Modern Exploitation and Defenses

CS-576 Systems Security

Instructor: Georgios Portokalidis

Spring 2018

Topics

Recap: Security mechanisms for software hardening

Attacks against client programs

- Browsers
- Heap spraying
- Mitigations

Back to return-to-libc

Return-oriented programming

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

Broadly Deployed Security Mechanisms

NX-bit → Prevent arbitrary code execution

Stack canaries → Detect and prevent stack overflows

ASLR → Introduce uncertainty on the location of injected shellcode and existing code in a running program

They have raised the bar for attackers

Shift in Target Selection

Clients | **Servers**

Shift in Target Selection

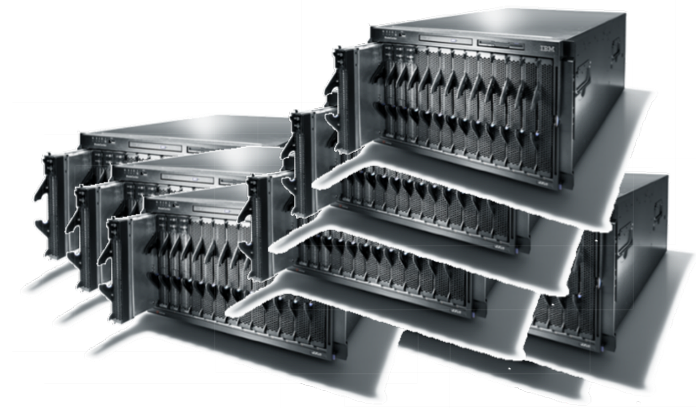
Clients

Web browsers

Flash

Acrobat Reader

Servers



Recap: Attacks Against Browsers

Very popular software

- Probably installed on every client device

Large and complex software

Dynamically translates and executes JavaScript

JavaScript Code

↓

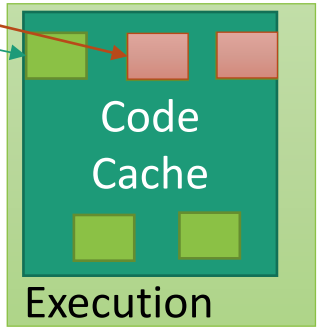
↓

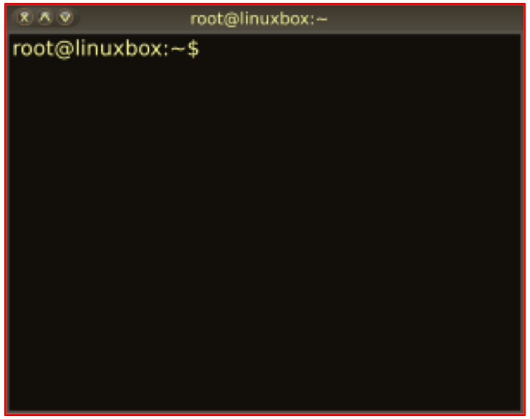
Native Code

Recap: Code Injection in the Code Cache

```
<html>
<body>
<script language='javascript'>

var myvar = unescape('%u\4F43%u\4552'); //
CORE
myvar += unescape('%u\414C%u\214E'); //
LAN!
alert("allocation done");

</script>
</body>
</html>
```

No ASLR → Code cache location known



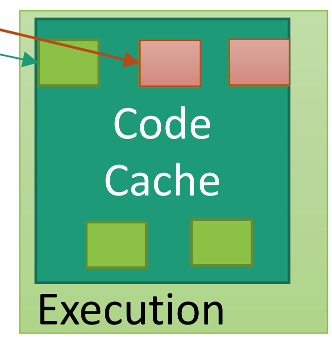


Exploit bug to control instruction pointer!

Recap: Code Injection in the Code Cache

ASLR → Code cache location **unknown**

```
<html>
<body>
<script language='javascript'>

var myvar = unescape('%u\4F43%u\4552'); //
CORE
myvar += unescape('%u\414C%u\214E'); //
LAN!
alert("allocation done");

</script>
</body>
</html>
```



Heap Spraying

Attempt to place shellcode at a predictable location

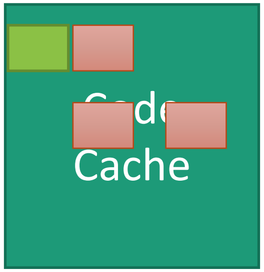
Mechanisms:

Dynamically expand buffer by appending copies of the shellcode

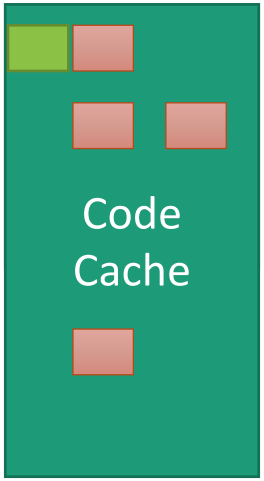
On the fly generate variables

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

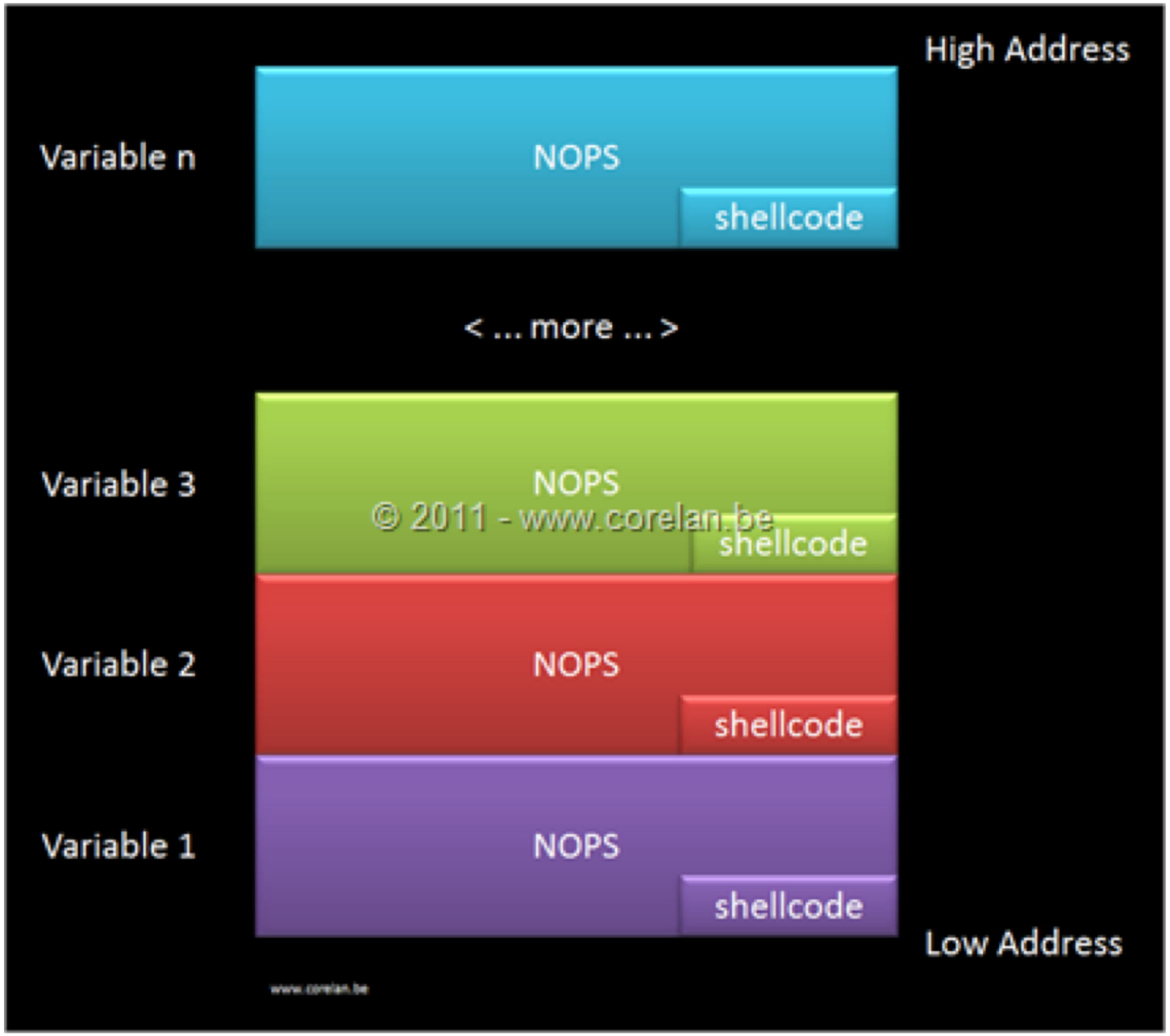
```
var v1 = “myshellcode”;
var v2 = “myshellcode”;
var v3 = “myshellcode”;
```

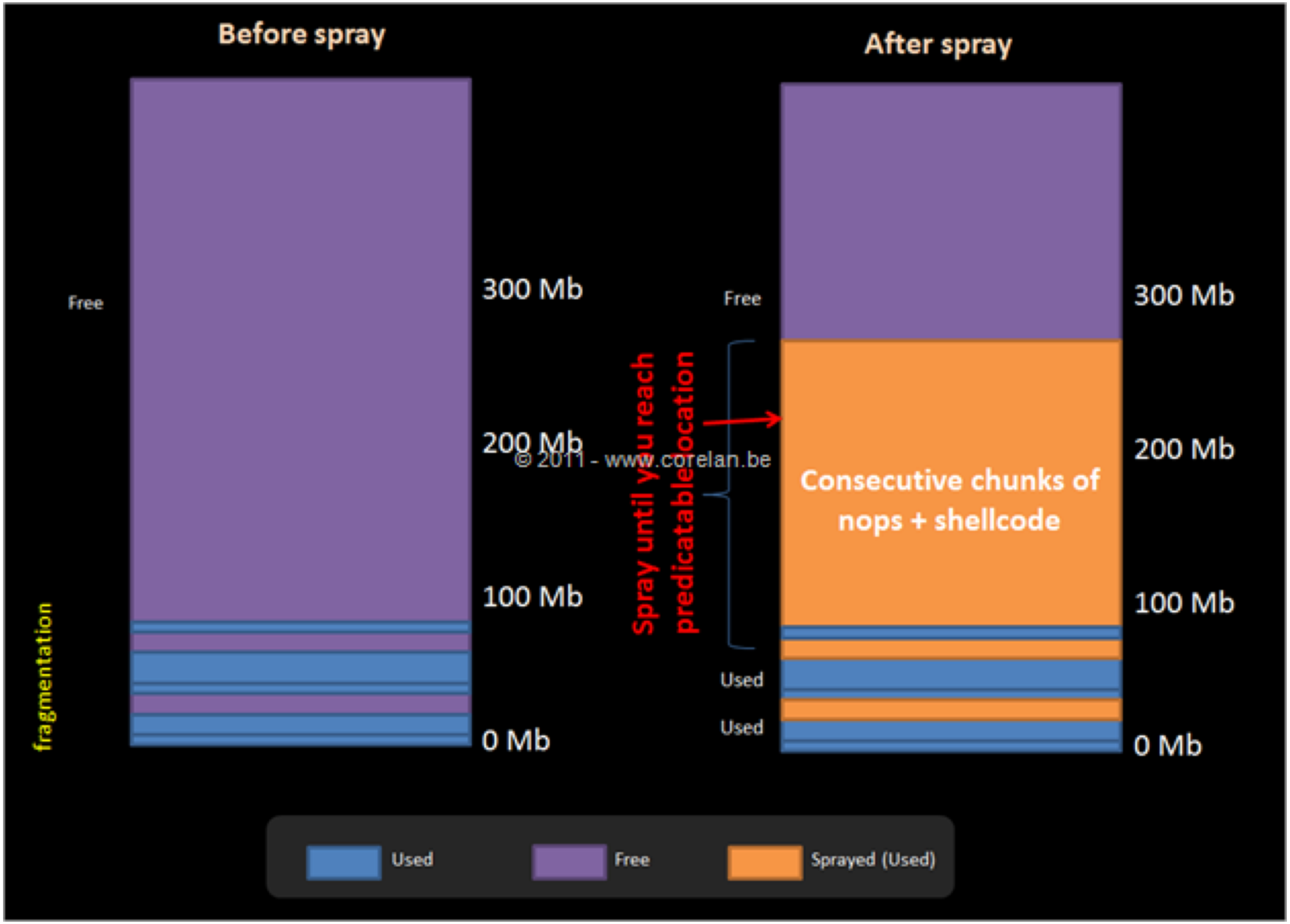


```
var v1 = "myshellcode";
var v2 = "myshellcode";
var v3 = "myshellcode";
var v4 = "myshellcode";
```



Large NOP Sleds




Before spray
After spray
Free
300 Mb
200 Mb
100 Mb
0 Mb
fragmentation
© 2011 - www.corelan.be
Spray until you reach predicatable location
Free
Consecutive chunks of nops + shellcode
Used
Used
300 Mb
200 Mb
100 Mb
0 Mb
Used
Free
Sprayed (Used)

Summary: Heap Spraying

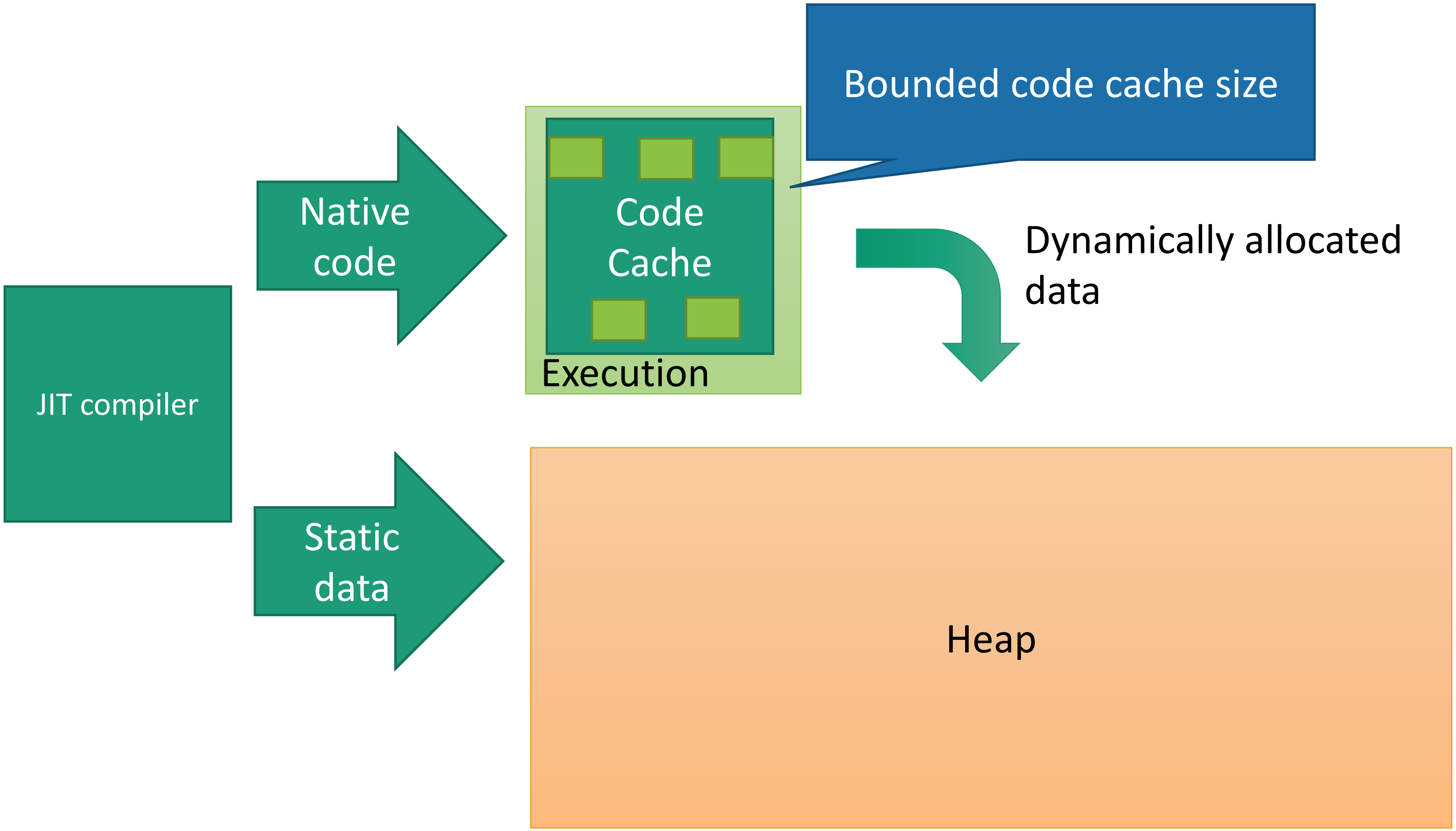
May require multiple attempts

Can possibly defeat ASLR

Heap fragmentation is in play

- May be worse in concurrent systems

Code/Data Separation in the Code Cache



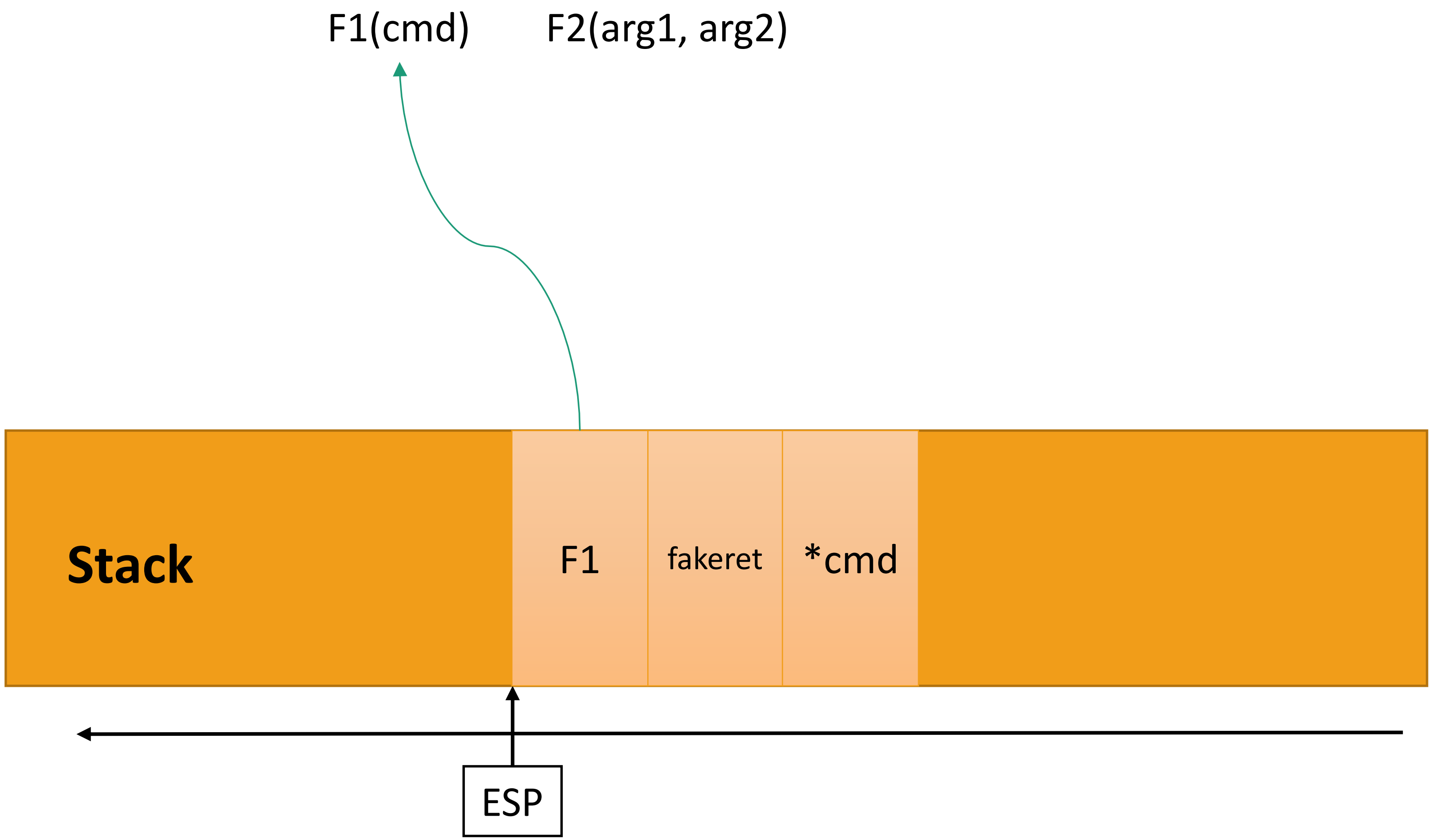
ASLR + Code/data Separation + Finite Code Cache

No More Code Injection

Back to return-to-libc

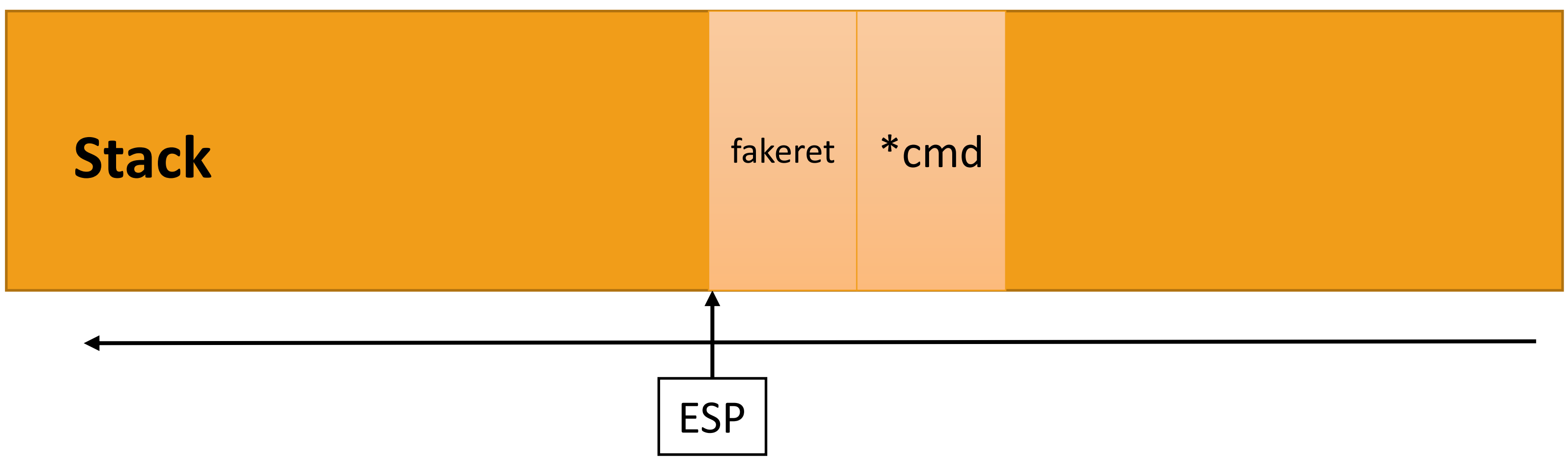

I don’t like calling system() every day

Chaining Functions with ret2libc

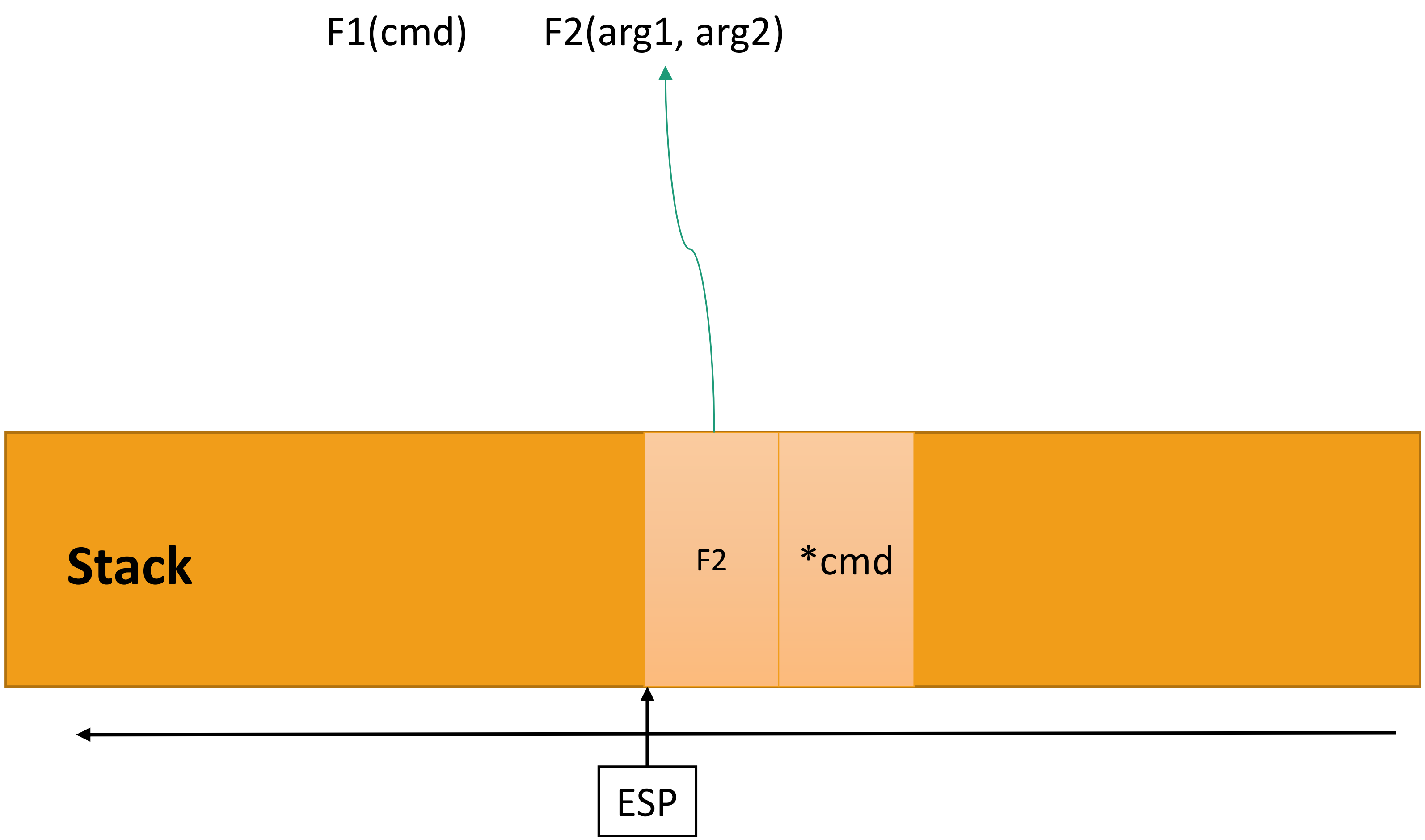


Chaining Functions with ret2libc

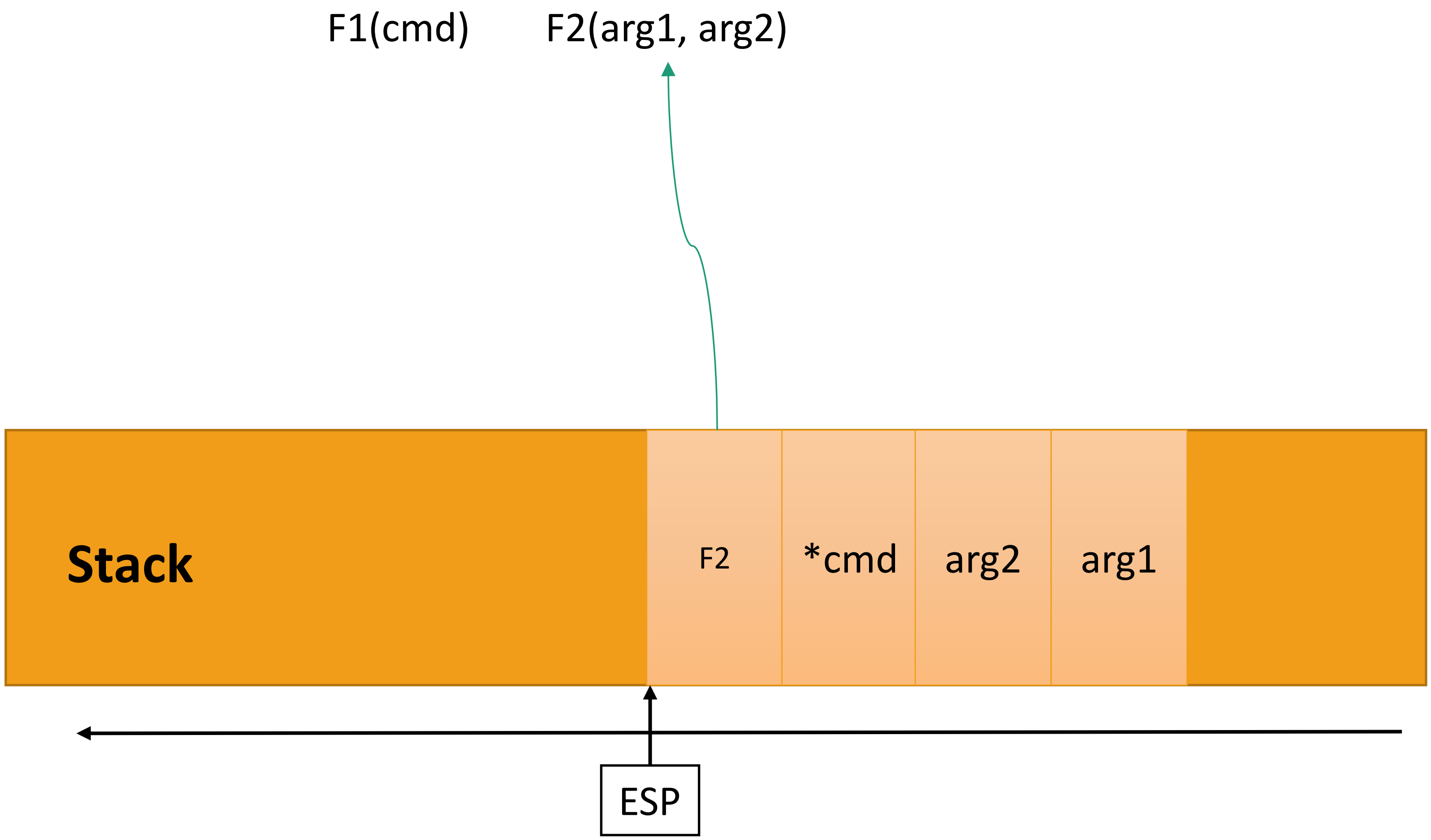
F1(cmd)   F2(arg1, arg2)



