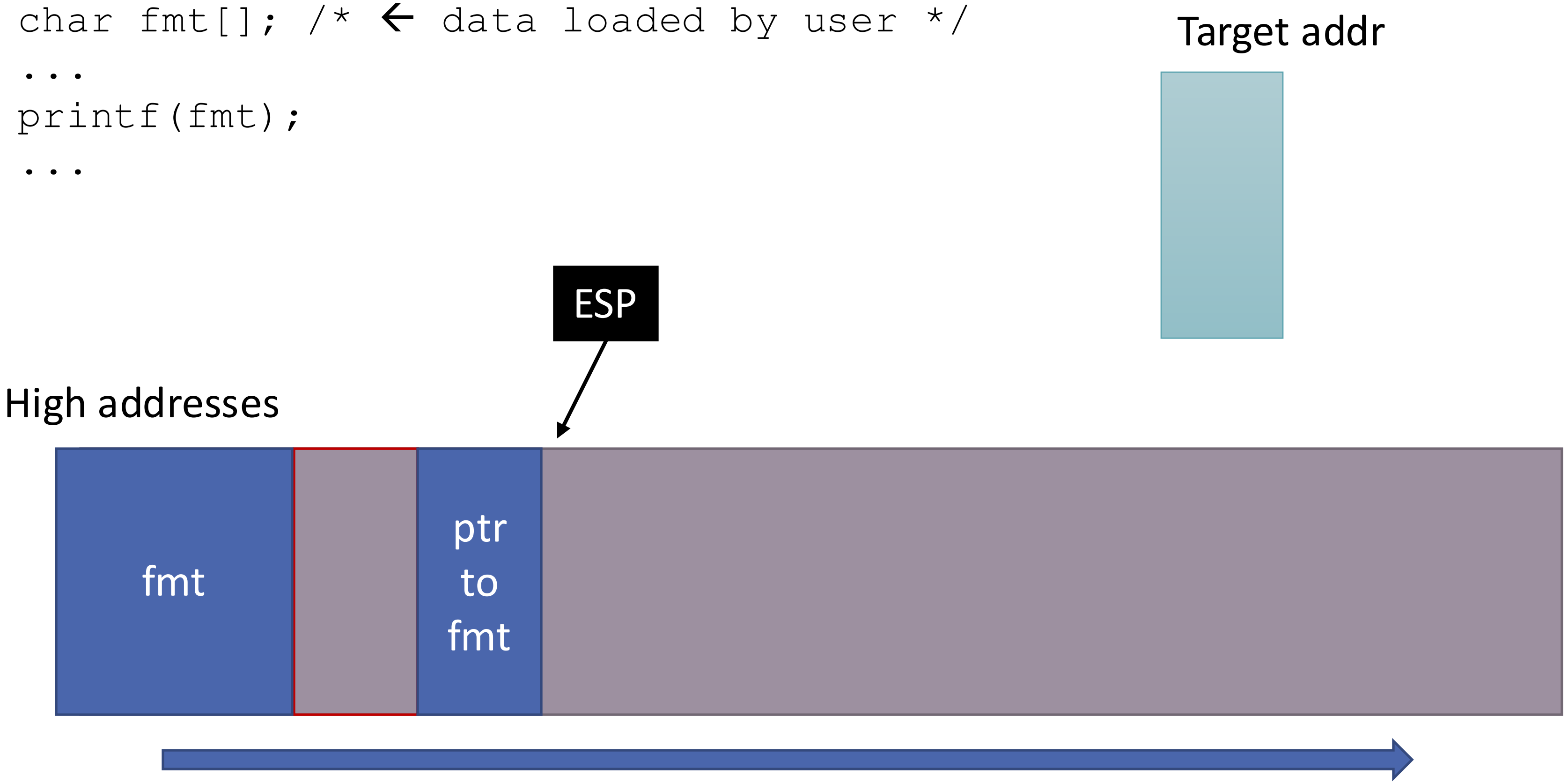
Arbitrary Write with printf()

```
{
    char fmt[]; /* ← data loaded by user */
    ...
    printf(fmt);
    ...
}
```



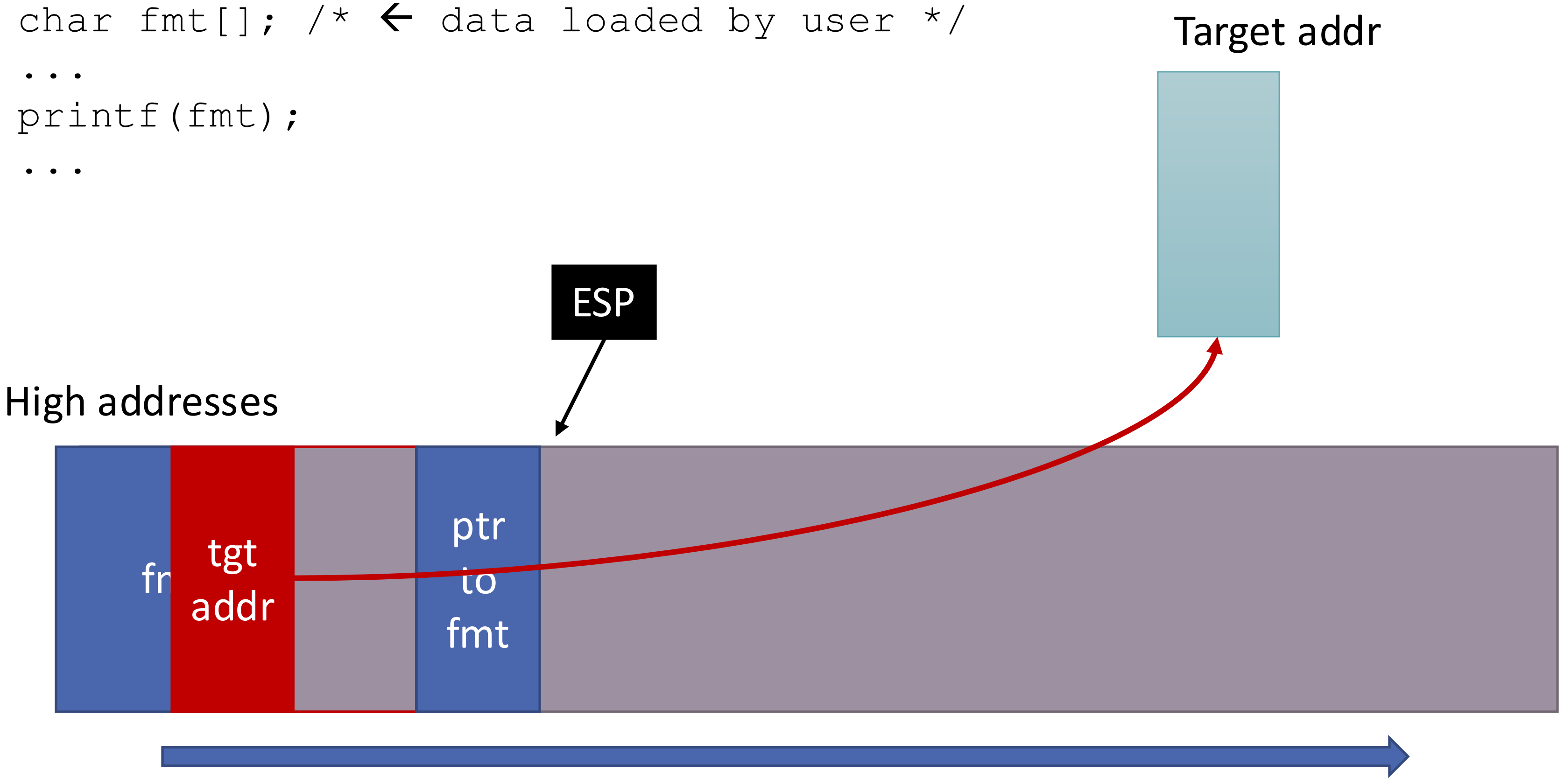
Arbitrary Write with printf()

```
{
    char fmt[]; /* ← data loaded by user */
    ...
    printf(fmt);
    ...
}
```



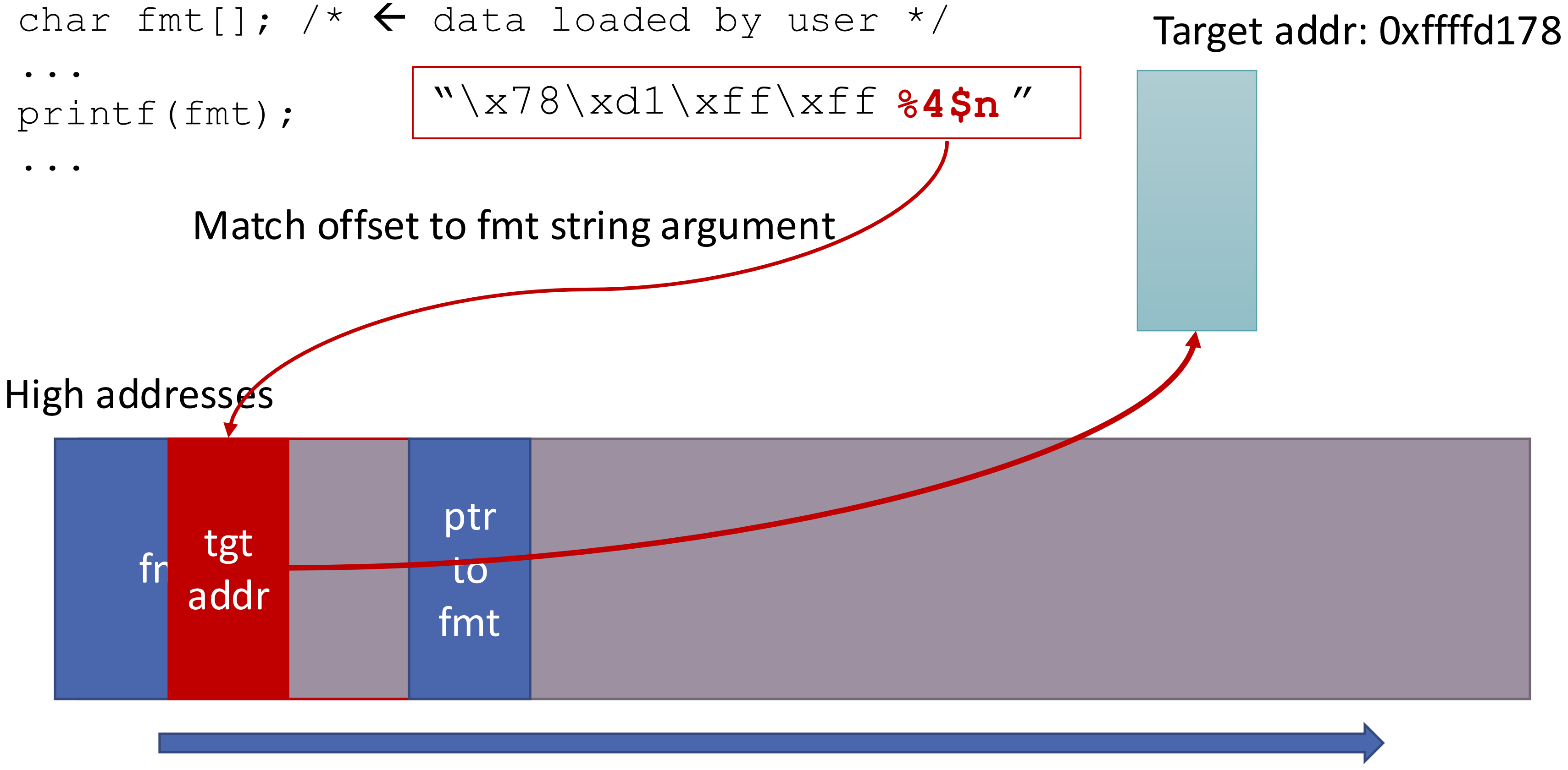
Arbitrary Write with printf()

```
{
    char fmt[]; /* ← data loaded by user */
    ...
    printf(fmt);
    ...
}
```



Arbitrary Write with printf()

```
{
    char fmt[]; /* ← data loaded by user */
    ...
    printf(fmt);
    ...
}
```



Defenses and Bypasses

- Defining _FORTIFY_SOURCE replaces calls to printf with safe version
  - “%n” is not allowed if stored in writable memory
  - No argument holes are allowed, so “%4$x” would be invalid but “%4$x %2$x %1$x %3x” is valid
- Bypasses propose overwriting the writable flag that enable the _FORTIFY_SOURCE variable
  - Possible but more complicated

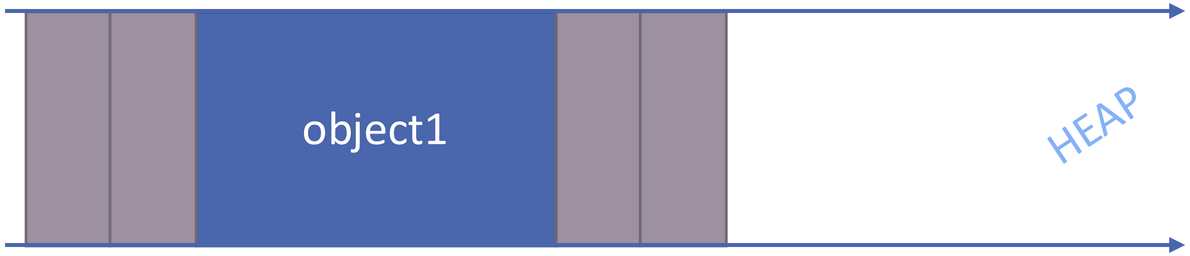
Format String Attacks Summary

- Powerful but you need to work with strings and their restrictions
  - Can’t use NUL bytes in a format string
- Depending on the program, these attacks can also be Turing complete
  https://github.com/HexHive/printbf
- _FORTIFY_SOURCE raises the bar for attackers
- Additional reading:
  - https://www.win.tue.nl/~aeb/linux/hh/formats-teso.html
  - http://phrack.org/issues/67/9.html

Use-after-Free

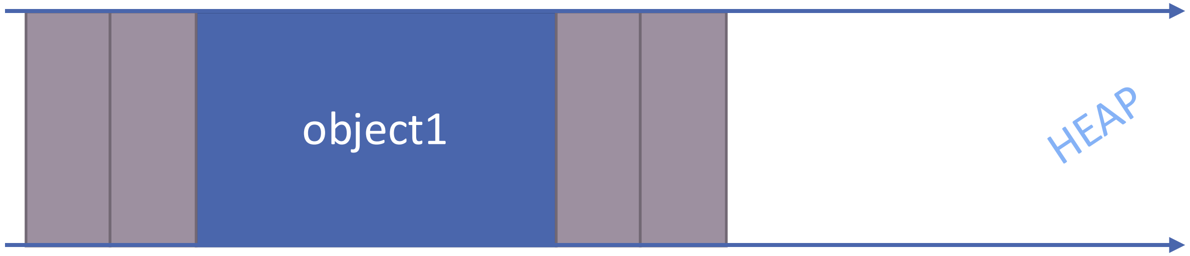
Use-after-Free Vulnerabilities

- A memory object is used after being freed



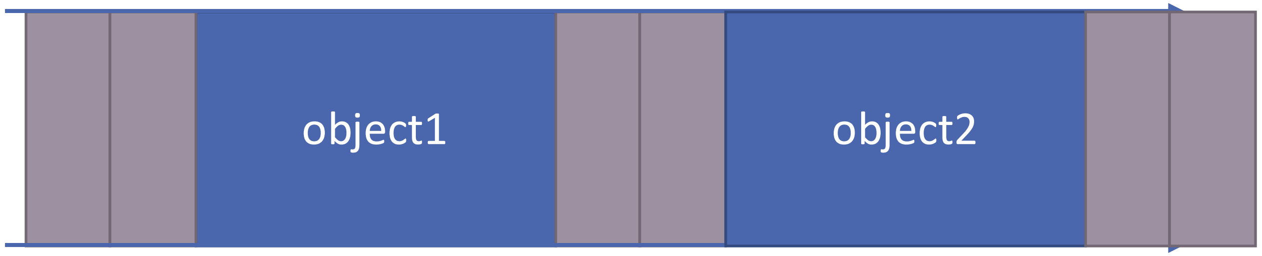
Use-after-Free Vulnerabilities

- A memory object is used after being freed
- Steps:



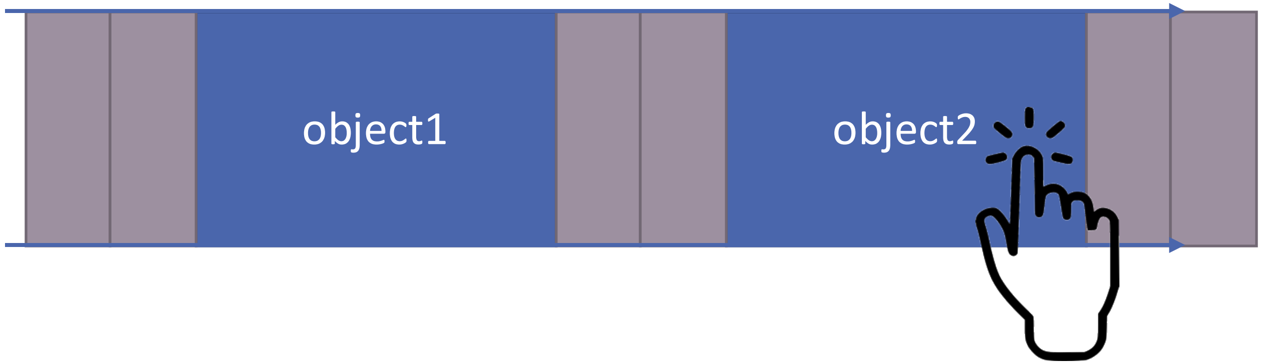
Use-after-Free Vulnerabilities

- A memory object is used after being freed
- Steps:
  - Allocate memory for object2



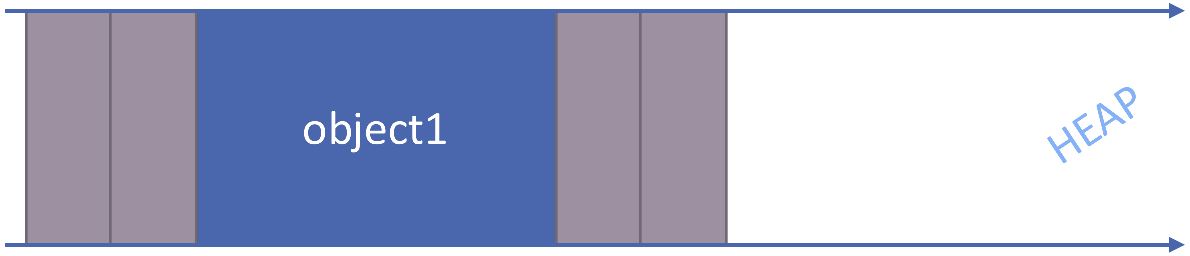
Use-after-Free Vulnerabilities

- A memory object is used after being freed
- Steps:
  - Allocate memory for object2
  - Use memory as object2



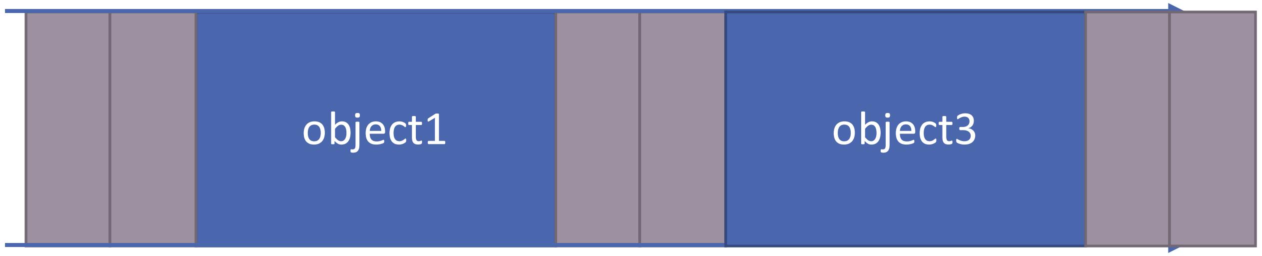
Use-after-Free Vulnerabilities

- A memory object is used after being freed
- Steps:
  - Allocate memory for object2
  - Use memory as object2
  - Free memory of object2



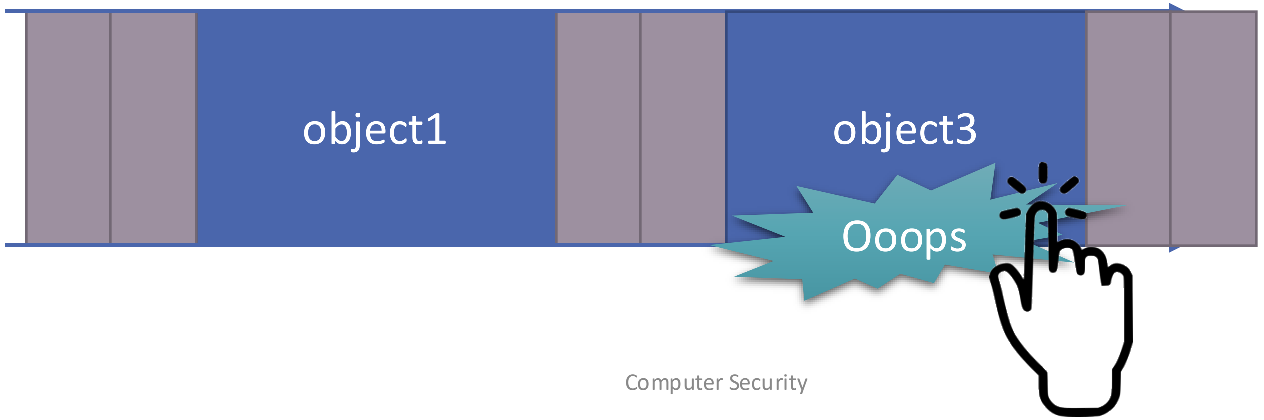
Use-after-Free Vulnerabilities

- A memory object is used after being freed
- Steps:
  - Allocate memory for object2
  - Use memory as object2
  - Free memory of object2
  - Allocate memory for object3



Use-after-Free Vulnerabilities

- A memory object is used after being freed
- Steps:
  - Allocate memory for object2
  - Use memory as object2
  - Free memory of object2
  - Allocate memory for object3
  - Use memory as object2

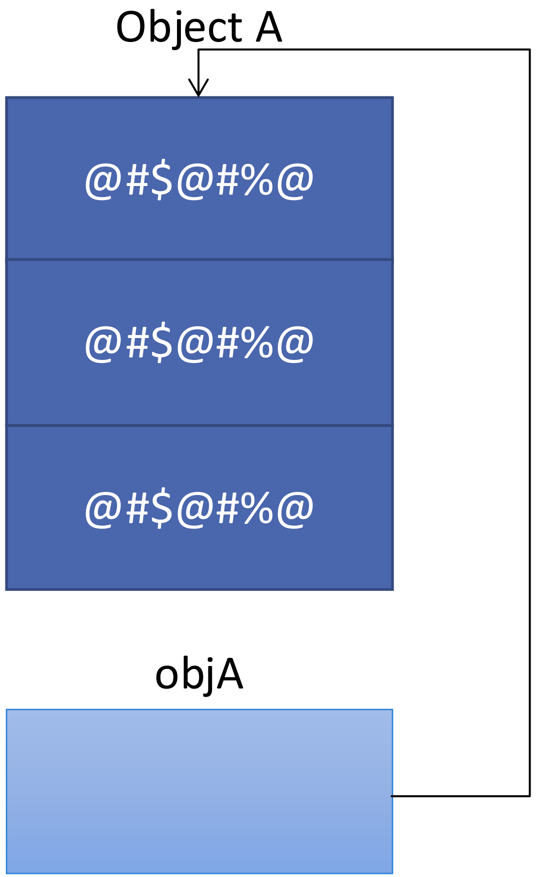


What Happens?

- Freeing an object creates a dangling pointer

Dangling Pointers

```
struct objectA *objA;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
writeA(objA); /* Uses dangling pointer */
```

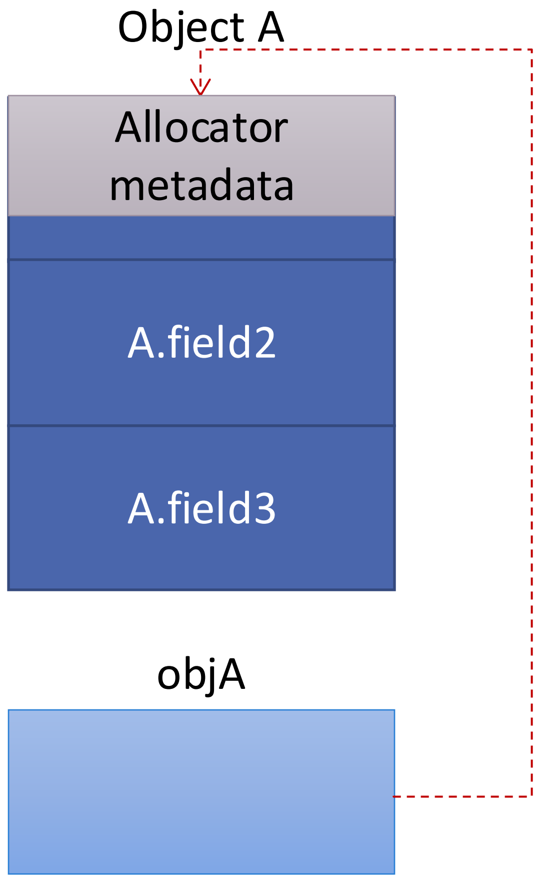


Dangling Pointers

```
struct objectA *objA;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
writeA(objA); /* Uses dangling pointer */
```

Dangling Pointers

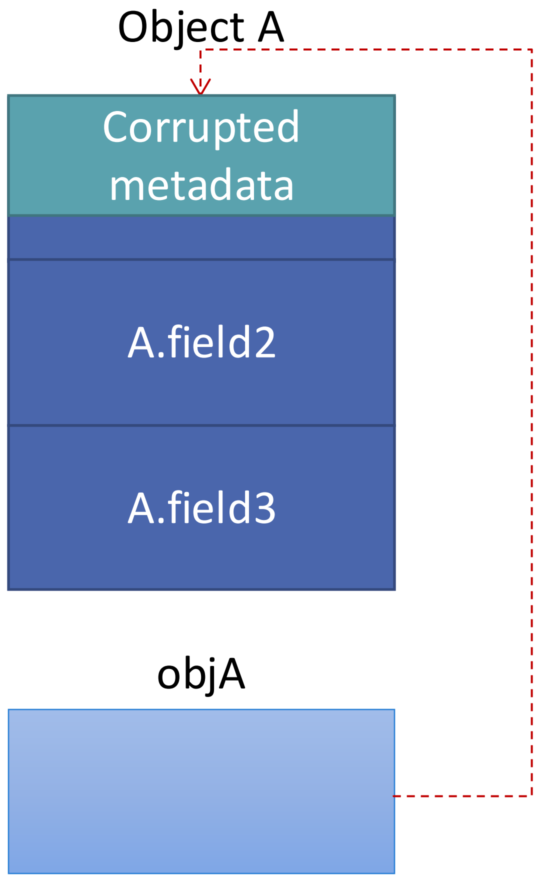
```
struct objectA *objA;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
writeA(objA); /* Uses dangling pointer */
```



Pointer is dangling → does not point to a valid location

Use-after-Free Case 1

```
struct objectA *objA;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
writeA(objA); /* Uses dangling pointer */
```

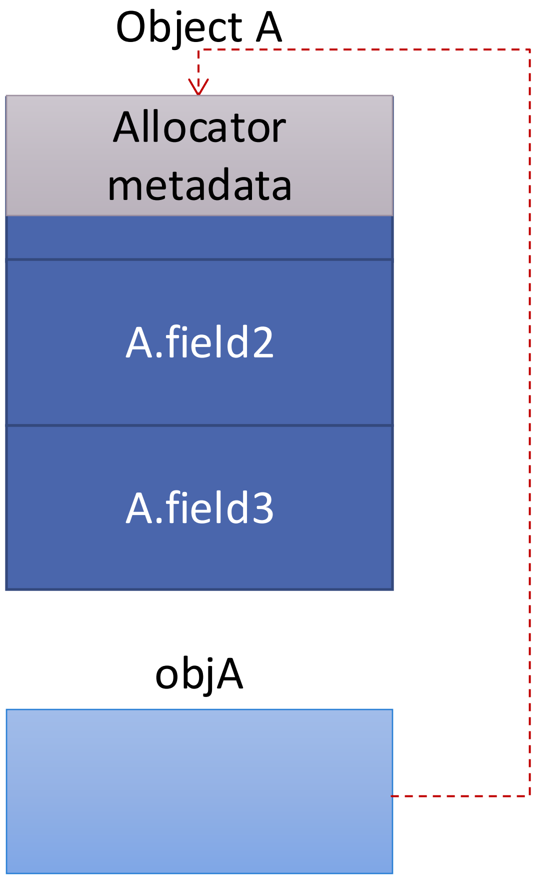


What Happens?