Chaining Functions with ret2libc

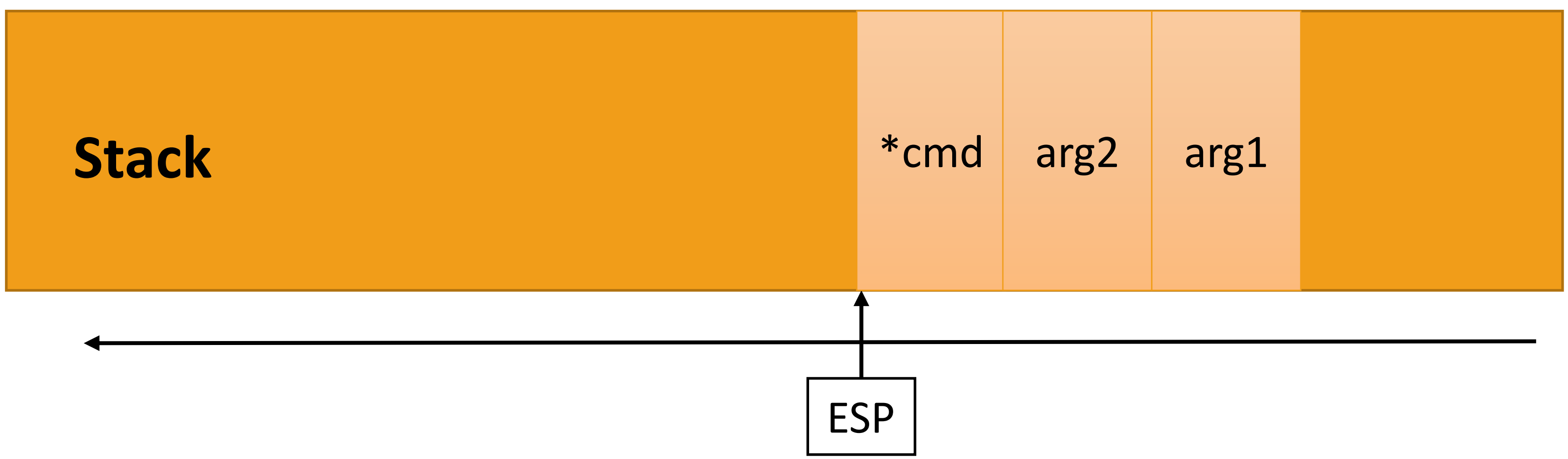


Chaining Functions with ret2libc



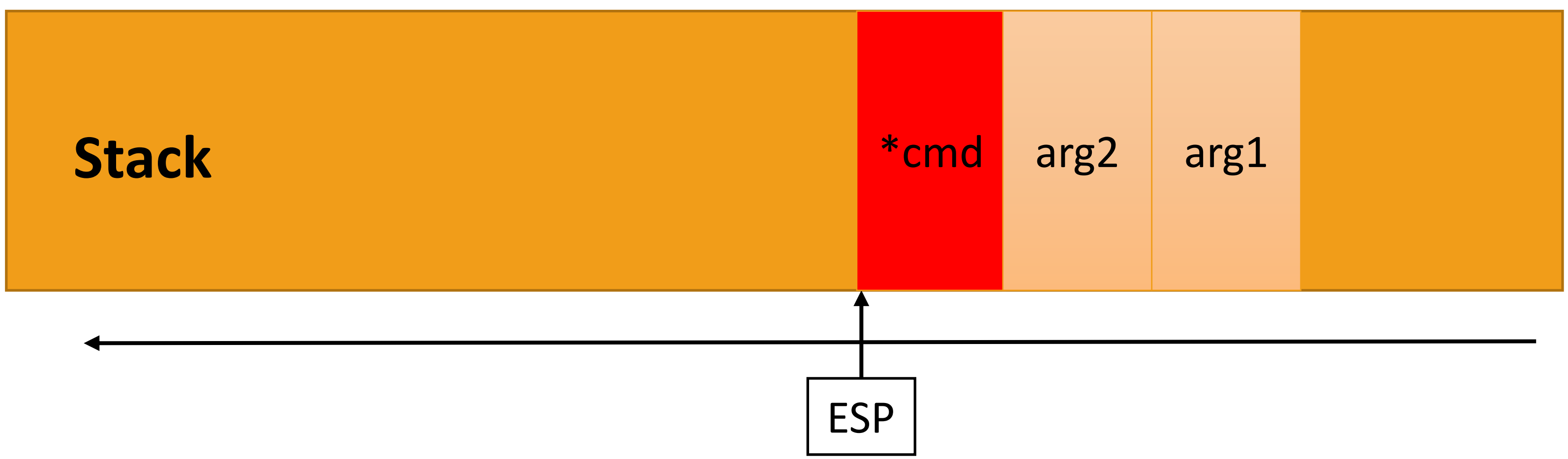
Chaining Functions with ret2libc

F1(cmd) F2(arg1, arg2) F3(arg3)



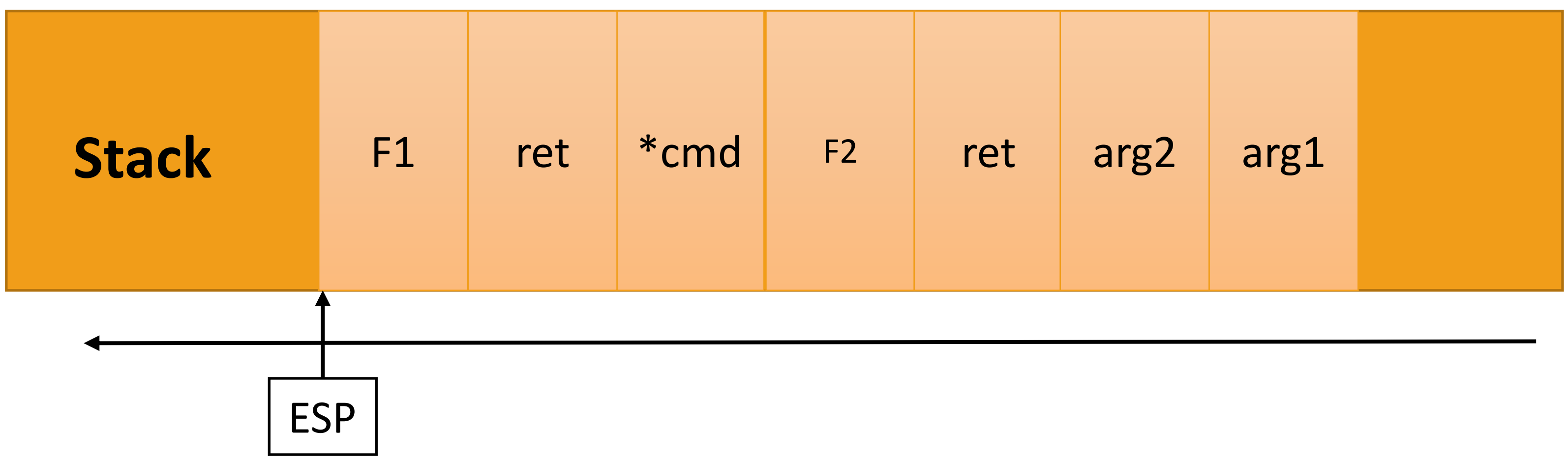
Chaining Functions with ret2libc

F1(cmd) F2(arg1, arg2) F3(arg3)



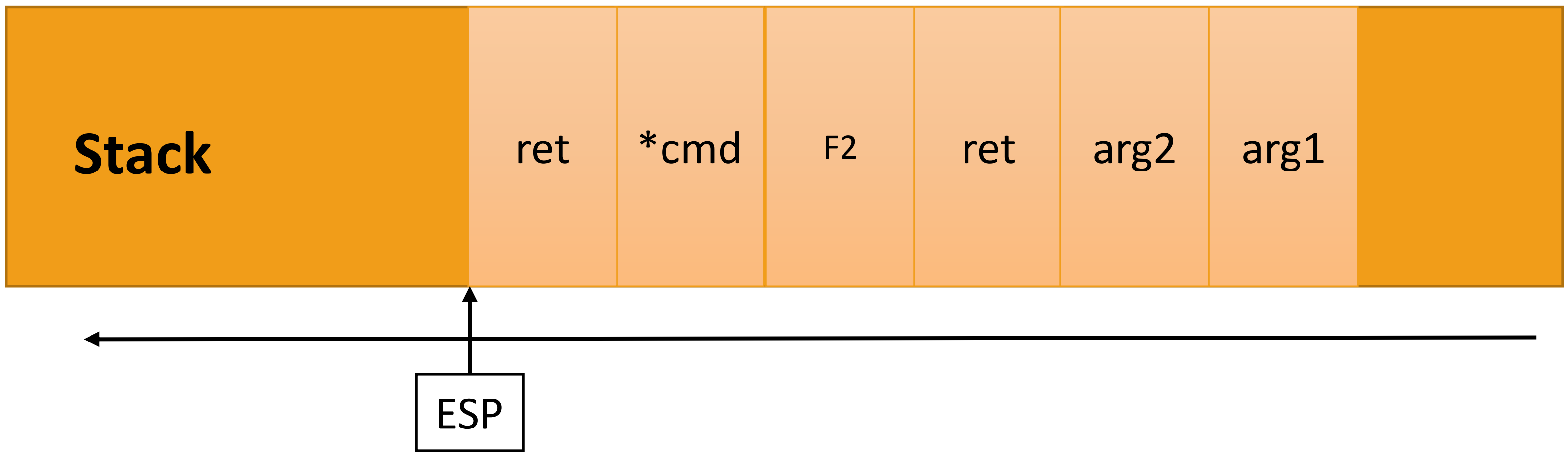
Chaining Functions with ret2libc

We need small gadgets to unwind the stack pointer in a controlled way



Chaining Functions with ret2libc

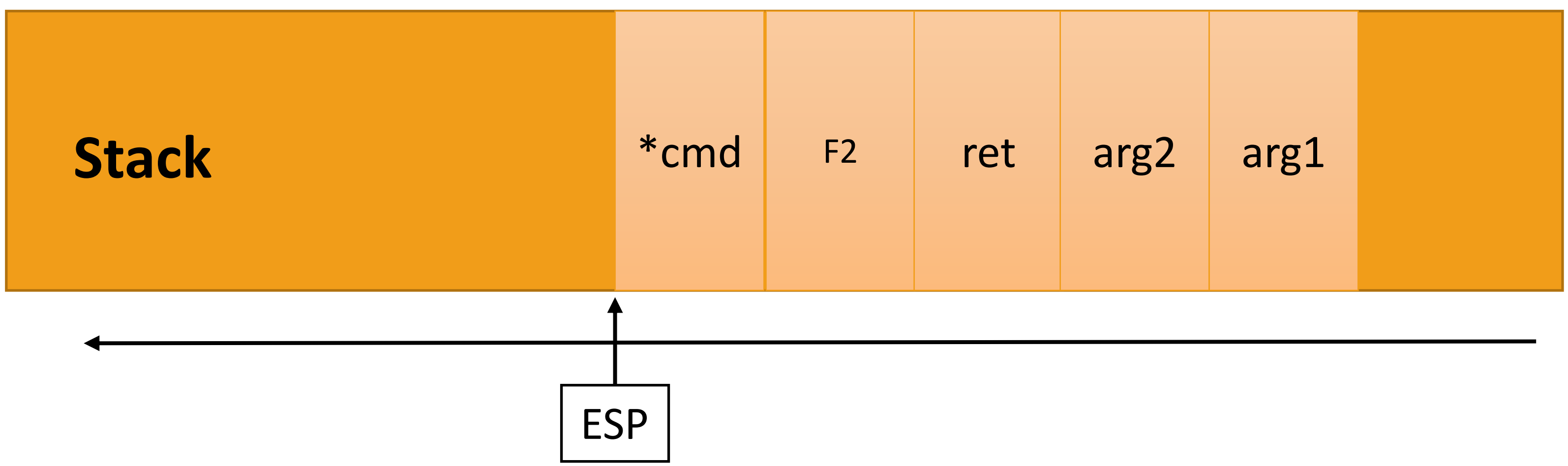
F1(cmd)



Chaining Functions with ret2libc

F1(cmd)

```
pop eax; ret
```

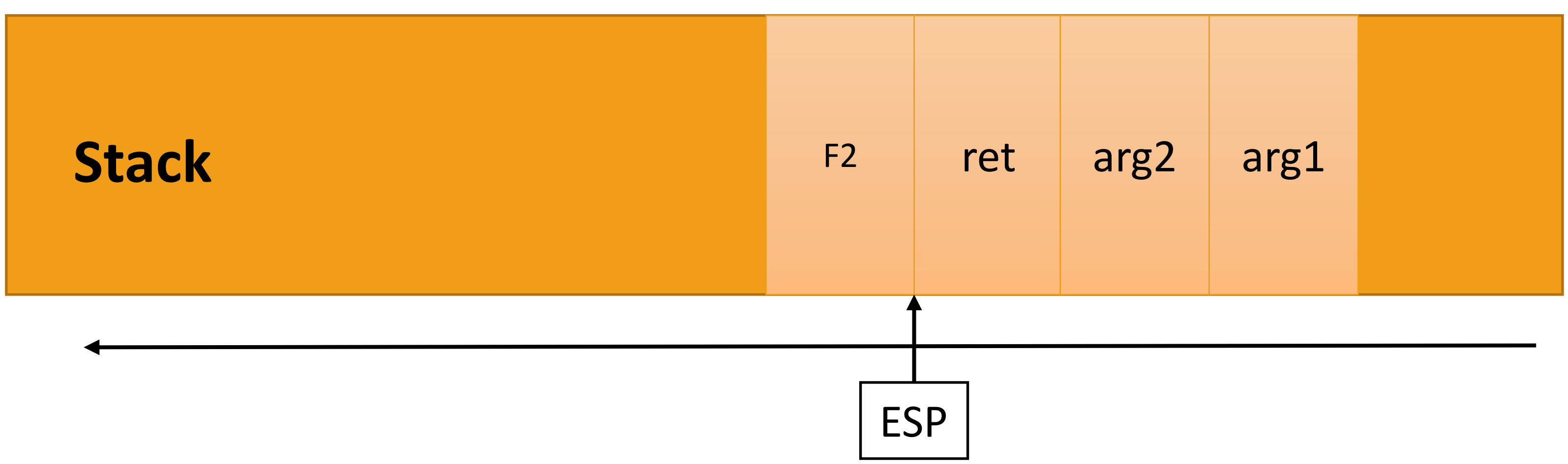


Chaining Functions with ret2libc

F1(cmd)

`pop eax; ret`

F1(arg1, arg2)



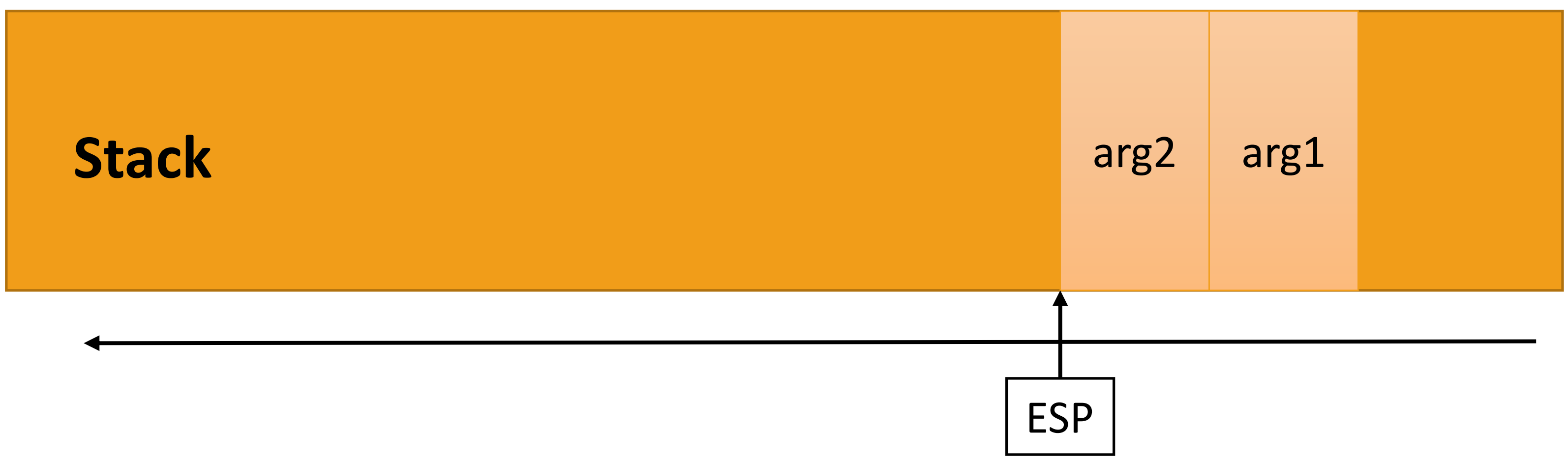
Chaining Functions with ret2libc

F1(cmd)

```
pop eax; ret
```

F1(arg1, arg2)

```
add 0x8,esp; ret
```



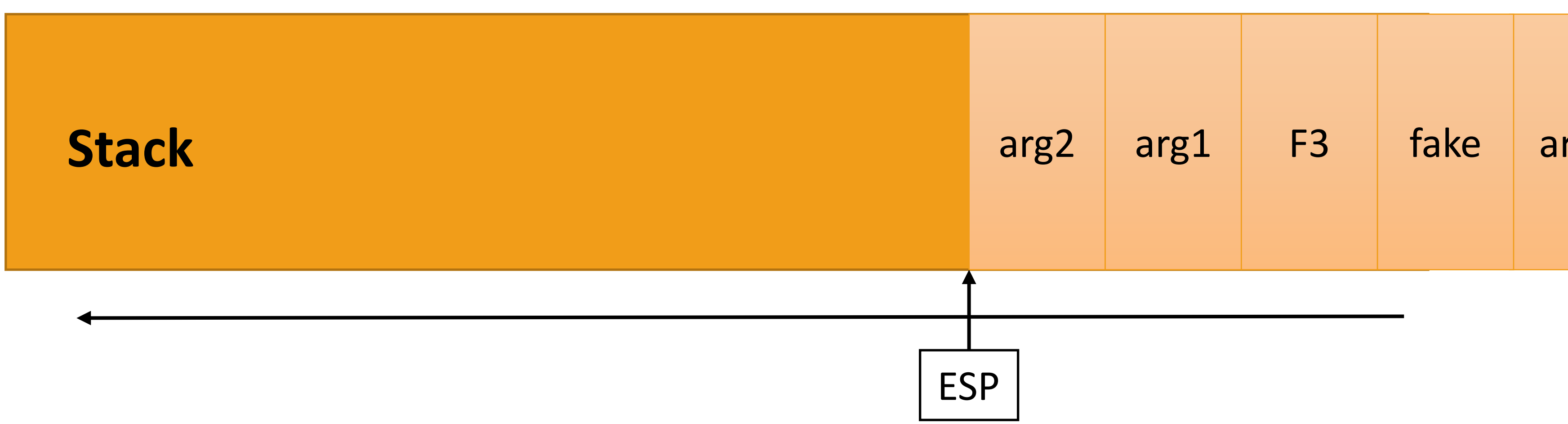
Chaining Functions with ret2libc

F1(cmd)

`pop eax; ret`

F1(arg1, arg2)

`add 0x8,esp; ret`



Chaining Functions with ret2libc

F1(cmd)

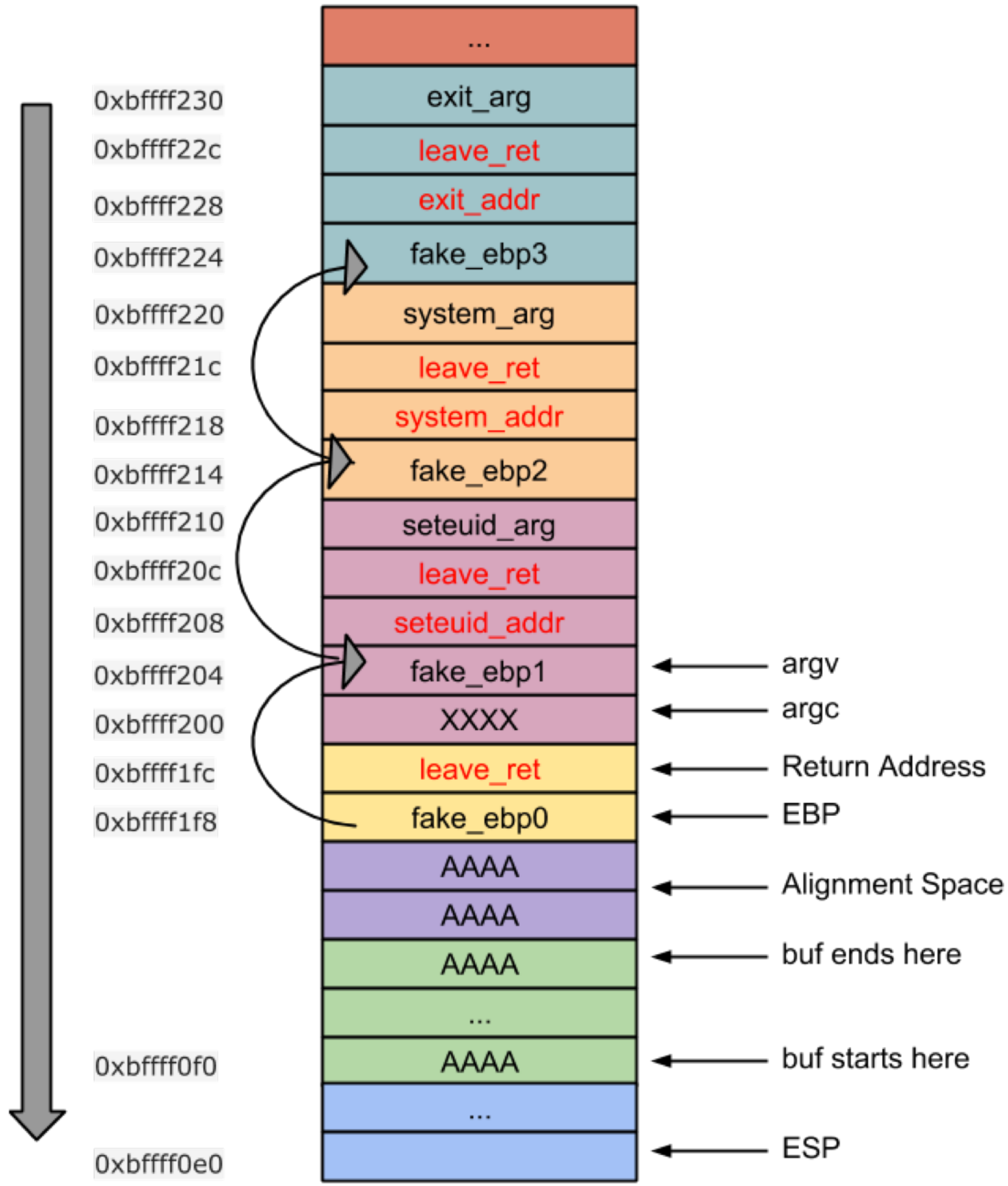
```
pop eax; ret
```

F1(arg1, arg2)

```
add 0x8,esp; ret
```

F3(arg3)

https://sploitfun.wordpress.com/2015/05/08/bypassing-nx-bit-using-chained-return-to-libc/



main() Stack Layout - Chained with multiple libc functions


I don’t like only
calling functions

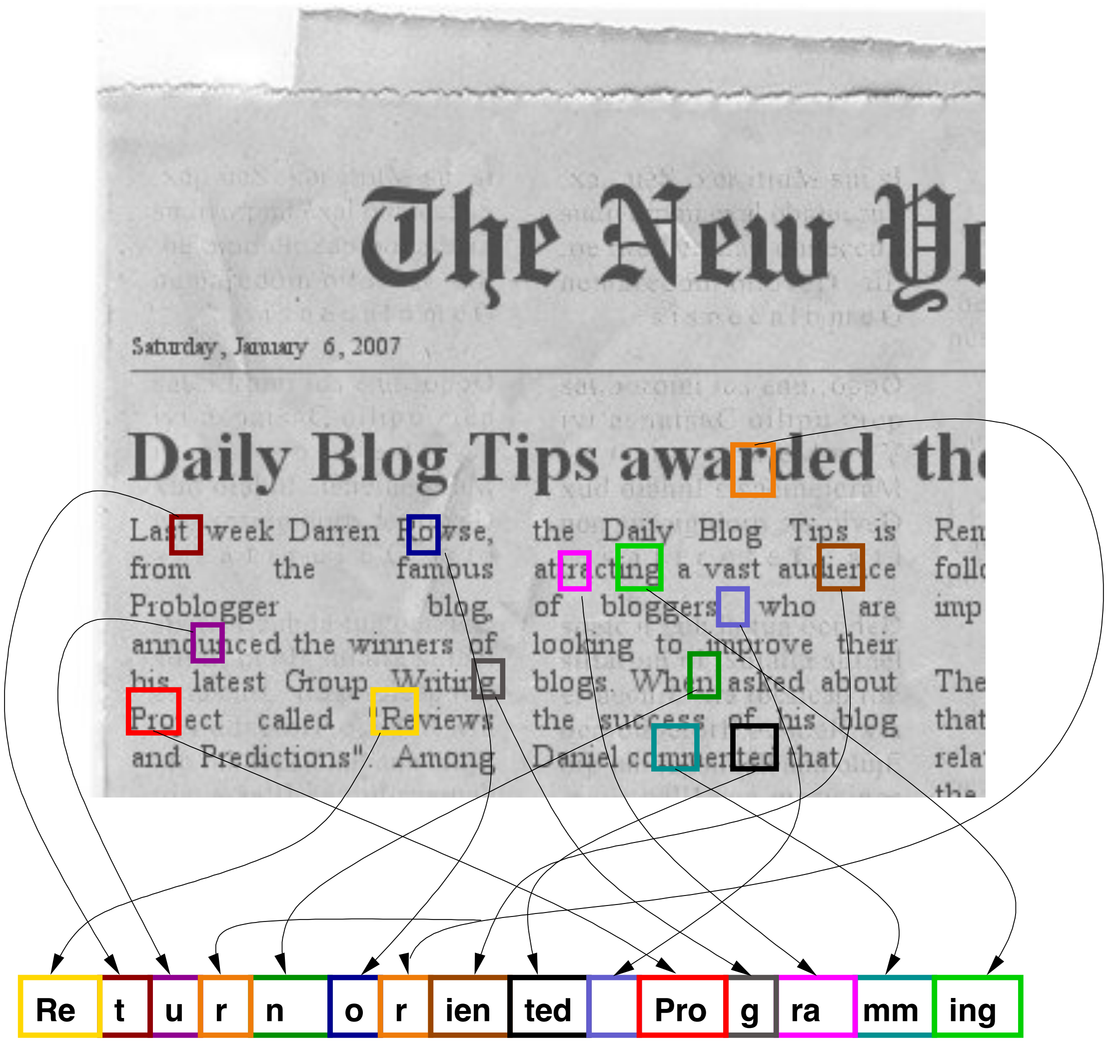
Enter Return-Oriented Programming

Re-use parts of the application's code to perform arbitrary computations

A Turing complete machine

Use the stack like a tape providing the data for the computation and the instruction pointer

A Code Collage



https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

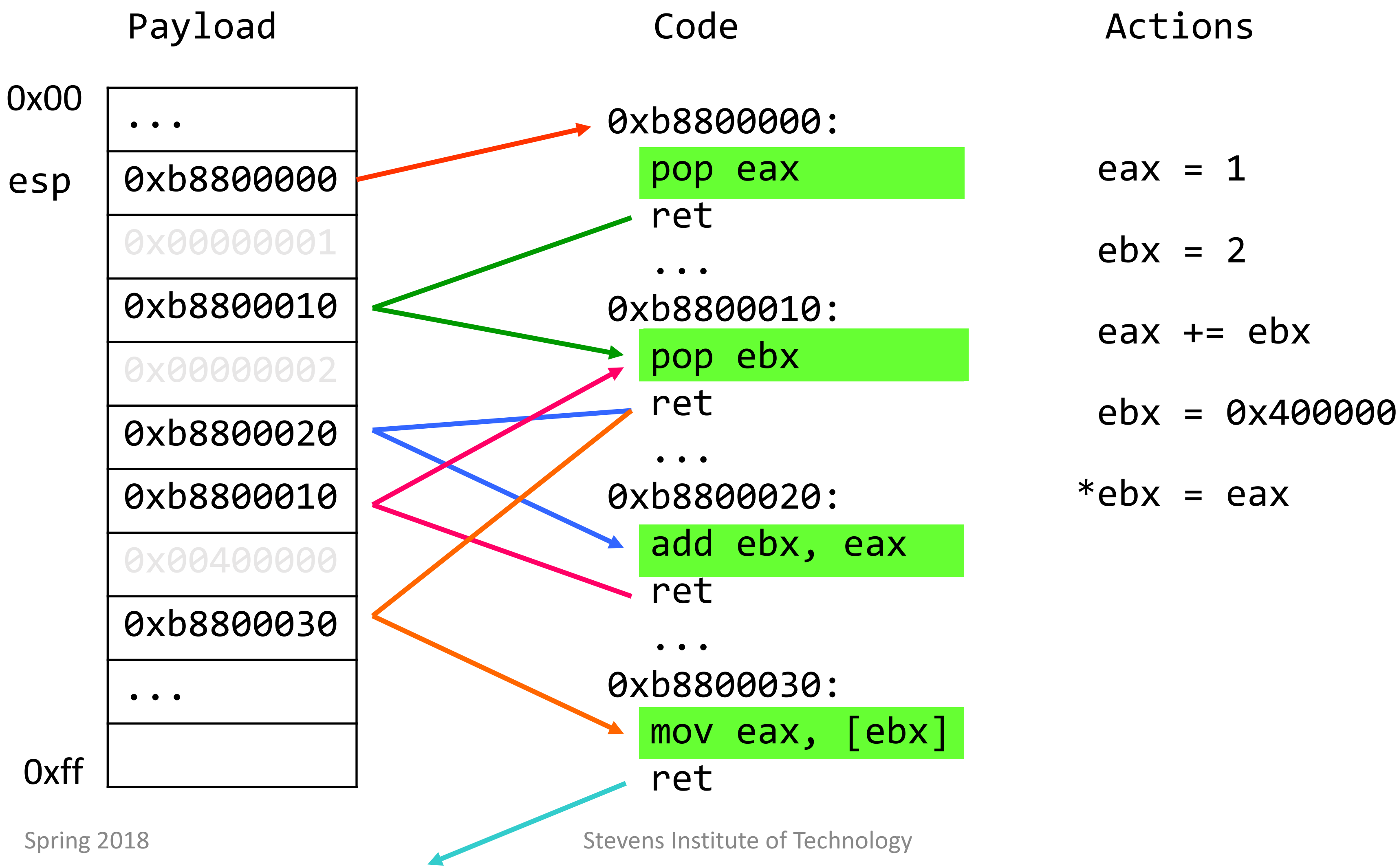
```
mov (%rcx),%rbx
test %rbx,%rbx
je 41c523 <main+0x803>
mov %rbx,%rdi
callq 42ab00
mov %rax,0x2cda9d(%rip)
cmpb $0x2d,(%rbx)
je 41c4ac <main+0x78c>
mov 0x2cda8d(%rip),%rax
ret
test %rbx,%rbx
mov $0x4ab054,%eax
cmove %rax,%rbx
mov %rbx,0x2cda6a(%rip)
test %rdi,%rdi
je 41c0c2 <main+0x3a2>
mov $0x63b,%edx
mov $0x4ab01d,%esi
callq 46cab0 <sh_xfree>
ret
```

```
mov %rax,0x2d2945(%rip)
mov 0x2cda16(%rip),%rax
test %rax,%rax
je 41c112 <
movzbl (%ra
callq 41b64
mov 0xb8(%r
cmp 0xc(%rsp),
mov %rax,0x2d2670(%rip)
je 41c214 <main+0x4f4>
xchg %ax,%ax
mov (%rsp),%rdx
movslq %r15d,%rax
mov (%rdx,%rax,8),%r14
ret
je 41c214 <main+0x4f4>
cmpb $0x2d,(%r14)
jne 41c214 <main+0x4f4>
movzbl 0x1(%r14),%r12d
movl $0x0,0x18(%rsp)
cmp $0x2d,%r12b
```

Gadgets

```
je 41c440 <main+0x720>
xor %ebp,%ebp
mov $0x4c223a,%ebx
add $0x1,%r14
jmp 41c1a3 <main+0x483>
cmp (%rbx),%r12b
mov %ebp,%r13d
jne 41c188 <main+0x468>
mov %rbx,%rsi
test %eax,%eax
xchg %ax,%ax
jne 41c188 <main+0x468>
movslq %ebp,%rax
ret
cmpl $0x1,0x4ab3c8(%rax)
je 41c461 <main+0x741>
mov (%rsp),%rcx
add $0x1,%r15d
movslq %r15d,%rdx
mov (%rcx,%rdx,8),%rdx
test %rdx,%rdx
je 41cefd <main+0x11dd>
```

An Example


Payload
Code
Actions
0x00
esp
0xff
...
0xb8800000
0x00000001
0xb8800010
0x00000002
0xb8800020
0xb8800010
0x00400000
0xb8800030
...
0xb8800000:
pop eax
ret
...
0xb8800010:
pop ebx
ret
...
0xb8800020:
add ebx, eax
ret
...
0xb8800030:
mov eax, [ebx]
ret
eax = 1
ebx = 2
eax += ebx
ebx = 0x400000
*ebx = eax

Current State of the Art

First-stage ROP code for bypassing NX

- Allocate/set W+X memory (VirtualAlloc, VirtualProtect, …)
- Copy embedded shellcode into the newly allocated area

Second stage jumps to injected code

Pure-ROP exploits

- In-the-wild exploit against Adobe Reader XI
- CVE-2013-0640

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

Control-flow Integrity aims to restrict the arbitrary manipulation of the program counter

Control Flow Manipulation

Function calls

```
my_function(arg1, arg2)
```

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

Function returns

```
return;
```

```
return 100;
```

If statements

```
if (cond) {
} else {
}
```

Loops

```
for () { }
```

```
while { }
```

```
do { } while
```

Break/continue

```
while (true) {
     if (cond)
          break;
}
```

```
while (cond) {
     if (cond2)
          continue;
}
```

Switch statement

```
switch (cond) {
   val1: … break;
   val2: … break;
}
```

goto statement

```
goto label1;
…
Label1:
```

Control-Flow Hijacking Prone Statements

Statements where the target statement cannot be known a priori

- Indirect control-flow transfers

```
return;
```

```
return 100;
```

```
switch (cond) {
   val1: … break;
   val2: … break;
}
```

Indirect calls, returns, and some switches

```
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

Machine Code

```
return;
```

```
return 100;
```

```
ret
```

```
switch (cond) {
   val1: … break;
   val2: … break;
}
```

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

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

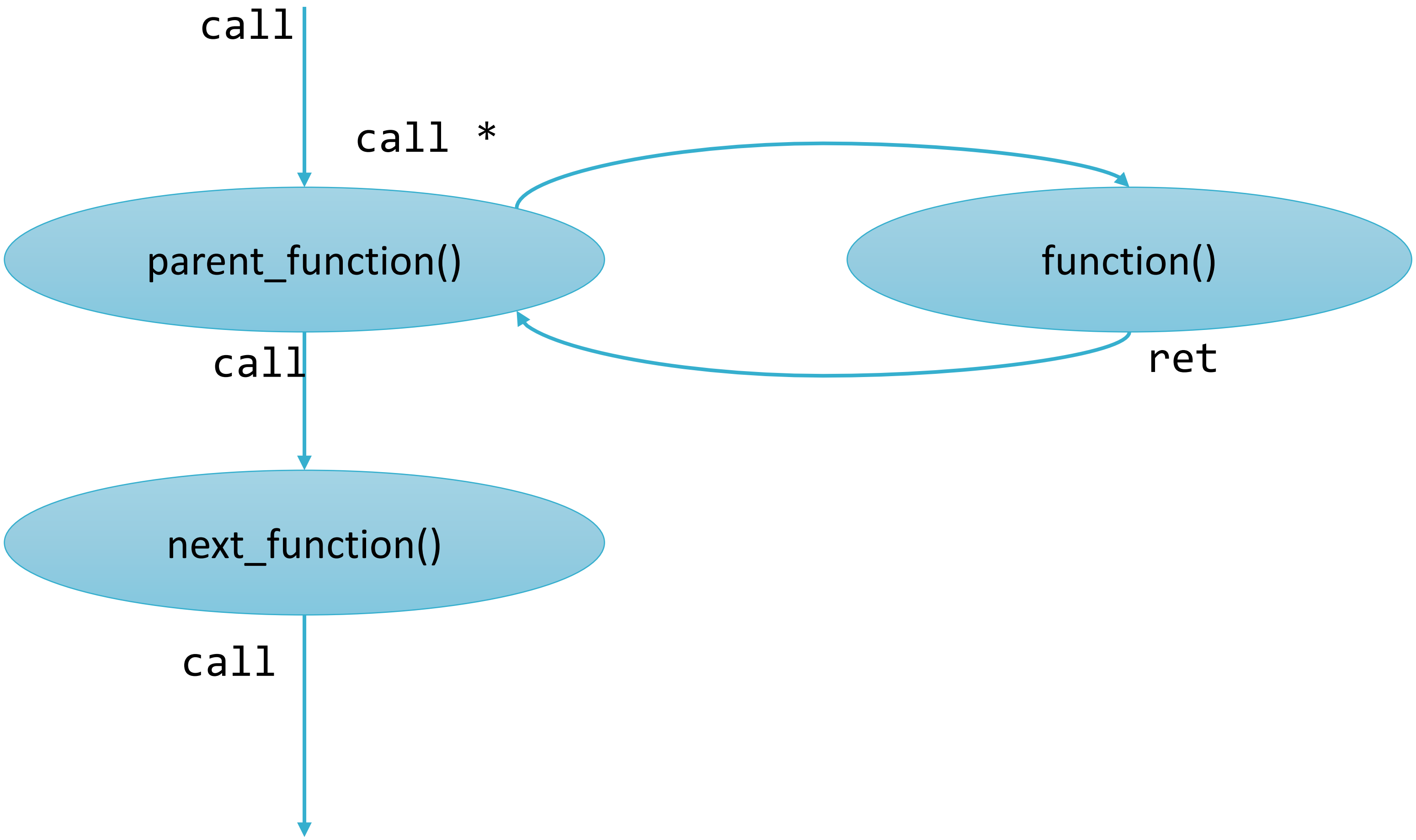
```
jmp *(%rax)
```

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

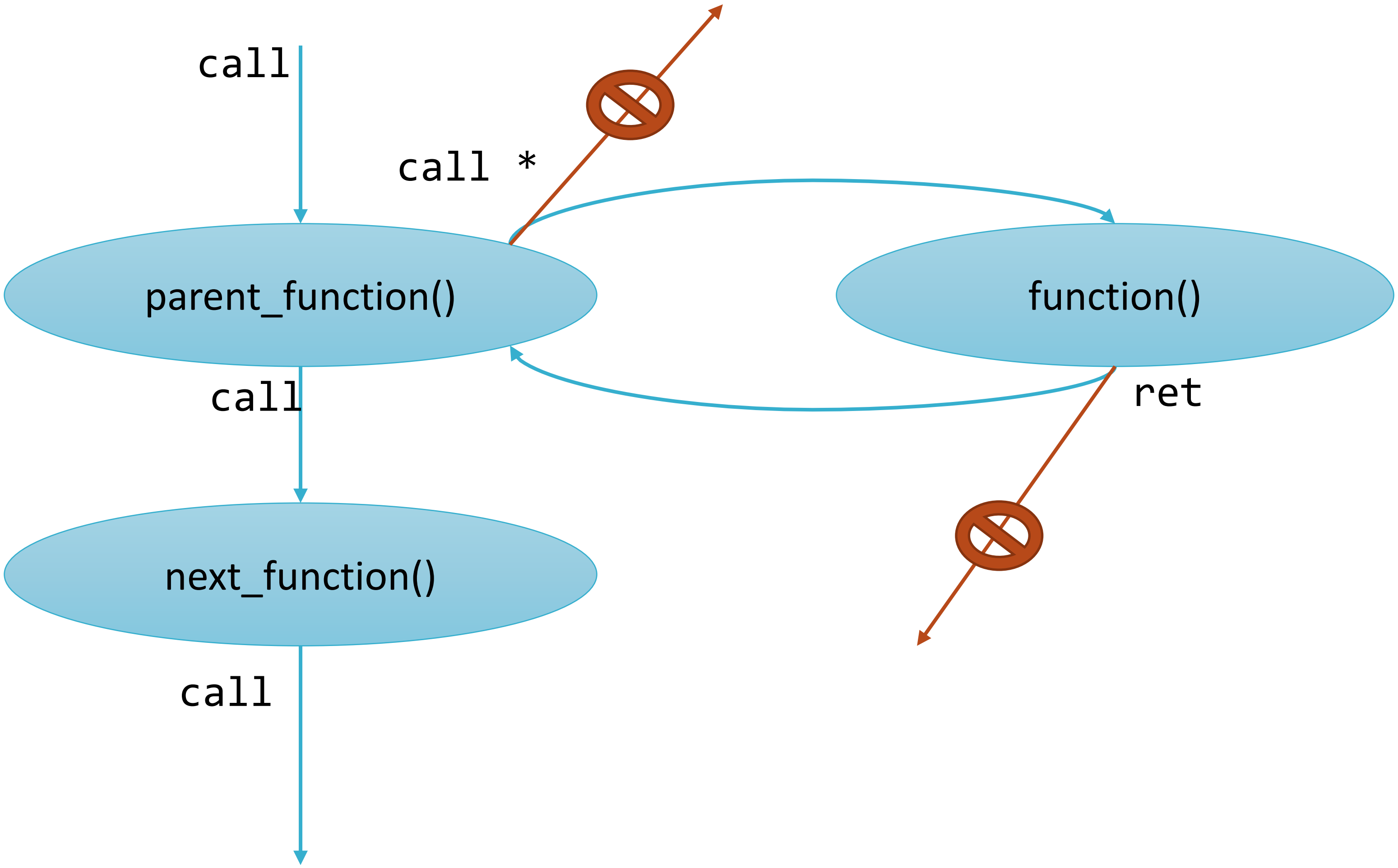
```
Class C {
   virtual void vcall(void);
}

C obj = new C();

obj->vcall():
```

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

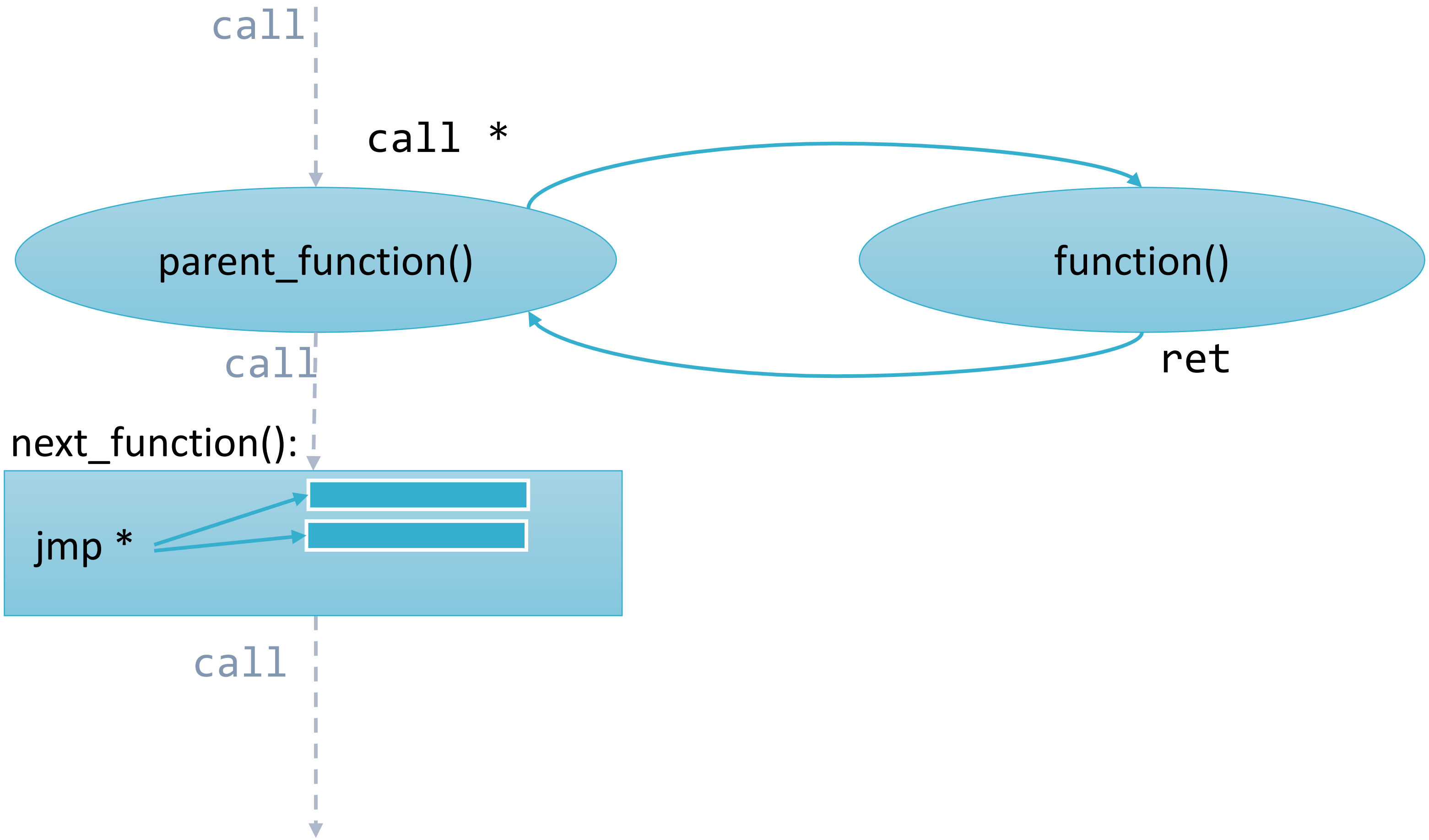
Function Call Graph (FCG)



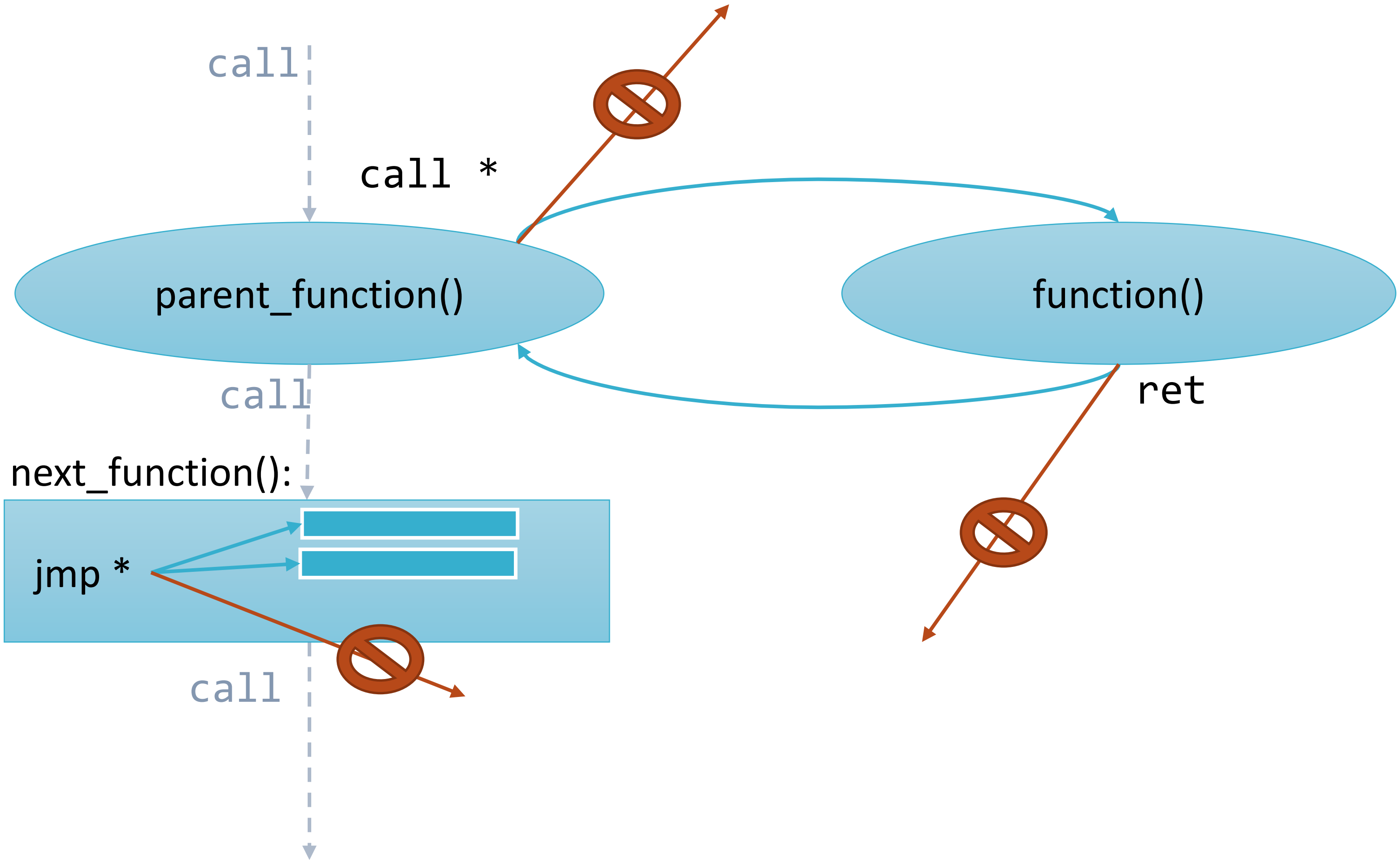
FCG Enforcement



Control-flow Graph (CFG)
Indirect flows only



CFI - CFG Enforcement



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);
        }
}
```

```
4028d1:     be 71 85 41 00          mov    $0x418571,%esi
4028d6:     bf 06 00 00 00          mov    $0x6,%edi
4028db:     e8 30 fe ff ff          callq  402710 <setlocale@plt>
4028e0:     be 3f 51 41 00          mov    $0x41513f,%esi
4028e5:     bf 28 51 41 00          mov    $0x415128,%edi
4028ea:     e8 51 fa ff ff          callq  402340 <bindtextdomain@plt>
4028ef:     bf 28 51 41 00          mov    $0x415128,%edi
4028f4:     e8 07 fa ff ff          callq  402300 <textdomain@plt>
4028f9:     bf c0 a1 40 00          mov    $0x40a1c0,%edi
4028fe:     c7 05 d8 9c 21 00 02    movl   $0x2,0x219cd8(%rip)        # 61c5e0 <_fini+0x20a054>
402905:     00 00 00
402908:     e8 63 fc 00 00          callq  412570 <__sprintf_chk@plt+0xfce0>
40290d:     48 b8 00 00 00 00 00    movabs $0x8000000000000000,%rax
402914:     00 00 80
402917:     c7 05 0f a8 21 00 00    movl   $0x0,0x21a80f(%rip)        # 61d130 <stderr+0xa80>
40291e:     00 00 00
402921:     c6 05 a8 a8 21 00 01    movb   $0x1,0x21a8a8(%rip)        # 61d1d0 <stderr+0xb20>
402928:     48 89 05 51 a9 21 00    mov    %rax,0x21a951(%rip)        # 61d280 <stderr+0xbd0>
40292f:     8b 05 97 9c 21 00       mov    0x219c97(%rip),%eax        # 61c5cc <_fini+0x20a040>
402935:     48 c7 05 50 a9 21 00    movq   $0x0,0x21a950(%rip)        # 61d290 <stderr+0xbe0>
40293c:     00 00 00 00
402940:     48 c7 05 3d a9 21 00    movq   $0xffffffffffffffff,0x21a93d(%rip)        # 61d288 <stderr+0xbd8>
402947:     ff ff ff ff
40294b:     c6 05 9e a8 21 00 00    movb   $0x0,0x21a89e(%rip)        # 61d1f0 <stderr+0xb40>
402952:     83 f8 02                cmp    $0x2,%eax
402955:     0f 84 83 08 00 00       je     4031de <__sprintf_chk@plt+0x94e>
40295b:     83 f8 03                cmp    $0x3,%eax
40295e:     74 2f                   je     40298f <__sprintf_chk@plt+0xff>
402960:     83 e8 01                sub    $0x1,%eax
402963:     74 05                   je     40296a <__sprintf_chk@plt+0xda>
402965:     e8 b6 f8 ff ff          callq  402220 <abort@plt>
40296a:     bf 01 00 00 00          mov    $0x1,%edi
40296f:     e8 0c f9 ff ff          callq  402280 <isatty@plt>
402974:     85 c0                   test   %eax,%eax
402976:     0f 84 2c 0e 00 00       je     4037a8 <__sprintf_chk@plt+0xf18>
40297c:     c7 05 ca a8 21 00 02    movl   $0x2,0x21a8ca(%rip)        # 61d250 <stderr+0xba0>
402983:     00 00 00
```

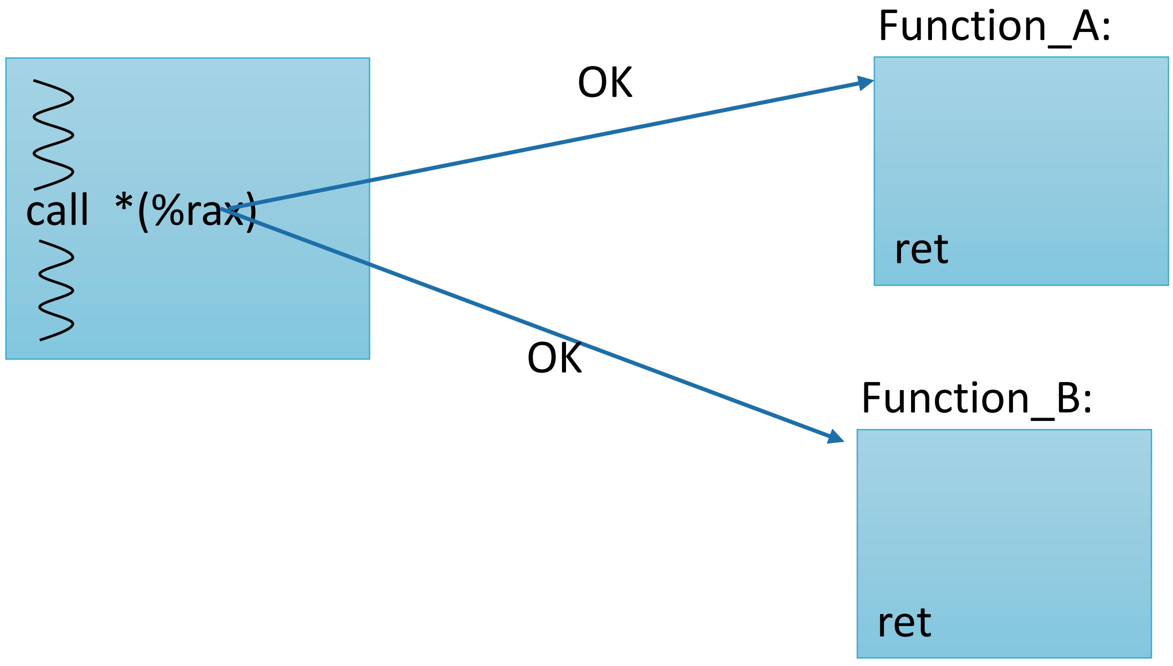
Working with an Imperfect CFG

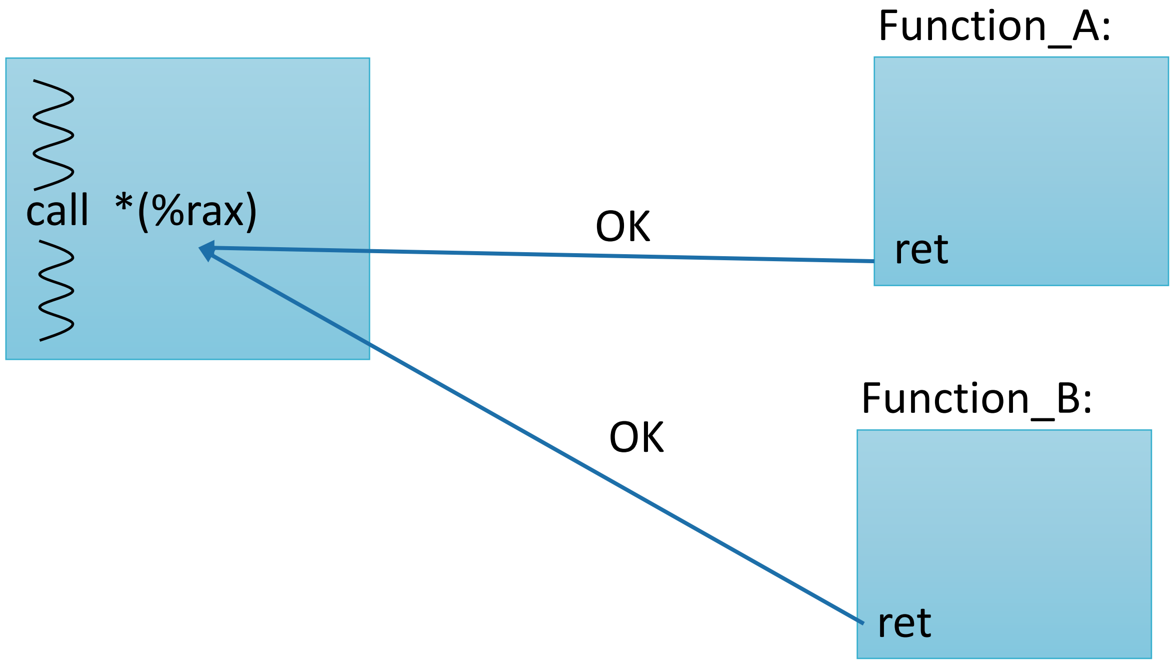
Lets assume that we know/can learn

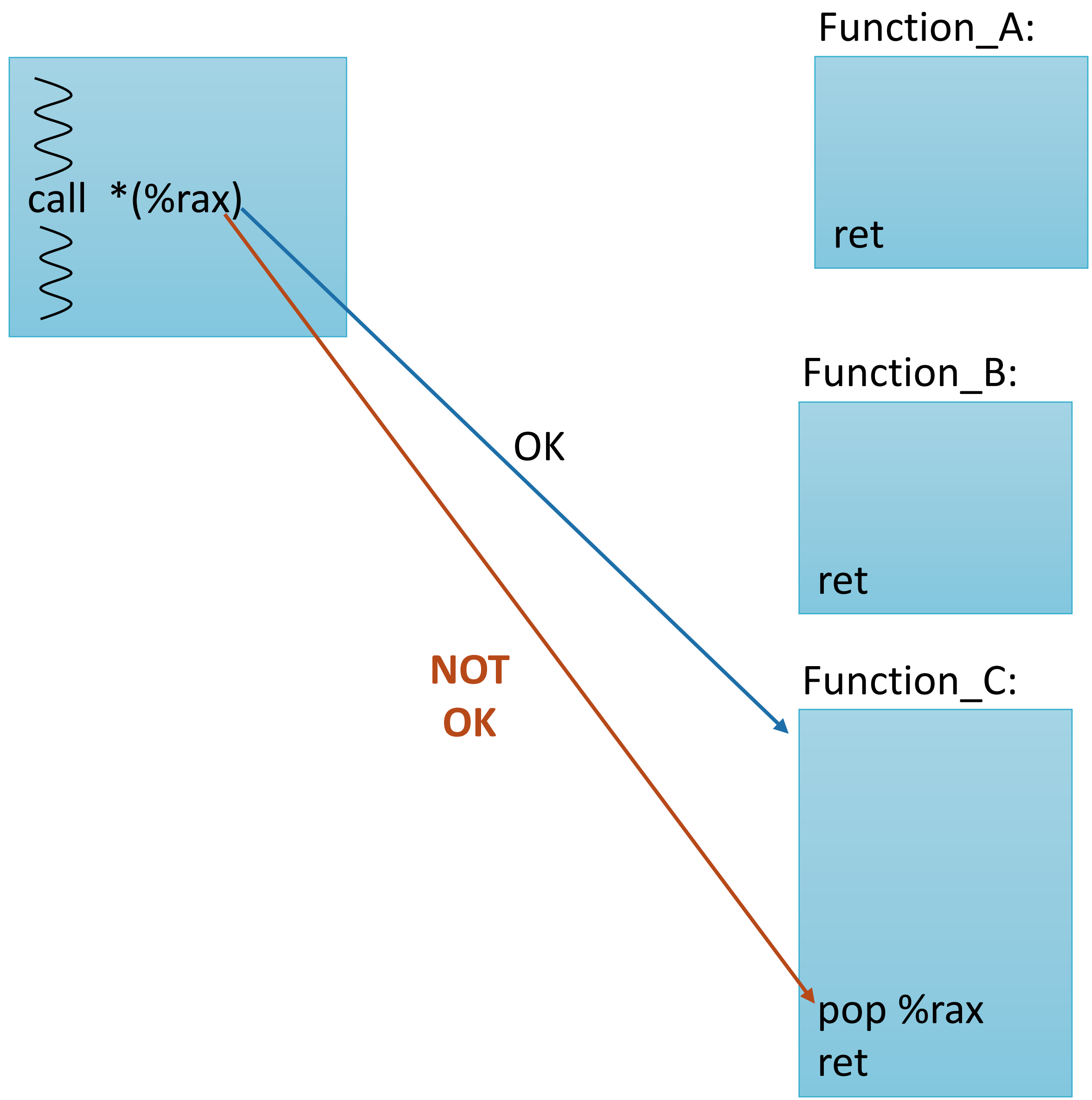
- The location of every function
- The location of every indirect branch instruction