- Freeing an object creates a dangling pointer
- Case 1: Pointer is used to write data immediately → metadata may be corrupted leading to unpredictable behavior

Use-after-Free Case 2

```
struct objectA *objA;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
readA(objA); /* Uses dangling pointer */
```

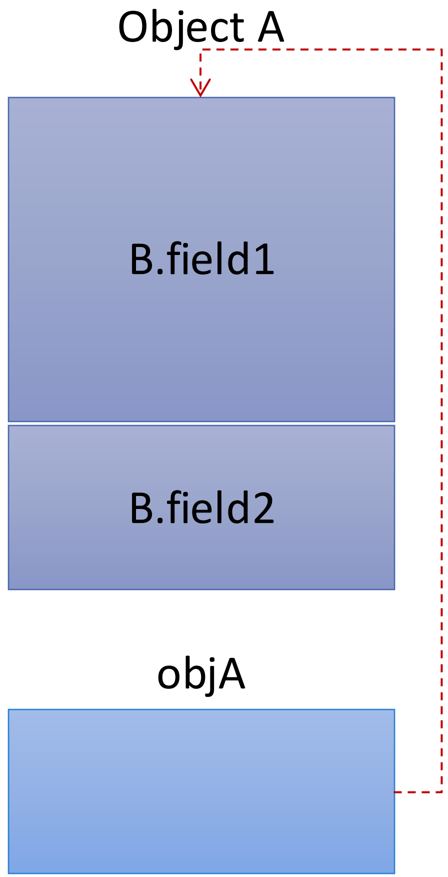


What Happens?

- Freeing an object creates a dangling pointer
- Case 1: Pointer is used to write data immediately → metadata may be corrupted leading to unpredictable behavior
- Case 2: Pointer is used to read data immediately → metadata may be exfiltrated, leaking memory layout (threat to ASLR)

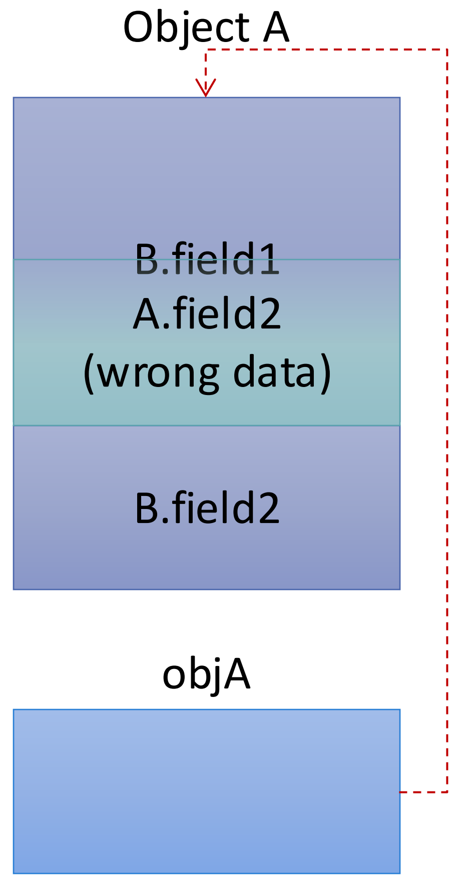
Use-after-Free Case 3

```
struct objectA *objA;
struct objectB *objB;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
objB = malloc(sizeof(struct object B));
...
writeB(objB); /* writes in objB */
...
readA(objA); /* Uses dangling pointer */
```



Use-after-Free Case 3

```
struct objectA *objA;
struct objectB *objB;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
objB = malloc(sizeof(struct object B));
...
writeB(objB); /* writes in objB */
...
readA(objA); /* Uses dangling pointer */
```



What Happens?

- Freeing an object creates a dangling pointer
- Case 1: Pointer is used to write data immediately → metadata may be corrupted leading to unpredictable behavior
- Case 2: Pointer is used to read data immediately → metadata may be exfiltrated, leaking memory layout (threat to ASLR)
- Case 3: Another object takes the freed space, writes to it, and the dangling pointer is used to read from it → invalid (potentially dangerous) data are used in the wrong way

What Happens?

- Freeing an object creates a dangling pointer
- Case 1: Po d leading to unpredict
- Case 2: Po d, leaking memory l
- Case 3: Ar nter is used to read fr y

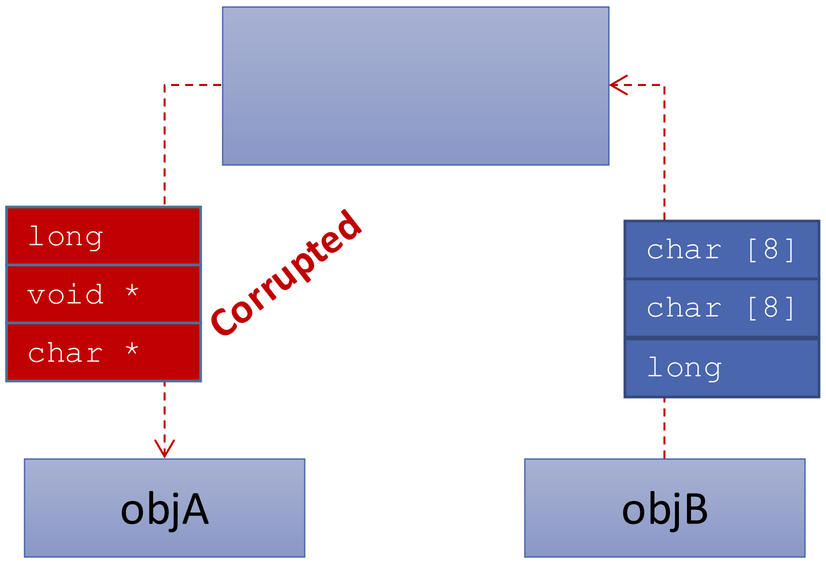
Most common meaning of use-after-free

Use-after-Free Case 3

```
struct objectA {
        long n;
        void (*fptr)();
        char *string;
}
```

```
struct objectB {
        char b[16];
        long n;
}
```

```
struct objectA *objA;
struct objectB *objB;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
objB = malloc(sizeof(struct object B));
...
writeB(objB); /* writes in objB */
...
readA(objA); /* Uses dangling pointer */
```



Summary

- The vulnerability can allow an attacker to control a code pointer, which will be later dereferenced using a dangling pointer
- Enables control-flow hijacking
- Requires careful timing
- Also appears due to concurrency bugs
  - Example: Thread 1 is still using a pointer to ObjectA, while Thread 2 frees the object

Type Confusion Bugs

Type Confusion

```
struct objectB *objB;
...
objB = malloc(sizeof(struct object B));
...
writeB(objB); /* writes in objB */
...
handle_object(objB);
```

```
struct objectA {
      long n;
      void (*fptr)();
      char *string;
}
```

```
struct objectB {
      char b[16];
      long n;
}
```

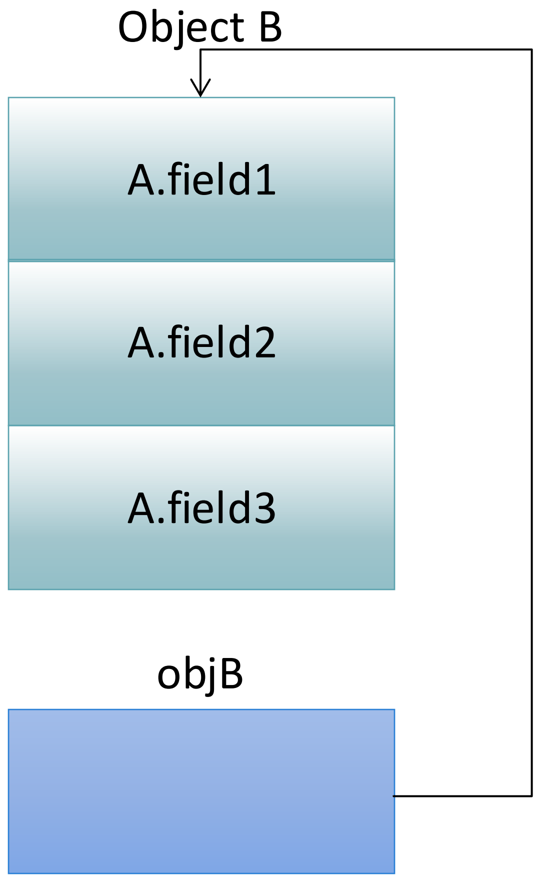
```
void readA(void *obj) {
     ...
     if ( ... ) { /* code treats object as objectA */
          struct objectA *objA = obj;
          readA(objA); /* Uses object with the wrong type */
     }
}
```

Uninitialized Memory

Uninitialized Memory

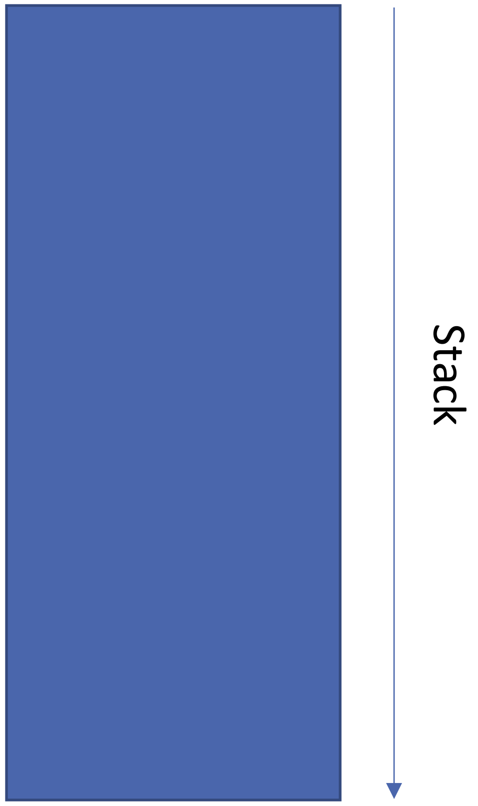
- Use of uninitialized data that can be potentially affected by attacker input

```
struct objectA *objA;

objA = malloc(sizeof(struct object A));
...
writeA(objA); /* writes in objA */
...
free(objA);
...
objB = malloc(sizeof(struct object B));

readB(objB); /* Uses uninitialized data */
```



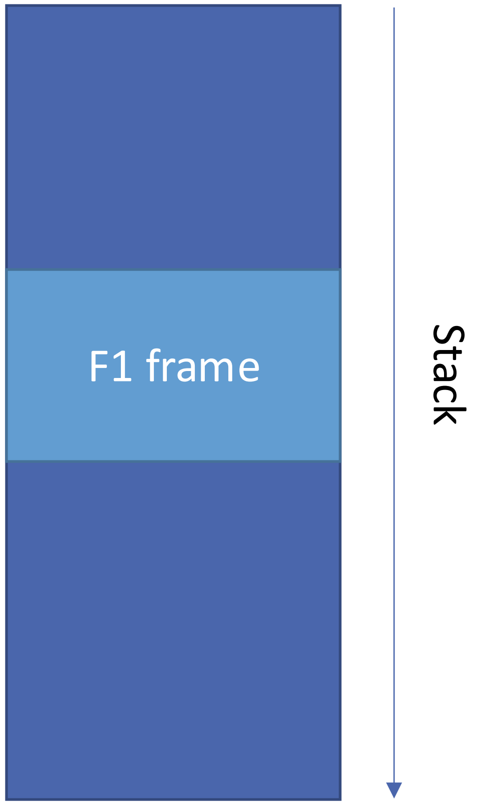
Also, in the Stack

```
f1() {
    char buf[16];

    read(1, buf, 16);
    ...
}

f2() {
    void (*fptr)();
    ... /* Does not init fptr */
    fptr();
}
main() {
    f1();
    ...
    f2();
}
```



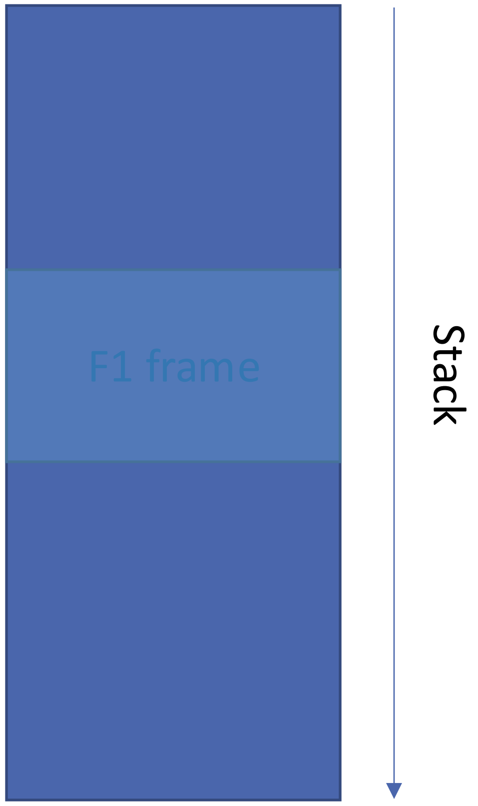
Also, in the Stack

```
f1() {
    char buf[16];

    read(1, buf, 16);
    ...
}

f2() {
    void (*fptr)();
    ... /* Does not init fptr */
    fptr();
}

main() {
    f1();
    ...
    f2();
}
```



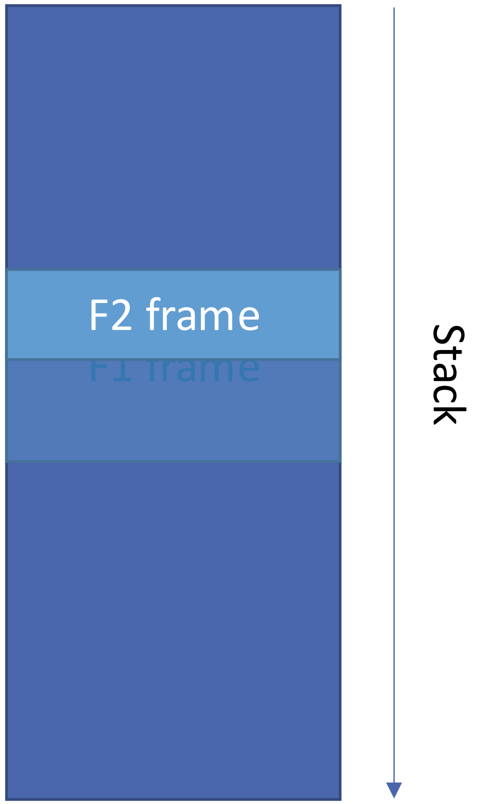
Also, in the Stack

```
f1() {
    char buf[16];

    read(1, buf, 16);
    ...
}

f2() {
    void (*fptr)();
    ... /* Does not init fptr */
    fptr();
}

main() {
    f1();
    ...
    f2();
}
```



Also, in the Stack

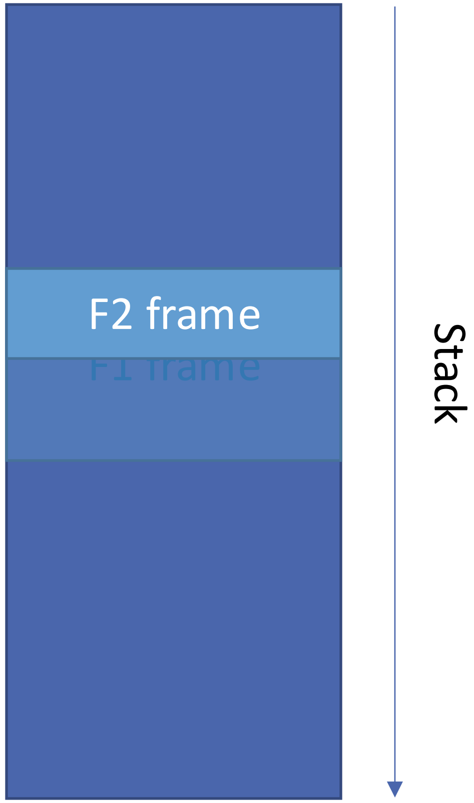
```
f1() {
    char buf[16];

    read(1, buf, 16);
    ...
}

f2() {
    void (*fptr)();
    ... /* Does not init fptr */
    fptr();
}

main() {
    f1();
    ...
    f2();
}
```



Also, in the Stack

```
f1() {
    char buf[16]

    read(1, buf,
    ...
}

f2() {
    void (*fptr)();
    ... /* Does not init fptr */
    fptr();
}

main() {
    f1();
    ...
    f2();
}
```

Could be controlled by previous input



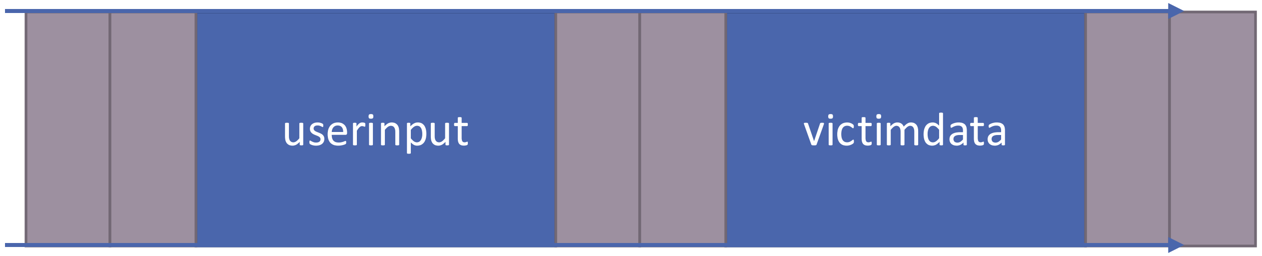
C++ Specific Problems

Reminder: Heap Overflow and Function Pointers

- Function pointers stored in the heap are at risk of heap overflows

```
struct callback {
        int priority;
        int (*cb_func)(void *data);
};
```

- On use of function pointer → control-flow hijacking



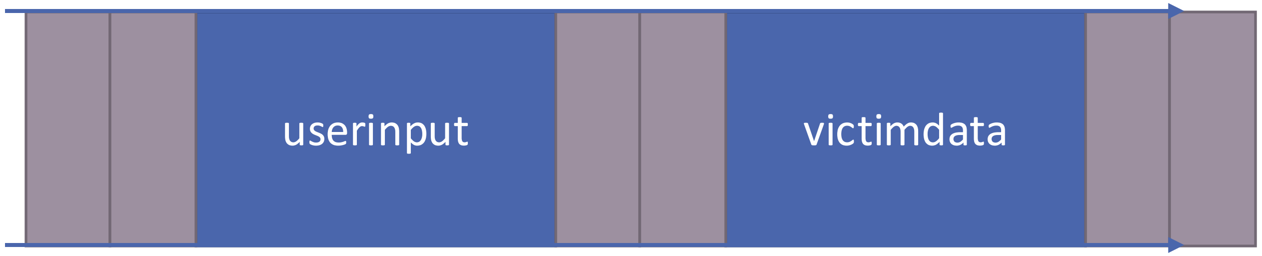
Reminder: Heap Overflow and Function Pointers

- Function pointers stored in the heap are at risk of heap overflows

```
struct callback {
        int priority;
        int (*cb_func)(void *data);
};
```

- On use of function pointer → control-flow hijacki

What if the developer is not using function pointers?



C++ Virtual Functions

```
class ClassA {
…
virtual void vfunc1() { /* code Avf1 */ }
void func1() { /* code Af1 */ }
};
```

```
class ClassB : ClassA {
…
virtual void vfunc1() { /* code Bvf1 */ }
virtual void vfunc2() { /* code Bvf2 */ }
void func1() { /* code Bf1 */ }
};
```

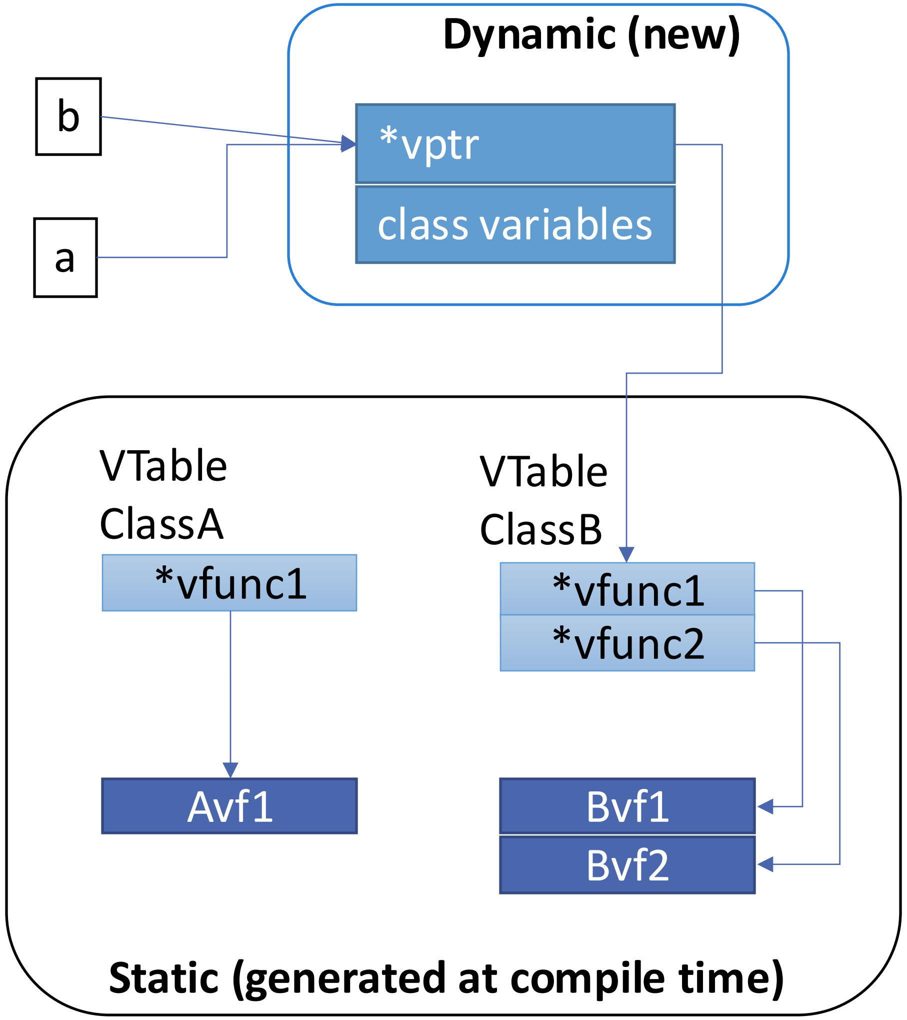
```
int main(int argc, char **argv)
{
        ClassA *a;
        ClassB *b;

        b = new ClassB();
        ….
        a = b;
        a->vfunc1();
        a->func1();
        ….
```

Which functions are called?

Late Binding and VTables

- The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation
- VTables are used to enable late binding

```
a->vfunc1();          mov 0(vptr), rax
b->vfunc1();          call *(%rax)

a->func1();           call avf1
```



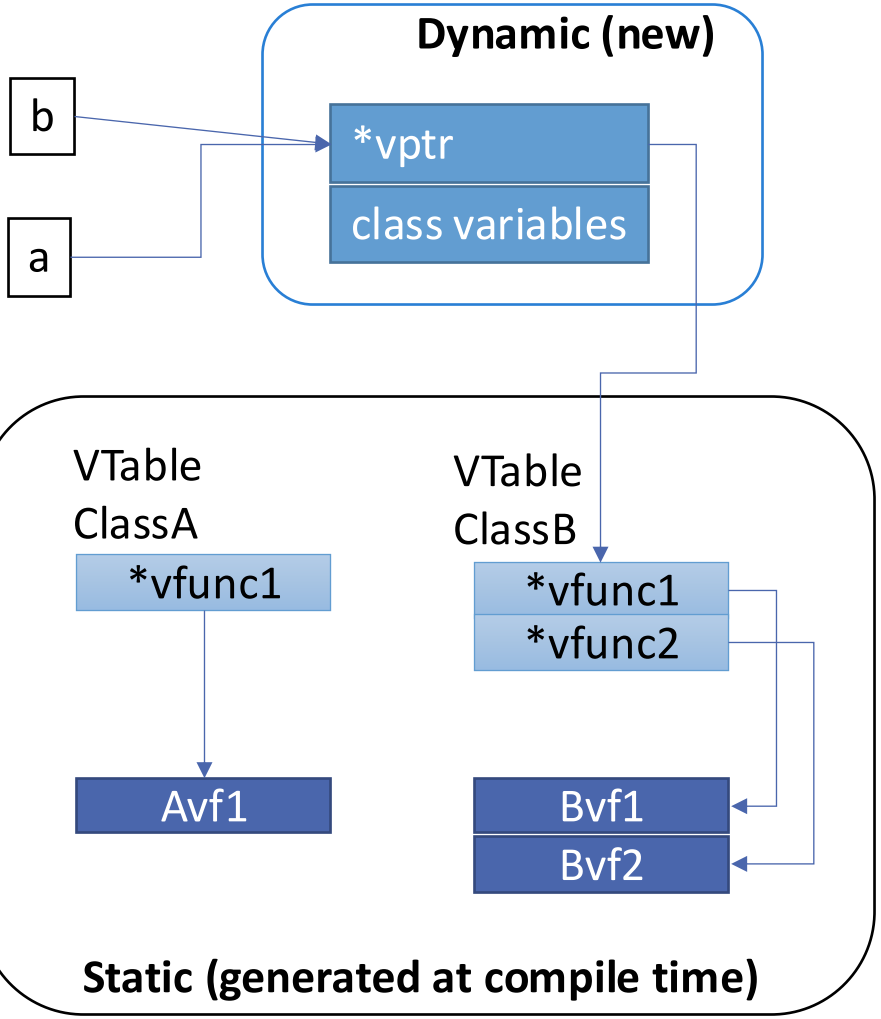
Late Binding and VTables

- The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation
- VTables are used to enable late binding

Becomes this pointer

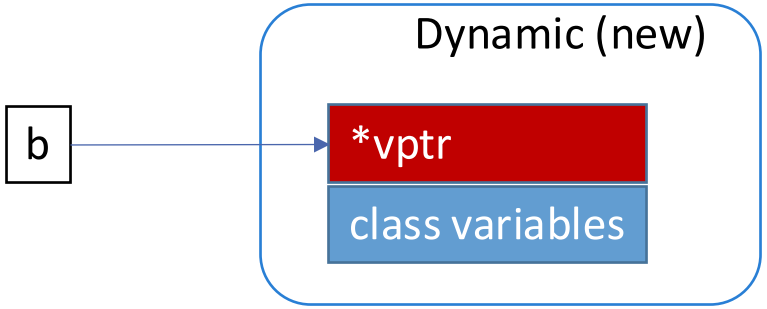
```
a->vfunc1();        mov 0(vptr), rax
b->vfunc1();        call *(%rax)

a->func1();         call avf1
```

```
classA::vfunc1(ClassA *this)
classA::func1(ClassA *this);
…
```



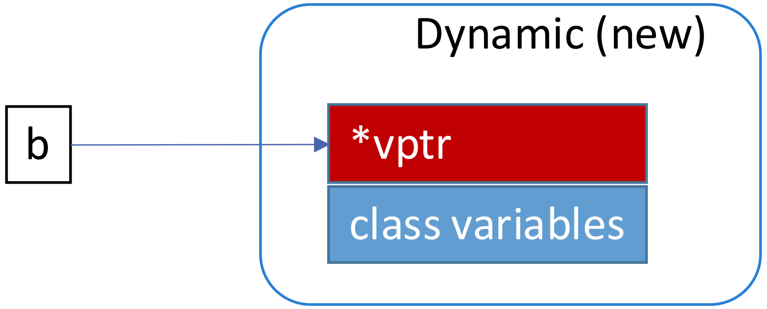
Late Binding and VTables

- The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation
- VTables are used to enable late binding
- **Heap overflows can be used to corrupt the object and vptr**



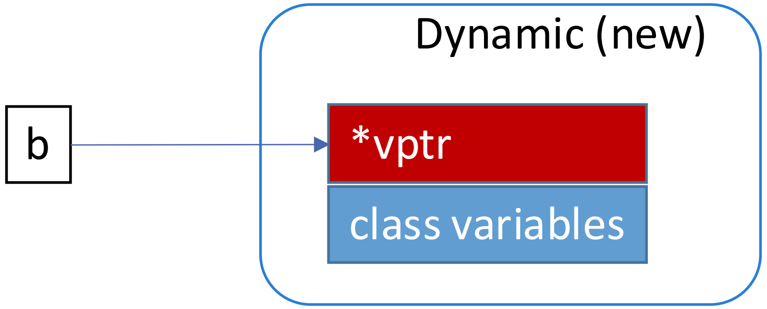
Late Binding and VTables

- The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation
- VTables are used to enable late binding
- **Heap overflows can be used to corrupt the object and vptr**
- **Corrupted vptr points to wrong VTable or fake, injected VTable**