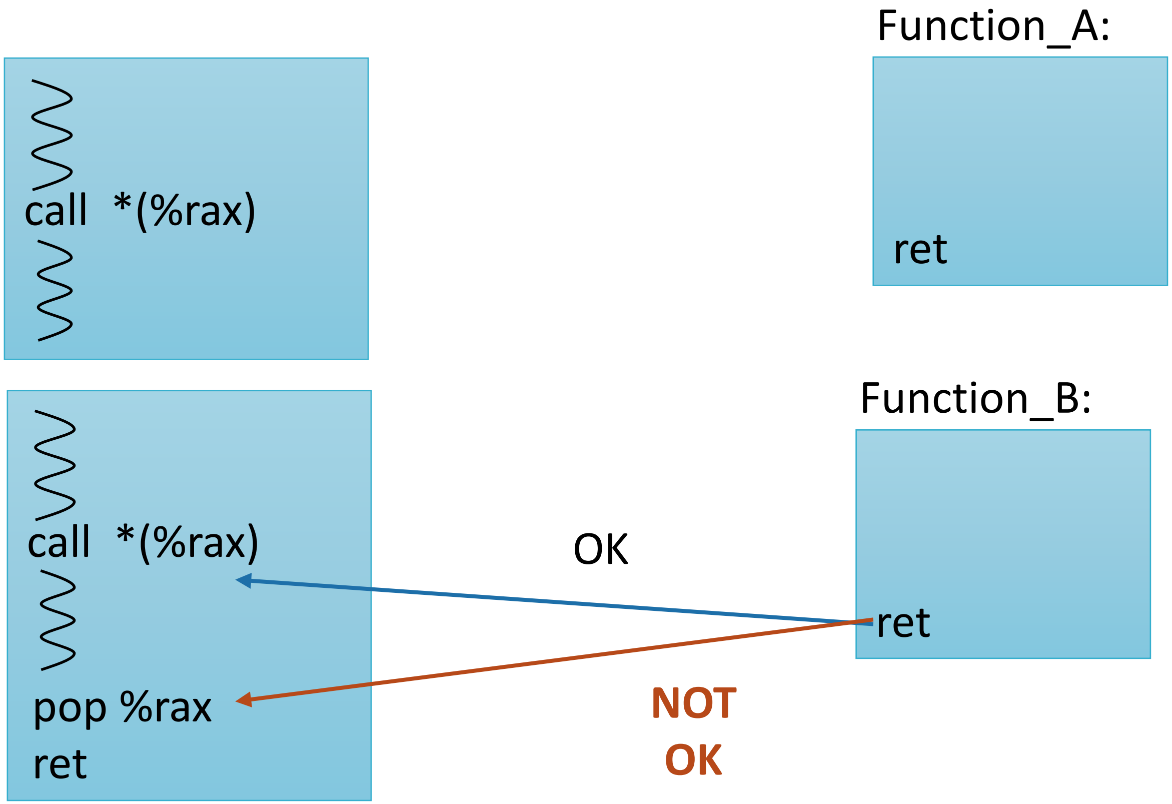
Coarse-grained CFI can enforce the following

- 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


Function_A:
Function_B:
call *(%rax)
OK
OK
ret
ret


Function_A:
call *(%rax)
OK
ret
Function_B:
OK
ret

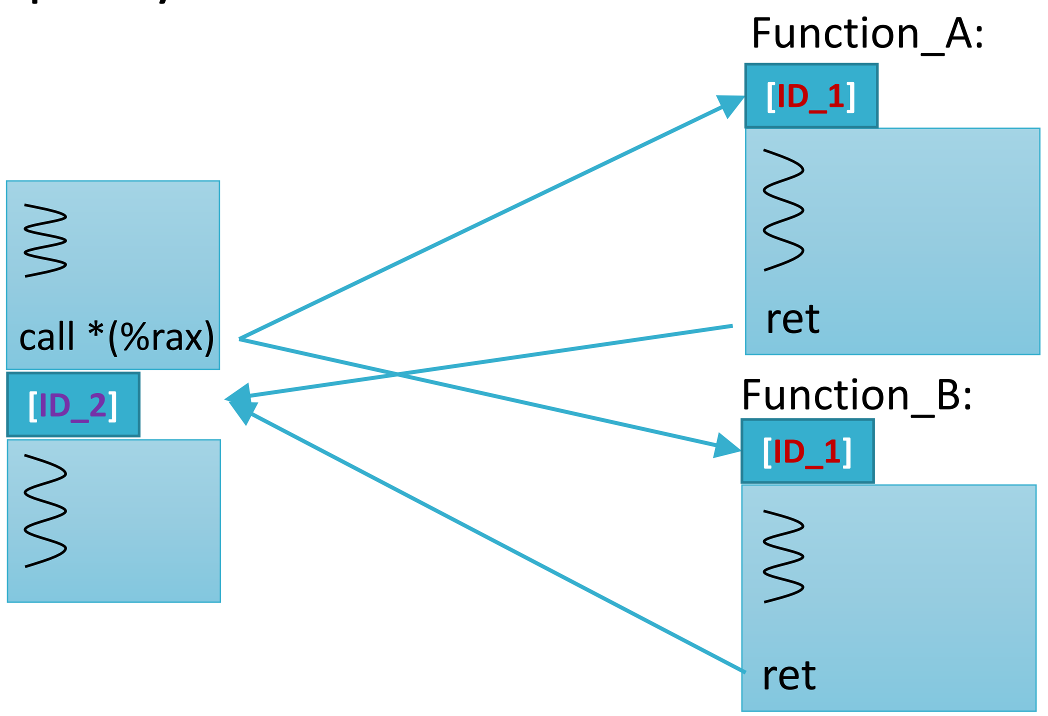

call *(%rax)
Function_A:
ret
Function_B:
ret
OK
NOT
OK
Function_C:
pop %rax
ret


Function_A:
ret
call *(%rax)
Function_B:
call *(%rax)
OK
ret
pop %rax
ret
NOT OK

Enforcing Through Embedded IDs

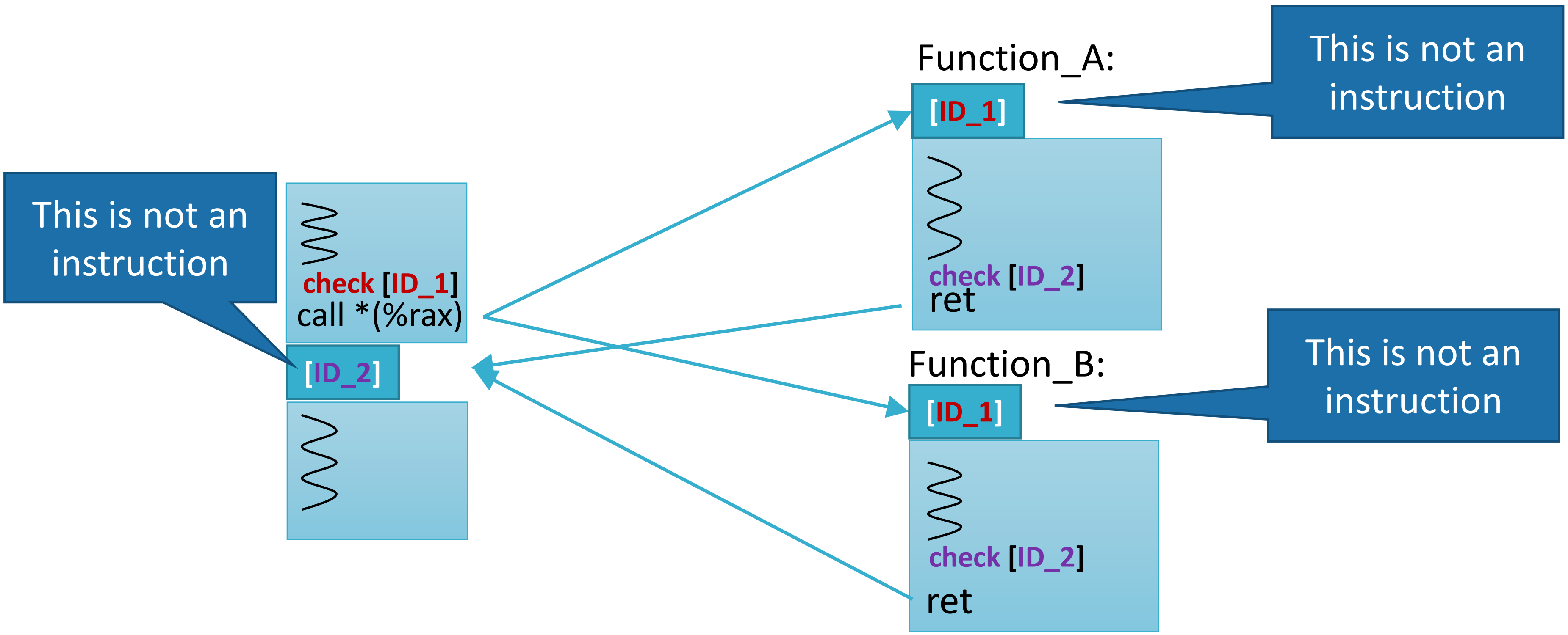
ID codes are embedded into the binary program to identify acceptable targets

- 2-ID policy

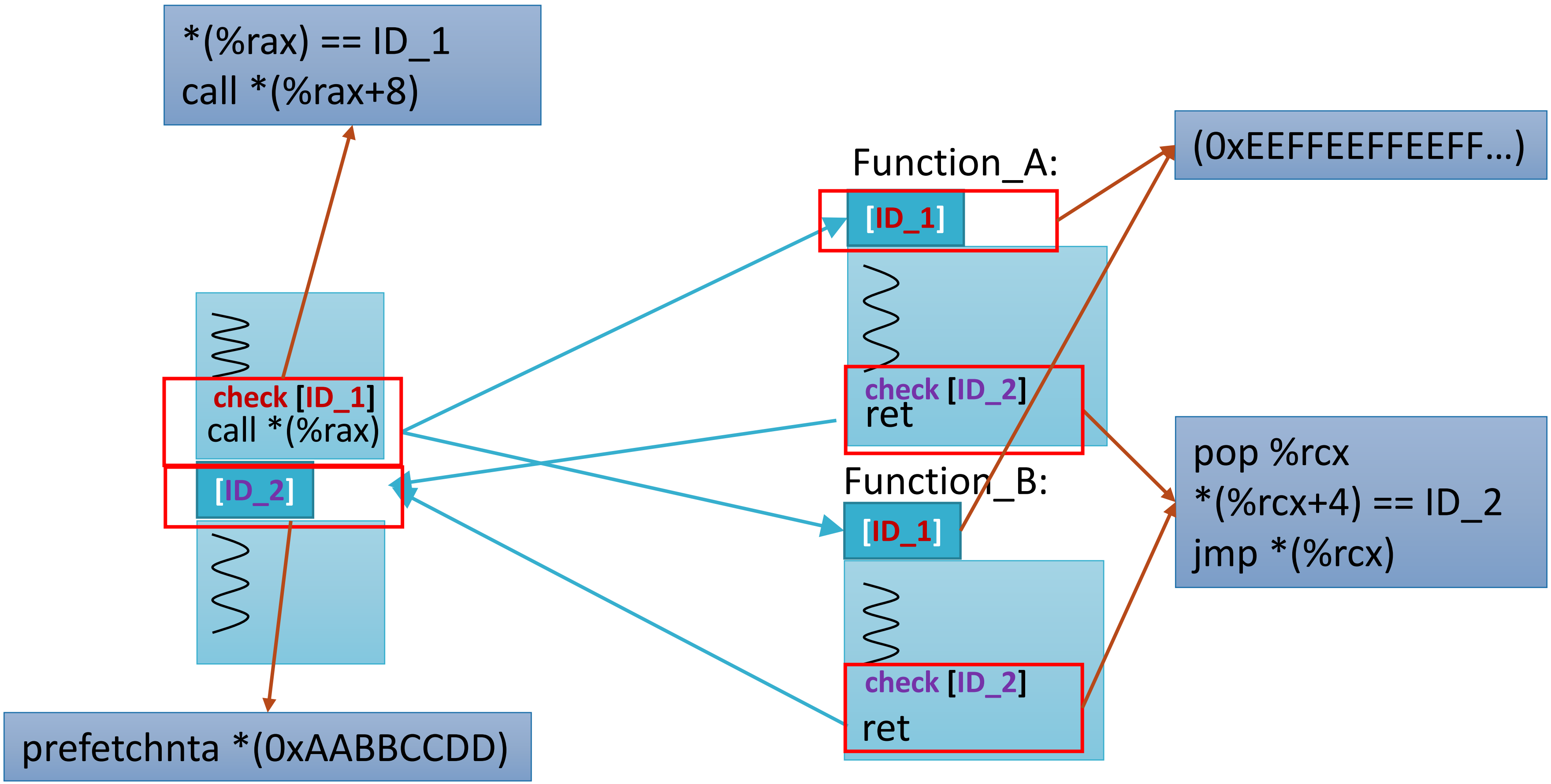


Enforcing Through Embedded IDs

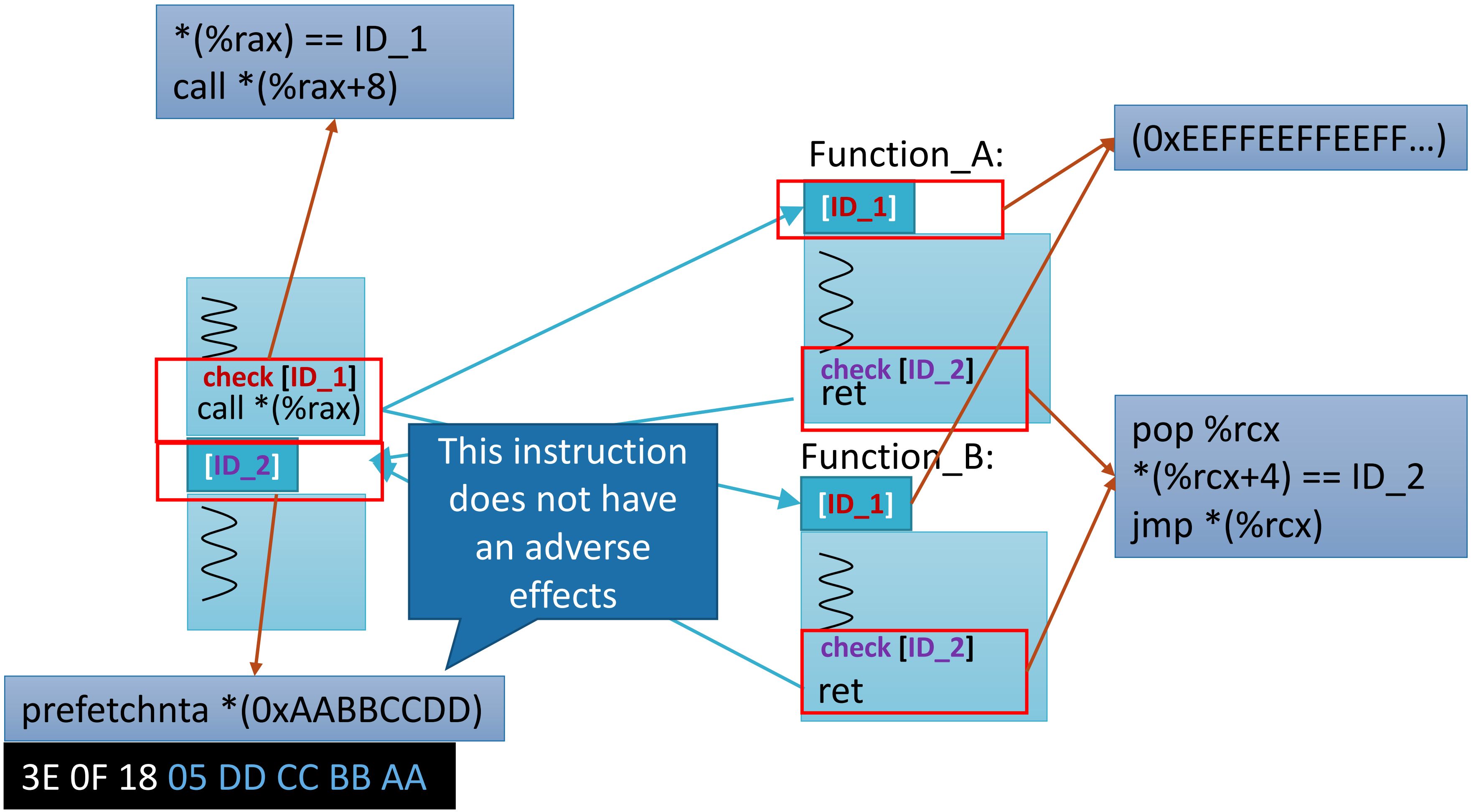
Checks are introduced right before the control transfer



Modifications for CFI Enforcement



Modifications for CFI Enforcement



Control-flow integrity

Martín Abadi	University of California, Santa Cruz and Microsoft Research, Santa Cruz, CA
Mihai Budiu	Microsoft Research
Úlfar Erlingsson	Reykjavík University and Microsoft Research
Jay Ligatti	University of South Florida, Tampa, FL

ACM Transactions on Information and System Security (TISSEC)

http://dl.acm.org/citation.cfm?id=1609960

Limitations:

- Code integrity must be ensured (no code injection)
- Incremental deployment is not supported (all or nothing)
- Only 2 IDs are supported for enforcing CFI

Practical Control Flow Integrity and Randomization for Binary Executables

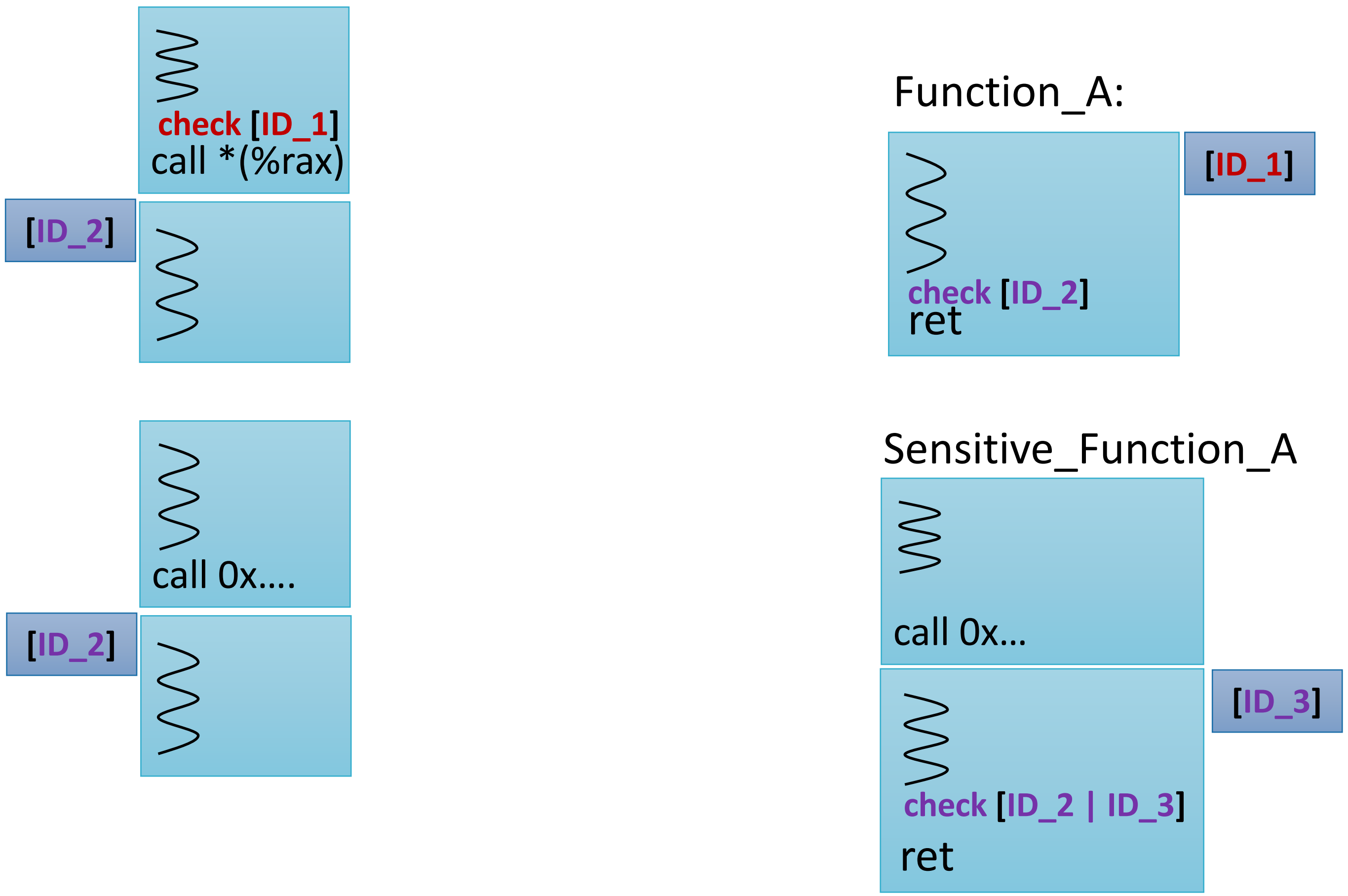
Chao Zhang
Tao Wei
Zhaofeng Chen
Lei Duan
Laszlo Szekeres
Stephen McCamant
Dawn Song
Wei Zou

Proceedings of the 2013 IEEE Symposium on Security and Privacy

http://dl.acm.org/citation.cfm?id=2498134

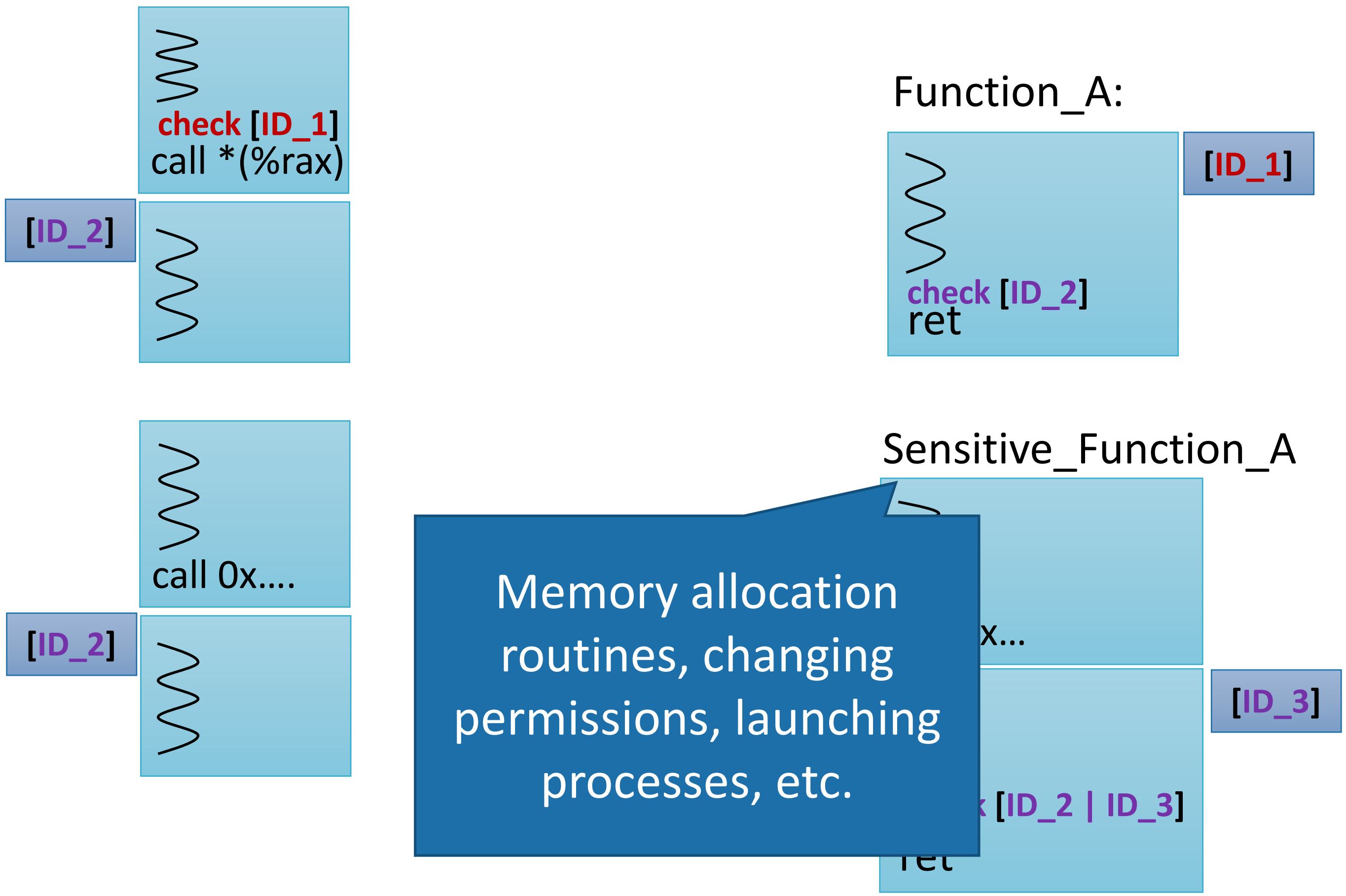
CCFIR

Three IDs are used to restrict control flow



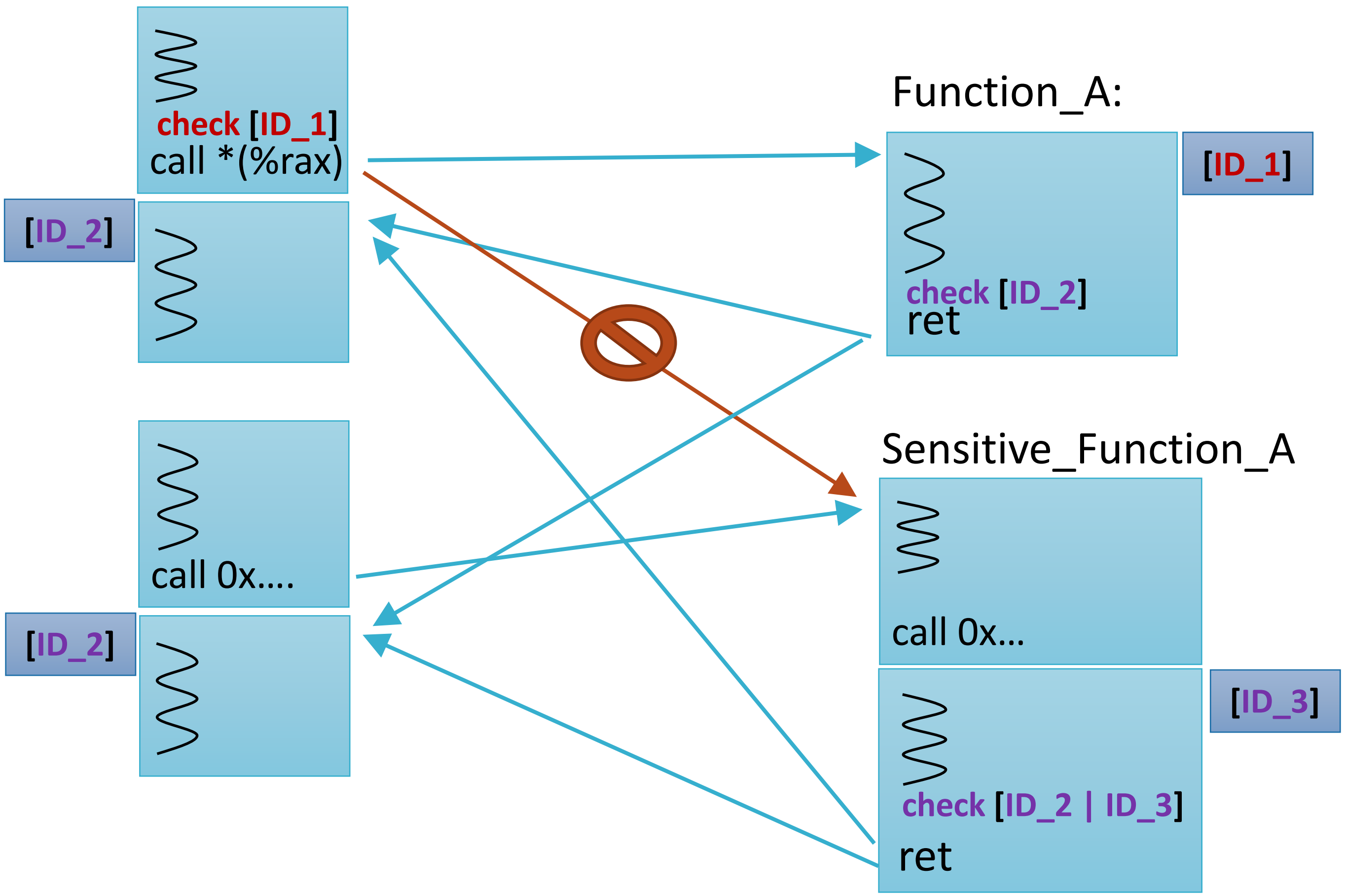
CCFIR

Three IDs are used to restrict control flow



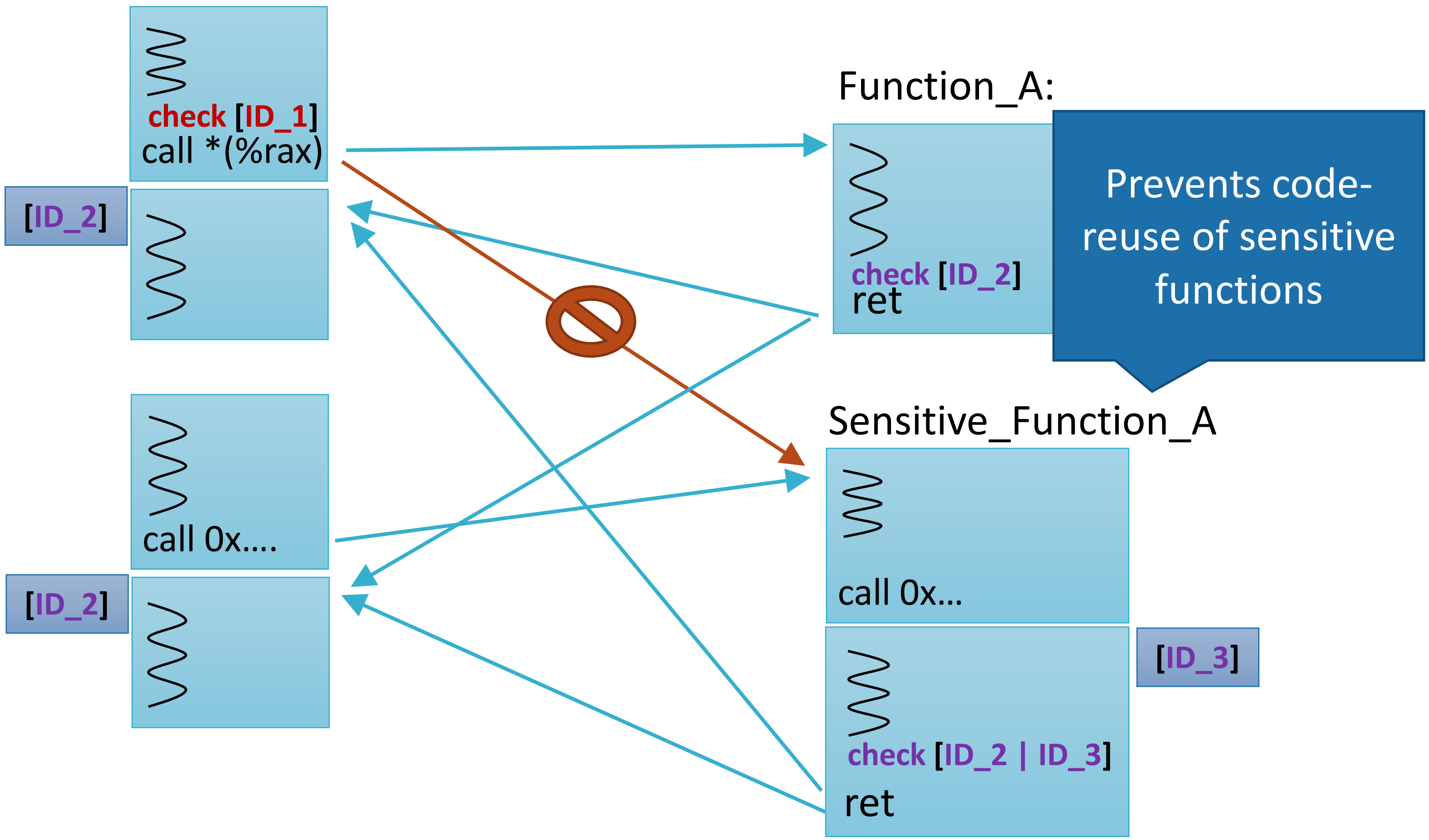
CCFIR

Three IDs are used to restrict control flow

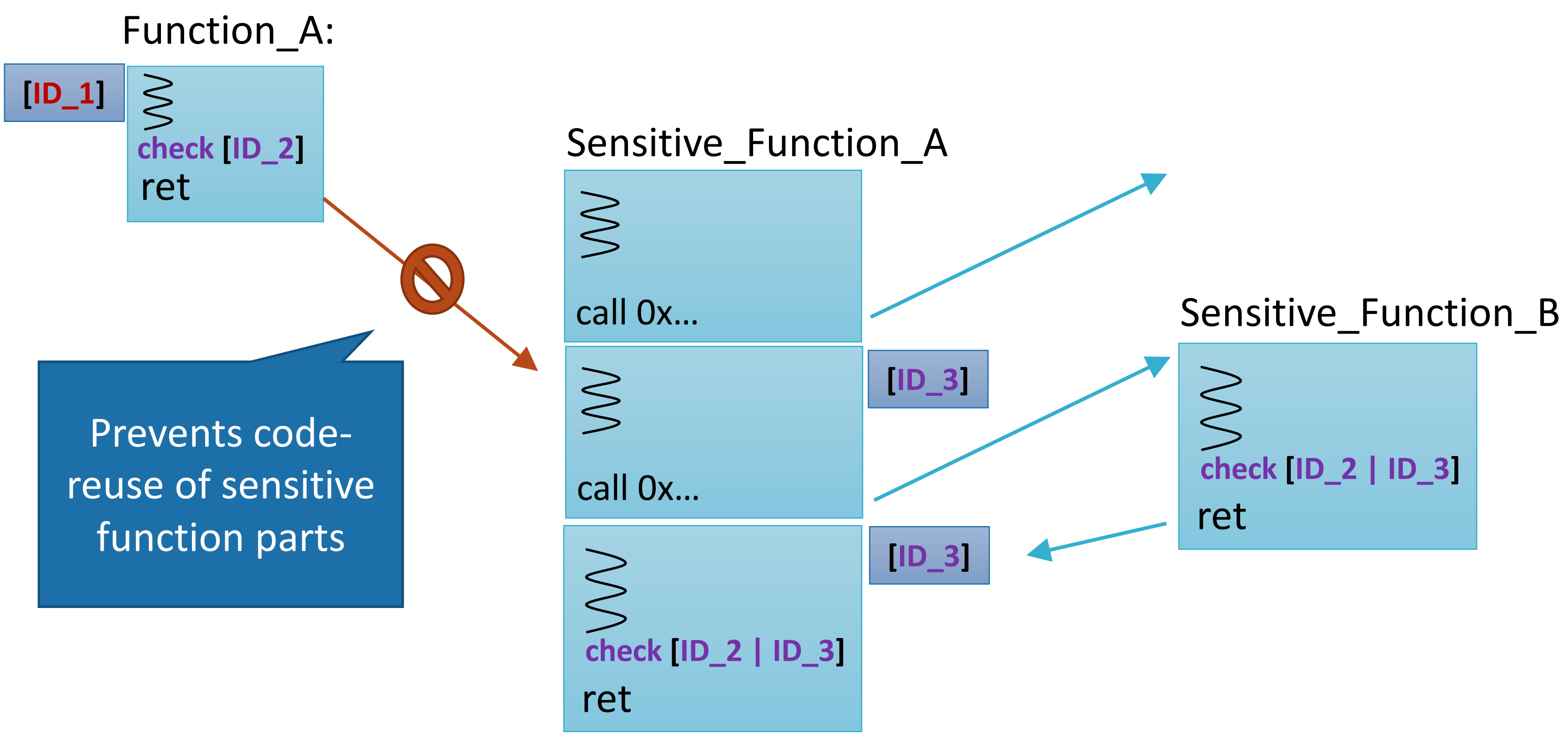


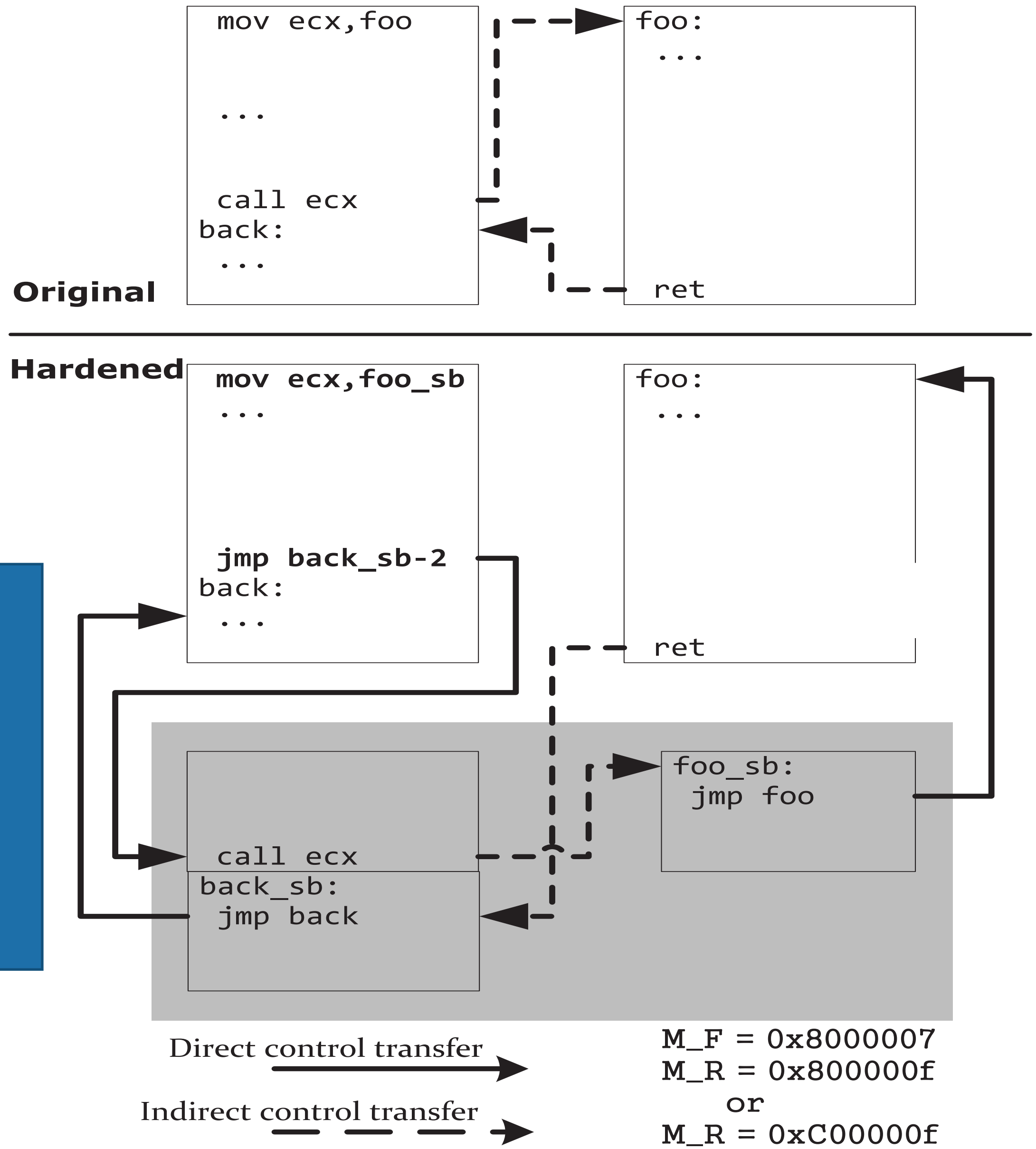
CCFIR

Three IDs are used to restrict control flow



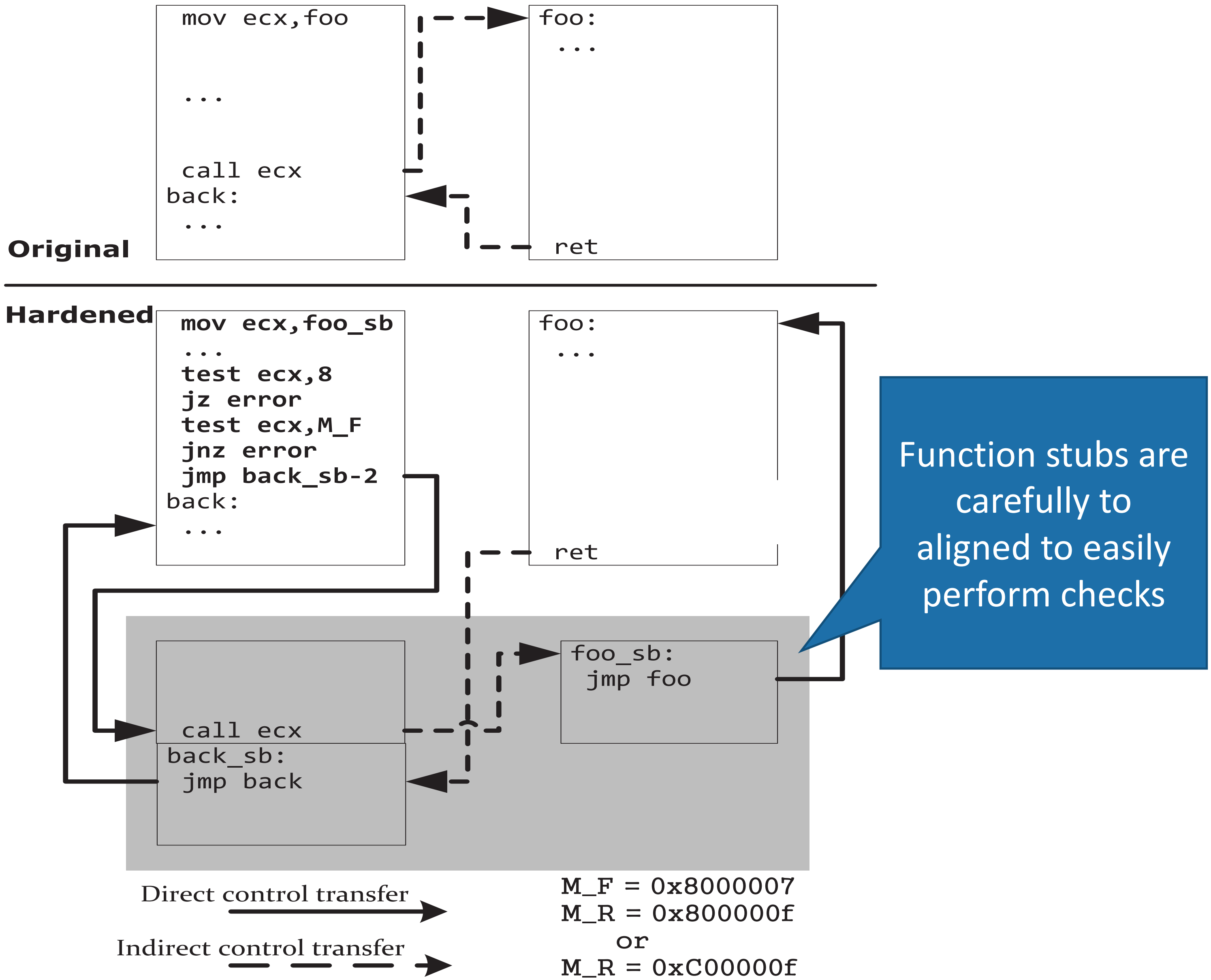
Sensitive Functions Heuristic





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

Targeted functions are called indirectly through another trampoline

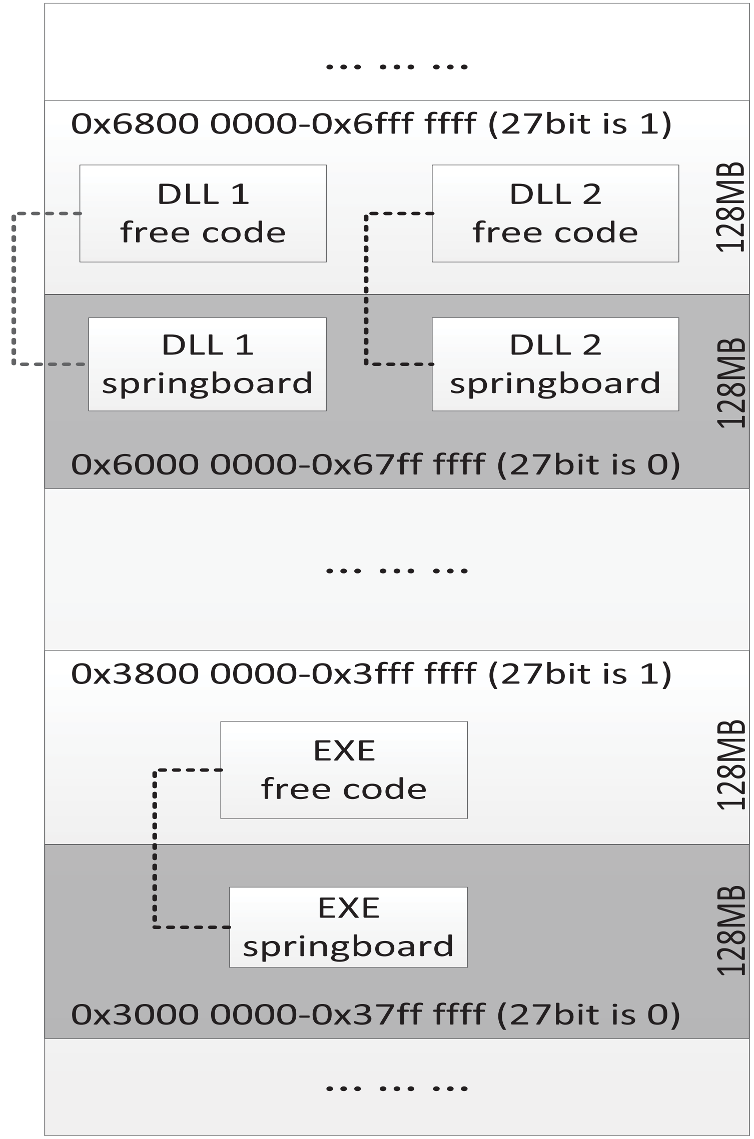


mov ecx,foo
...
call ecx
back:
...
Original
foo:
...
ret
Hardened
mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...
foo:
...
ret
call ecx
back_sb:
jmp back
foo_sb:
jmp foo
Function stubs are carefully to aligned to easily perform checks
Direct control transfer
Indirect control transfer
M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f