Late Binding and VTables

- The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation
- VTables are used to enable late binding
- **Heap overflows can be used to corrupt the object and vptr**
- **Corrupted vptr points to wrong VTable or fake, injected VTable**
- **aka VTable hijacking attacks**



Control-Flow Integrity

Attacker Modus Operandi

- **Find memory corruption bug**
  - **Manipulate to take over program counter**
- Find ASLR bypass
  - Leak memory layout
  - Spray memory
  - Weakly or non-randomized sections/memory
- Inject ROP payload
  - Break W^X semantics
- Inject code

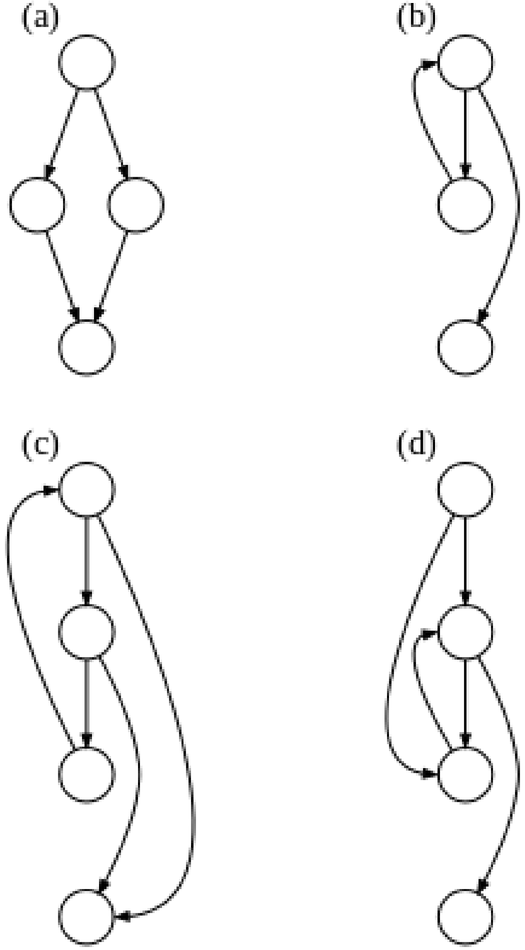
Attacker Modus Operandi

- **Find memory corruption bug**
  - **Manipulate to take over program counter**
- Find ASLR bypass
  - Leak me
  - Spray m
  - Weakly
- Inject RO
  - Break W
- Inject cod

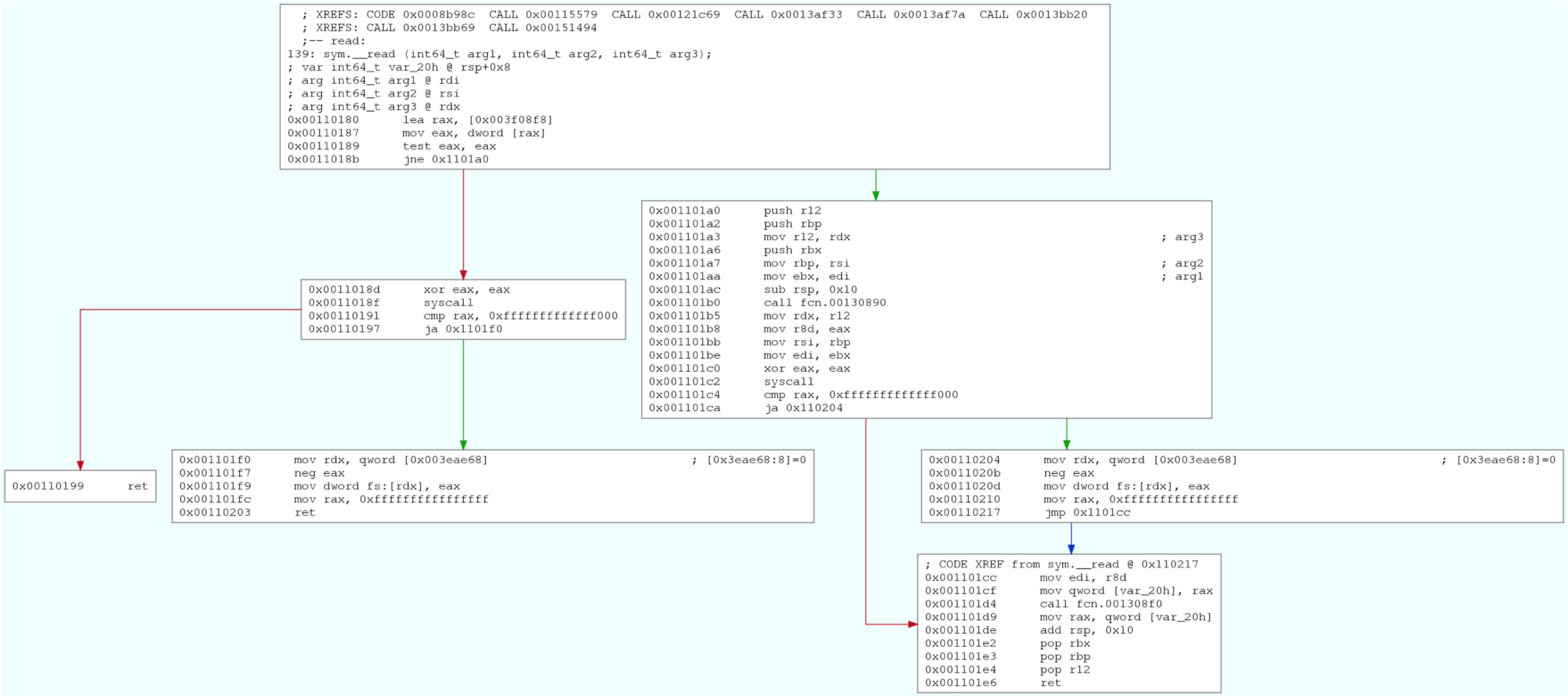
Control-flow Integrity aims to restrict the arbitrary manipulation of the program’s control flow

CFI and the Control-flow Graph (CFG)

- A **control flow graph** (CFG) in computer science is a representation, using **graph** notation, of all paths that might be traversed through a program during its execution. –Wikipedia
- Nodes are **basic blocks** (BBLs): a sequence of instructions with a single entry and single exit



CFG Example (Rendered)



CFI and the Control-flow Graph (CFG)

- A **control flow graph** (CFG) in computer science is a representation, using **graph** notation, of all paths that might be traversed through a program during its execution. –Wikipedia
- Nodes are **basic blocks** (BBLs): a sequence of instructions with a single entry and single exit
- CFI aims to enforce the program's CFG
  - Focus on hijackable control-flow edges

Control-Flow Hijacking Prone Statements

- Indirect calls, returns

```
return;
```

```
return 100;
```

```
void (*fptr)(arg1_type, arg2_type) = &my_function;

fptr(arg1, arg2);
```

- Calls to virtual functions are indirect calls

```
Class C {
   virtual void vcall(void);
}

C obj = new C();

obj->vcall():
```

Easily Observable in Machine Code

C Code

```
return;
```

```
return 100;
```

Machine Code

```
ret
```

```
pop %rax
jmp *(%rax)
```

```
void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);
```

```
jmp *(%rax)
```

```
call *(%rax)
```

```
Class C {
   virtual void vcall(void);
}

C obj = new C();

obj->vcall():
```

```
call *(%rax)
```

Extracting the CFG

- **With** source code
  - More reliable
  - Still not perfect
  - How to handle
    - Dynamically loaded libraries?
    - Callbacks
- Without source code
  - Requires accurate disassembly
  - Cannot accurately define all paths
  - Shared libraries are easier to handle

```
static void (*fptr)(char *string, int len);

void set_callback(void *ptr)
{
    fptr = ptr;
}

void process_items()
{
    for (string *s : items) {
        fptr(s->c_str, s->len);
    }
}
```

First CFI Proposal

- **Control-flow integrity (2009)**
  - http://dl.acm.org/citation.cfm?id=1609960
- Assumes code integrity is ensured (no code injection)
- Applied during compilation on the binary and all libraries
  - Incremental deployment is not supported (all or nothing)

Working with an Imperfect CFG

- Let's assume that we know/can learn
  - The location of every function
  - The location of every indirect branch instruction

- **Coarse-grained CFI can enforce the following**
  - Indirect calls should only transfer control to functions
    - Same for most jumps
  - Returns should only transfer control to instructions following a indirect call or jump
  - More permissive than the actual (potentially unknown) CFG but better than before

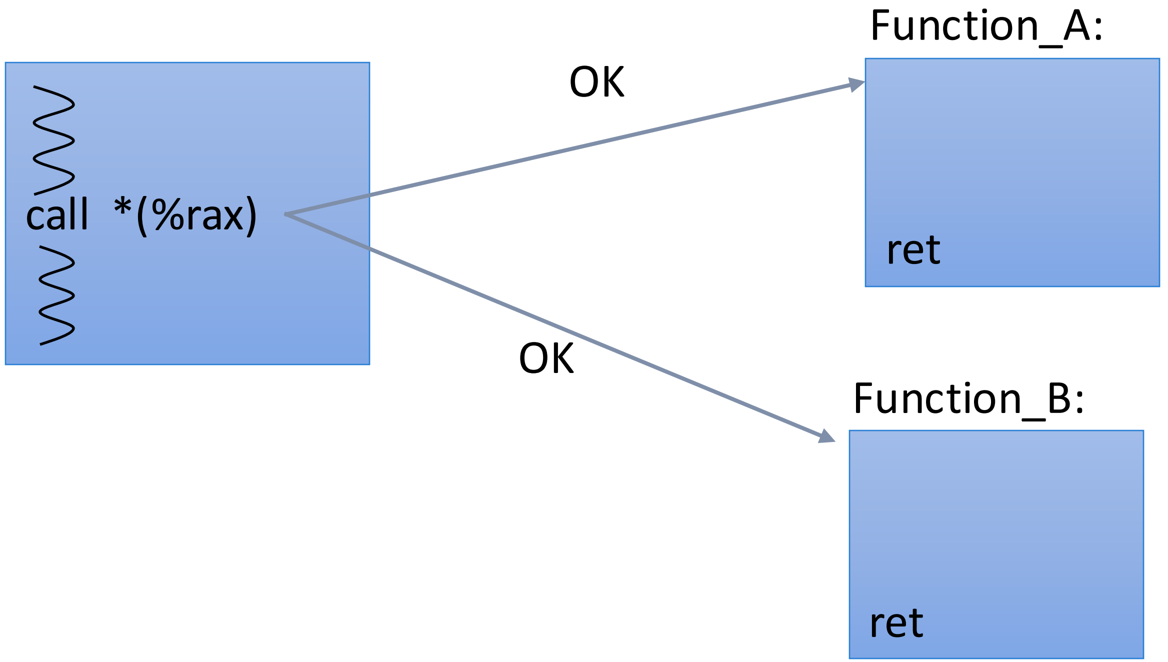
Forward and Backward in CFI

- Forward edges → Edges that correspond to indirect function calls in the FCG

- Backward edges → Edges that correspond to function returns in the FCG

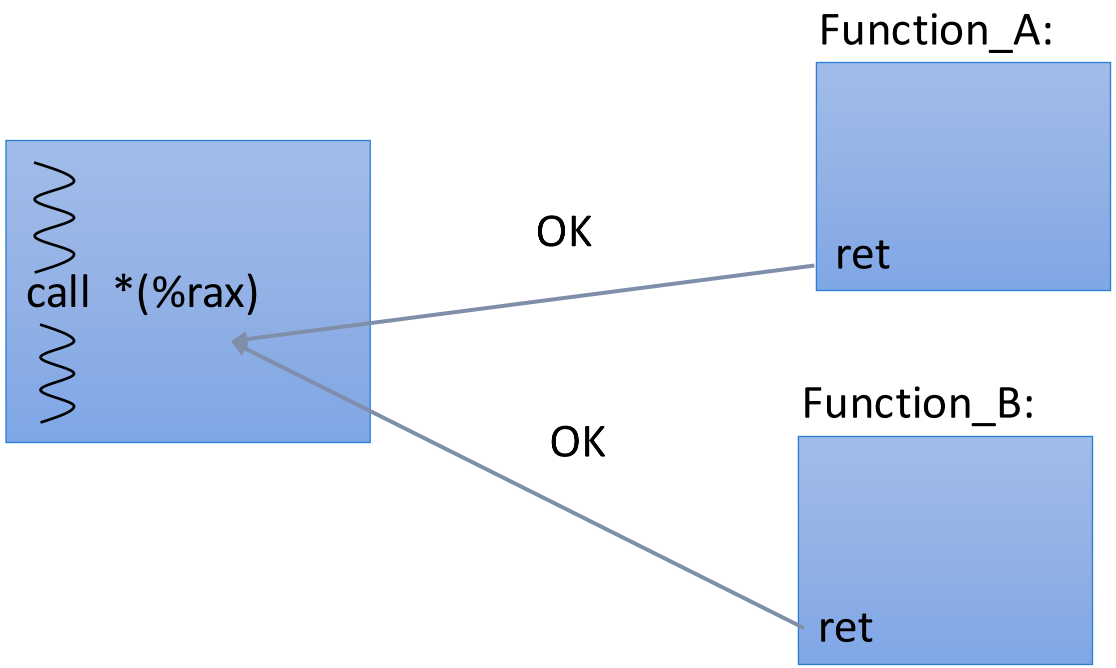
What is Allowed (Forward)

- Indirect calls should only transfer control to functions



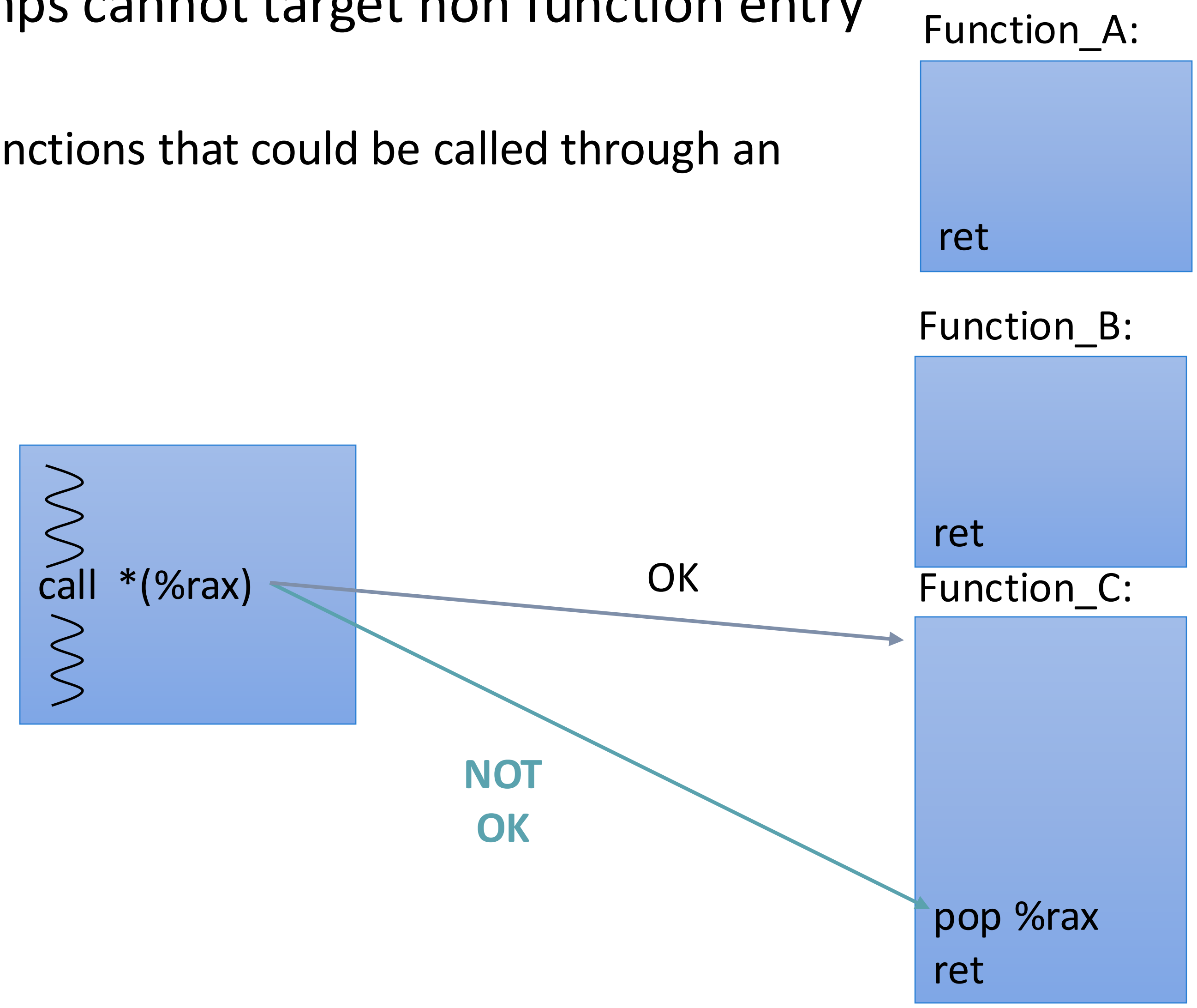
What is Allowed (Backward)

- Returns should only transfer control to instructions following a call or jump



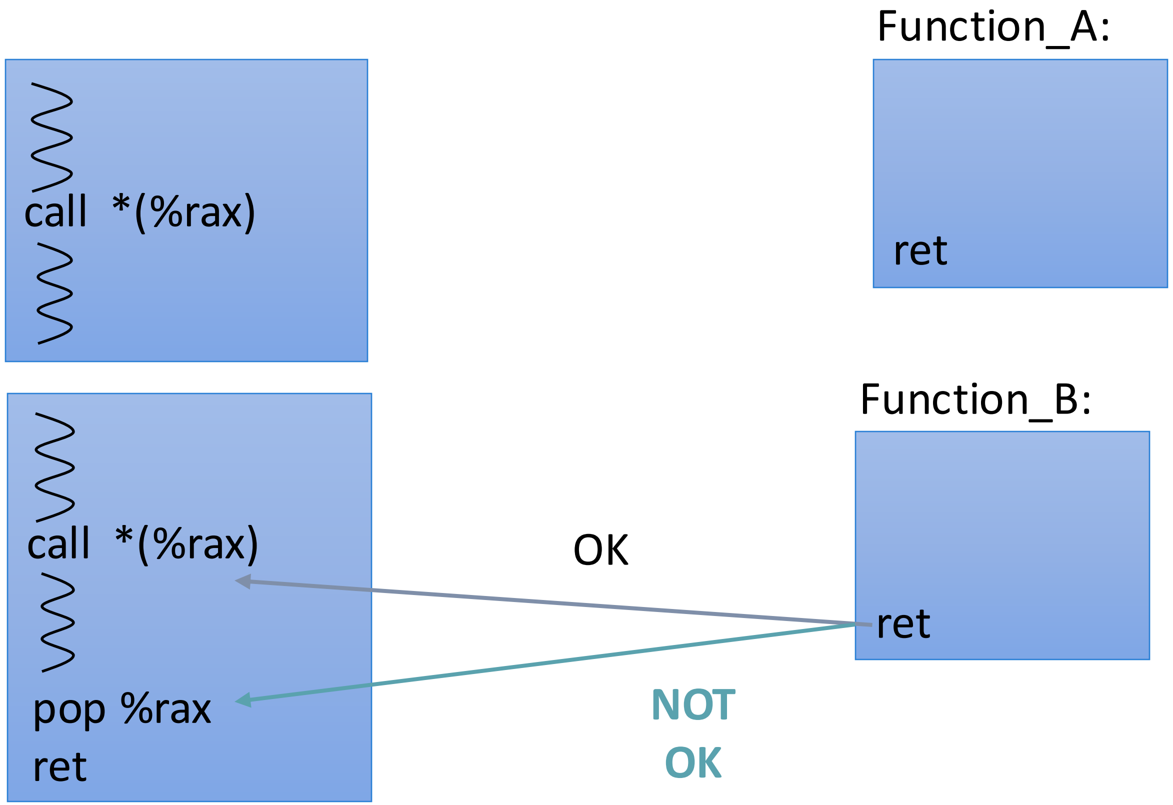
What is Not Allowed (Forward)

- Indirect calls/jumps cannot target non function entry points
  - But can target functions that could be called through an indirect call



What is Not Allowed (Backward)

- Returns cannot target bytes not following a call/jump



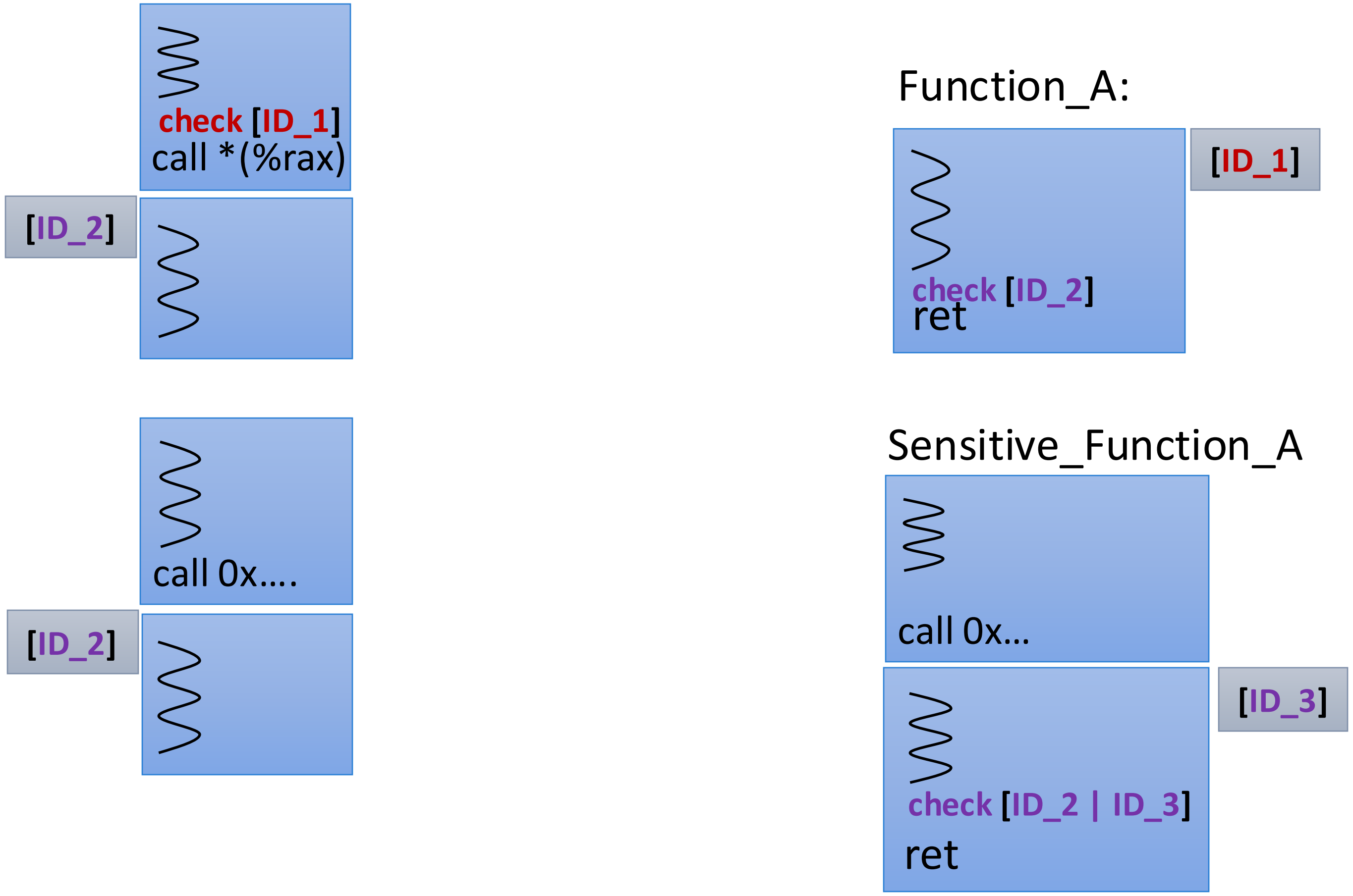
The Return of CFI

Coarse-Grained CFI for Binaries

- **Practical Control Flow Integrity and Randomization for Binary Executables (2013)**
  - http://dl.acm.org/citation.cfm?id=2498134
- Applied on binaries
- Extended (coarse-grained) restrictions
  - Only functions that can be called through a pointer can be targeted by indirect calls/jumps
    - Exported/imported functions
    - Address taken functions (functions that have an associated pointer)
  - The concept of sensitive functions is introduced
    - Functions important in exploits