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

↓

```
0
```

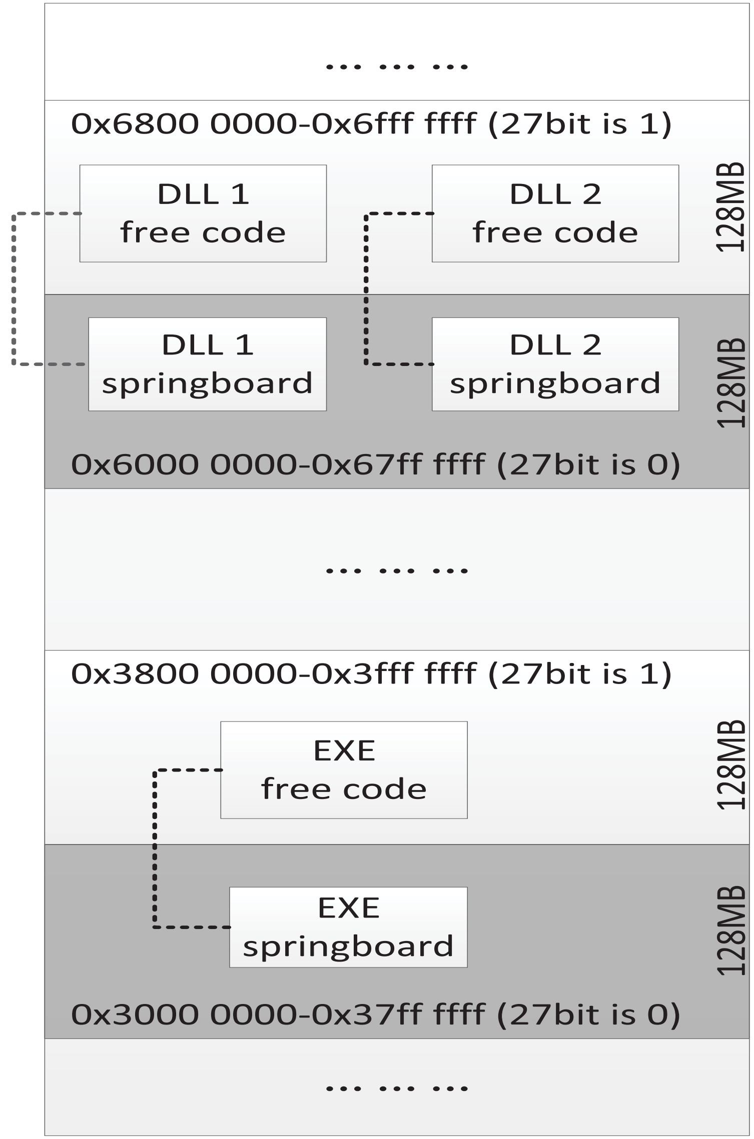


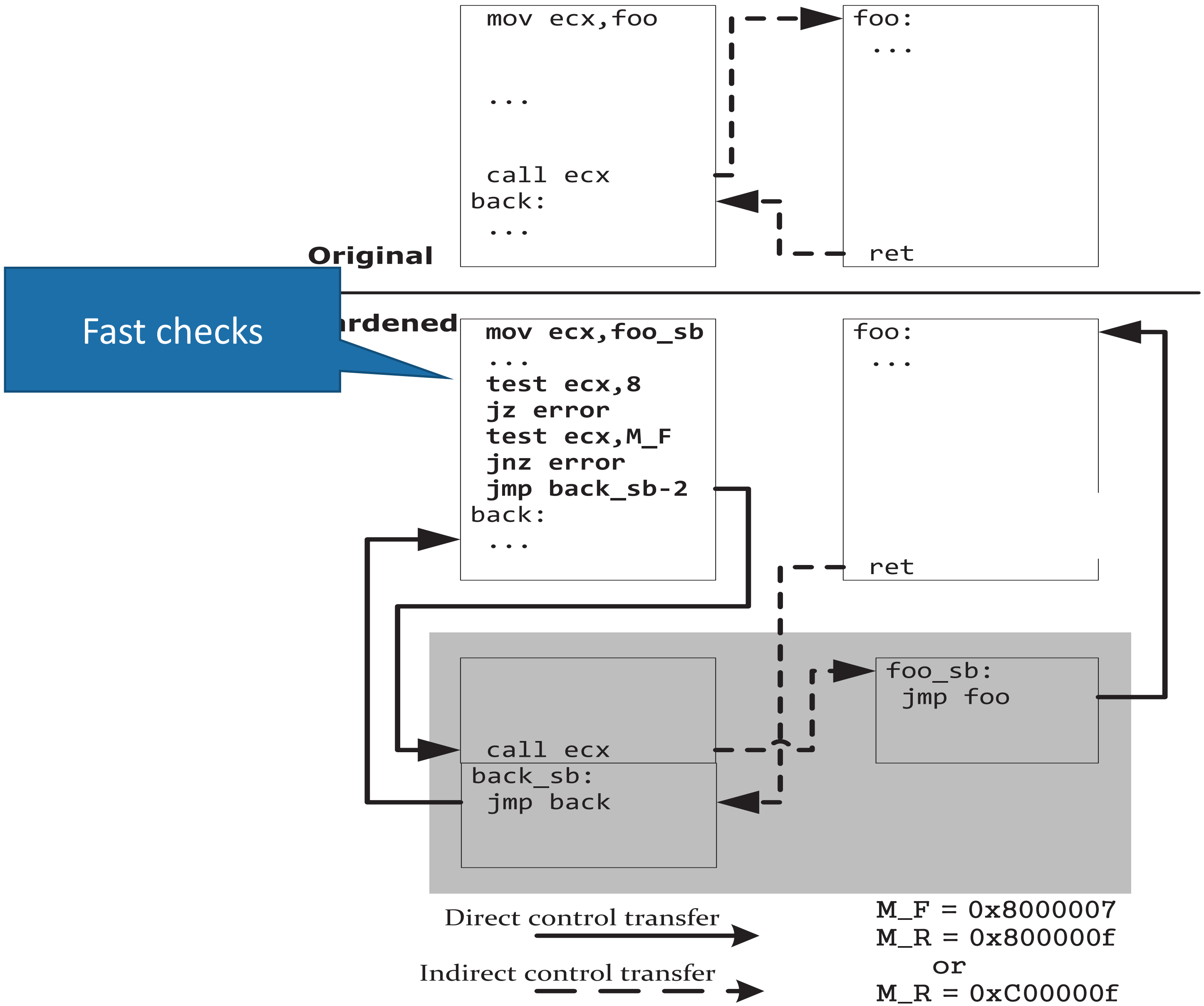
128MB segments

Function stu address
AND
M_F = 0x8000007

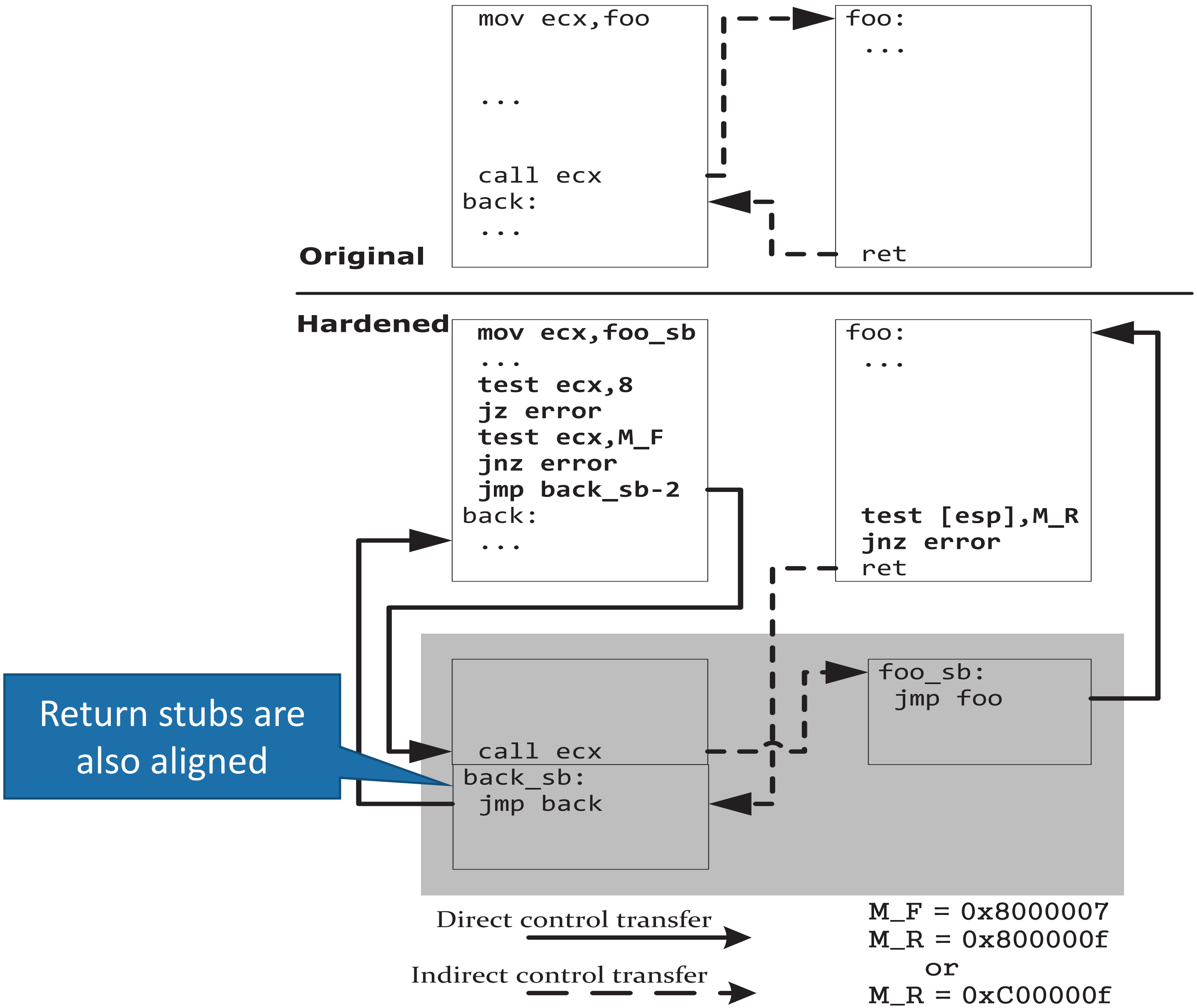
8-byte aligned slots

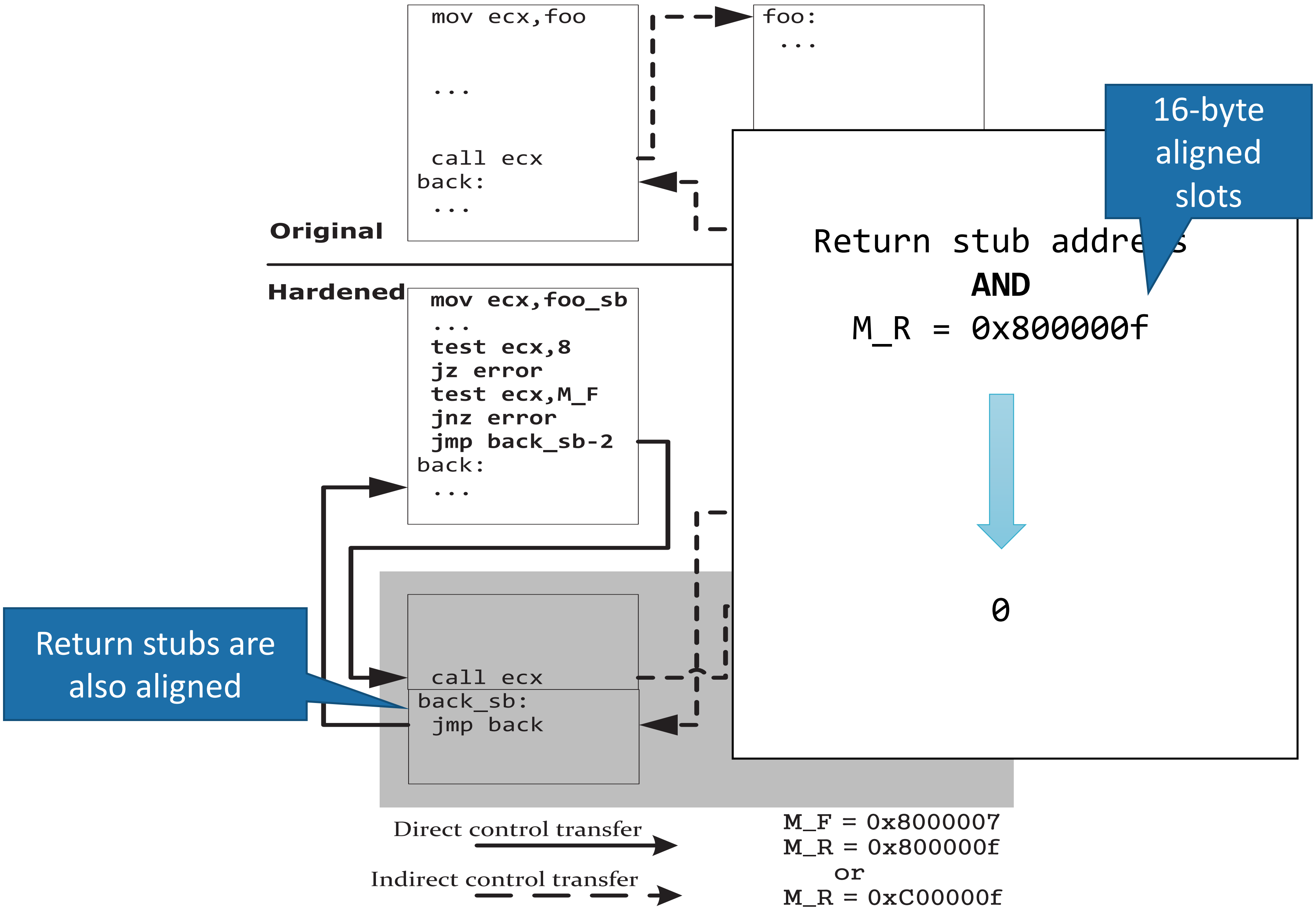
0



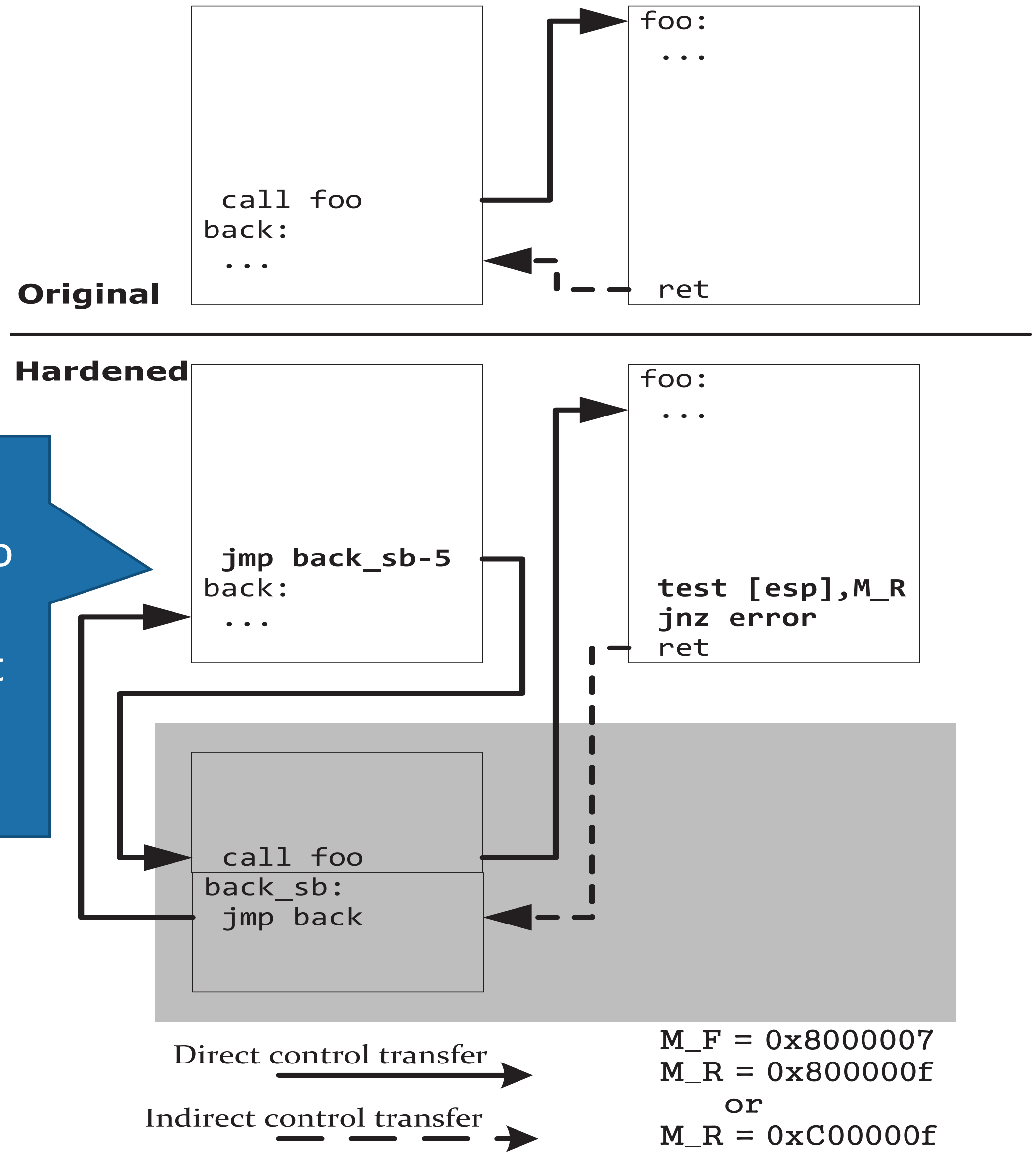


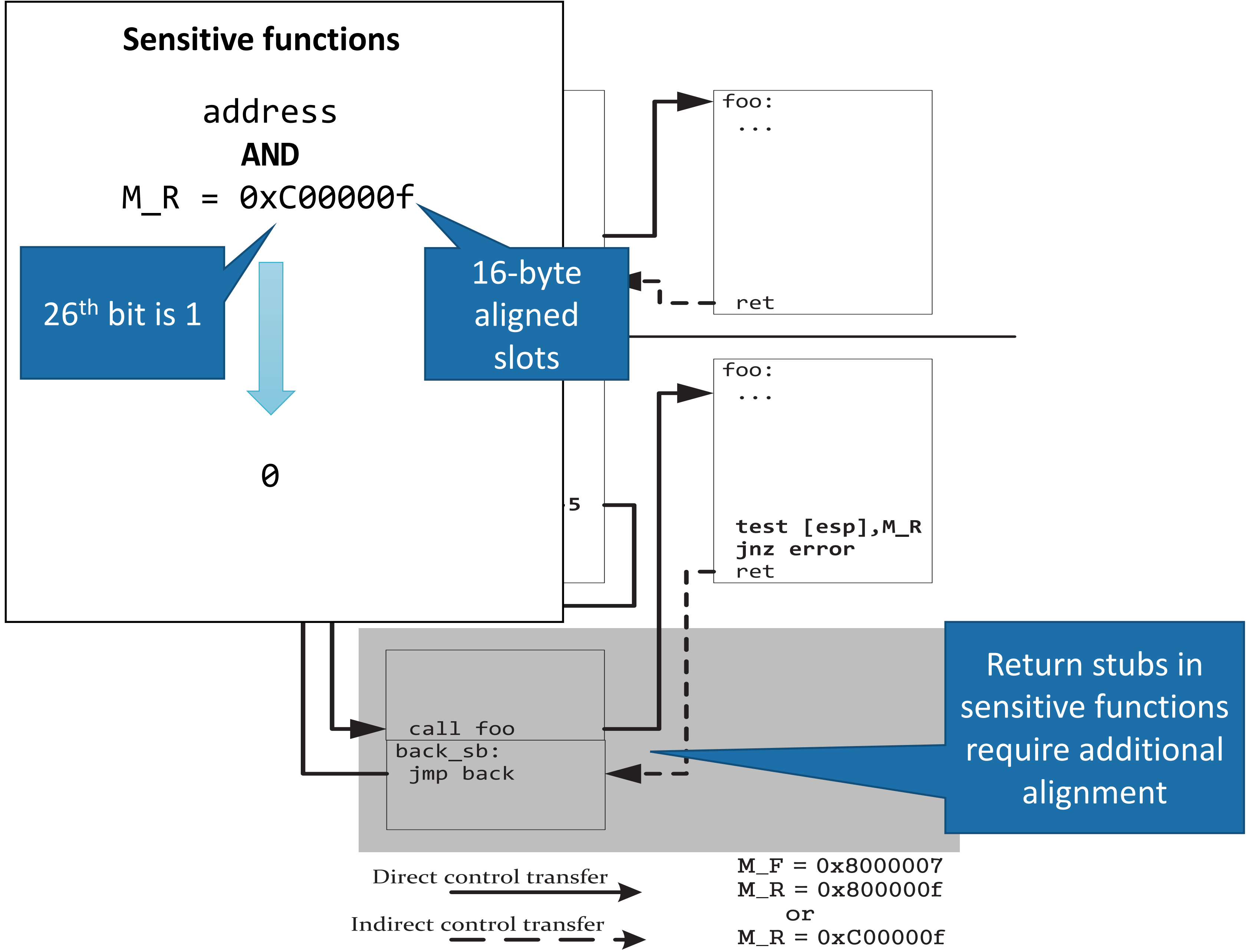
mov ecx,foo
...
call ecx
back:
...
foo:
...
ret
Original
Fast checks
rdened
mov ecx,foo_sb
...
test ecx,8
jz error
test ecx,M_F
jnz error
jmp back_sb-2
back:
...
foo:
...
ret
call ecx
back_sb:
jmp back
foo_sb:
jmp foo
Direct control transfer
Indirect control transfer
M_F = 0x8000007
M_R = 0x800000f
or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



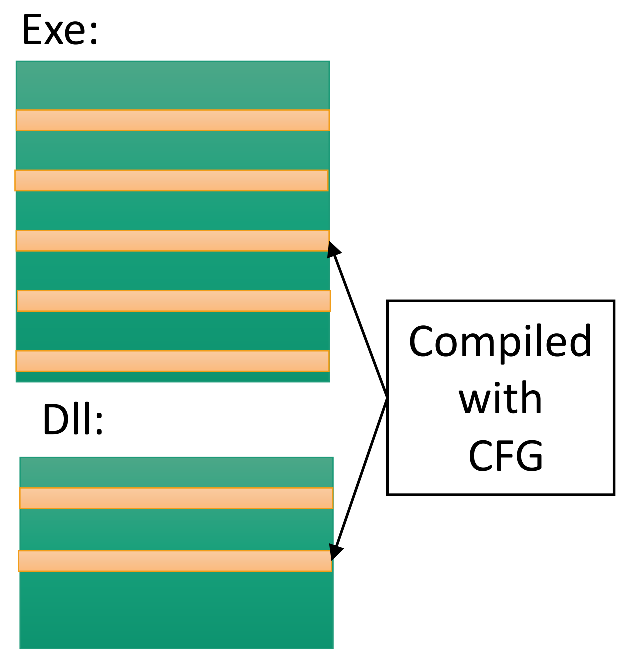

Sensitive functions
address
AND
M_R = 0xC00000f
26th bit is 1
16-byte aligned slots
0
foo:
...
ret
foo:
...
test [esp],M_R
jnz error
ret
call foo
back_sb:
jmp back
Return stubs in sensitive functions require additional alignment
Direct control transfer
Indirect control transfer
M_F = 0x8000007
M_R =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

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

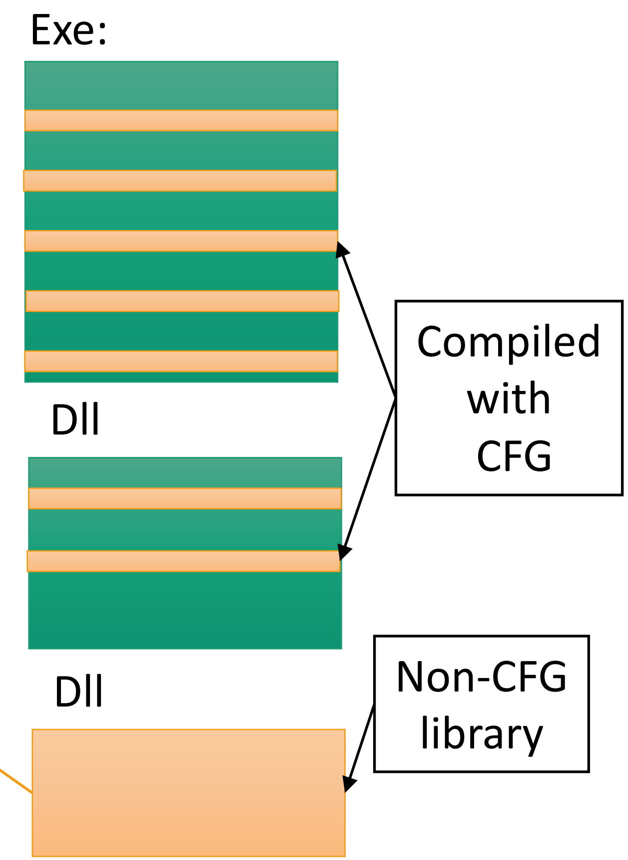
Inserts control-flow checks before indirect calls during compilation

A bitmap marks the allowed targets

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

bitmap:

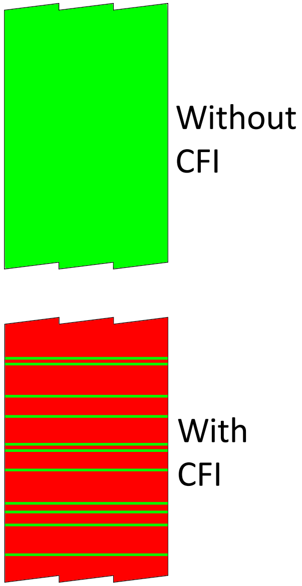
1 bit per 8 or 16-byte slot



Reachable Targets Under CFI

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

Targetable locations in code pages:



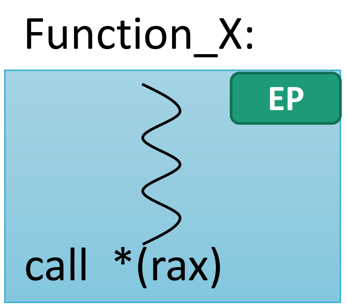
What is Left

Call Sites (**CS**)

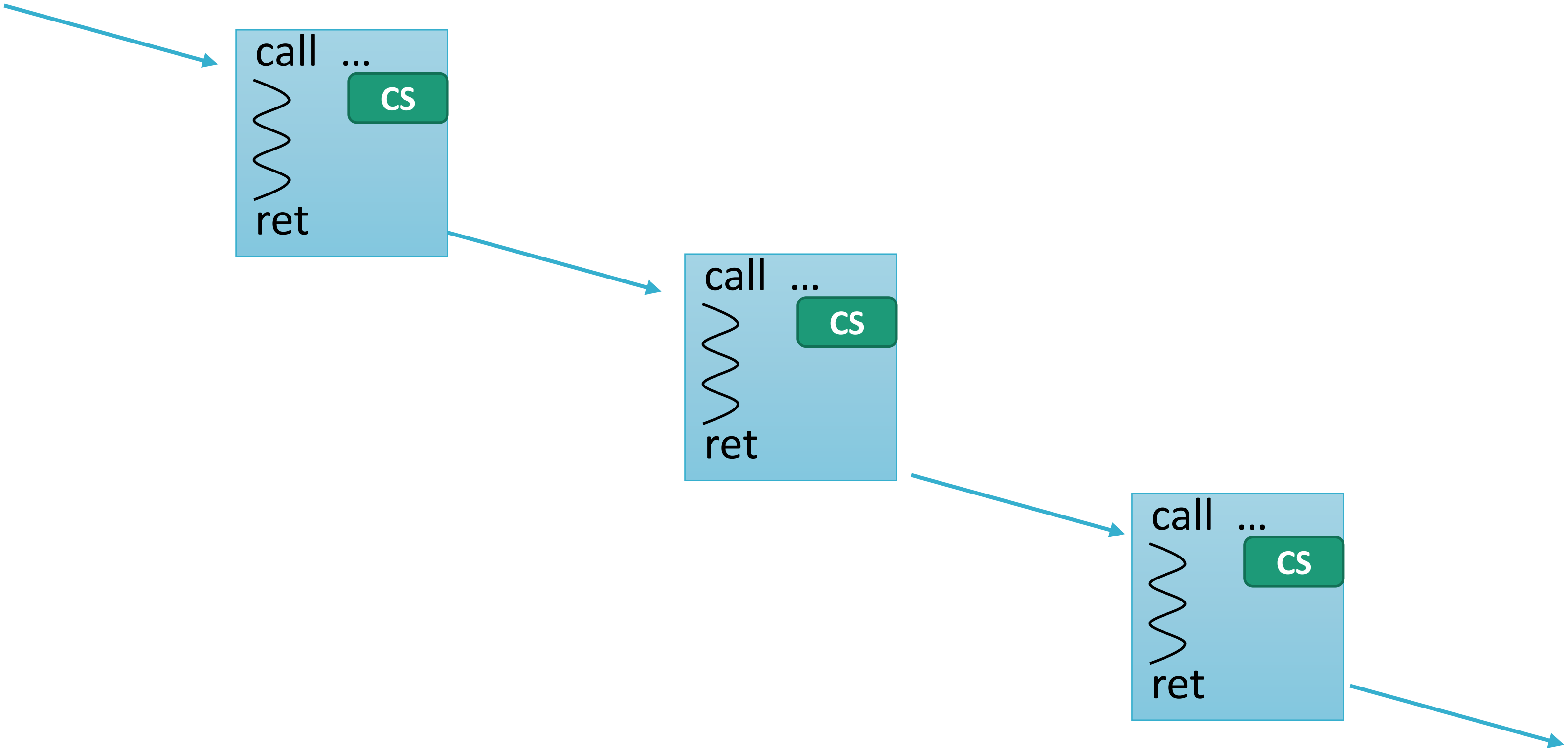
- Targetable by **return** instructions
- CS gadgets
- Return Oriented Programming (ROP)

Function Entry Points (**EP)**

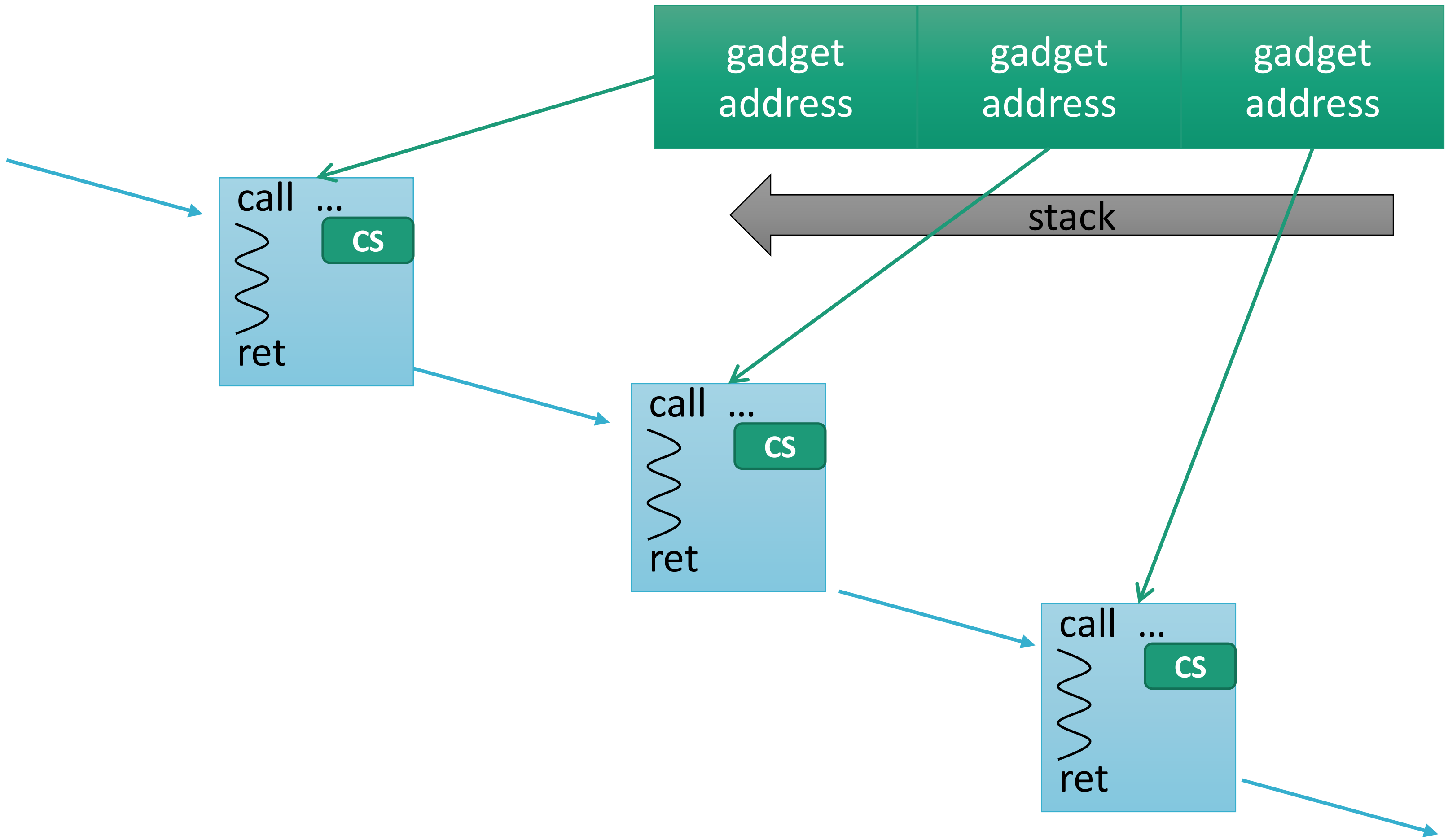
- Targetable by **indirect call** and **indirect jump** instructions
- EP gadgets
- Call Oriented Programming (COP)



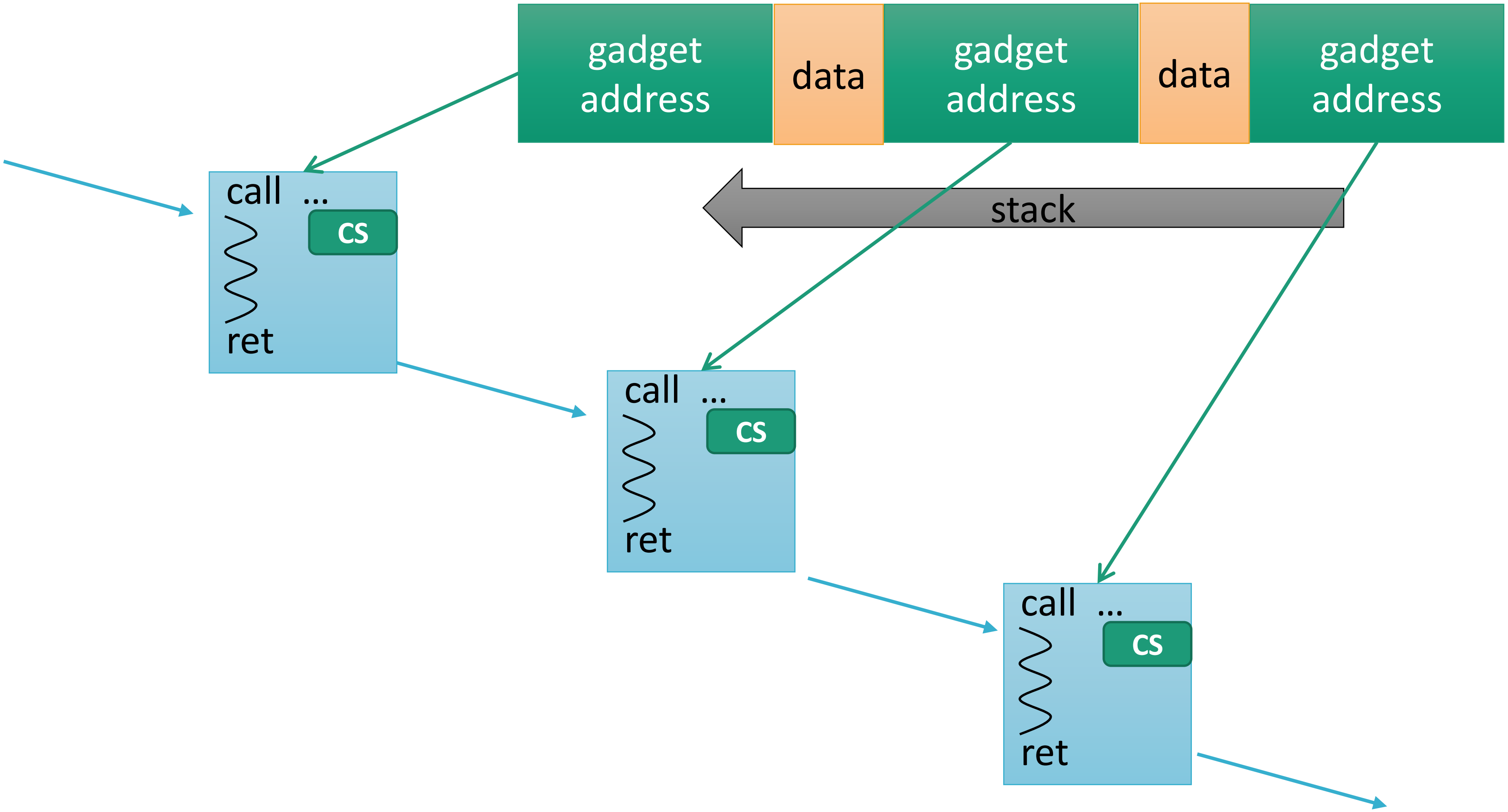
CS gadgets: Linking



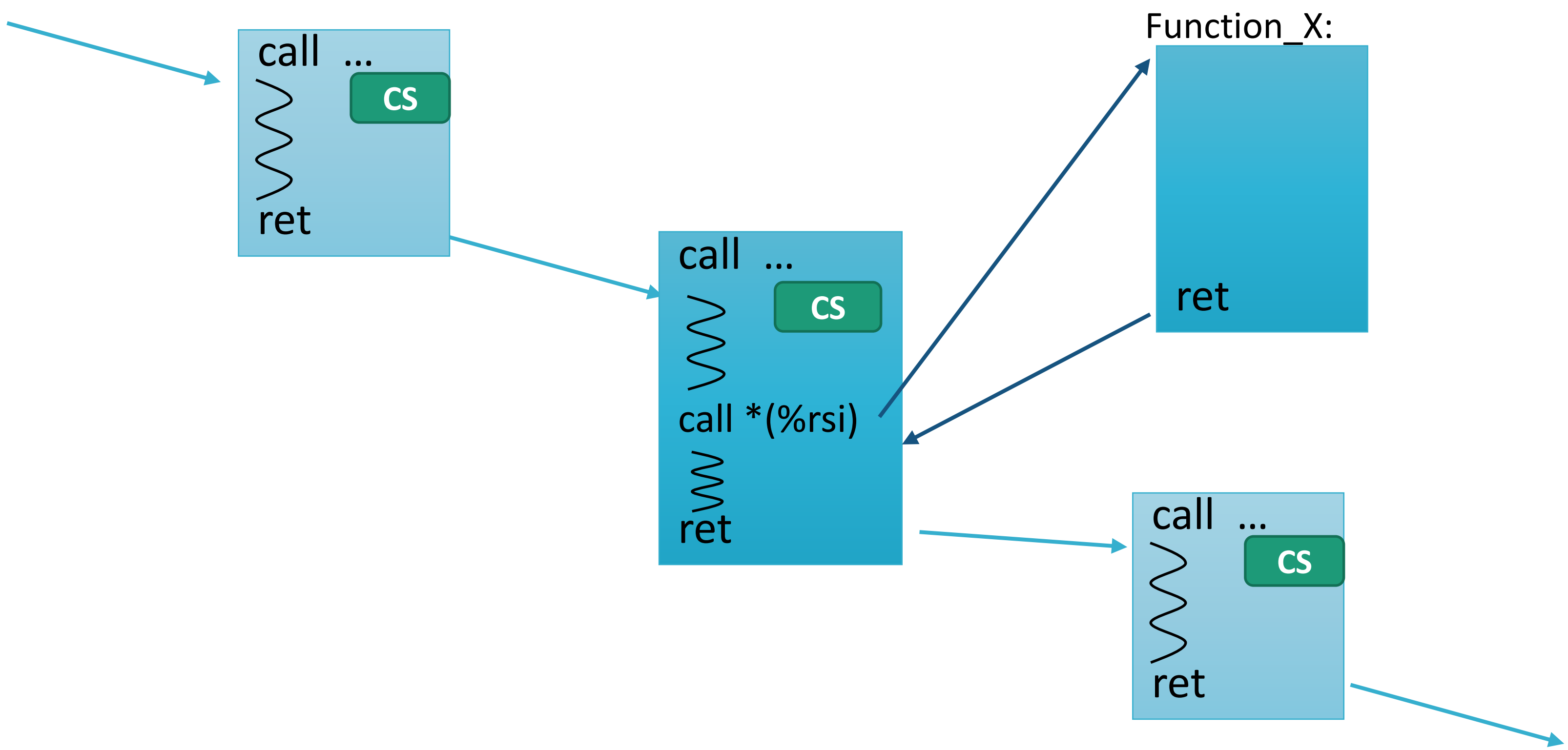
CS gadgets: Linking



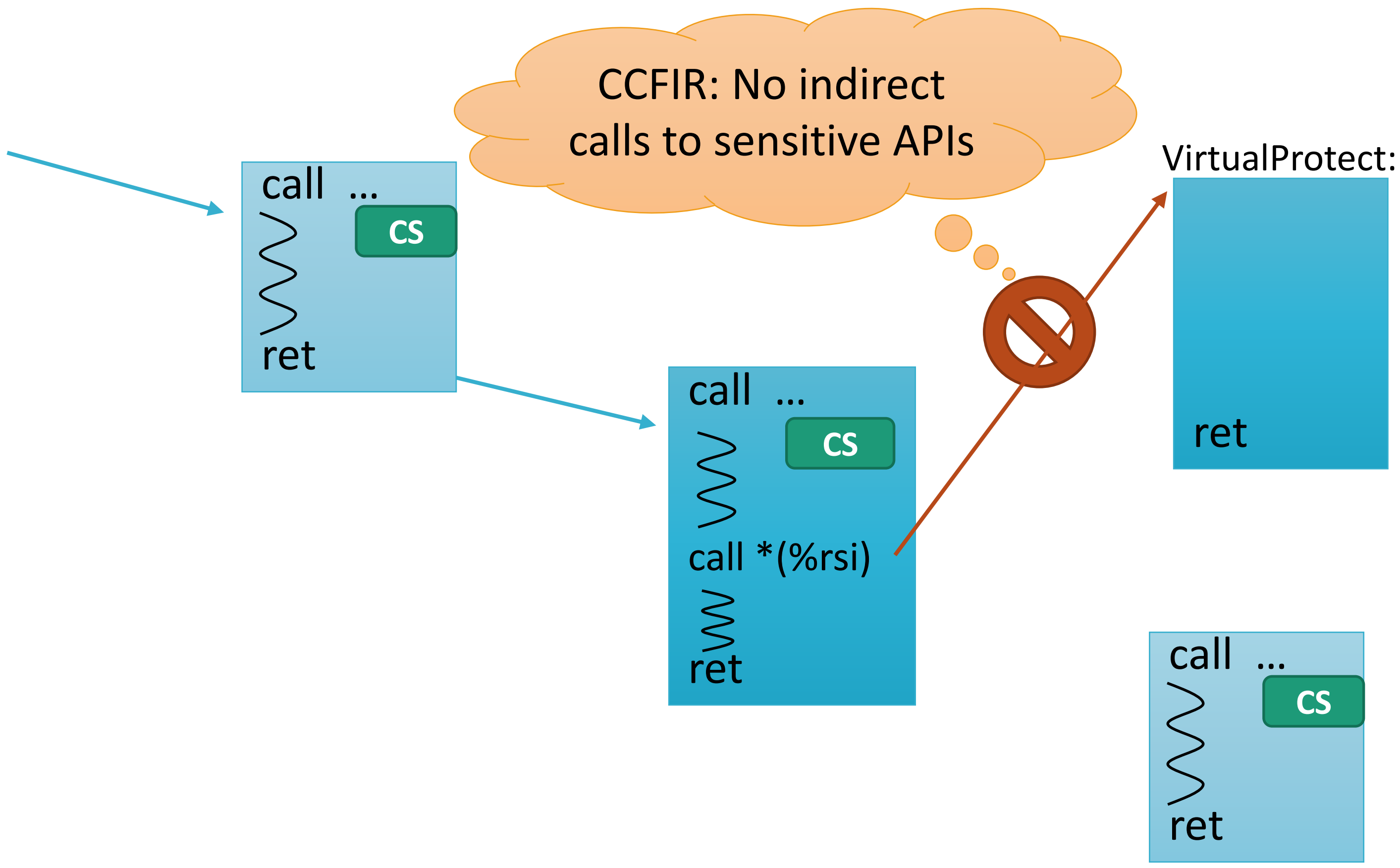
CS gadgets: Linking



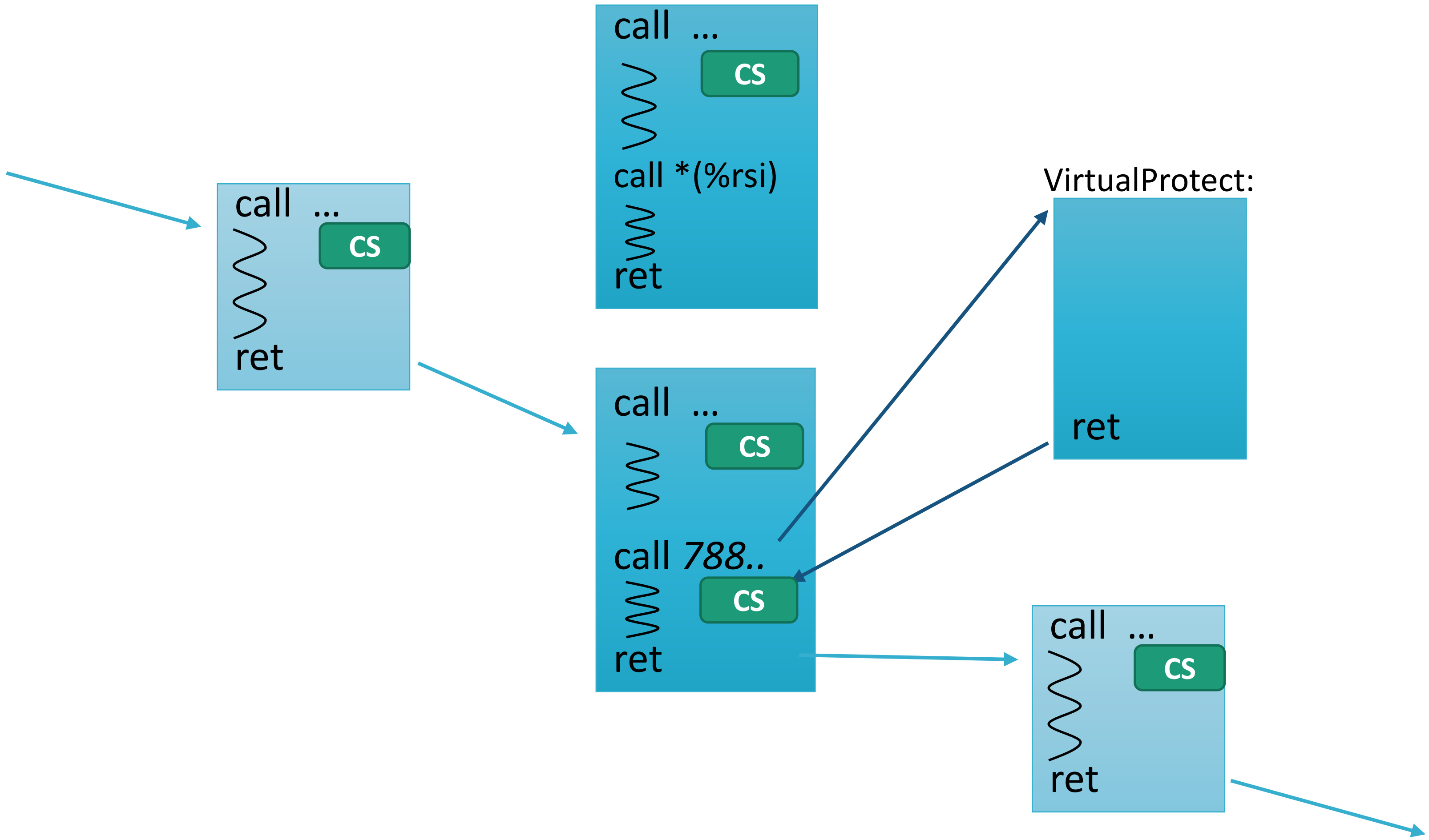
CS gadgets: Calling Functions



CS gadgets: Calling Sensitive Functions

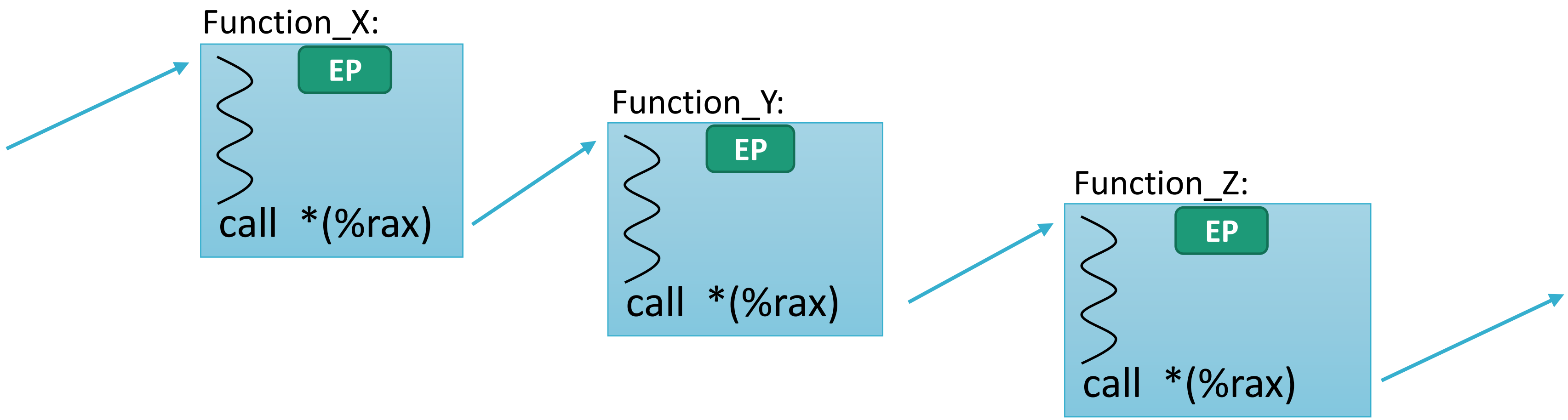


CS gadgets: Calling Sensitive Functions

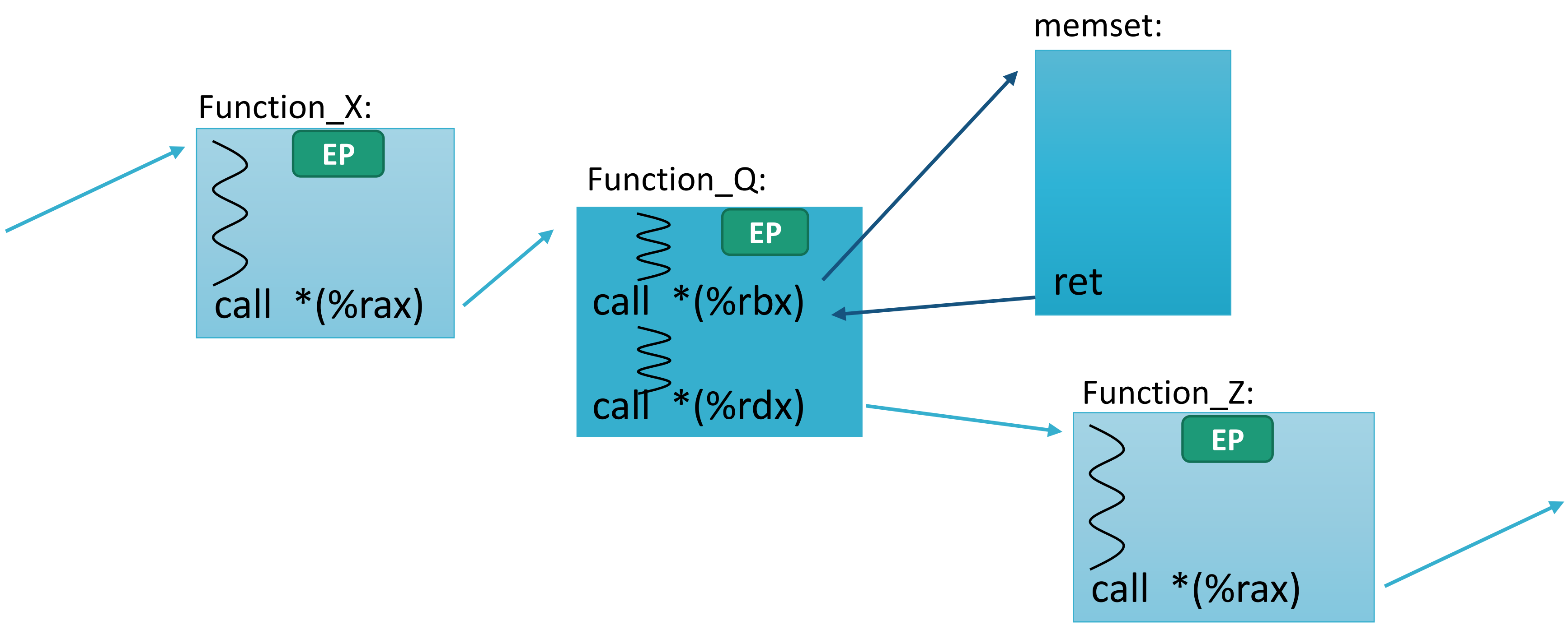


EP gadgets: Linking

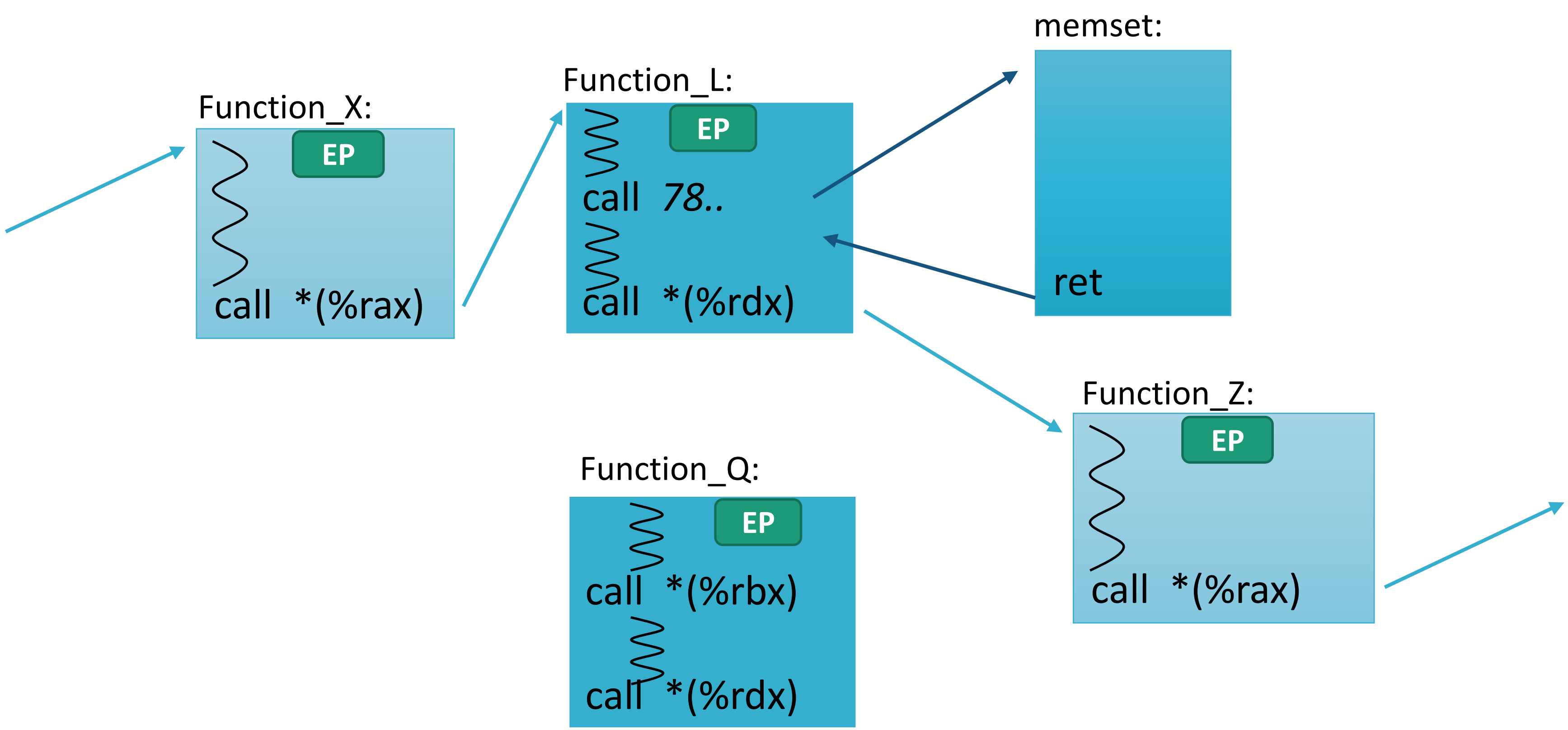
Chaining is significantly harder



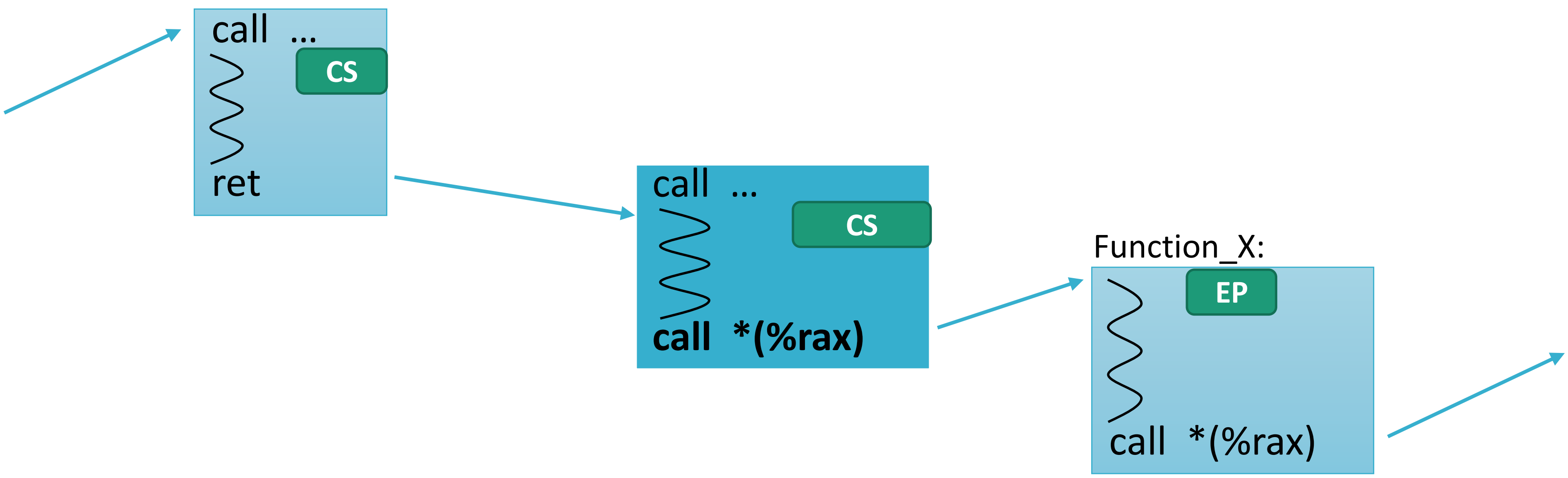
EP gadgets: Calling Functions



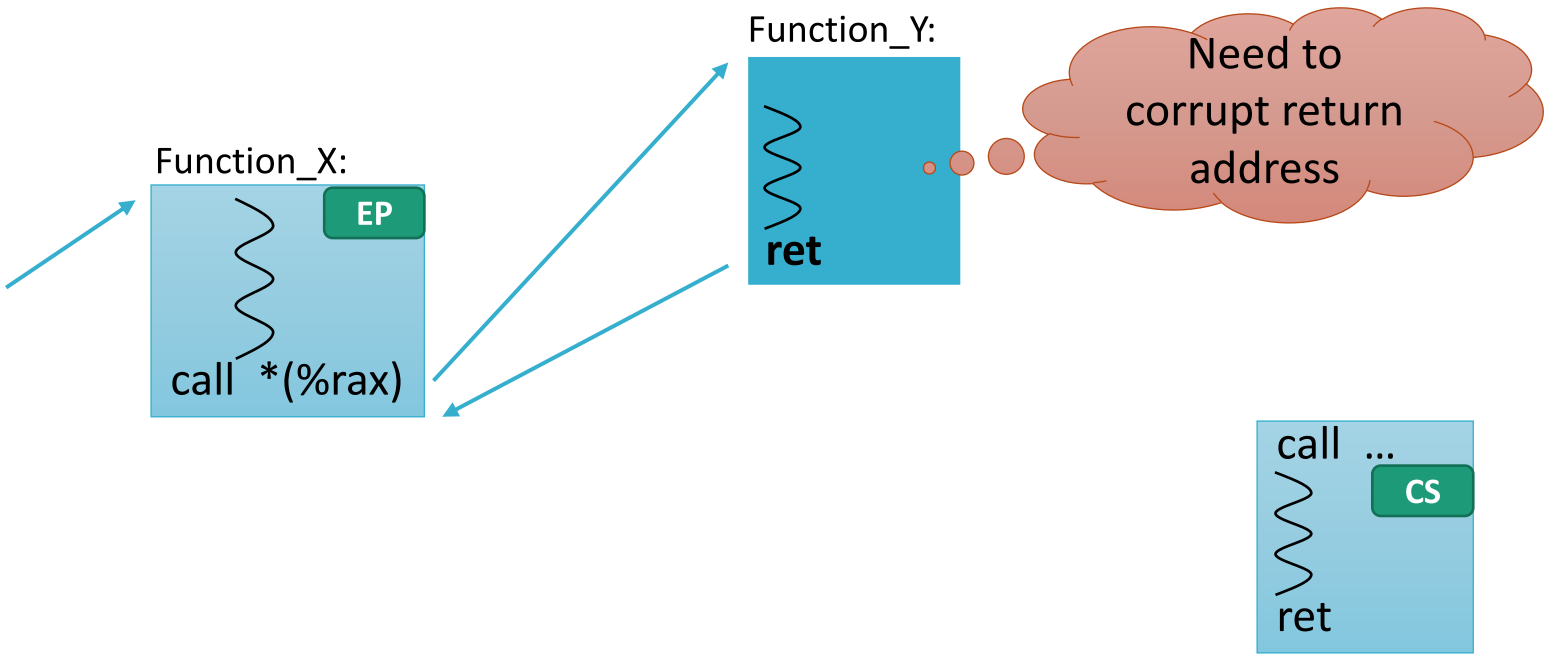
EP gadgets: Calling Functions



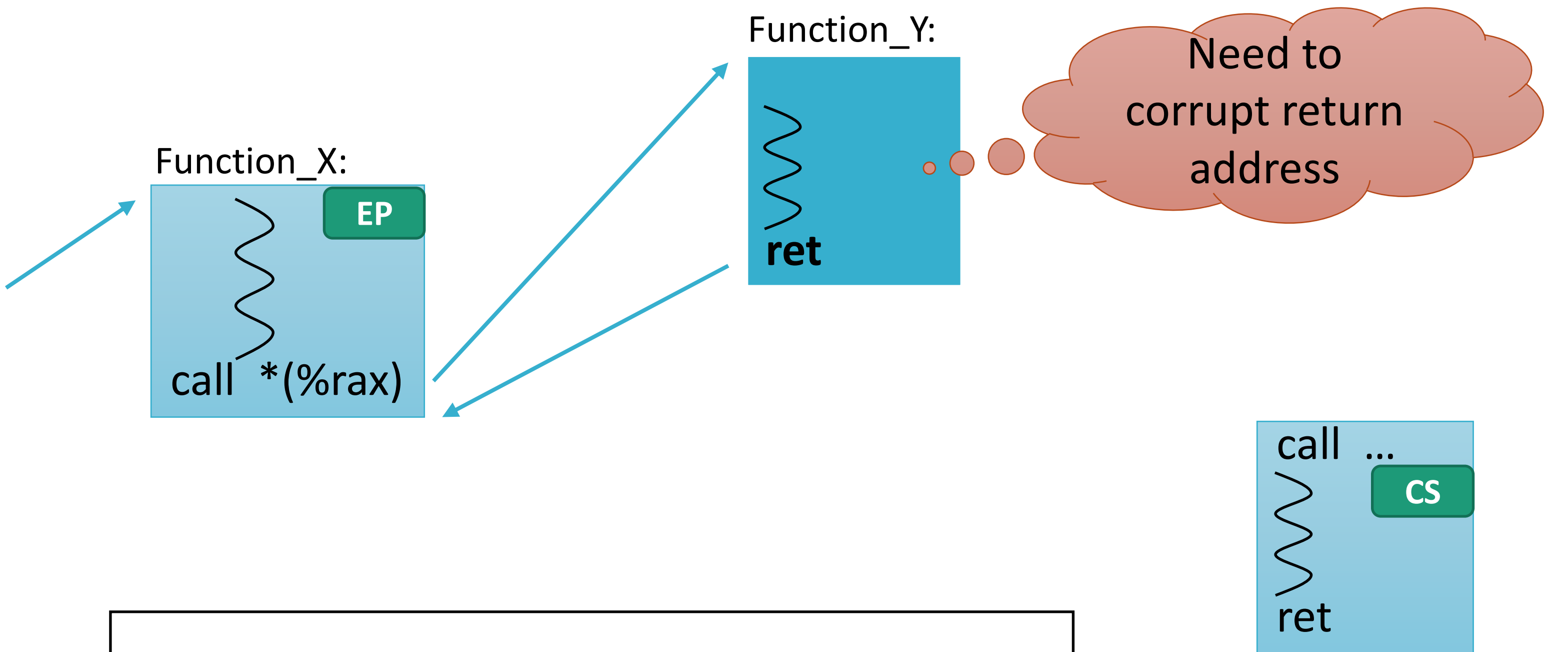
Switch Control: CS → EP



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.cs.stevens.edu/~gportoka/files/outofcontrol_oakland14.pdf

Exploiting **Internet Explorer 8**

- Vulnerability: Heap Overflow (CVE-2012-1876)
- More info about vulnerability @ http://www.vupen.com/blog

Assume **ASLR / DEP / CCFIR** in place

First controlled indirect branch instruction: **`jmp edx`**

(EP → CS) + VirtualProtect + memcpy = Code Injection

Compromising Coarse-grained CFI is Possible

https://www.cs.stevens.edu/~gportoka/files/outofcontrol_oakland14.pdf

Exploiting **Internet Explorer 8**

- Vulnerability: Heap Overflow (CVE-2012-1876)
- More info about vulnerability @ http://www.vupen.com/blog

Assume **ASLR / DEP / CCFIR** in place

First controlled indirect br `jmp edx`

(EP → CS) + VirtualProtec de Injection

Finer-Grained CFI

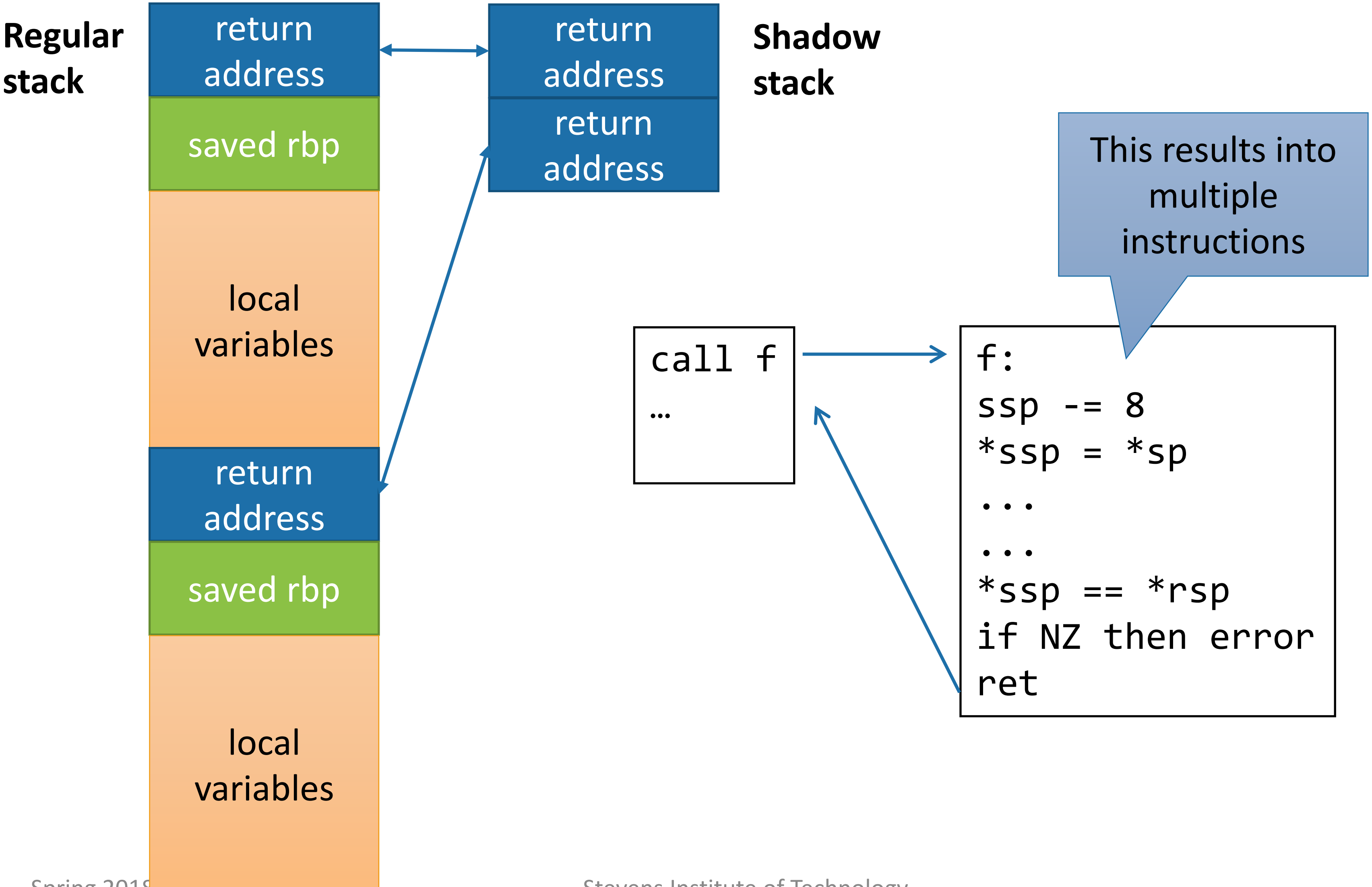
Various approaches to improve CFI

- More accurate CFG and more checks
- Only allow calls to target the functions they actually were intended to
  - **Better forward-edge CFI**

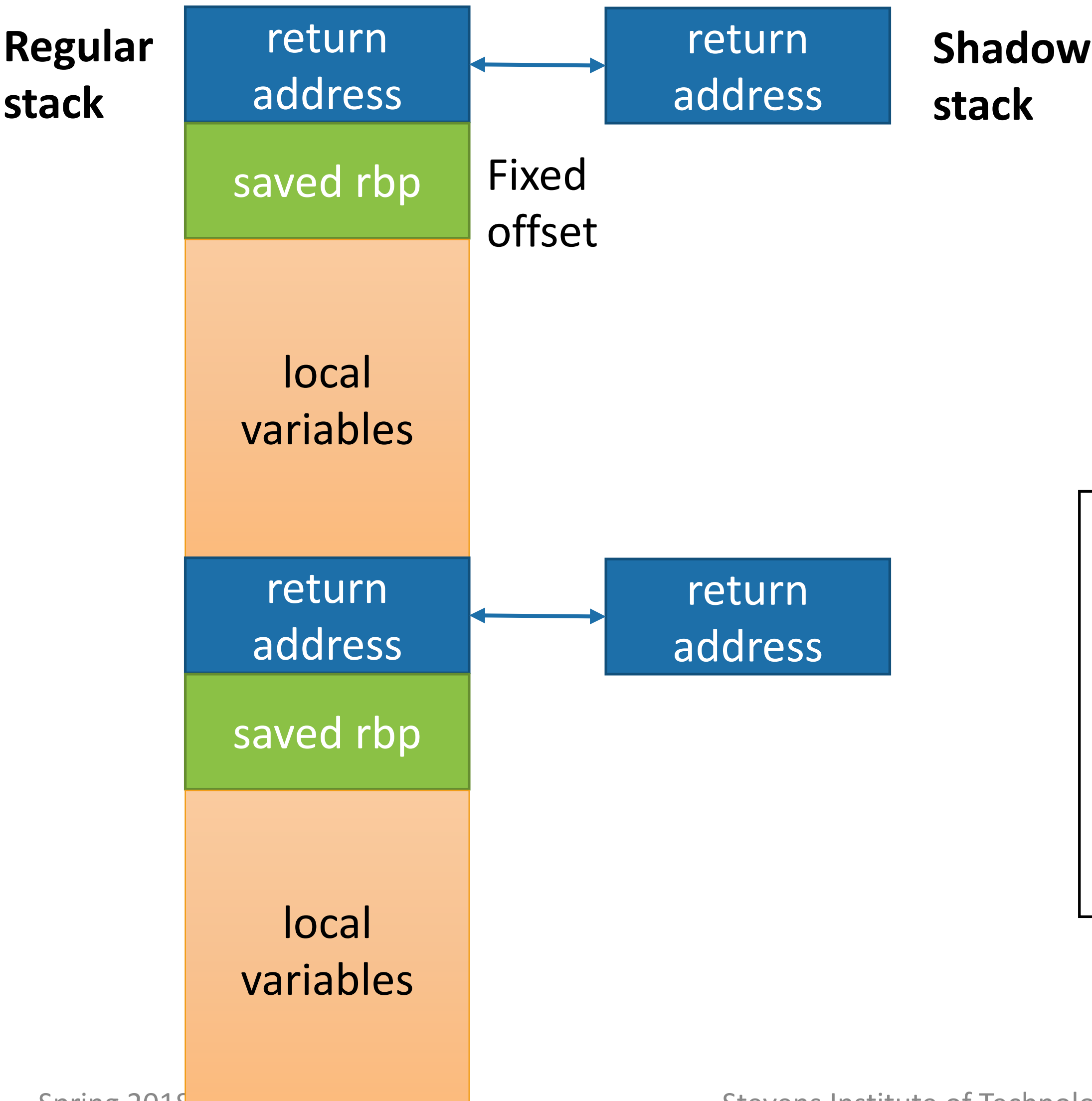
Context-sensitive control flow enforcement