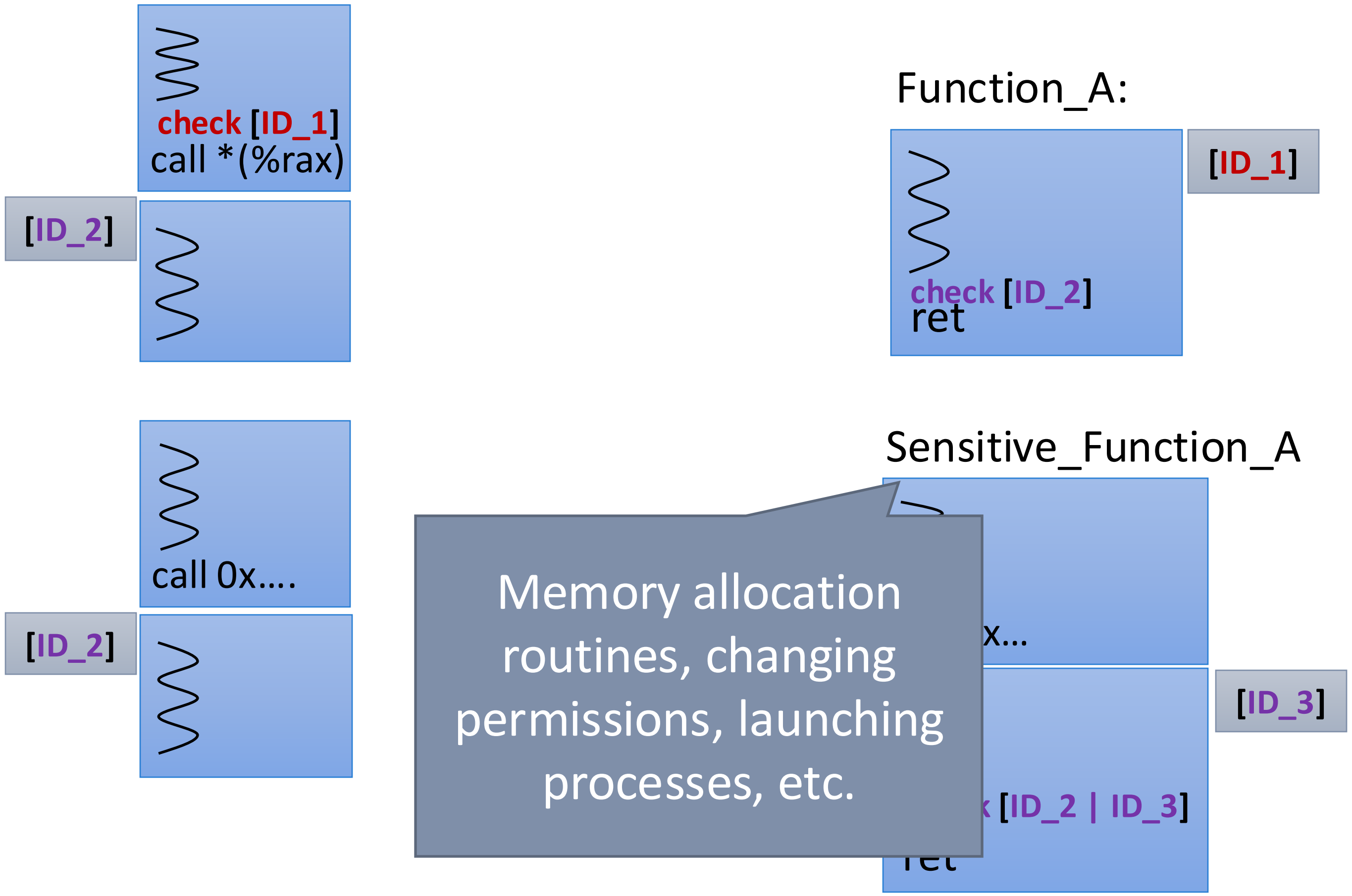
CCFIR

- Three IDs are used to restrict control flow



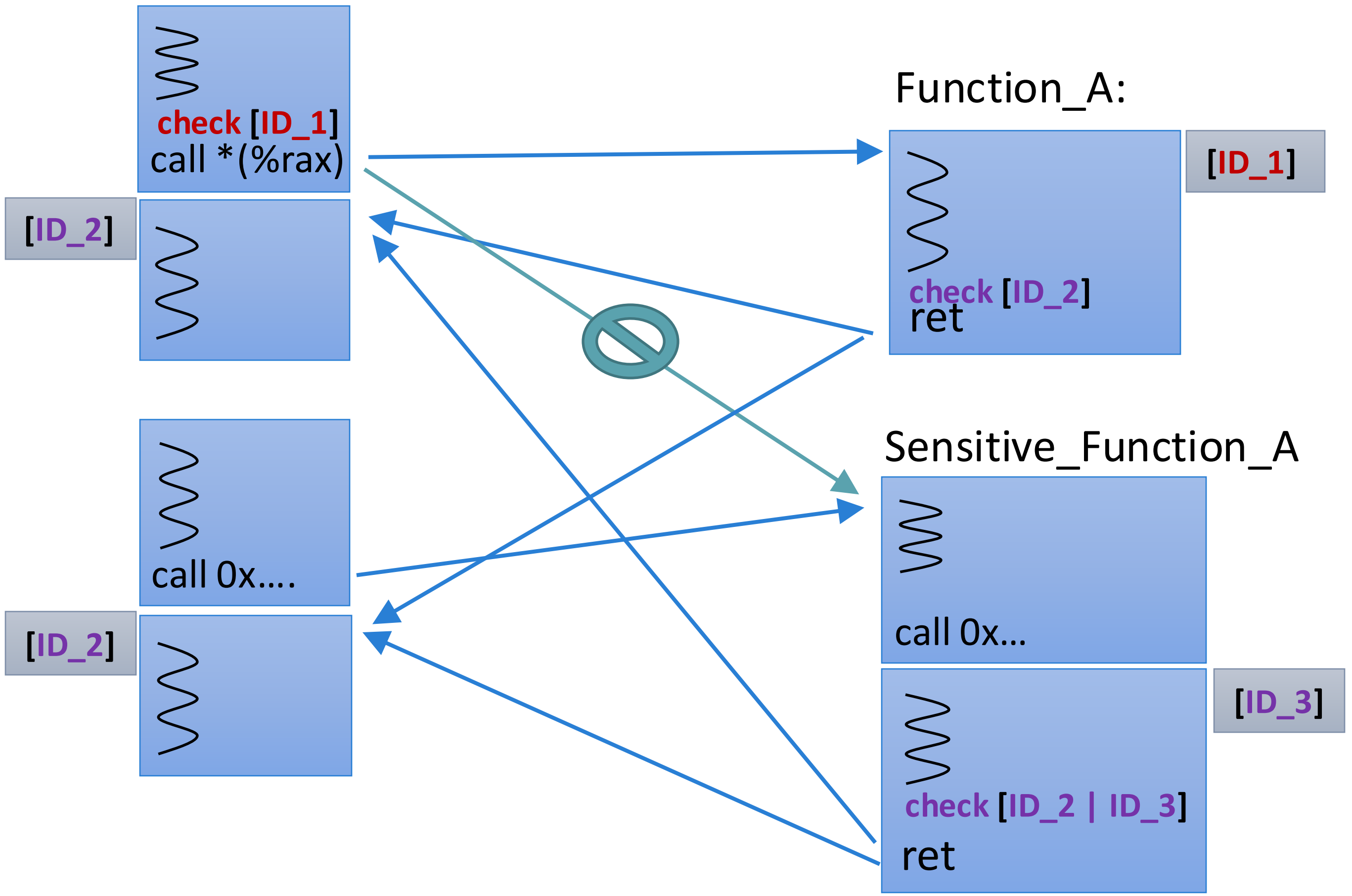
CCFIR

- Three IDs are used to restrict control flow



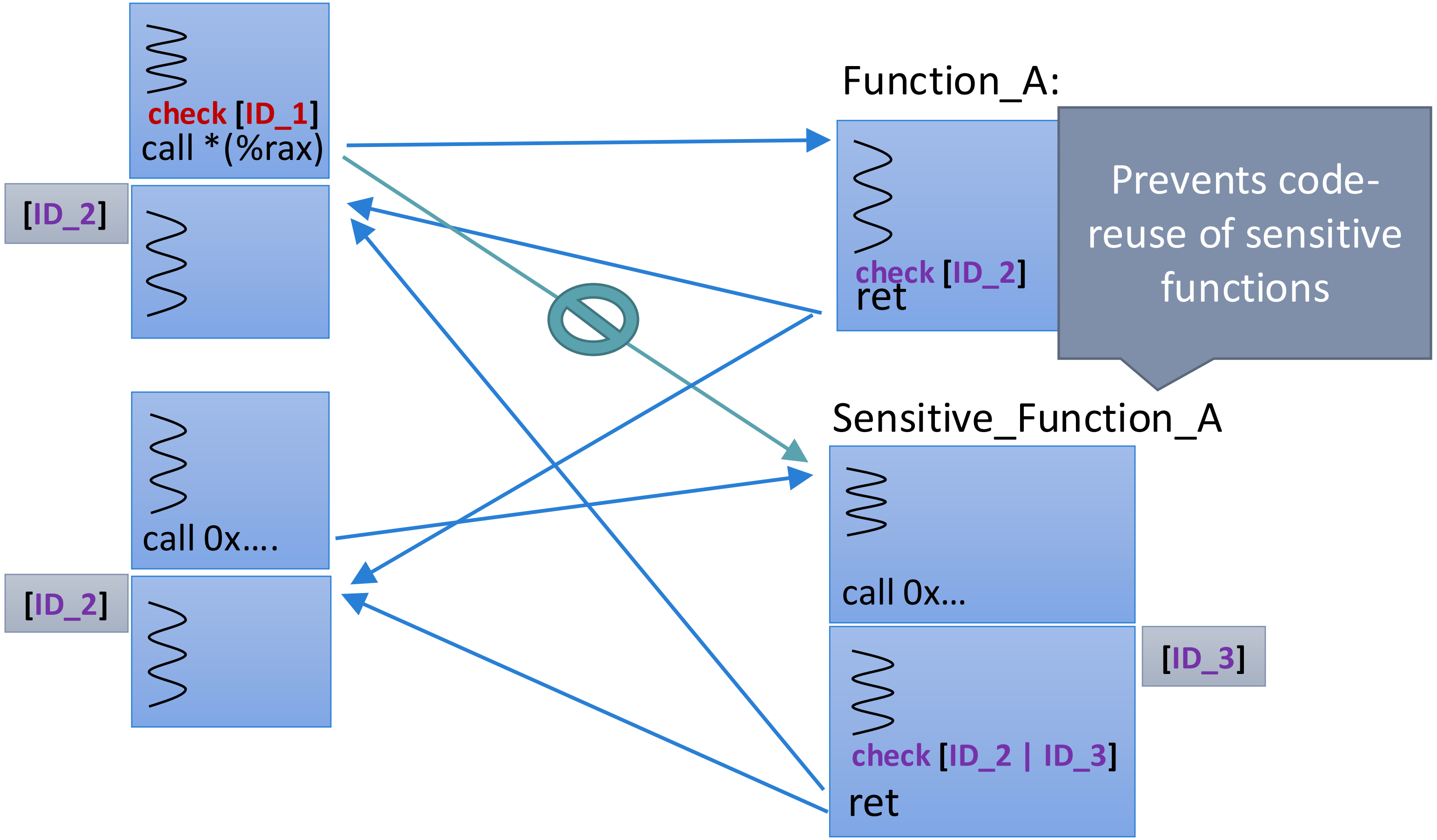
CCFIR

- Three IDs are used to restrict control flow

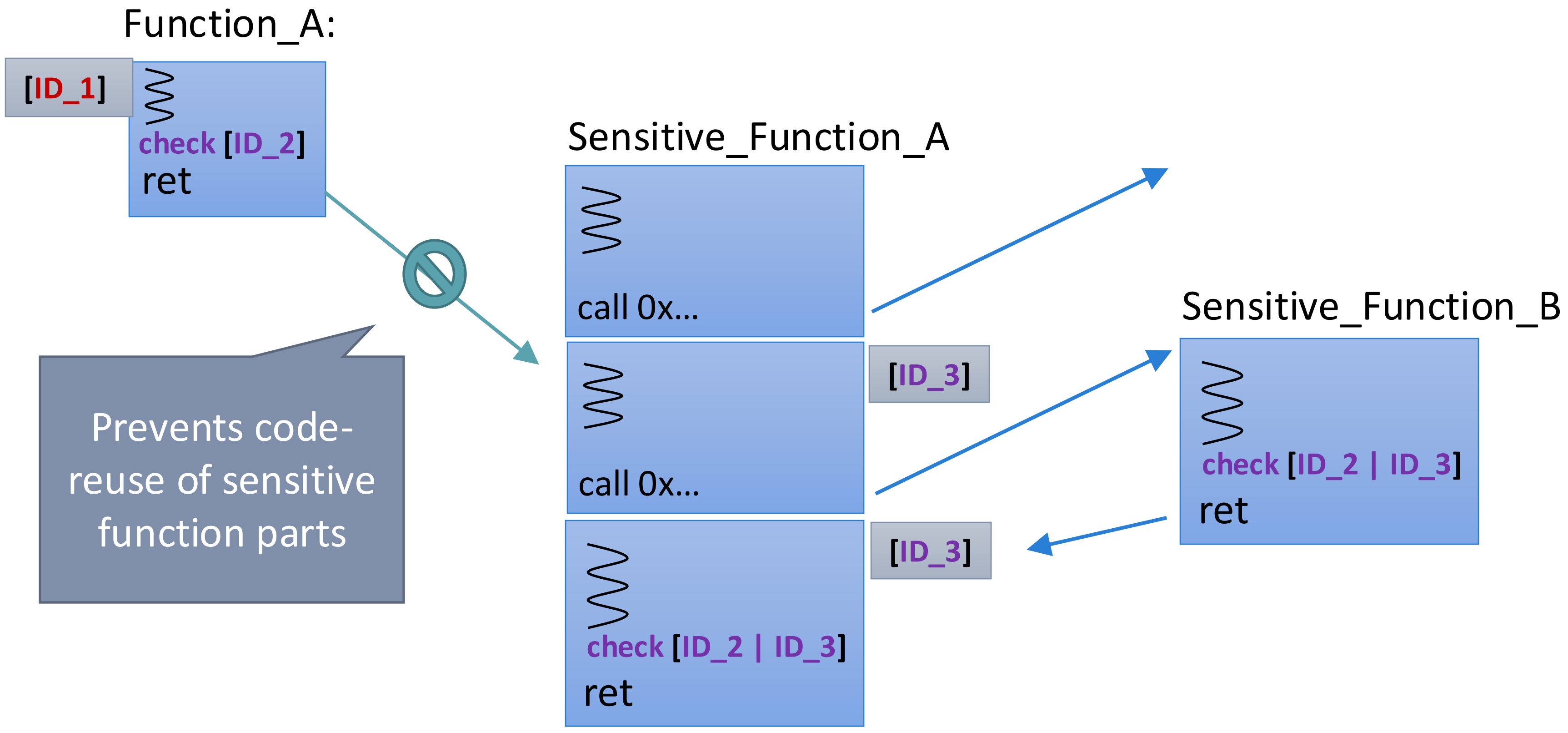


CCFIR

- Three IDs are used to restrict control flow



Sensitive Functions Heuristic



Supporting Legacy Libraries

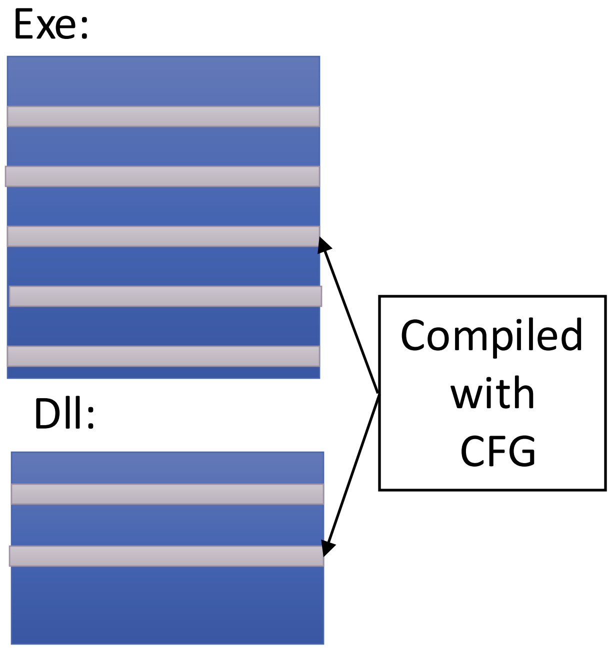
Microsoft’s Control-Flow Guard

- Included in MS Visual Studio
- Inserts control-flow checks before indirect calls during compilation
- A bitmap marks the allowed targets

```
check bitmap[%rax]
call *(%rax)
```

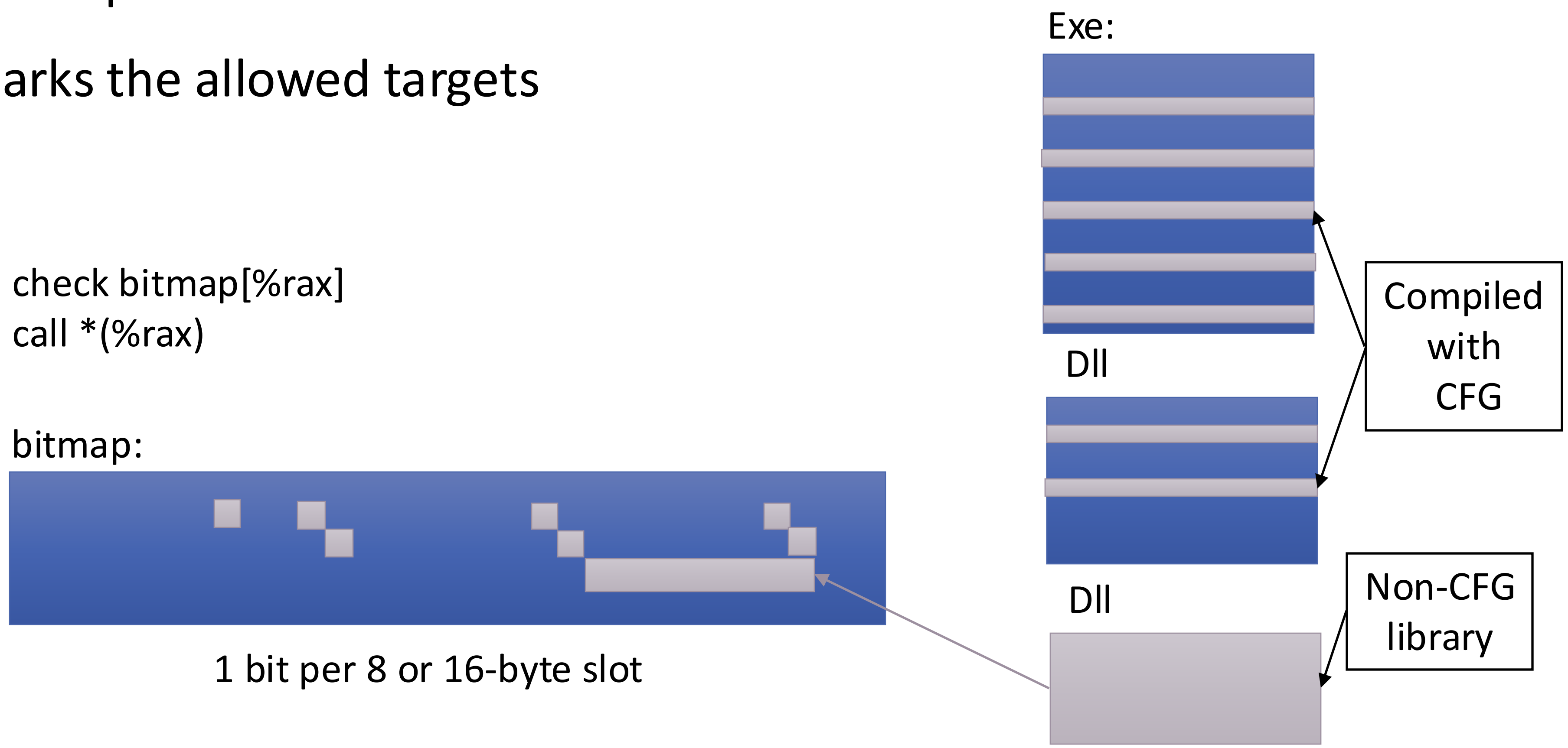
bitmap:

1 bit per 8 or 16-byte slot



Microsoft’s Control-Flow Guard

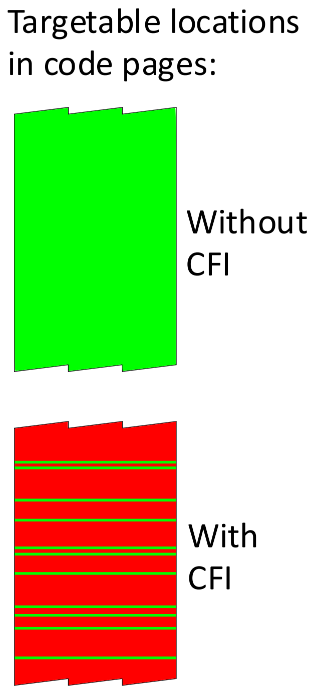
- Included in MS Visual Studio
- Inserts control-flow checks before indirect calls during compilation
- A bitmap marks the allowed targets



Attacking CFI

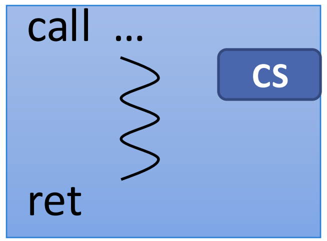
Reachable Targets Under CFI

- Most instructions cannot be targeted **(> 98%)**

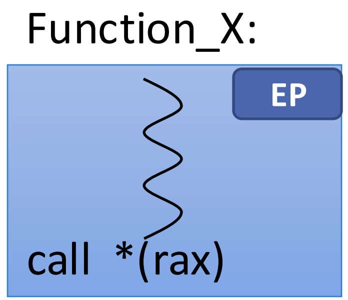


What is Left

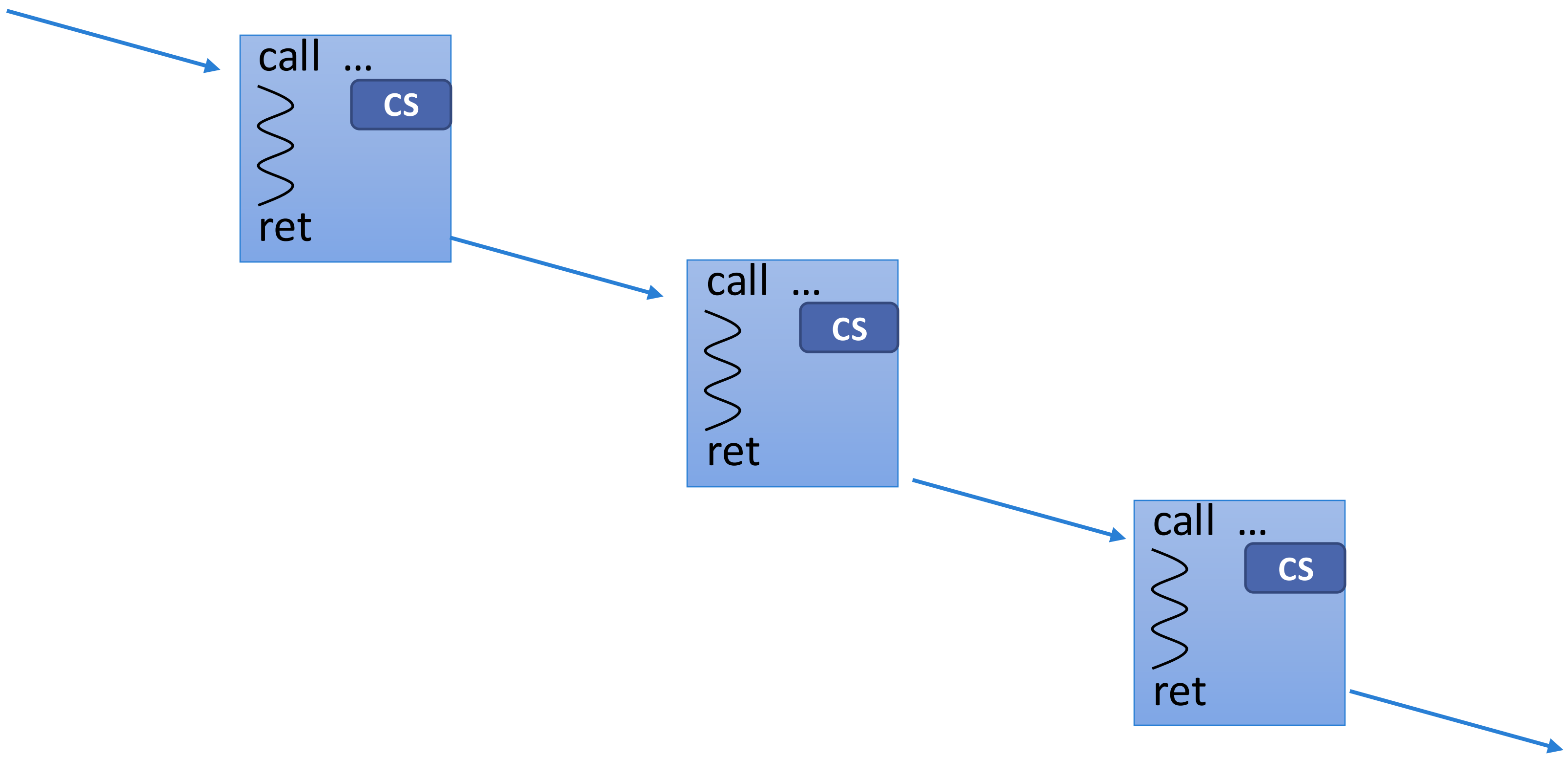
- Call Sites (**CS**)
  - Targetable by **return** instructions
  - CS gadgets
  - Return Oriented Programming (ROP)



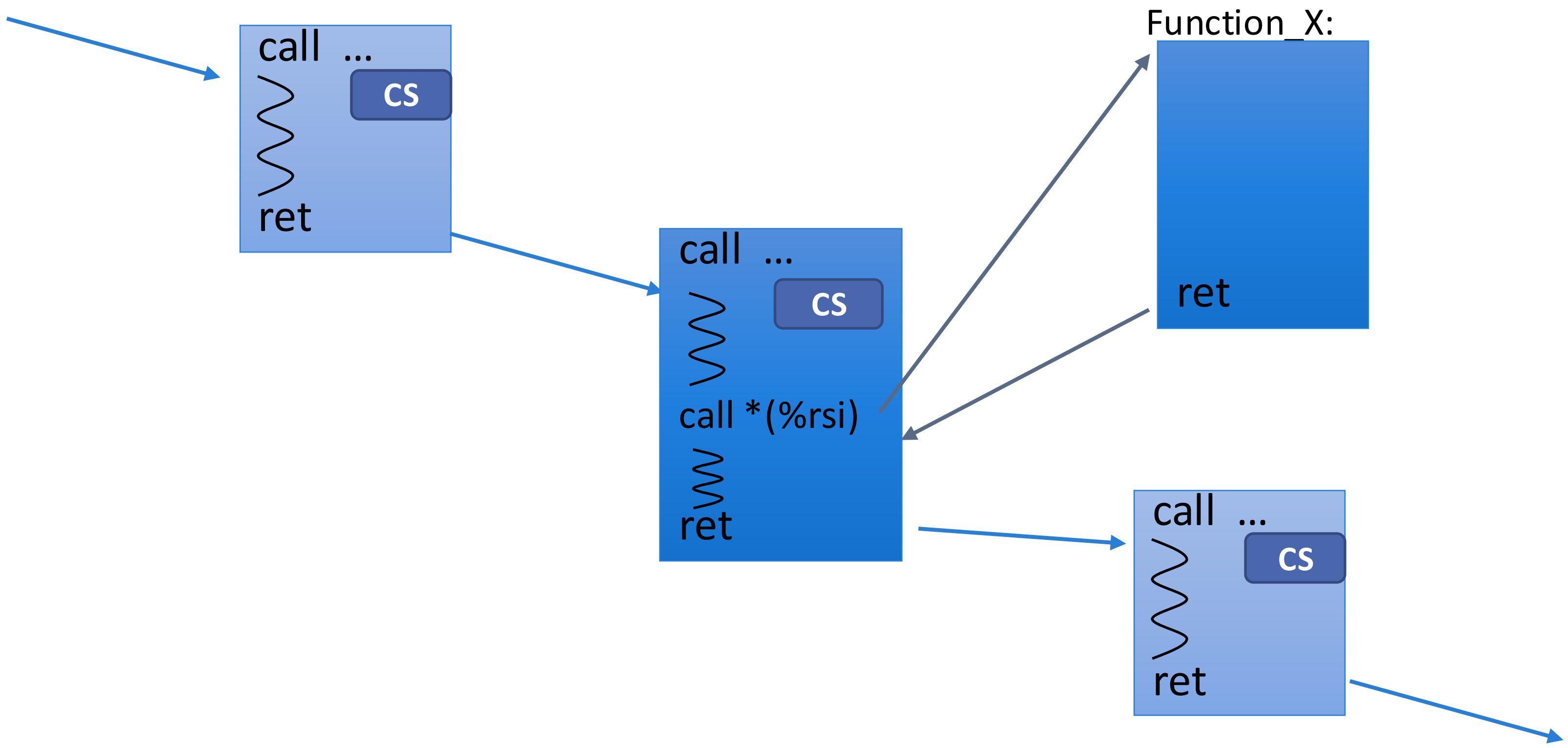
- Function Entry Points (**EP)**
  - Targetable by **indirect call** and **indirect jump** instructions
  - EP gadgets
  - Call Oriented Programming (COP)



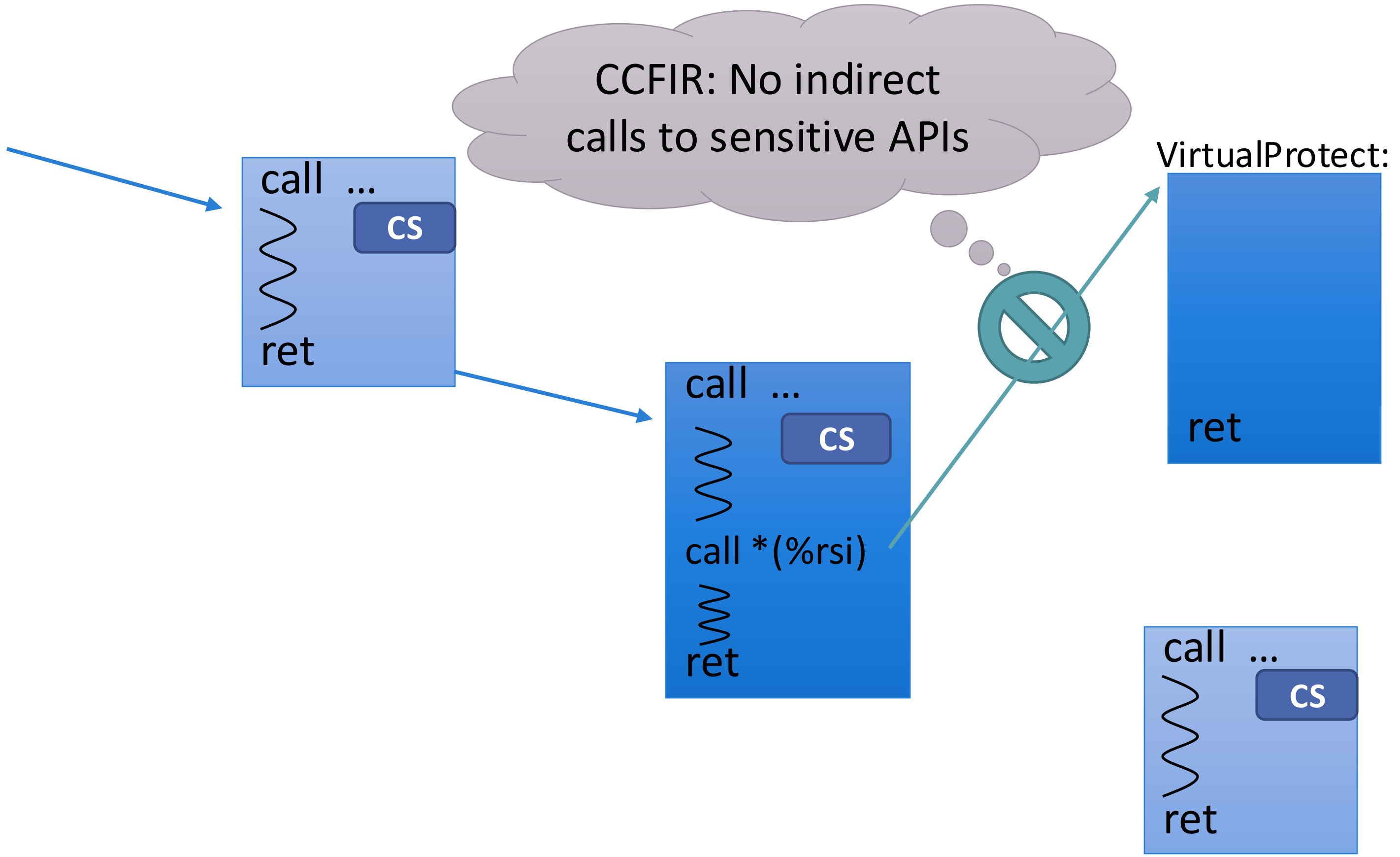
CS gadgets: Linking



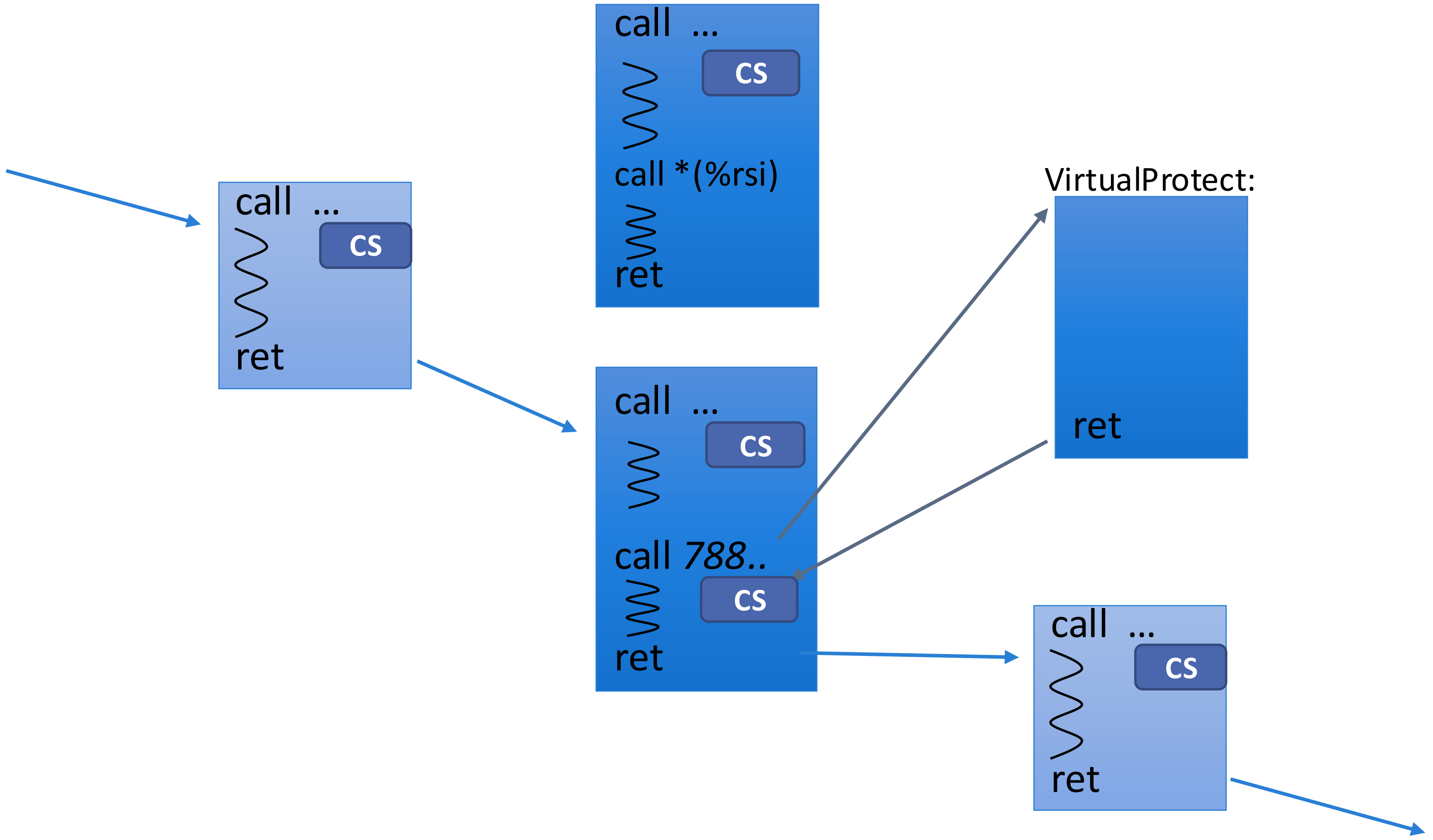
CS gadgets: Calling Functions



CS gadgets: Calling Sensitive Functions

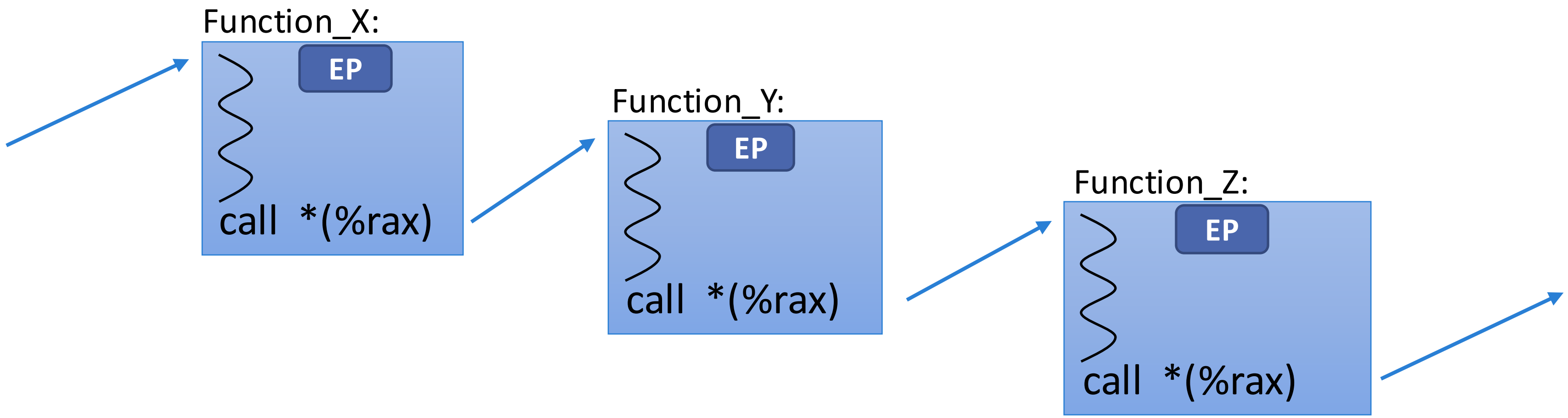


CS gadgets: Calling Sensitive Functions

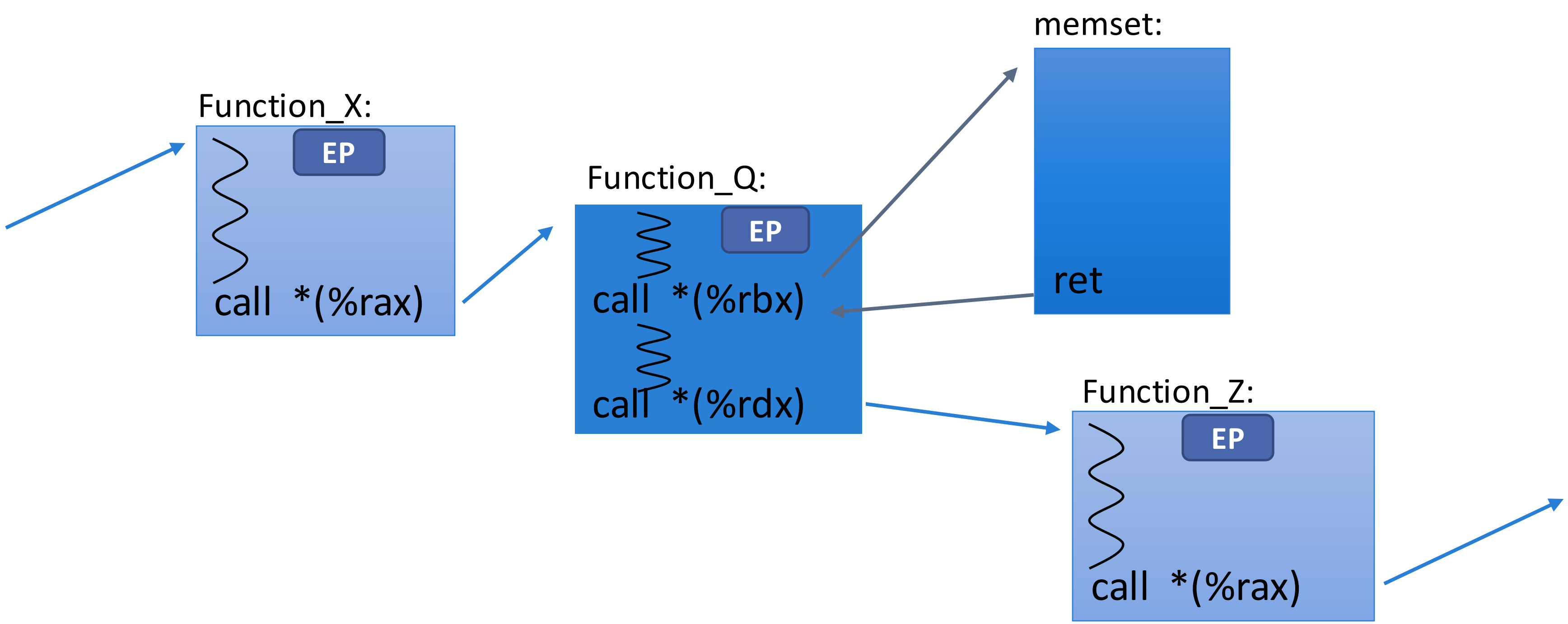


EP gadgets: Linking

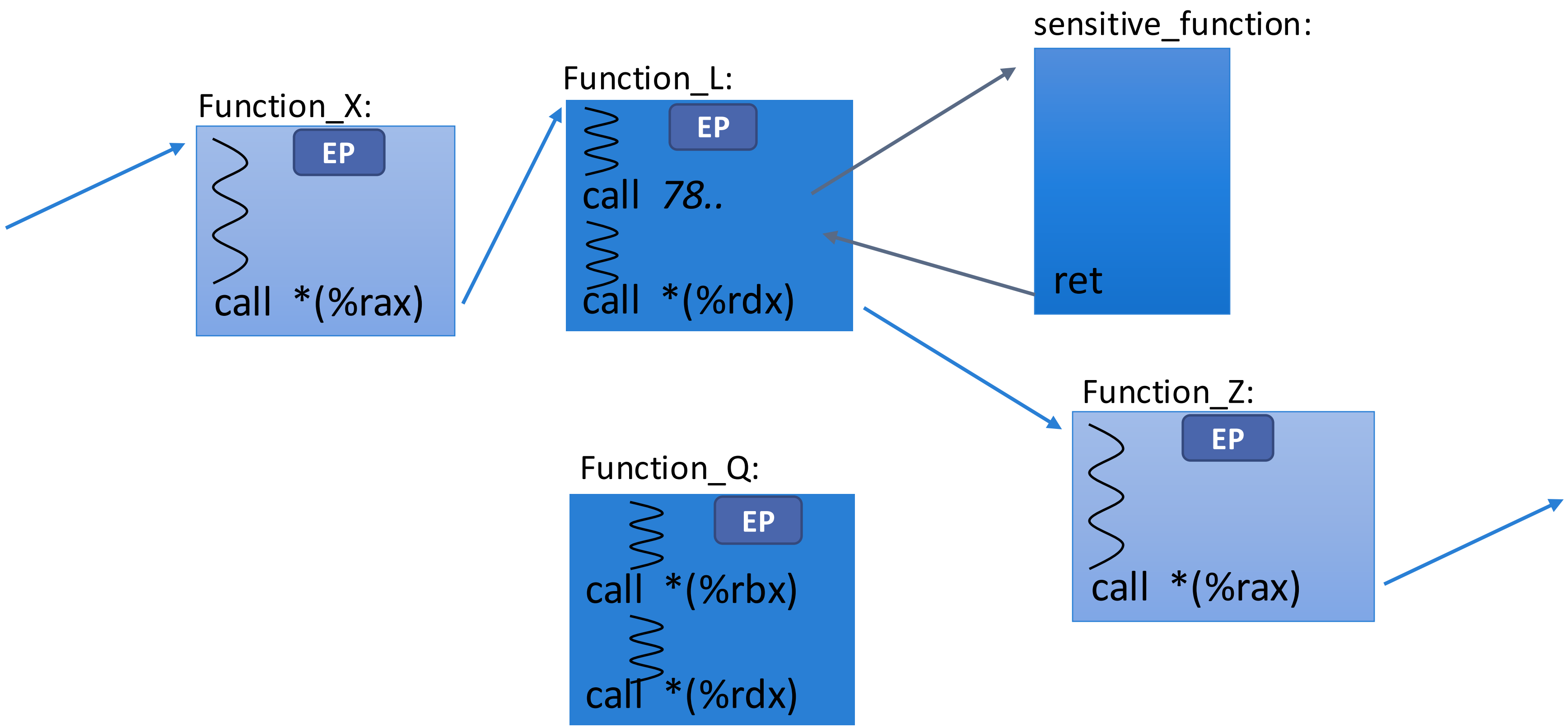
- Chaining is significantly harder



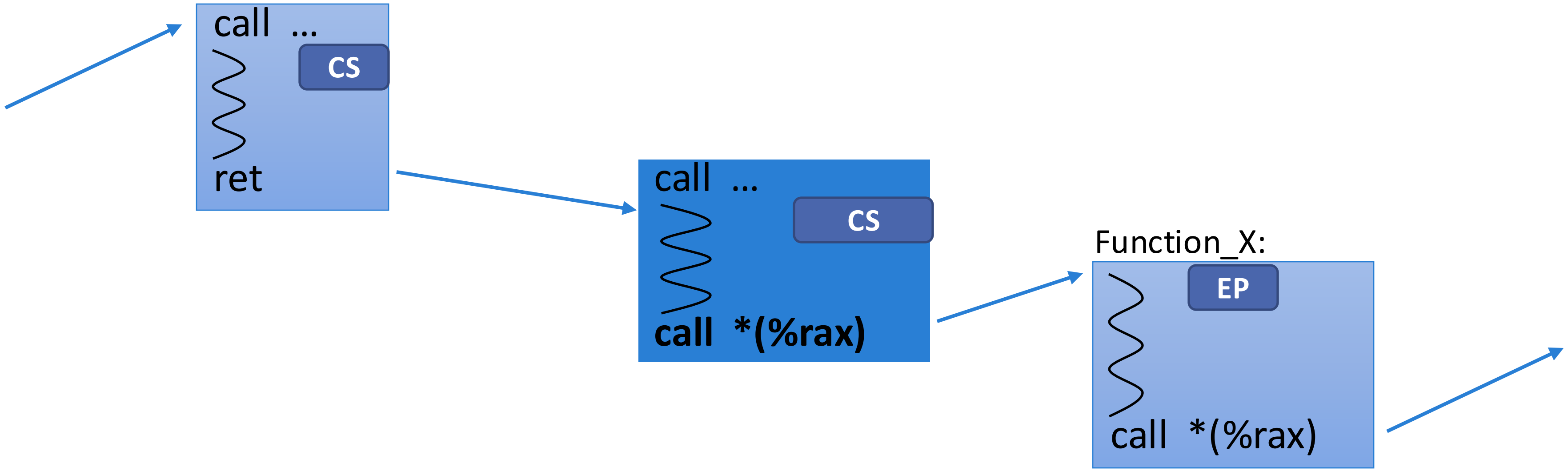
EP gadgets: Calling Functions



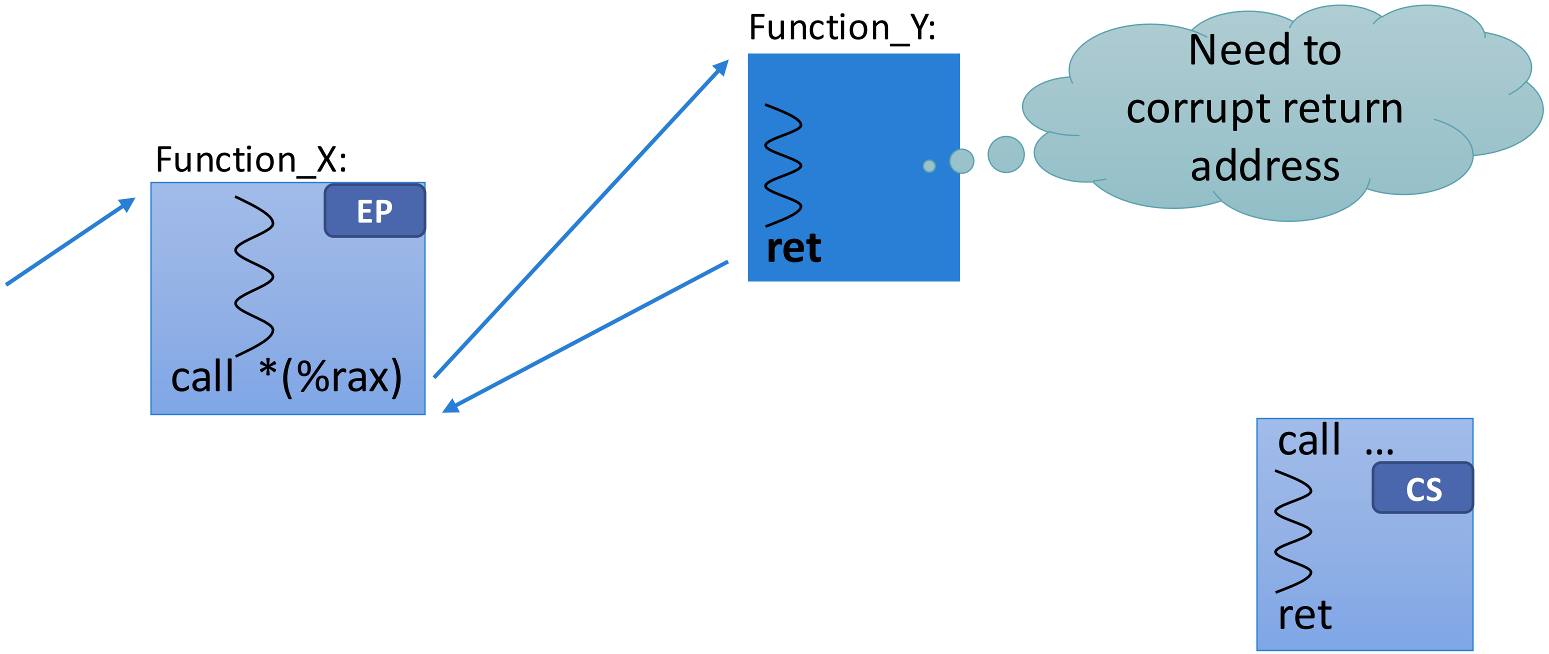
EP gadgets: Calling Functions



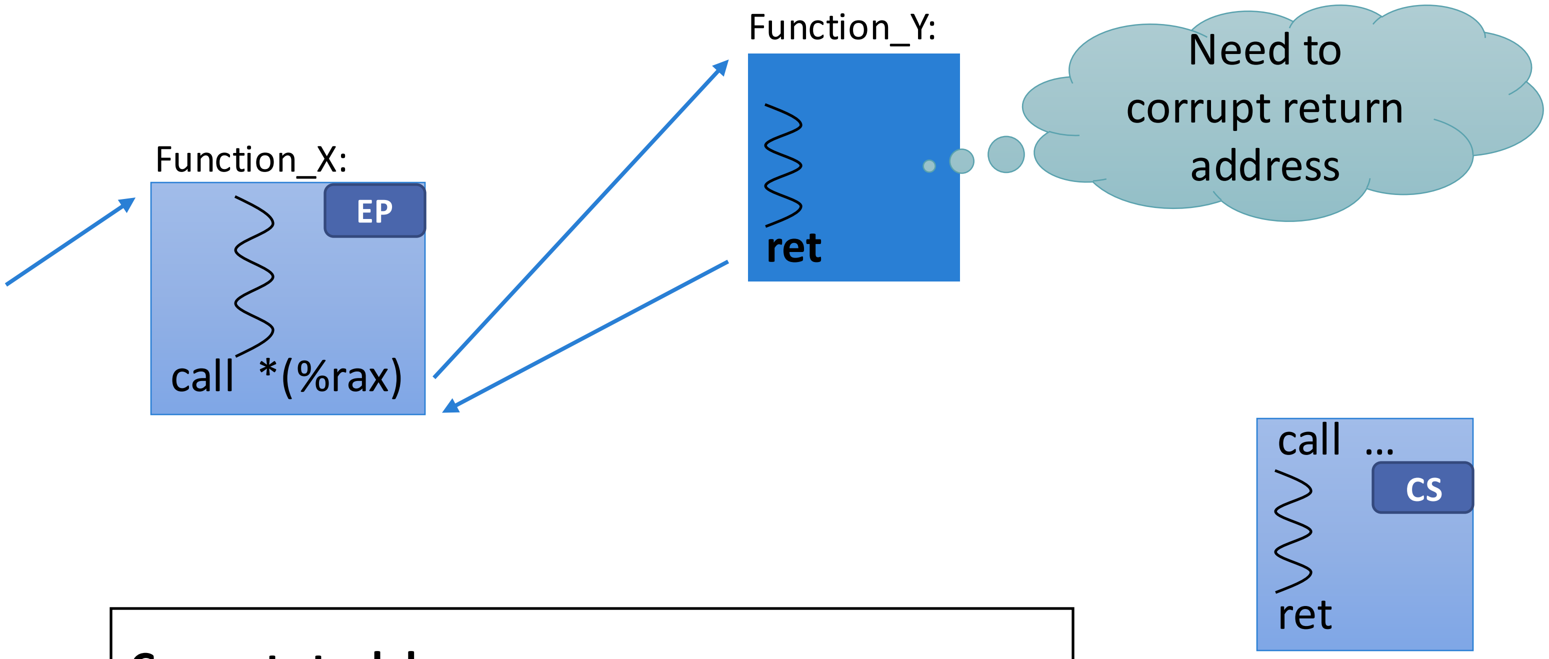
Switch Control: CS → EP



Switch Control: EP → CS



Switch Control: EP → CS



Corrupt stack by
- breaking calling conventions
- Self-corrupting function (e.g., memcpy())

Compromising Coarse-grained CFI is Possible

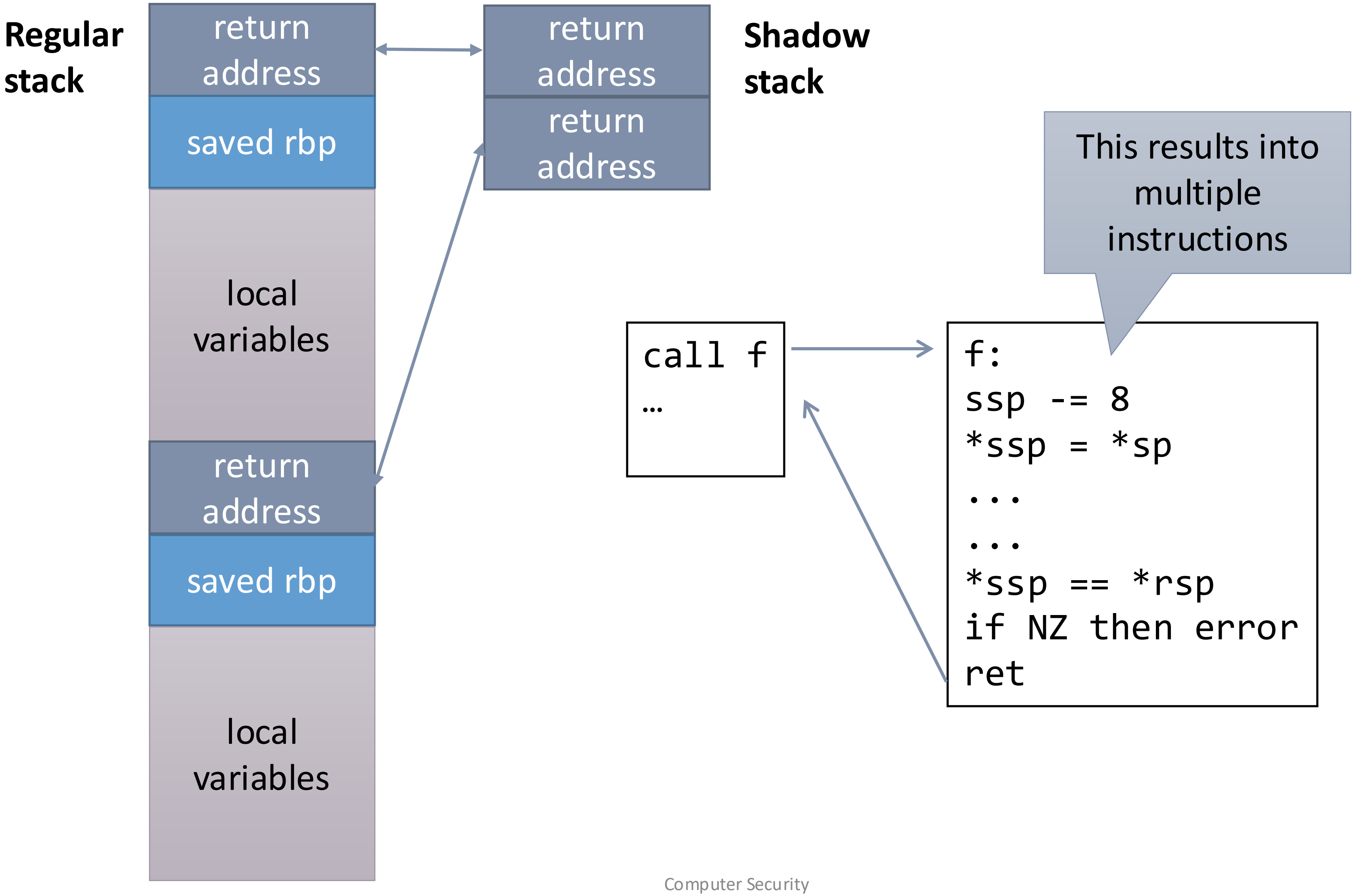
- https://www.portokalidis.net/files/outofcontrol_oakland14.pdf
- Exploiting **Internet Explorer 8**
  - Vulnerability: Heap Overflow (CVE-2012-1876)
  - https://web.archive.org/web/20150521040626/http://www.vupen.com:80/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php
- Assume **ASLR / DEP / CCFIR** in place
- First controlled indirect branch instruction: **`jmp edx`**
- (EP → CS) + VirtualProtect + memcpy = Code Injection
- Challenges: Larger gadgets have side effects that must be considered

Finer-Grained CFI

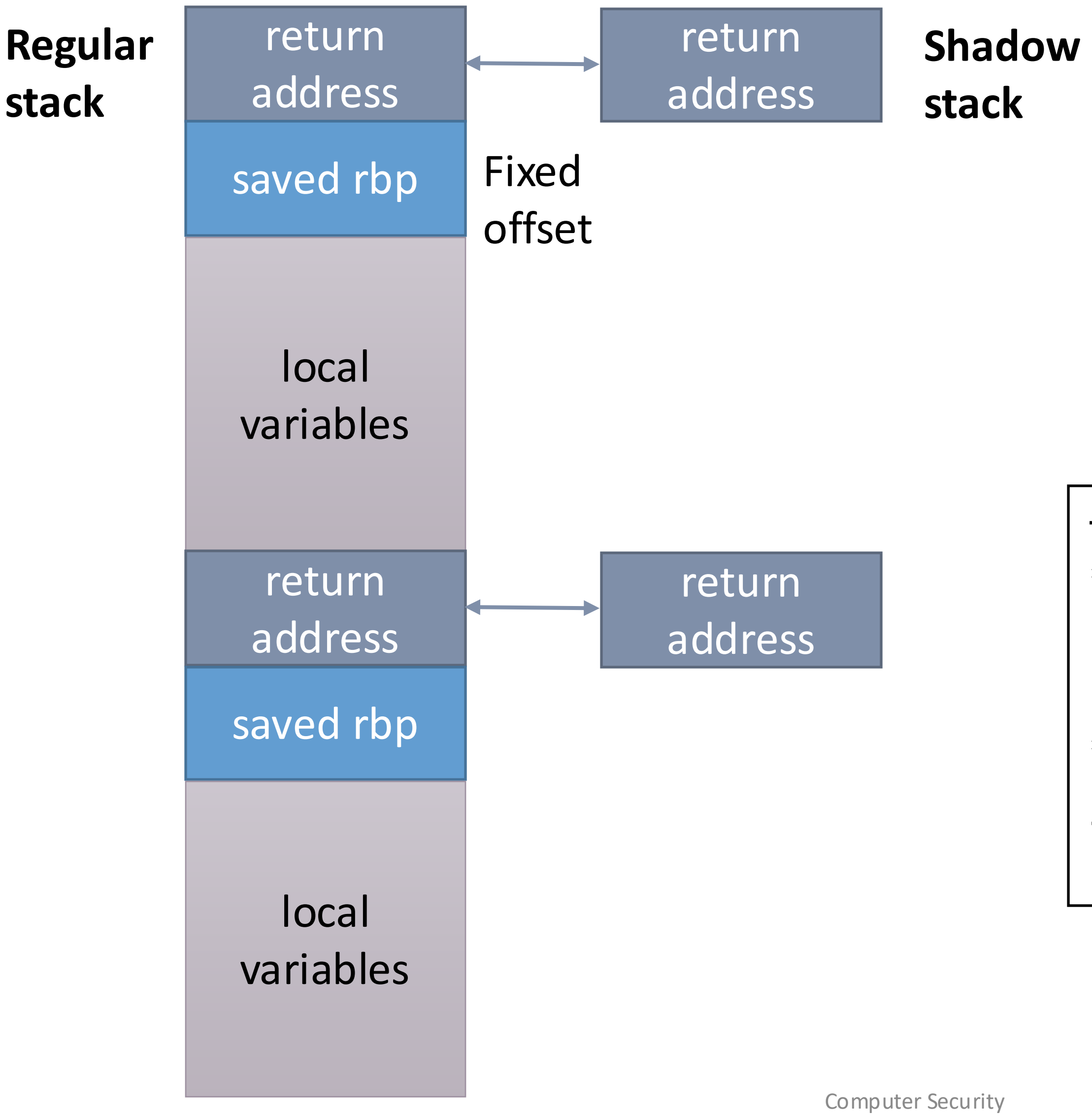
- More accurate CFG → only allow calls to target the functions they actually were intended to
  - Match # of arguments prepared at call site to the # of called function parameters
  - Resolve all possible values of a function pointer (harder)
  - Examples:
    - Modular Control-Flow Integrity http://www.cse.psu.edu/~gxt29/papers/mcfi.pdf
    - Practical Context-Sensitive CFI https://www.cs.vu.nl/~giuffrida/papers/ccs-2015.pdf

Shadow Stacks

Shadow Stacks



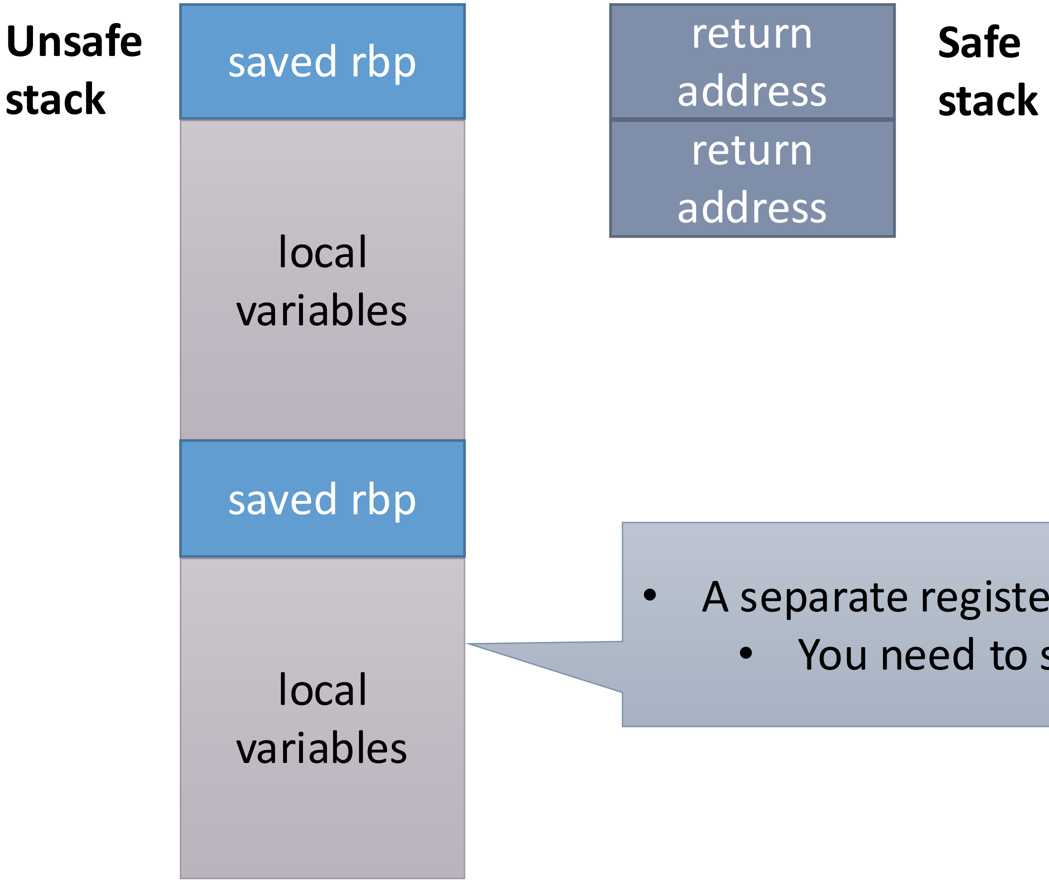
Shadow Stacks



This results into less instructions

```
f:
*(sp+off) = *sp
...
...
*(sp+off) == *sp
if NZ then error
ret
```

Shadow vs (Un)safe Stacks



- A separate register (not %rsp) used
  - You need to sacrifice a register

Shadow Stack Limitations

- Performance is the main obstacle for adoption
  - The Performance Cost of Shadow Stacks and Stack Canaries
  - https://people.eecs.berkeley.edu/~daw/papers/shadow-asiaccs15.pdf
- Time Of Check Time Of Use (TOCTOU) vulnerabilities
- CALL-RET mismatches can break applications
  - For example, when using setjmp/longjmp (exception handling, etc.)
- Certain implementations can be affected by various compilers optimization
  - For example: tail-call elimination
- How to support legacy libraries?

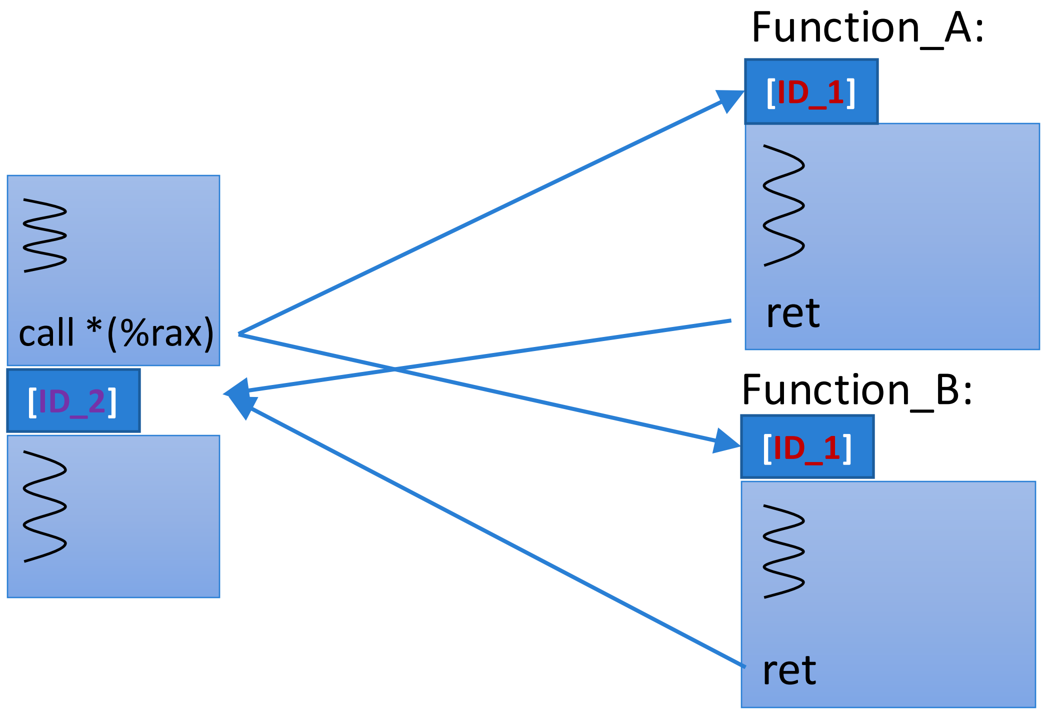
Appendix: Original CFI

First CFI Proposal

- **Control-flow integrity (2009)**
  - http://dl.acm.org/citation.cfm?id=1609960
- Assumes code integrity is ensured (no code injection)
- Applied during compilation on the binary and all libraries
  - Incremental deployment is not supported (all or nothing)

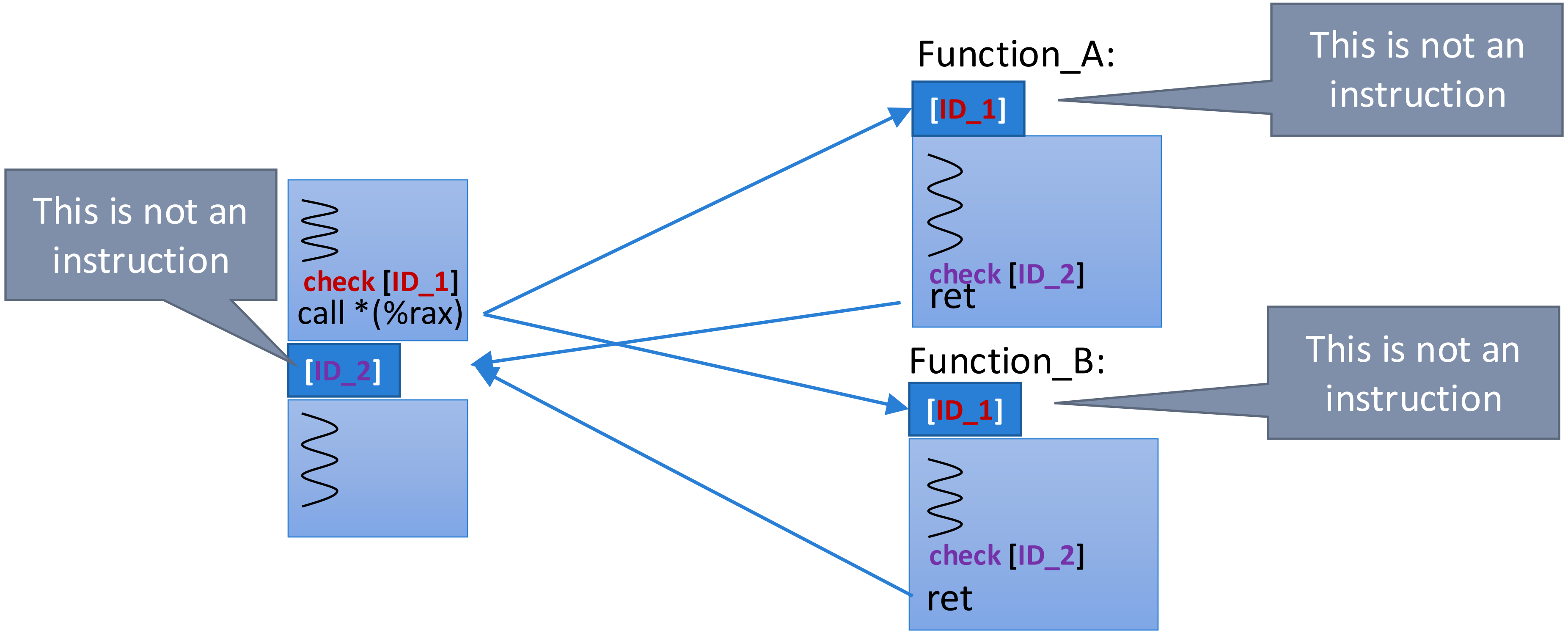
Enforcing Through Embedded IDs

- ID codes are embedded into the binary program to identify acceptable targets
  - 2-ID policy

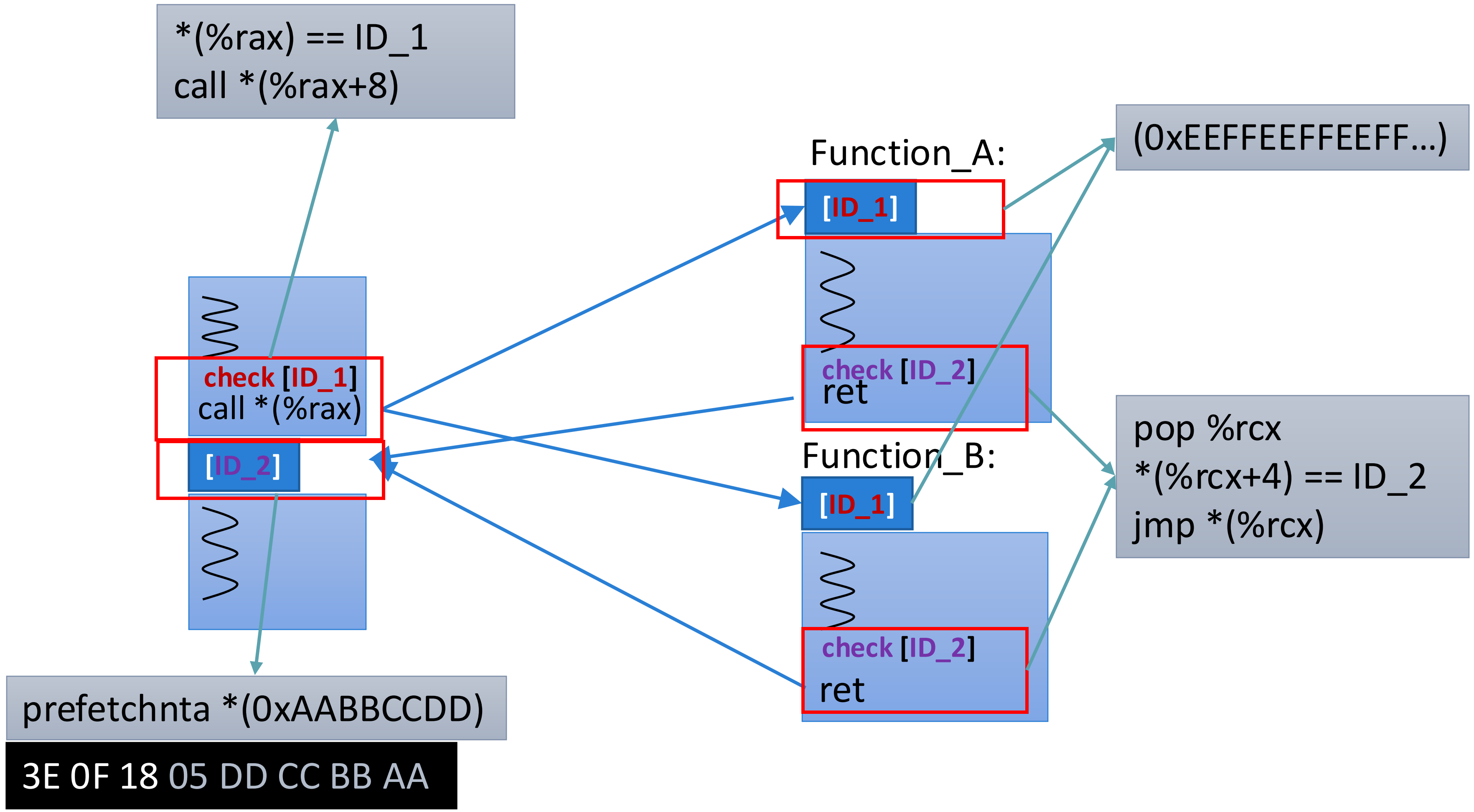


Enforcing Through Embedded IDs

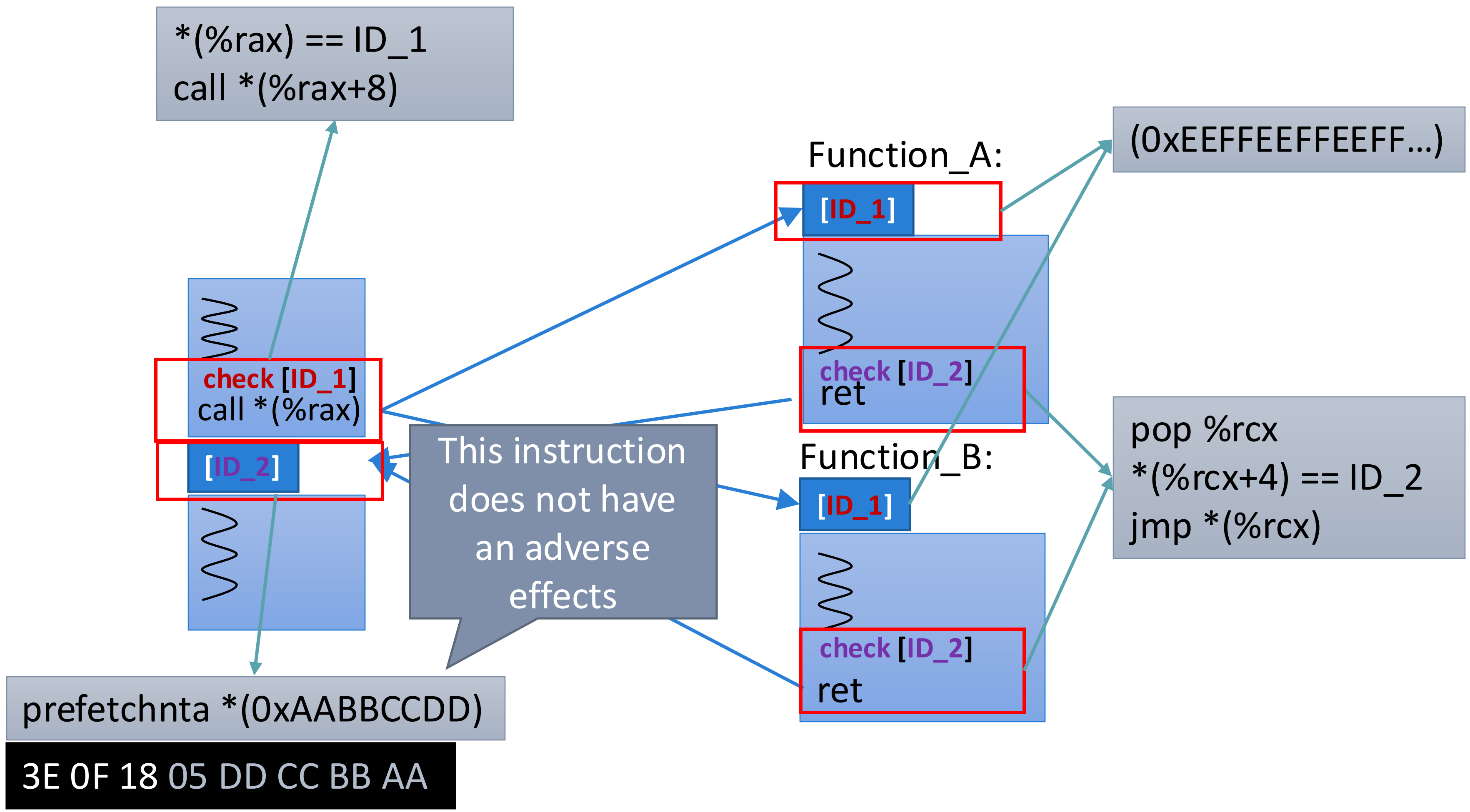
- Checks are introduced right before the control transfer



Modifications for CFI Enforcement



Modifications for CFI Enforcement



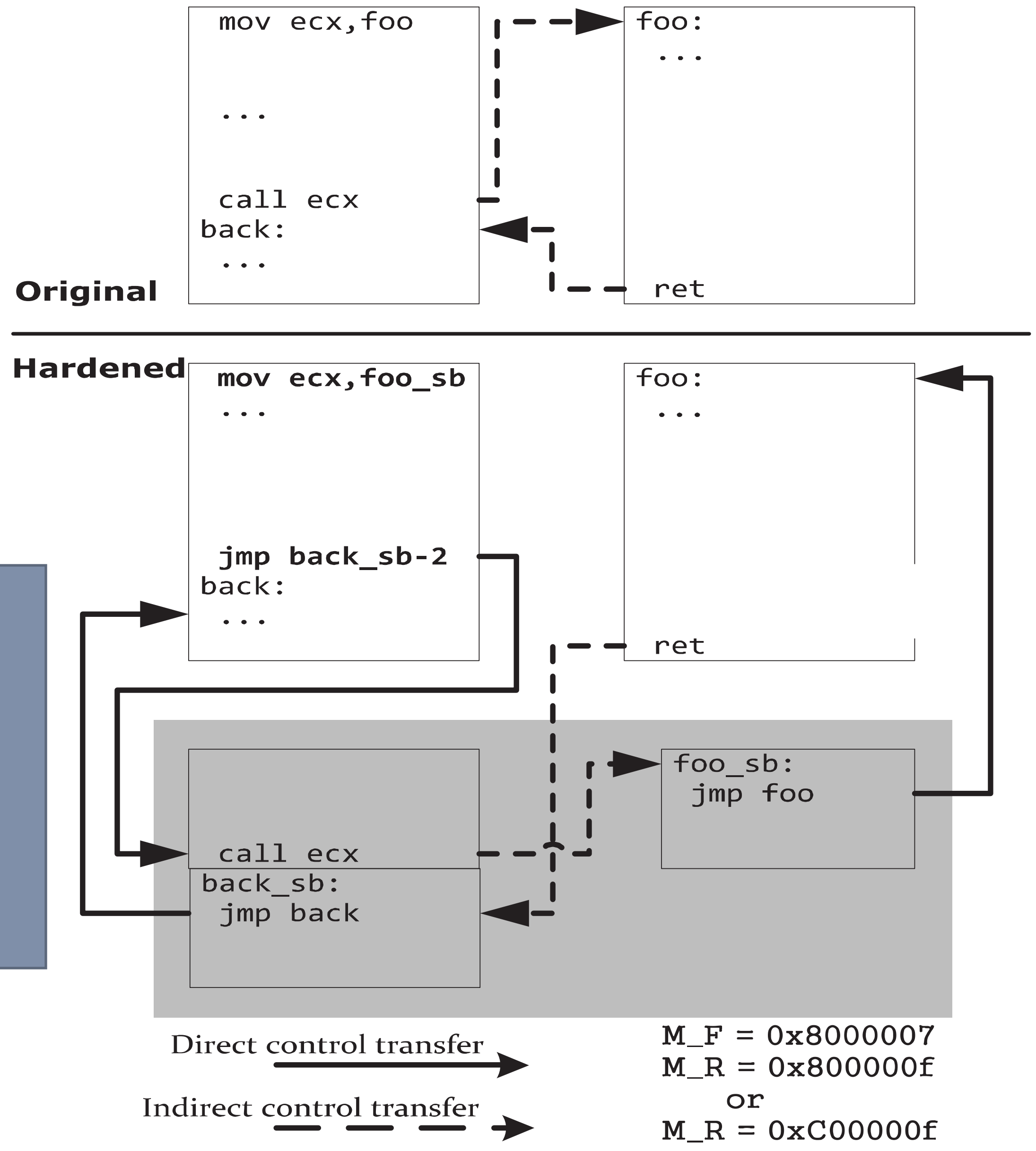
Discussion on Original CFI Proposal

- Efficient approach
  - Low overhead
  - Plays well with caches
- Limited CFG enforcement
  - Because only two IDs are used one each for forward and backward edges
- Can be potentially bypassed by chaining gadgets using still allowable transfers
  - Proposes coupling with another defense mechanism → shadow stacks

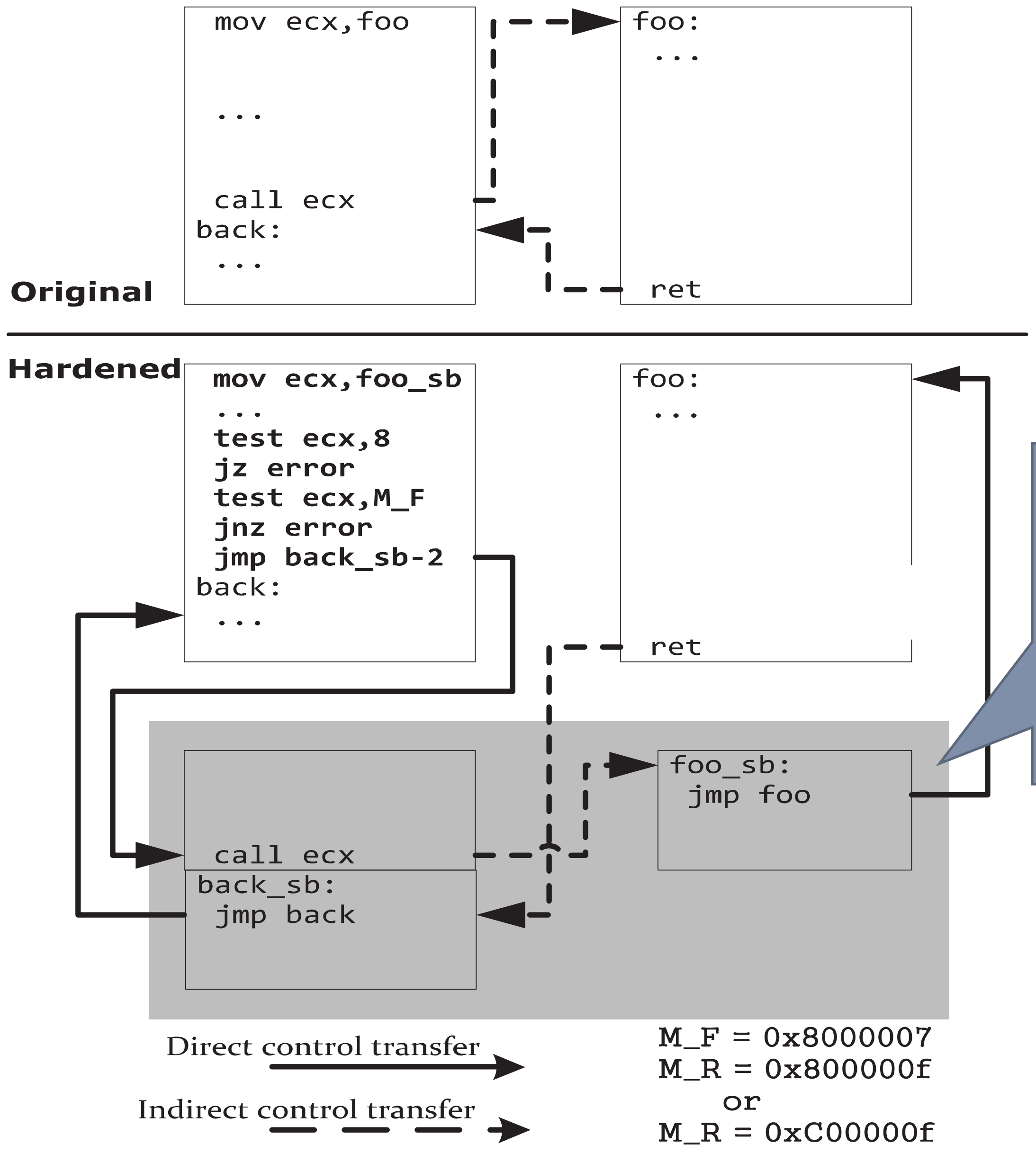
Appendix: CCFIR Implementation

- Practical Control Flow Integrity and Randomization for Binary Executables (2013)
  - http://dl.acm.org/citation.cfm?id=2498134

Each indirect call is redirected through a trampoline using a direct jump



Targeted functions are called indirectly through another trampoline

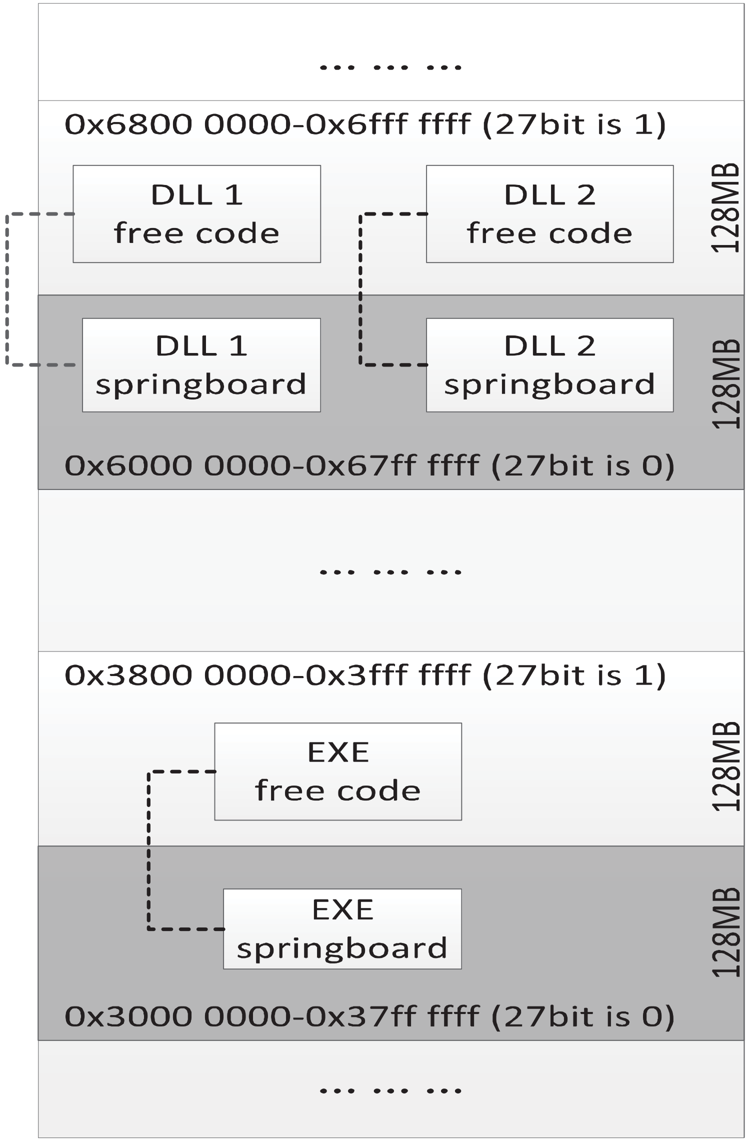


Function stubs are carefully to aligned to easily perform checks

```
Function stub address
        AND
  M_F = 0x8000007
```

$\downarrow$

0

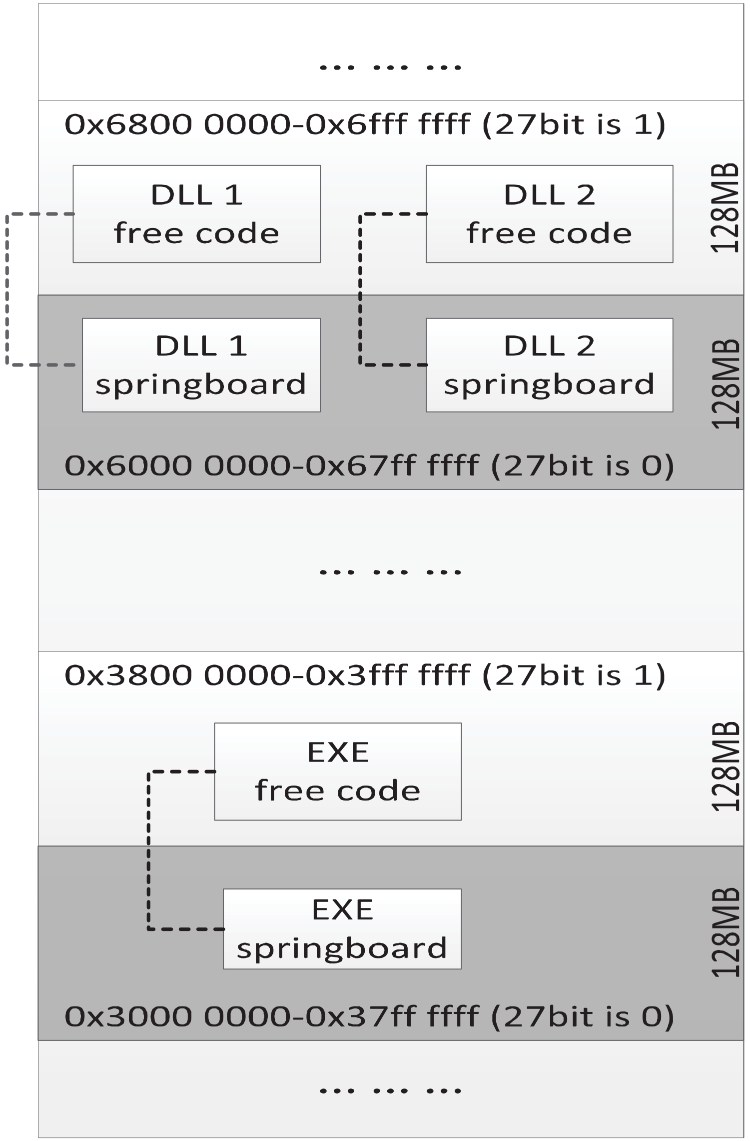


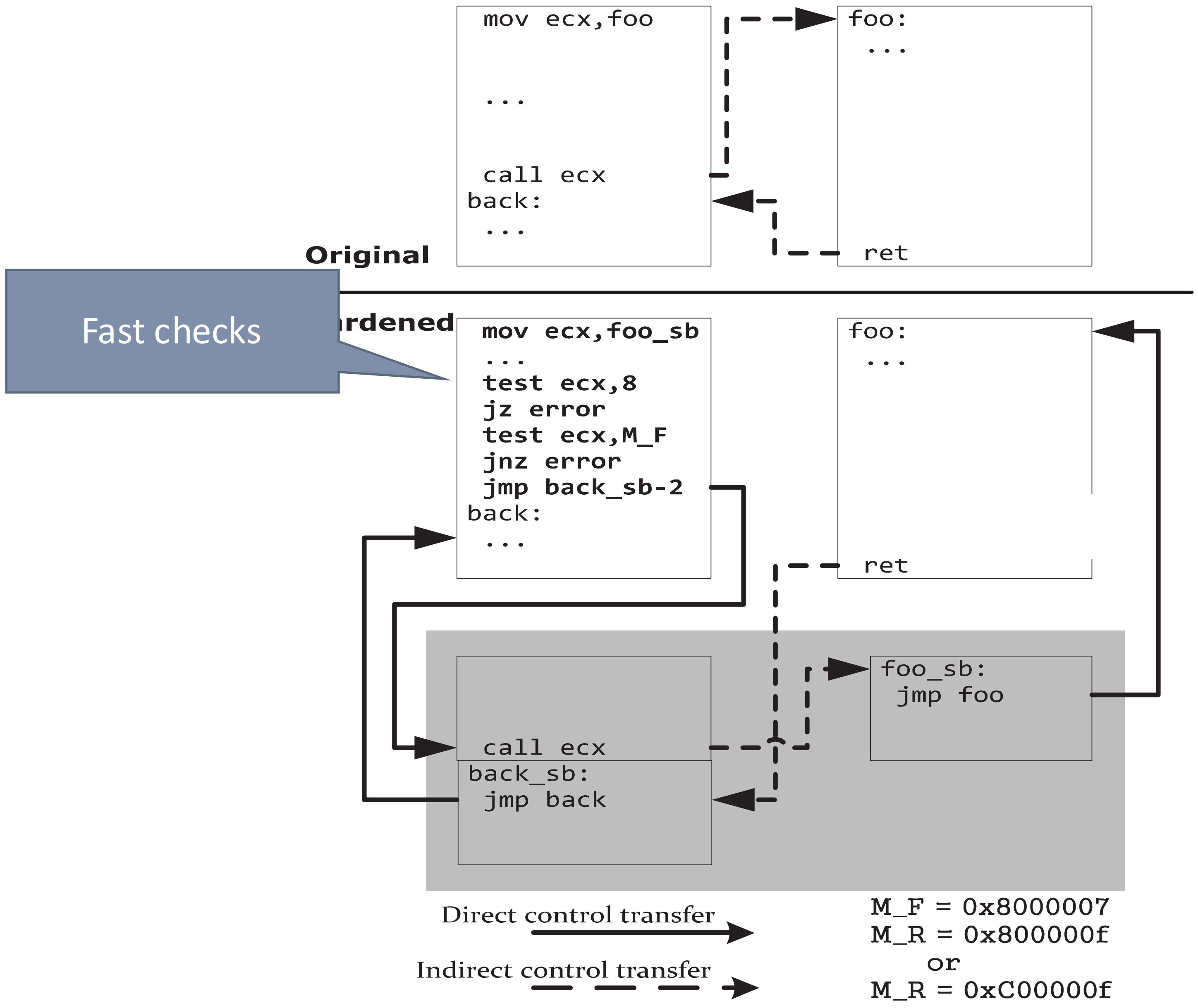
128MB segments

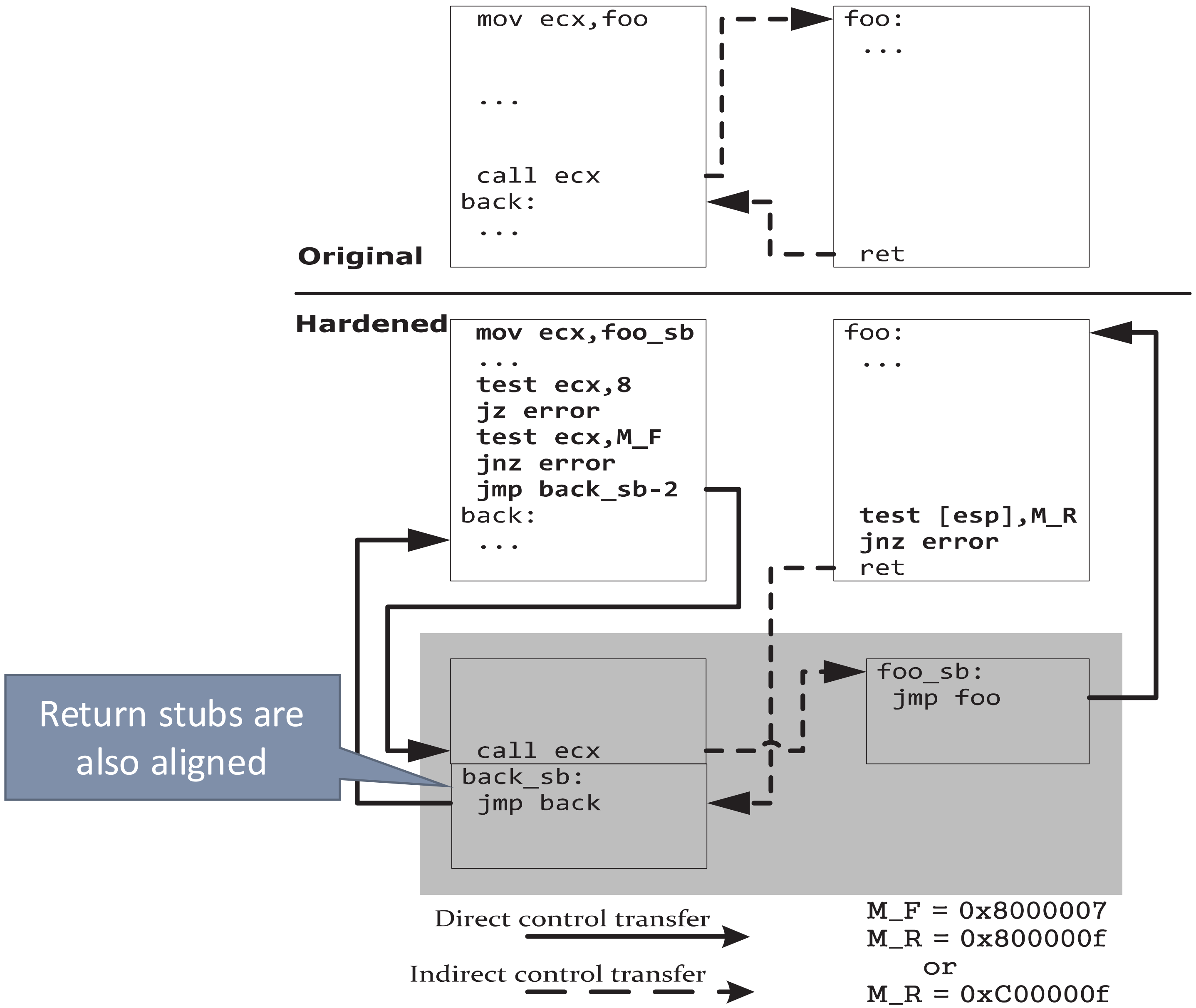
Function stu[ ] address
AND
M_F = 0x8000007

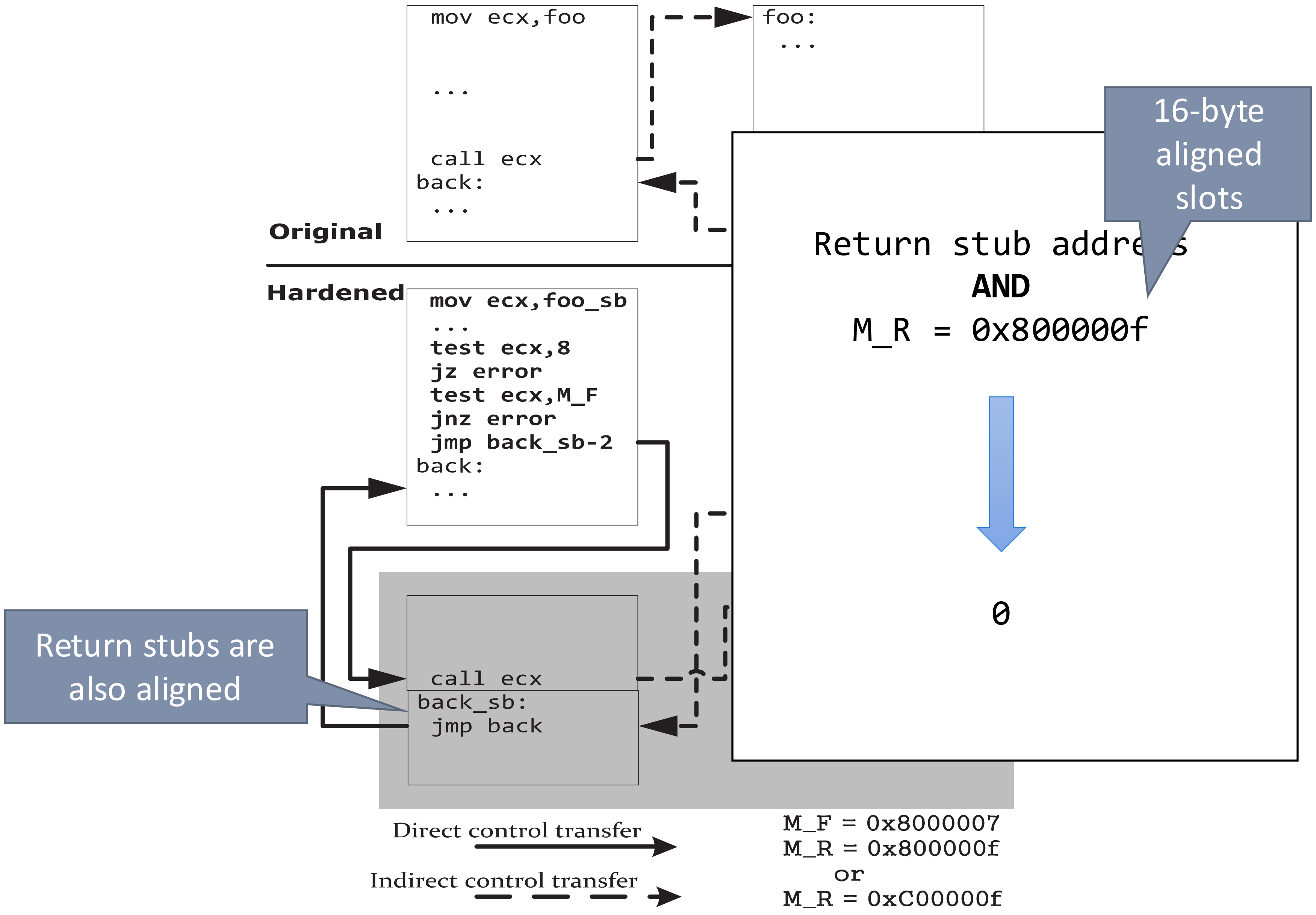
8-byte aligned slots

0

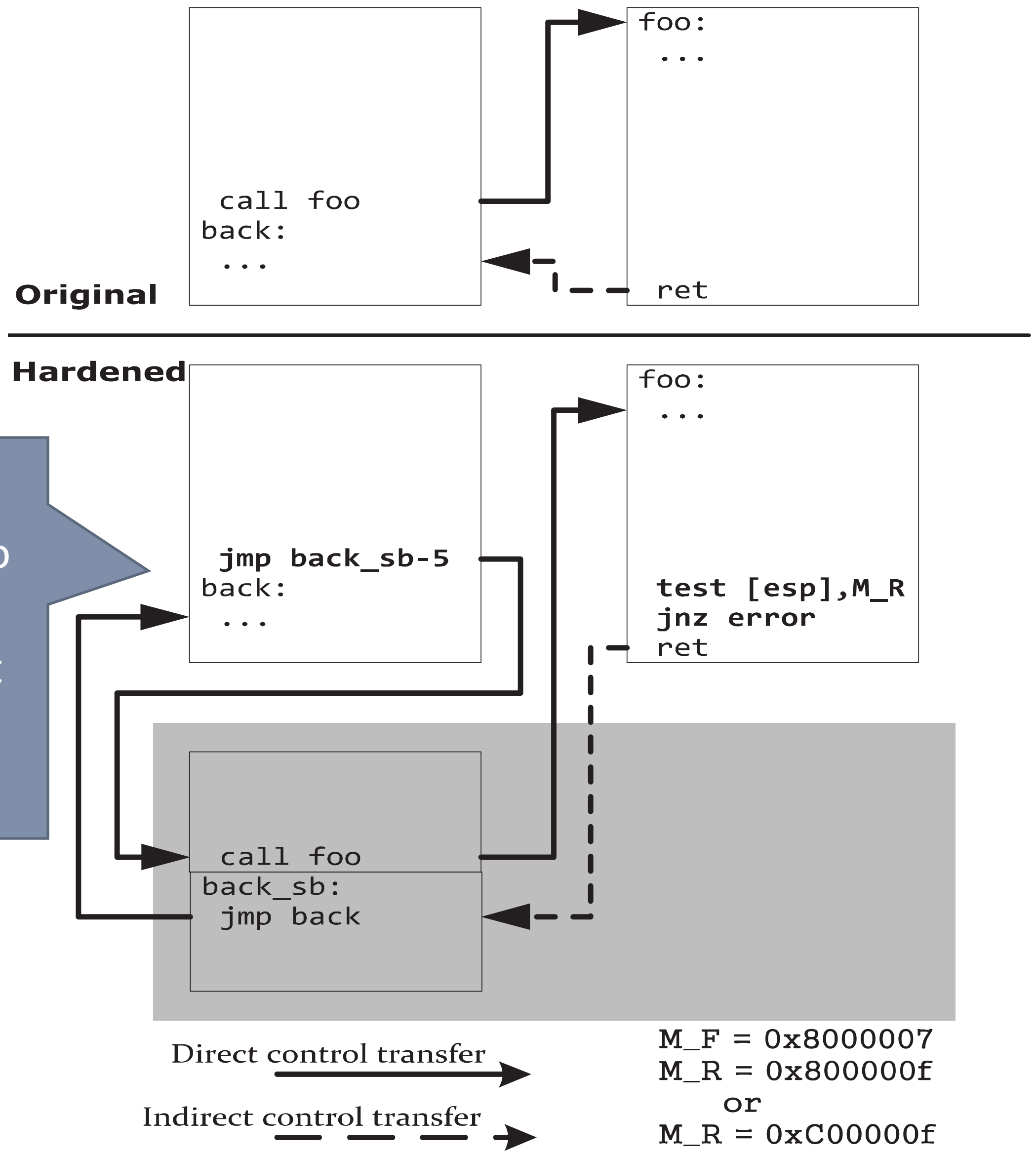


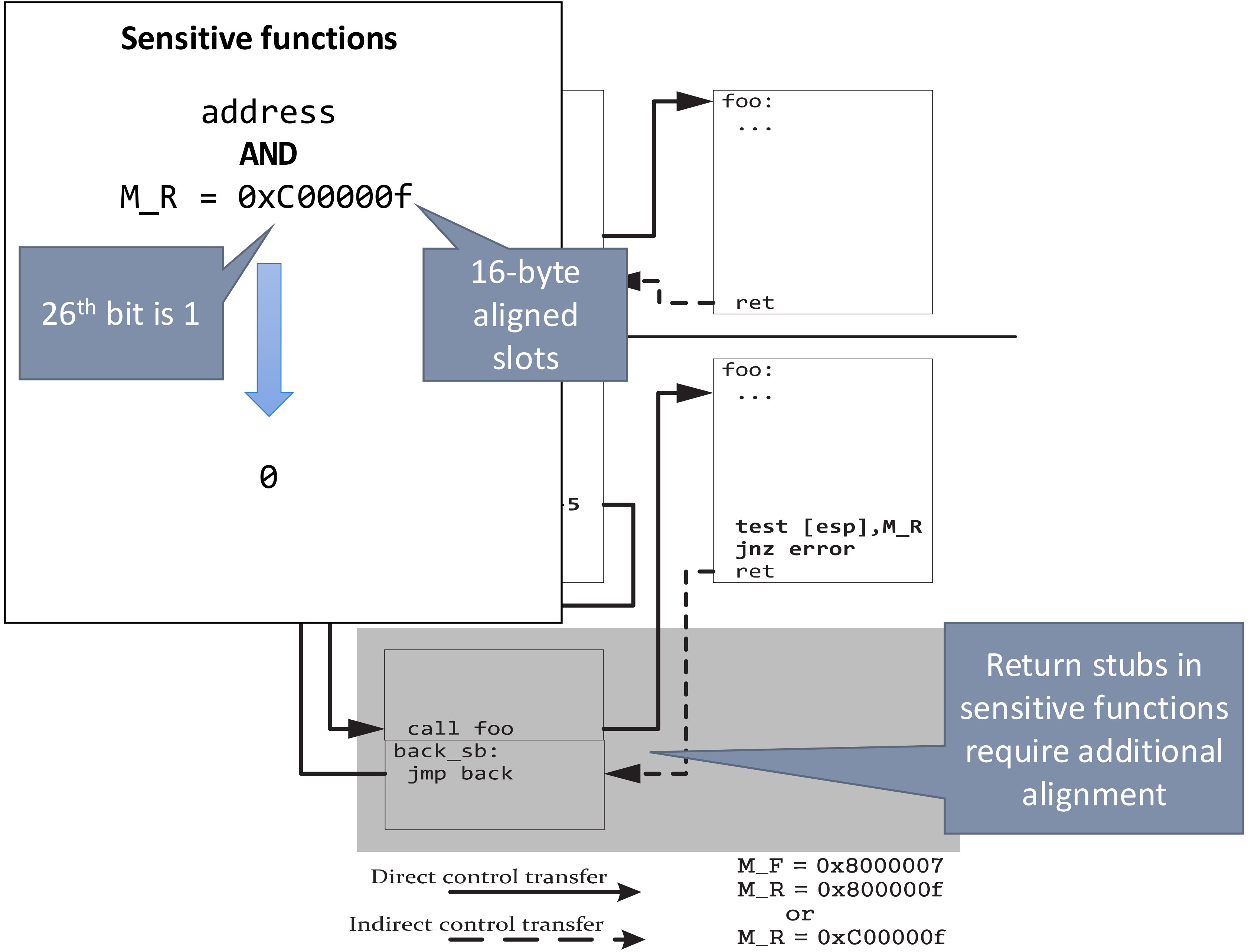

mov ecx,foo
...
call ecx
back:
...
foo:
...
ret
Original
Fast checks
rdened
mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...
foo:
...
ret
call ecx
back_sb:
jmp back
foo_sb:
jmp foo
Direct control transfer
Indirect control transfer
M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f


Original
mov ecx,foo
...
call ecx
back:
...
foo:
...
ret
Hardened
mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...
foo:
...
test [esp],M_R
jnz error
ret
call ecx
back_sb:
jmp back
foo_sb:
jmp foo
Return stubs are also aligned
Direct control transfer
Indirect control transfer
M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f


mov ecx,foo
...
call ecx
back:
...
Original
foo:
...
Hardened
mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...
call ecx
back_sb:
jmp back
Return stubs are also aligned
16-byte aligned slots
Return stub address
AND
M_R = 0x800000f
0
Direct control transfer
Indirect control transfer
M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f

Direct calls to functions also go through trampolines but no checks required



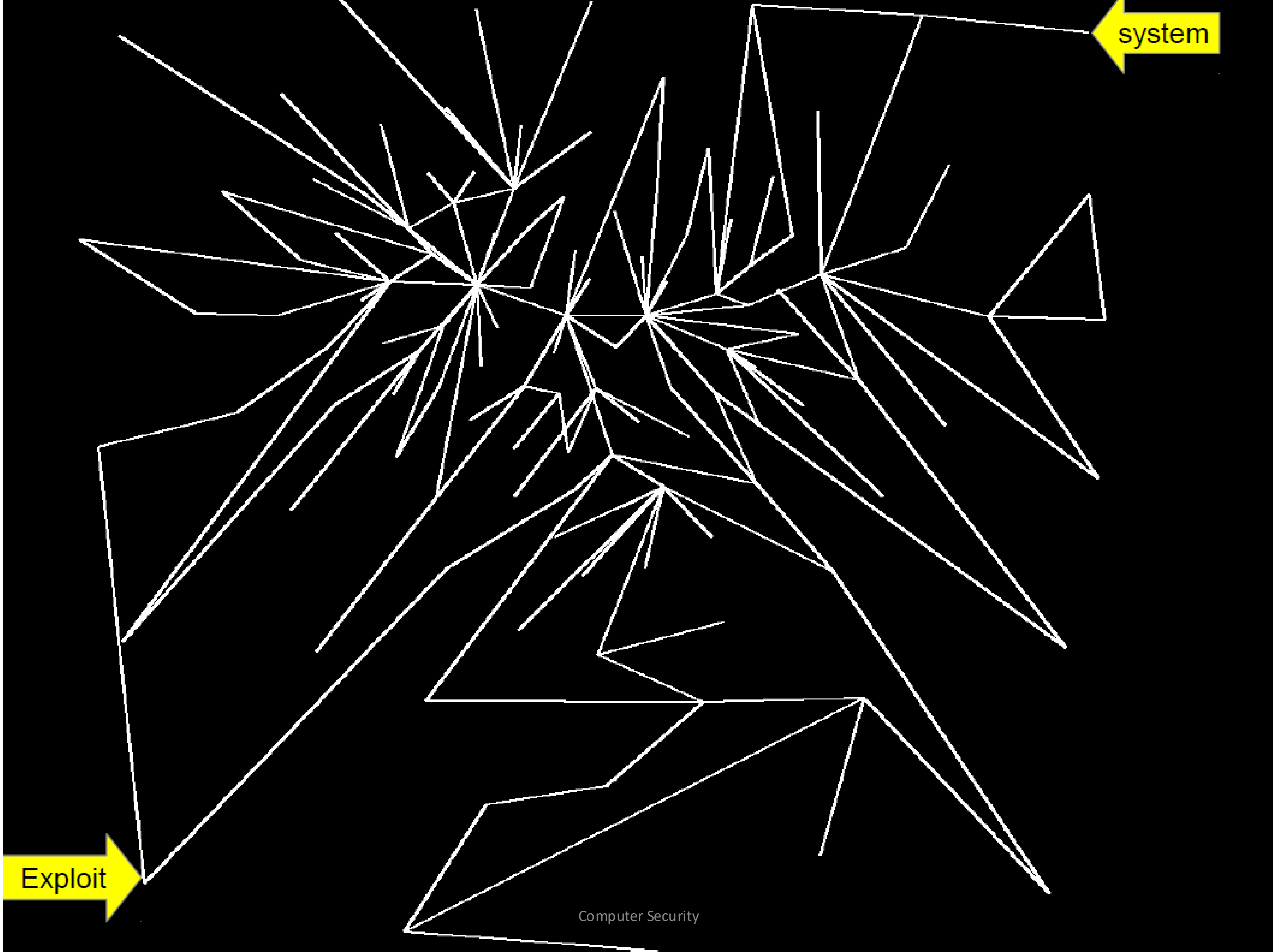


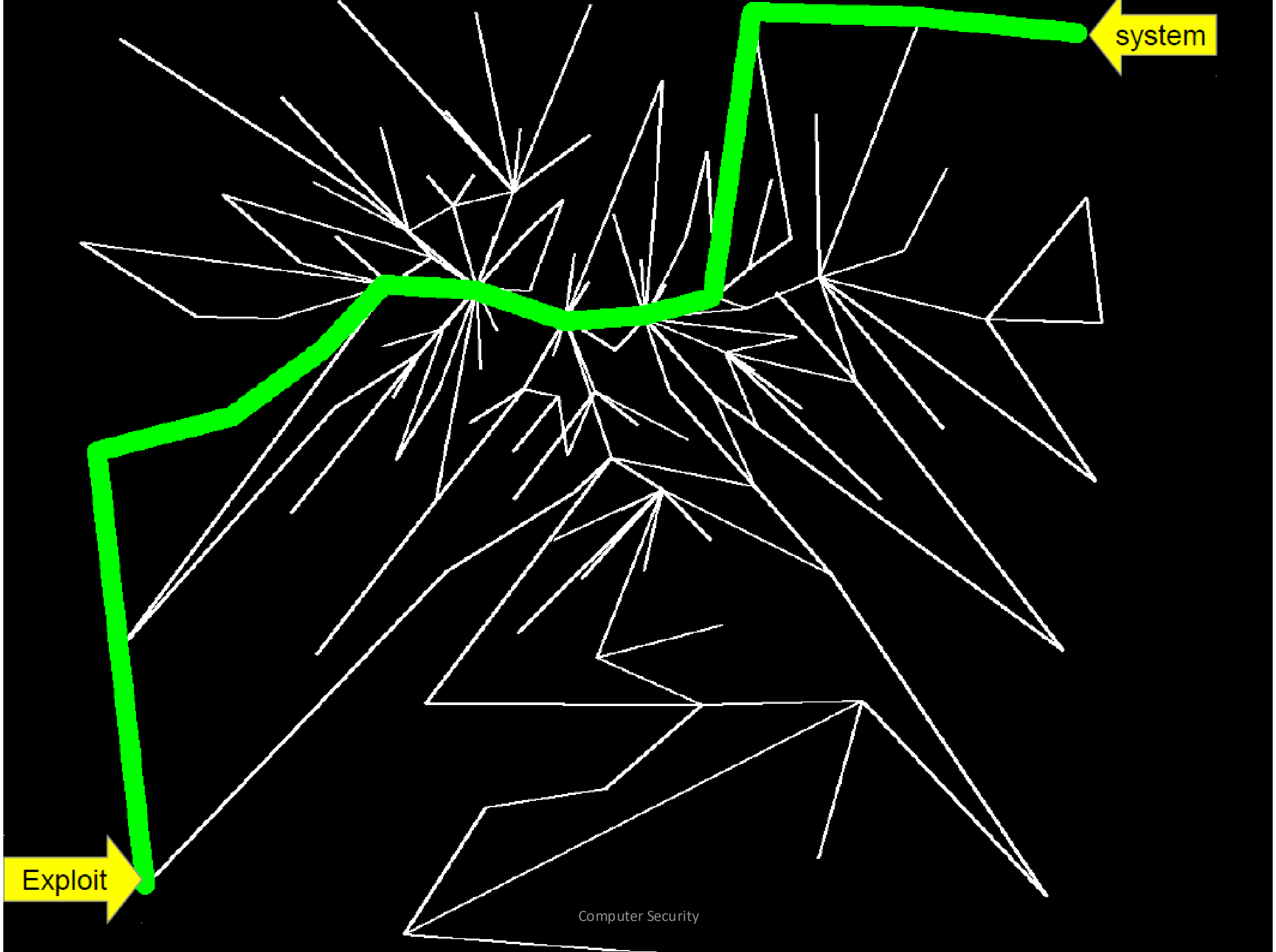
Sensitive functions
address
AND
M_R = 0xC00000f
26th bit is 1
0
16-byte aligned slots
foo:
...
ret
foo:
...
test [esp],M_R
jnz error
ret
5
call foo
back_sb:
jmp back
Return stubs in sensitive functions require additional alignment
Direct control transfer
Indirect control transfer
M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f

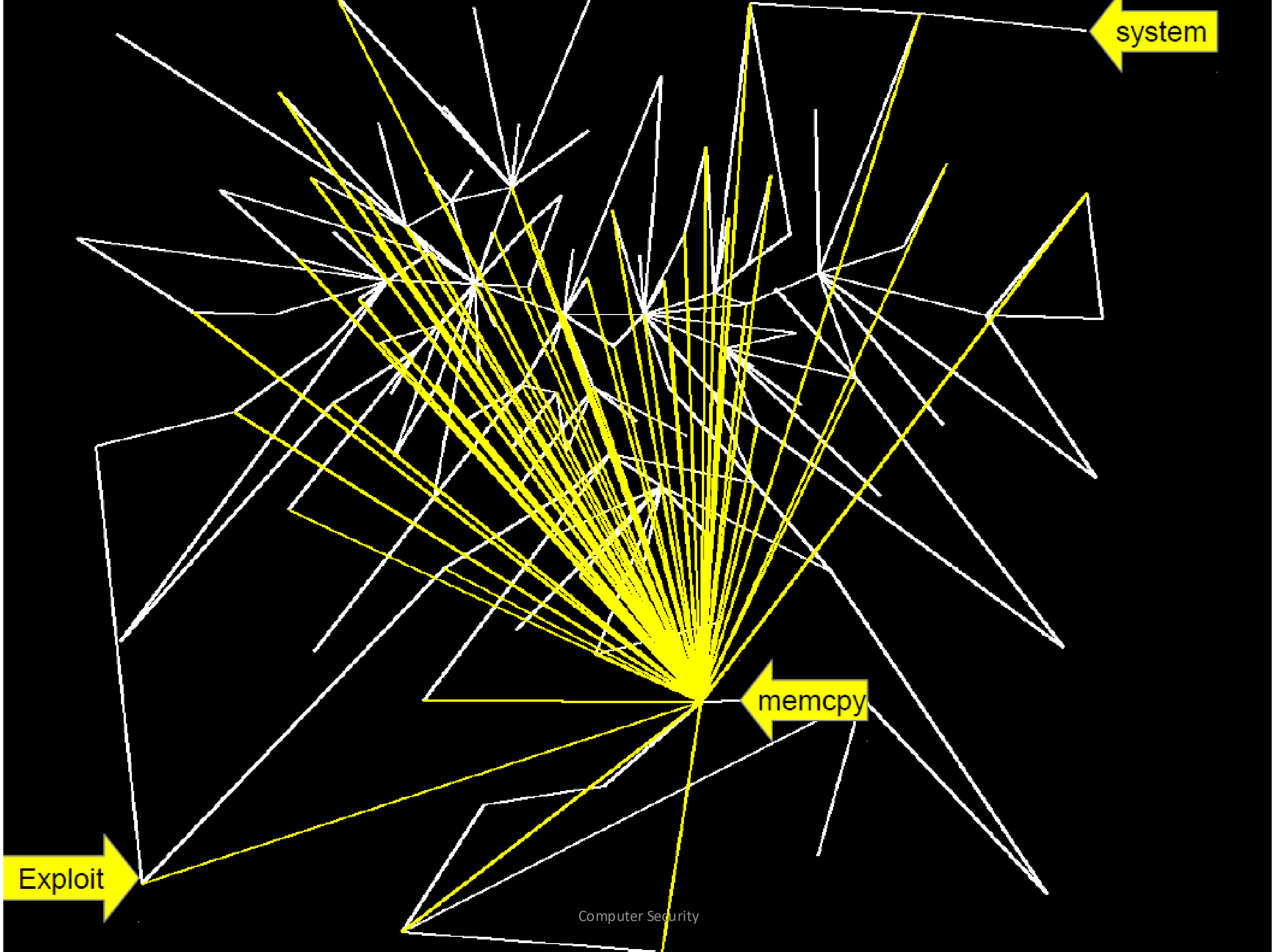
Appendix: What if We Had the Perfect CFG

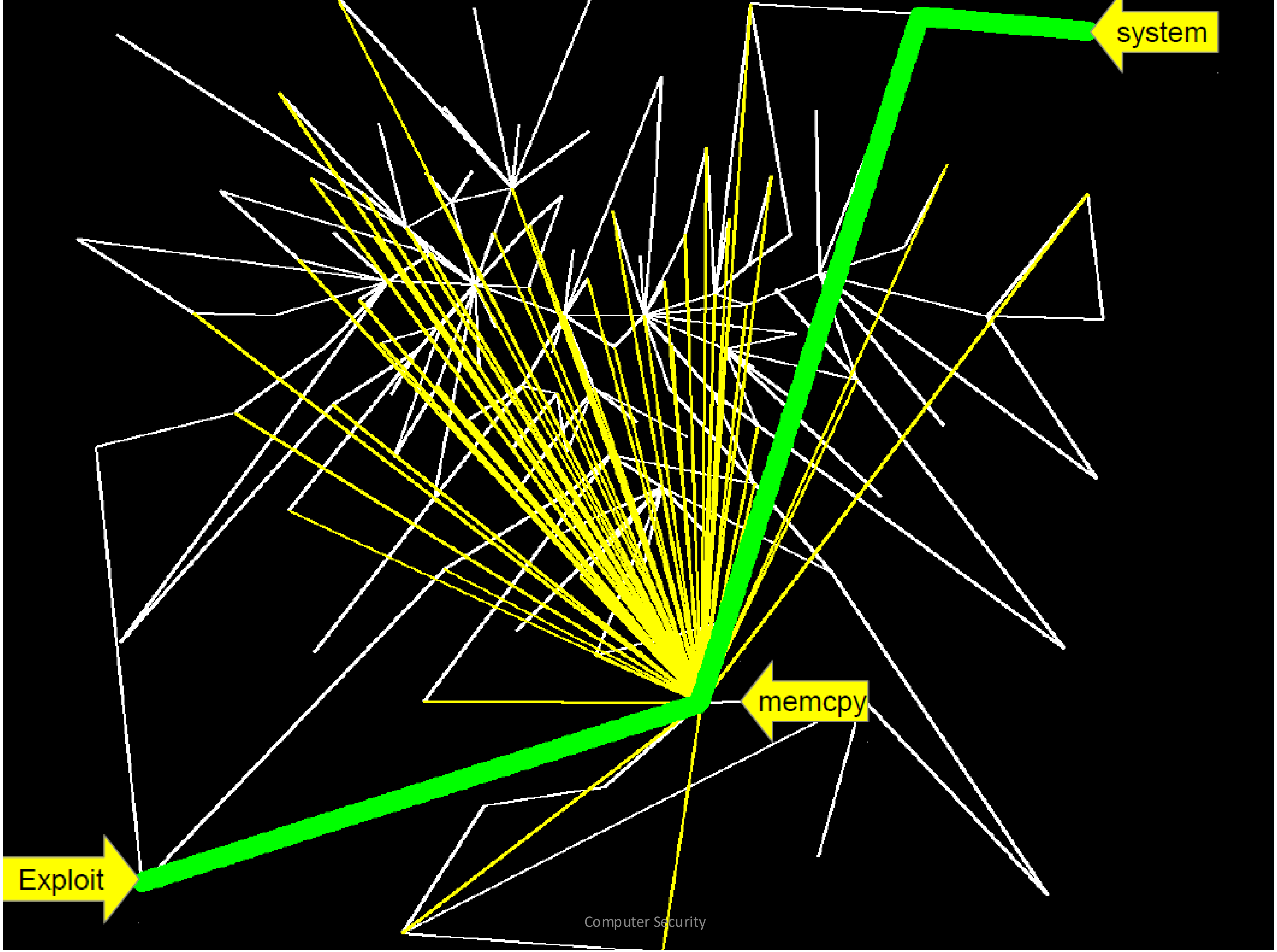
- We know exactly which functions are called from an indirect call
- We know exactly the call sites where a function's return is supposed to return
- But we still do not have a shadow stack

- **Control Flow Bending**
- https://www.usenix.org/sites/default/files/conference/protected-files/sec15_slides_carlini.pdf

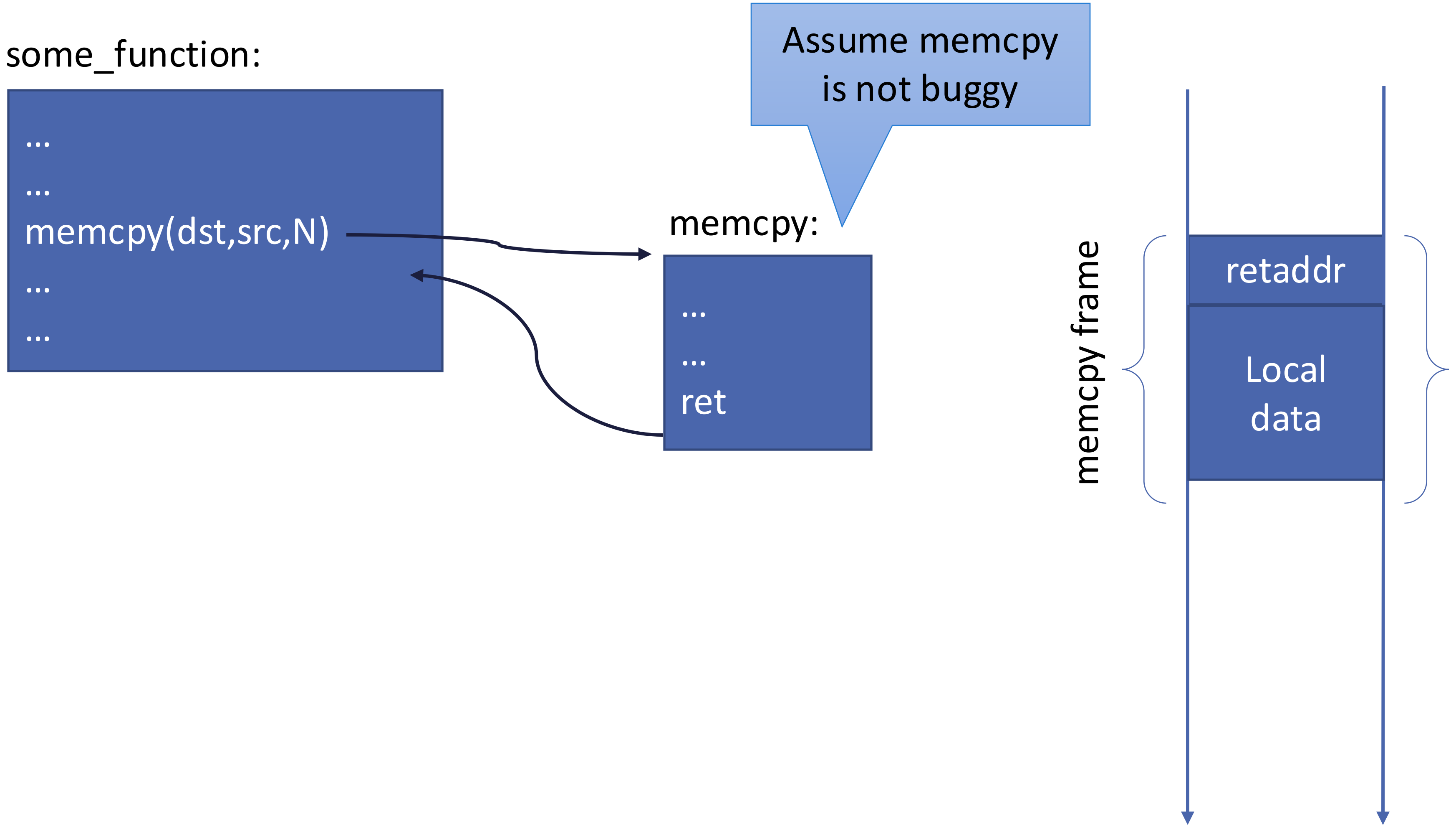

system
Exploit


system
Exploit

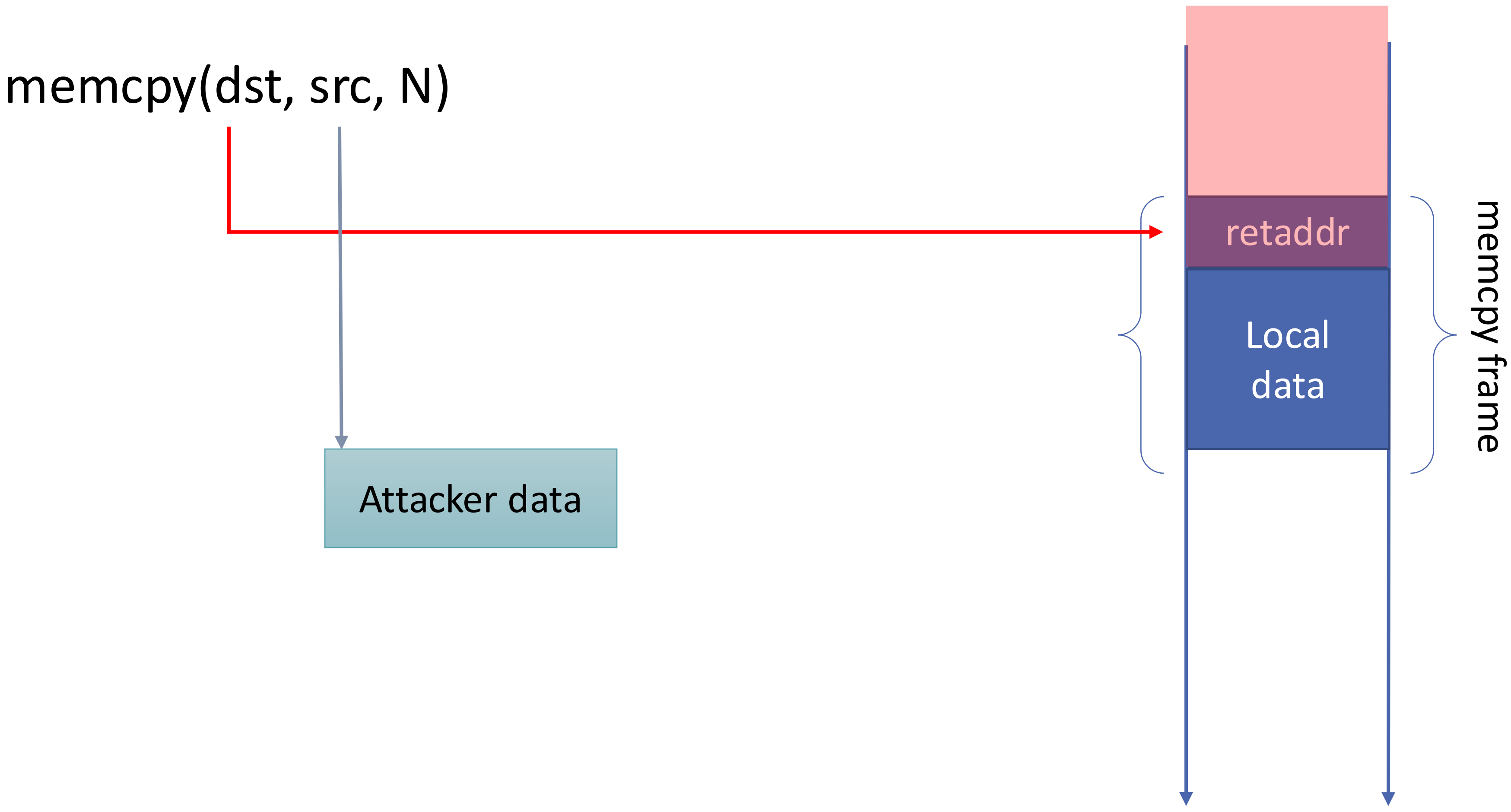

system
memcpy
Exploit


system
memcpy
Exploit

How to Exploit the memcpy() Hotspot



How to Exploit the memcpy() Hotspot



Dispatcher Function

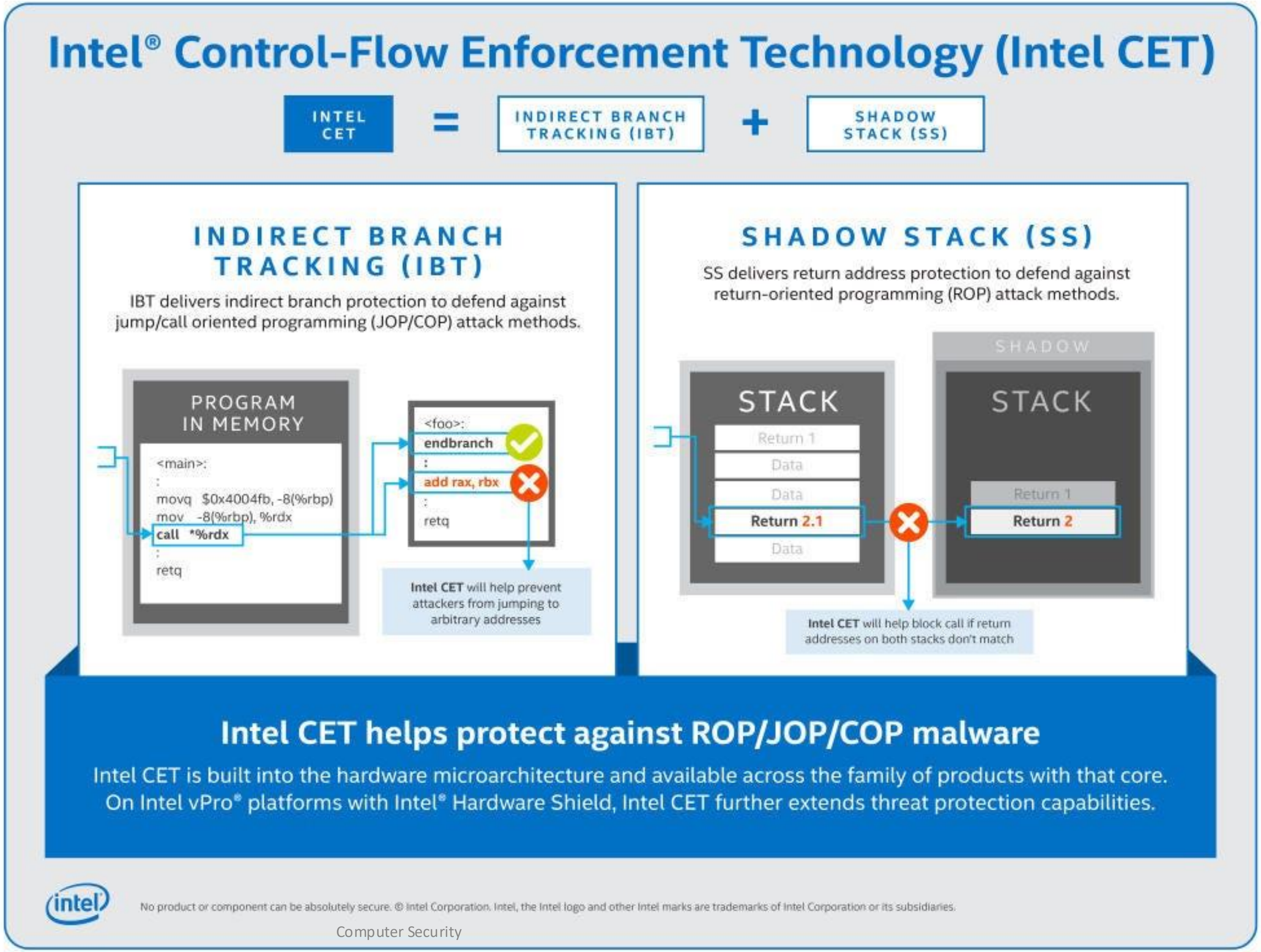
- memcpy() acts as a dispatcher function
  - Can be used to return to gadgets part of the CFG

- Other hot functions can act as dispatcher functions, as long as:
  - They are commonly called
  - Their arguments are under attacker control
  - Can overwrite their own return address

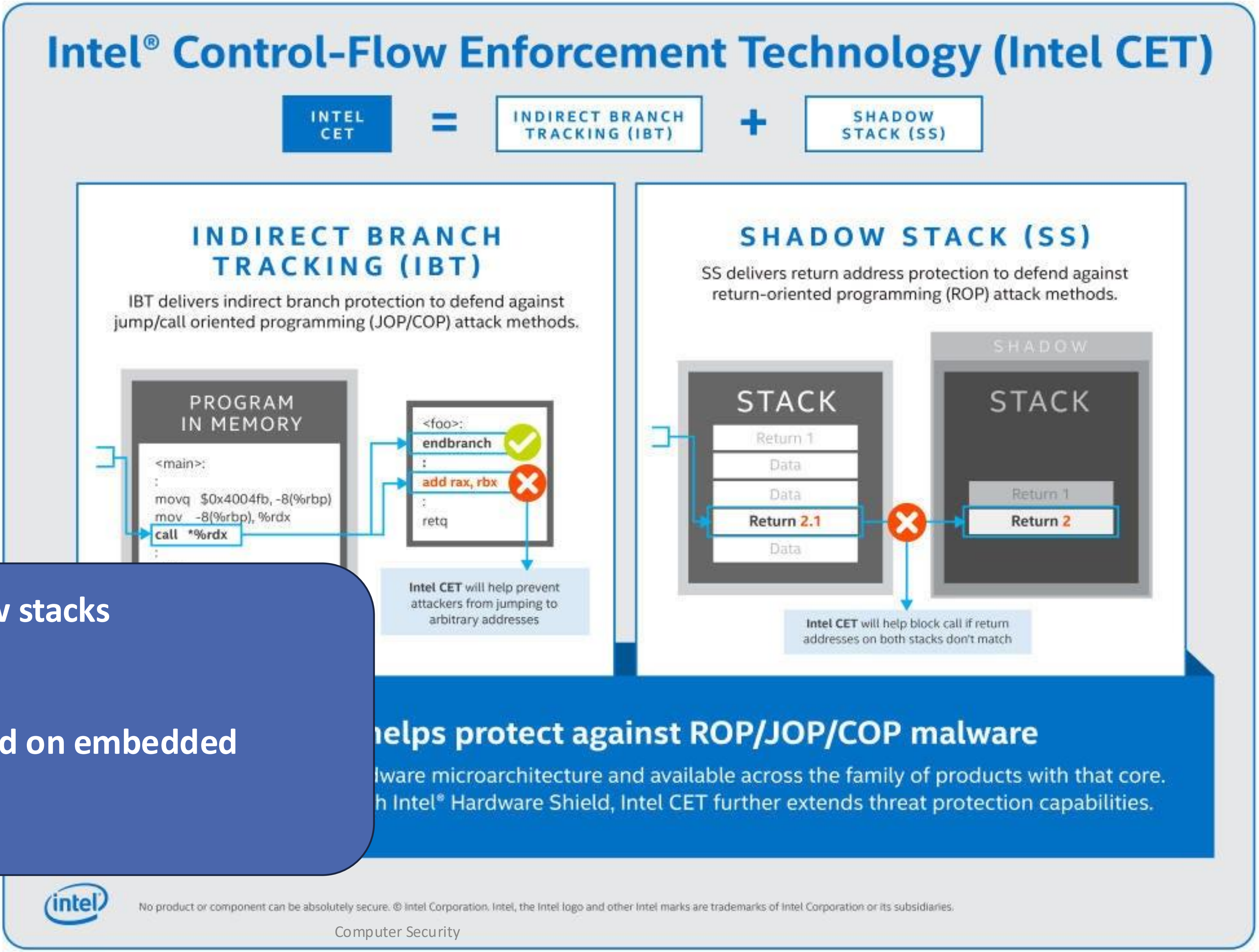
Summary

- CFI is a powerful security primitive
- Depends on the quality/accuracy of the CFG
- Even in the ideal case, it might fall to code-reuse attacks
  - Depends on the application
    - Complexity of the CFG
    - Availability of gadgets
- Securing the backward edge is crucial
  - Precision is required (for example, like with a shadow stack)

HW Support:
Intel Control-flow Enforcement Technology (CET)



HW Support: Intel Control-flow Enforcement Technology (CET)



- **Same issues with shadow stacks**
  - **Legacy code**
  - **Call-Ret mismatches**
- **Not (wont be?) supported on embedded systems**

Pointer Authentication Code

- HW support for enforcing pointer integrity on some ARM processors
- Example: PAC it up: Towards Pointer Integrity using ARM Pointer Authentication
  - https://www.usenix.org/system/files/sec19fall_liljestrand_prepub.pdf