- For example, a function should return to its caller not any caller

Shadow Stacks



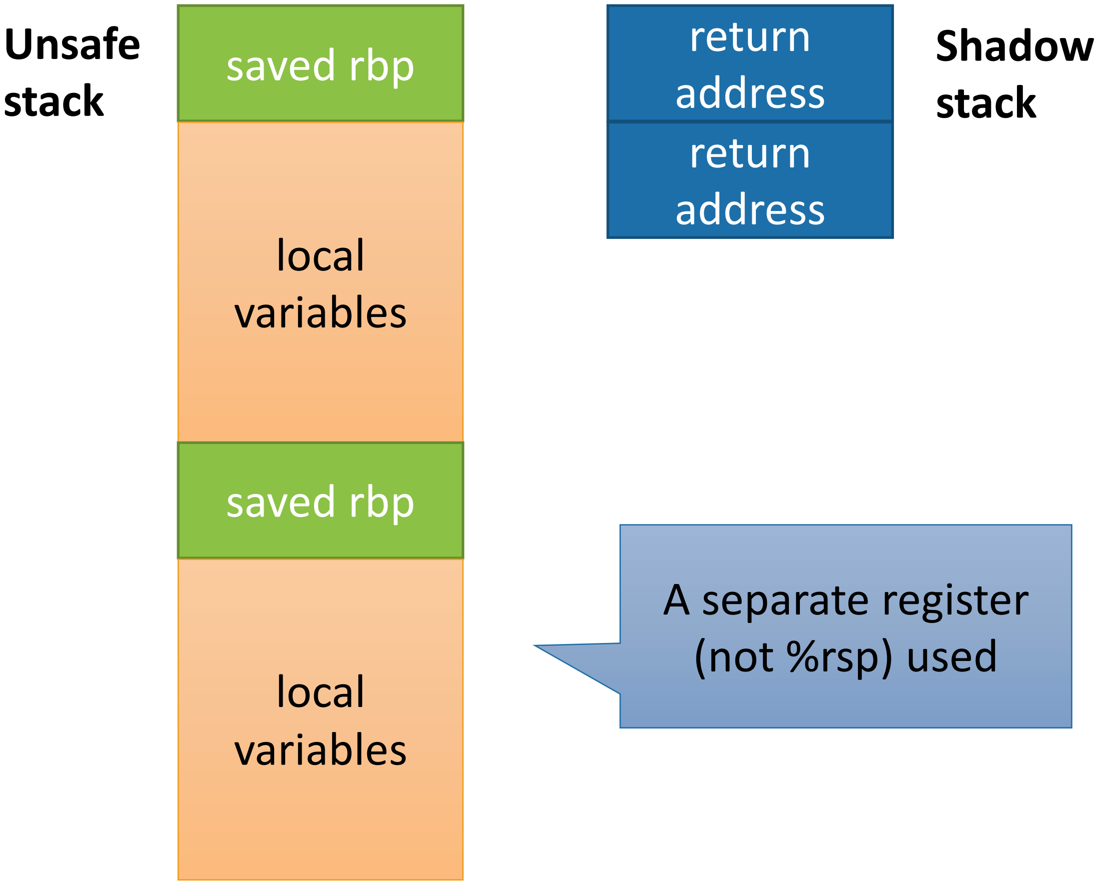
Shadow Stacks



This results into less instructions

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

Shadow vs Unsafe Stacks



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

Intel announced that hardware support for shadow stacks and CFI (called control-flow enforcement) will be made available on their future CPUs

- http://www.theregister.co.uk/2016/06/10/intel_control_flow_enforcement/

Heuristics-based Approaches

kBouncer: Efficient and Transparent ROP Mitigation

- Vassilis Pappas et al. [Usenix Security '13]
- Winner of Microsoft's Blue hat prize

Use HW debugging feature to detect abnormal control-flow transfers

- Low overhead!

Last Branch Record (LBR)

CPU registers store last branches taken by the program

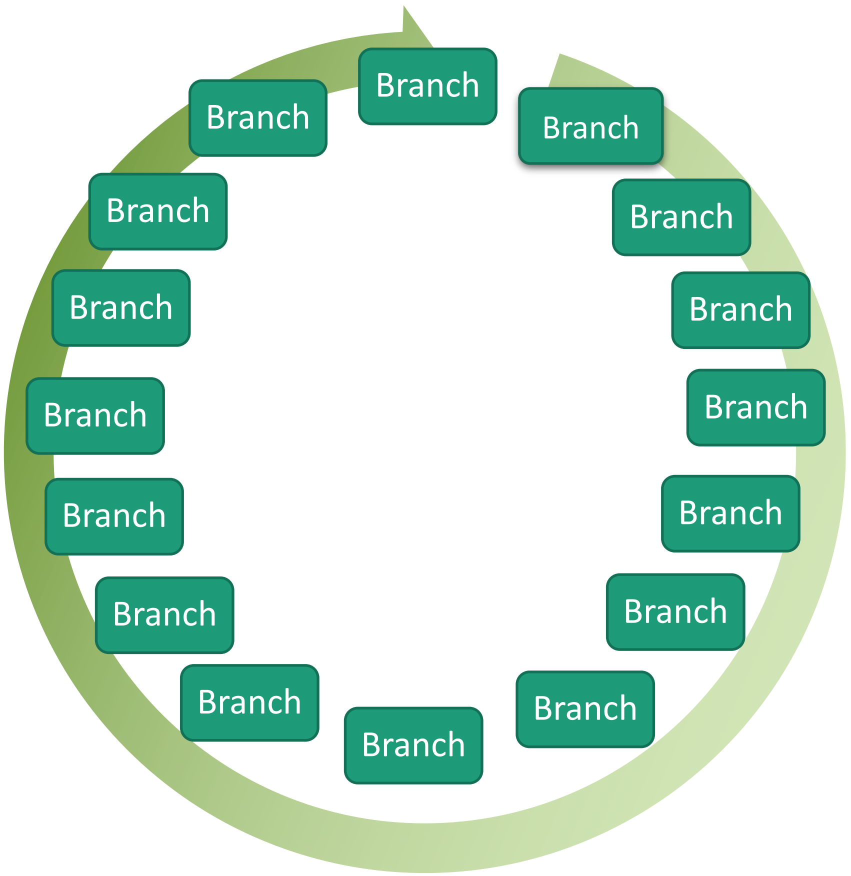
- Ring-buffer structure

Holds last 16 entries

- Store source:destination

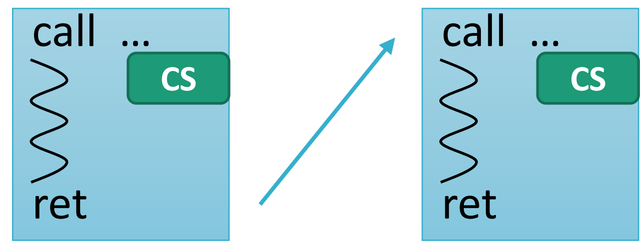
Configurable

- Example: Store only indirect calls

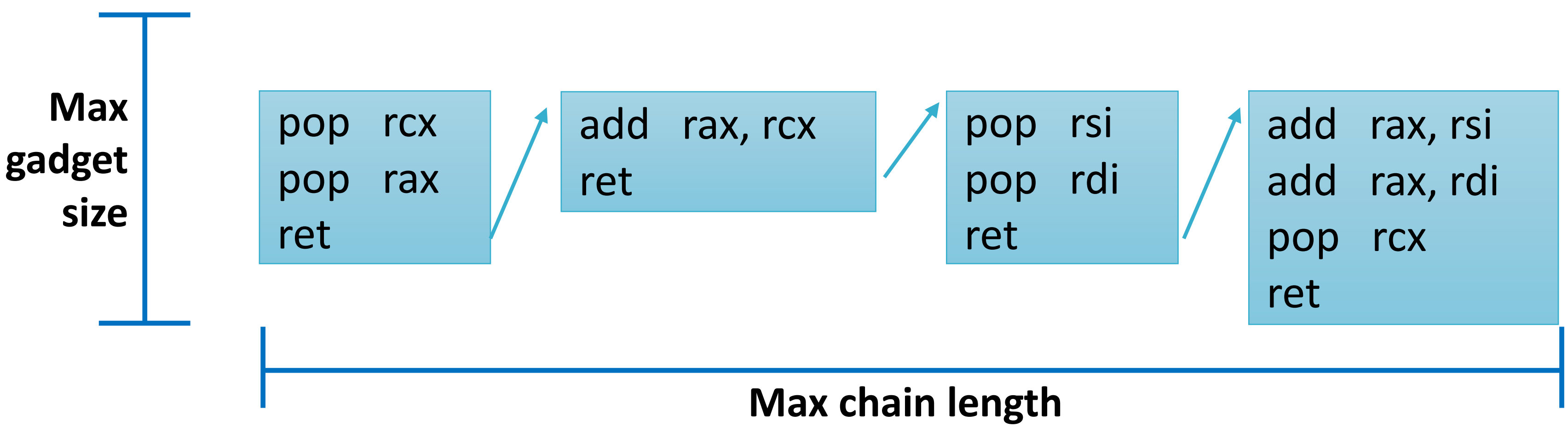


Detection Approach

1. Returns must target call sites



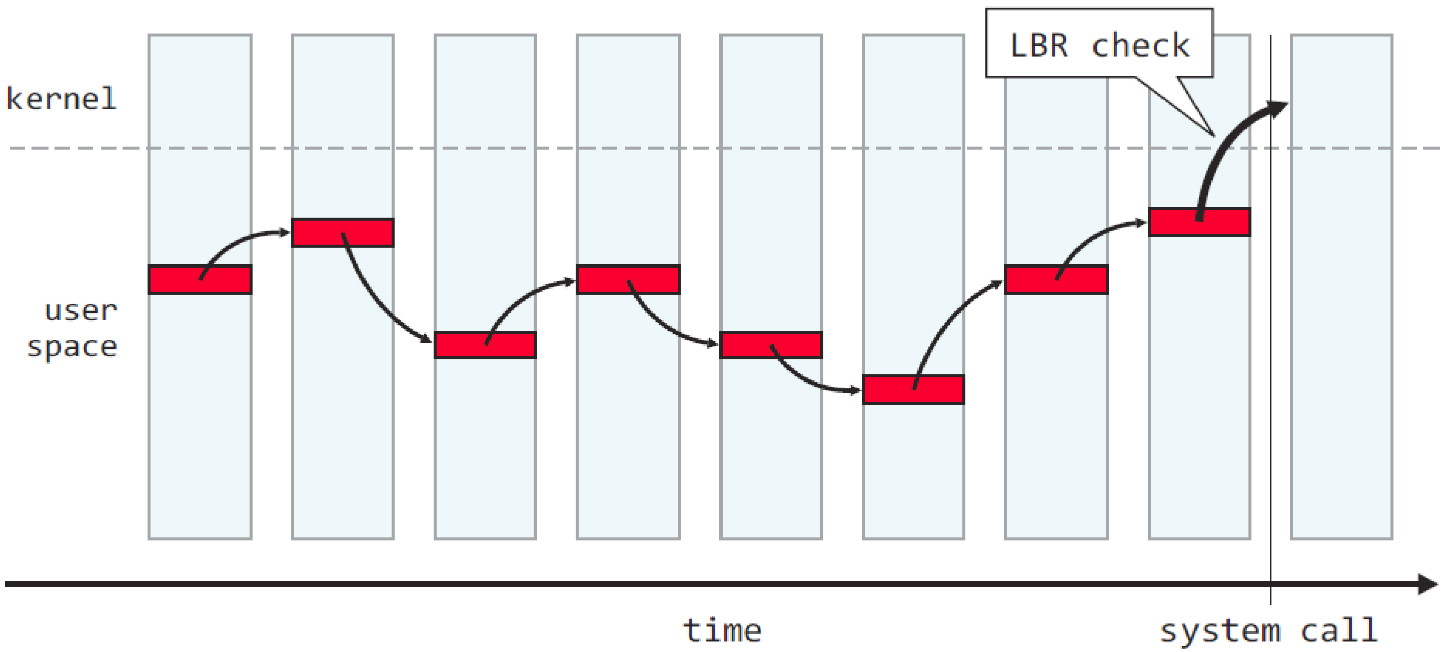
2. A limited number of small code fragments can be chained together



Fast Checks

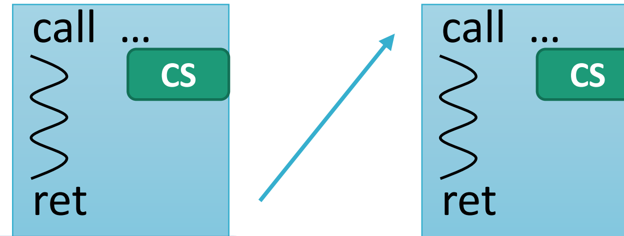
The payload will eventually interact with the OS through system calls

- Check for abnormal control transfers on system call entry



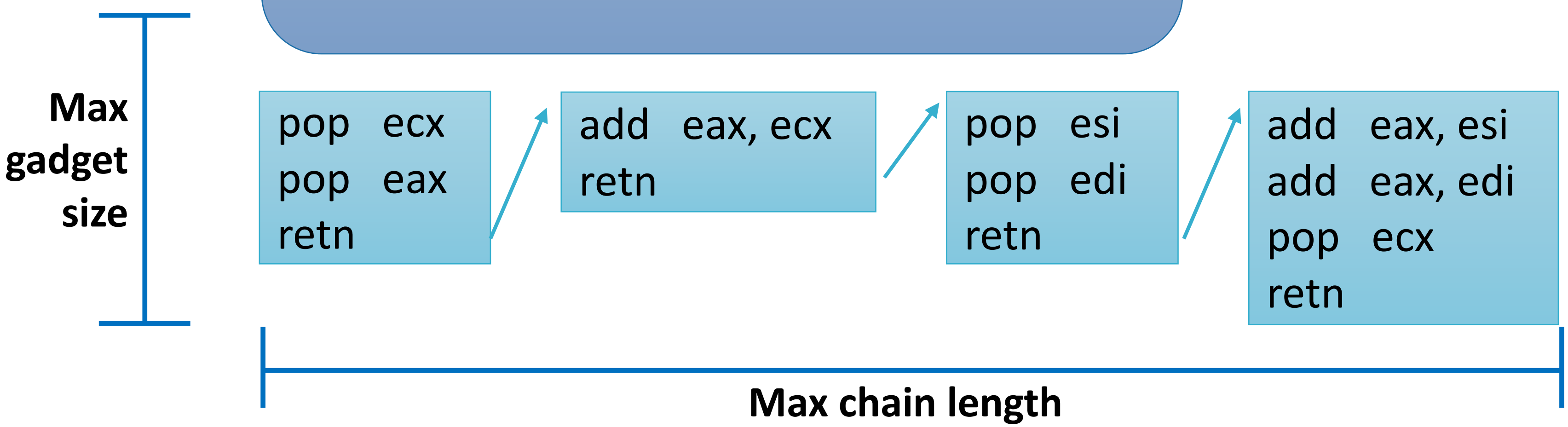
Detection Approach

1. Returns must target call sites



2. A limited can be
chained t

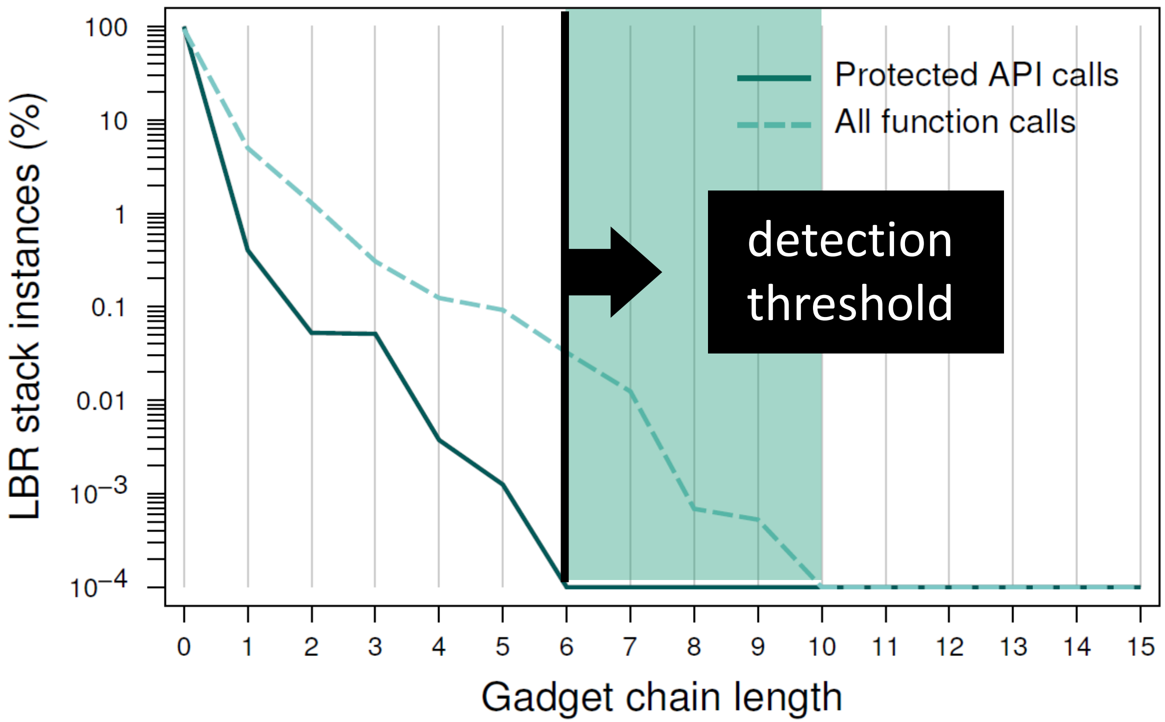
How can we establish the **max gadget size** and **max chain length?**



Establishing The Parameters

Set max gadget size to 19 (<20)

Evaluate max chain length **experimentally**



Dataset: Internet Explorer, Adobe Reader, Flash Player, Microsoft Office (Word, Excel, Powerpoint)

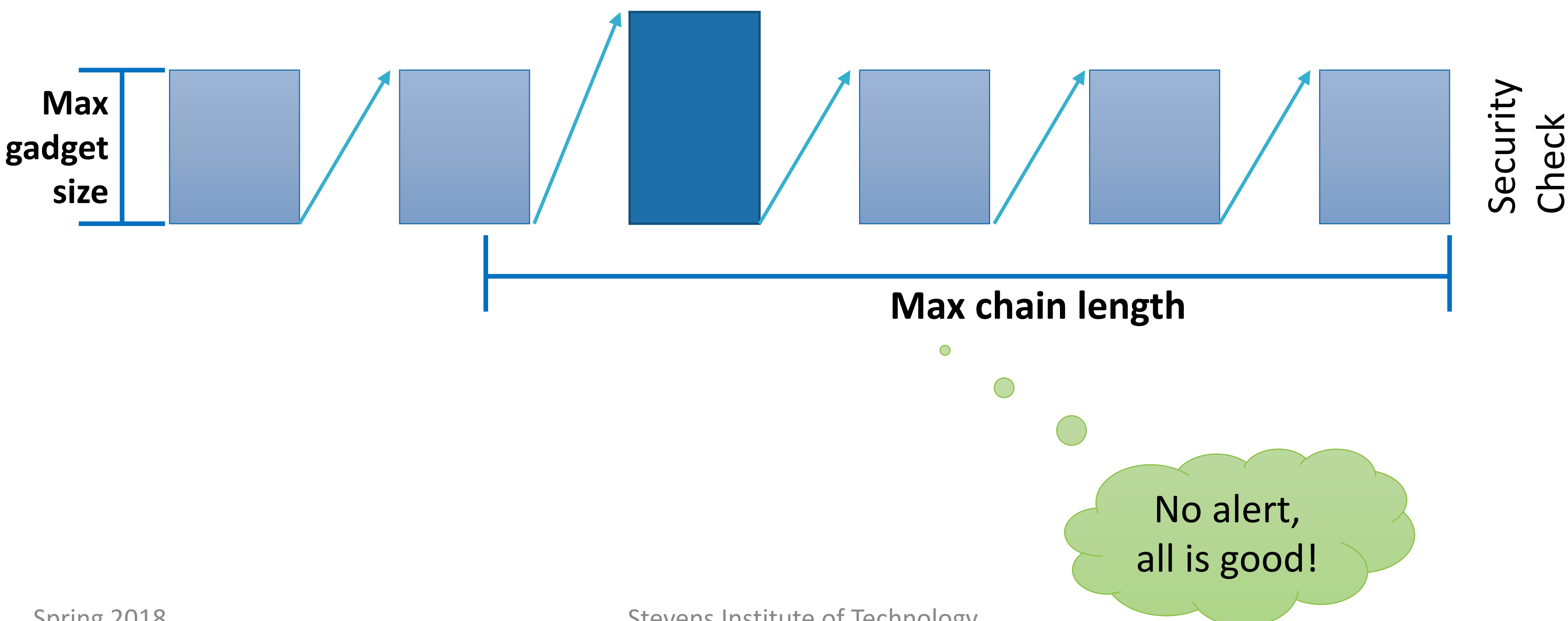
Chosen Parameters

Approach similar to kBouncer

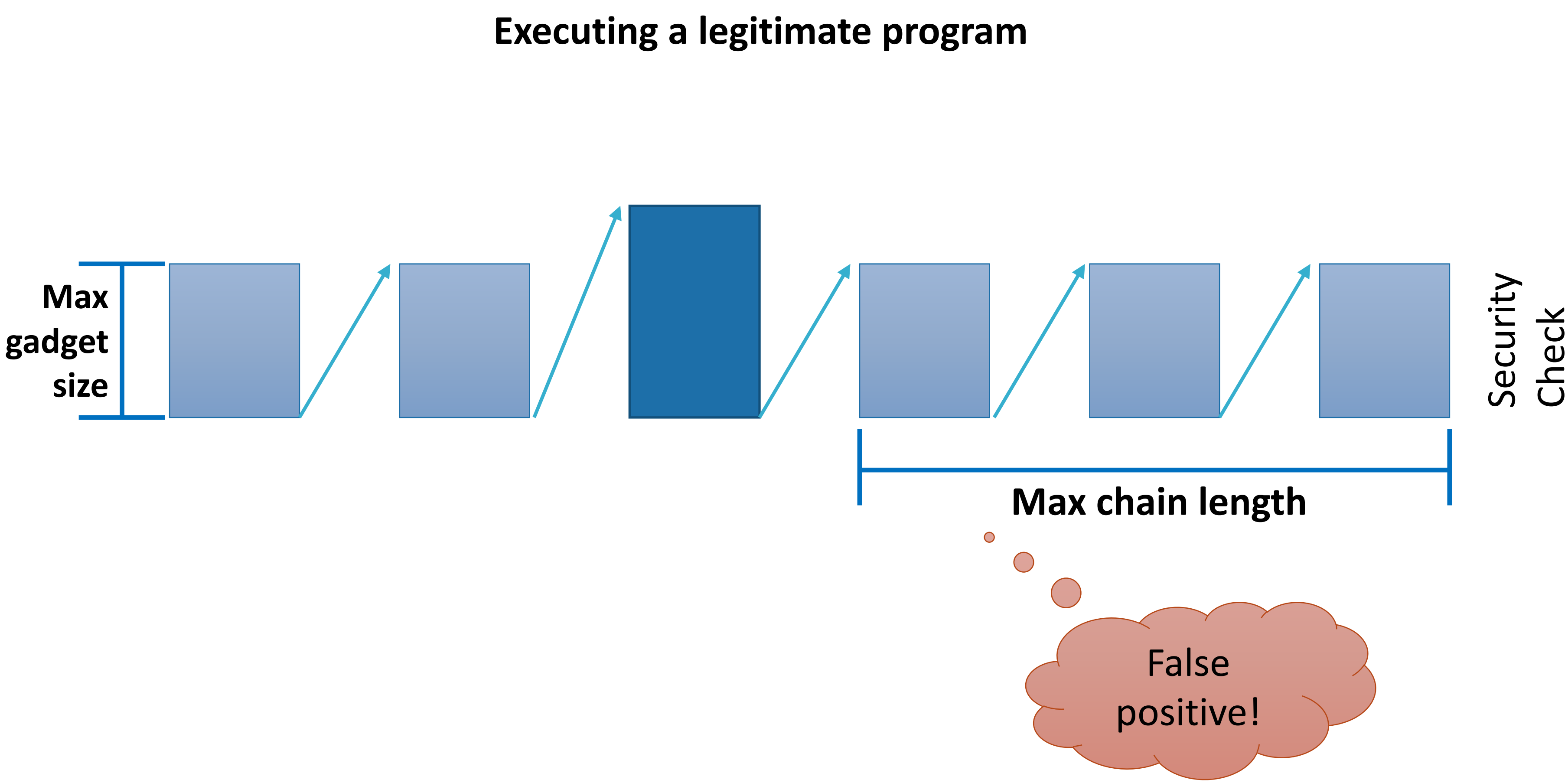
	kBouncer	**ROPecker**
Time-of-Check	Entry of Sensitive API	Entry of Sensitive API + during execution
Gadget Length	**20** instructions	**6** instructions
Inspect BH instances	Detected max "benign" gadget chain length: **5**	Detected max "benign" gadget chain length: **10**
Gadget Chain Length	**8** gadgets	**11** gadgets

Why Picking Parameters Is Hard

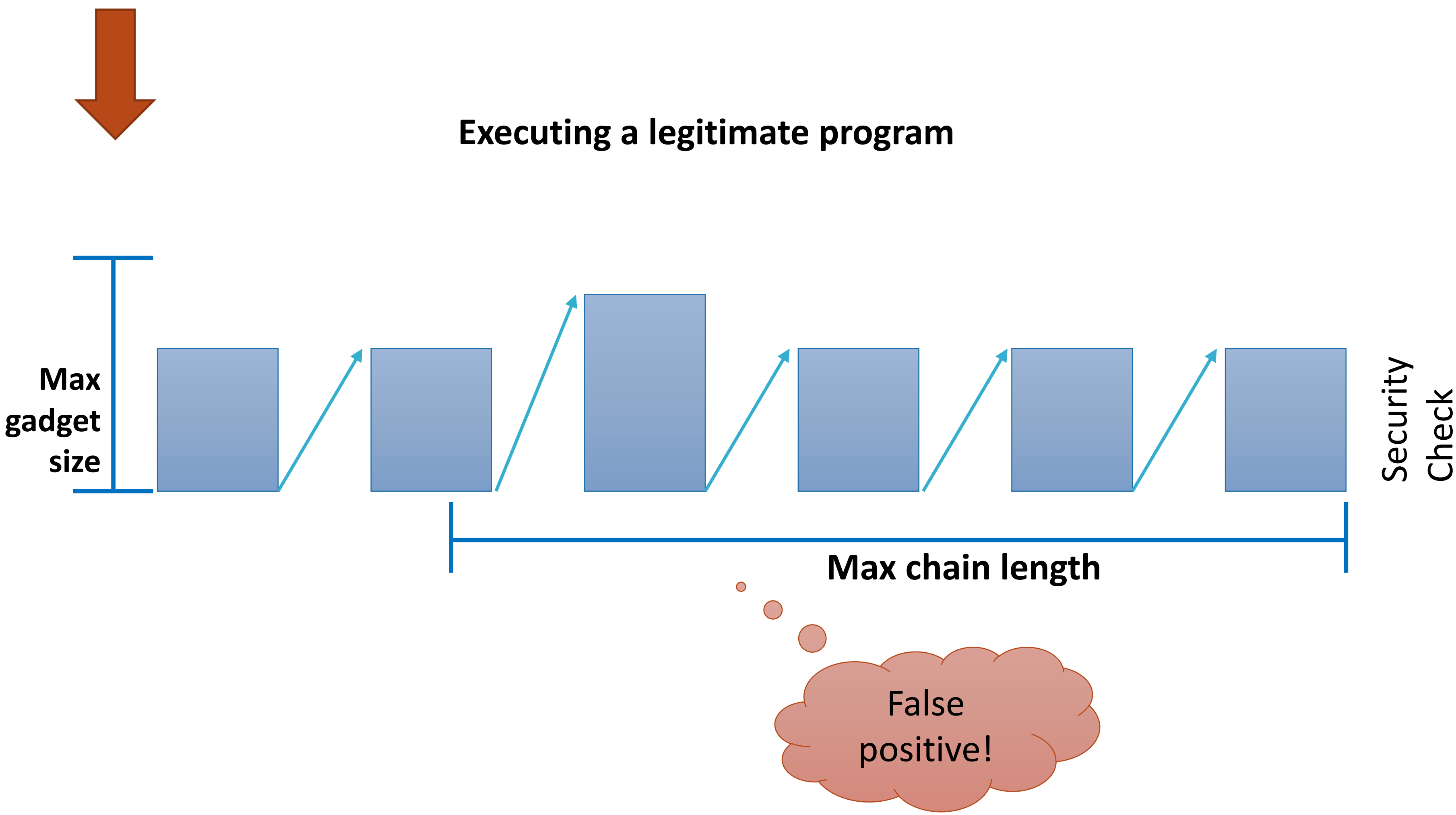
Executing a legitimate program



Why Picking Parameters Is Hard

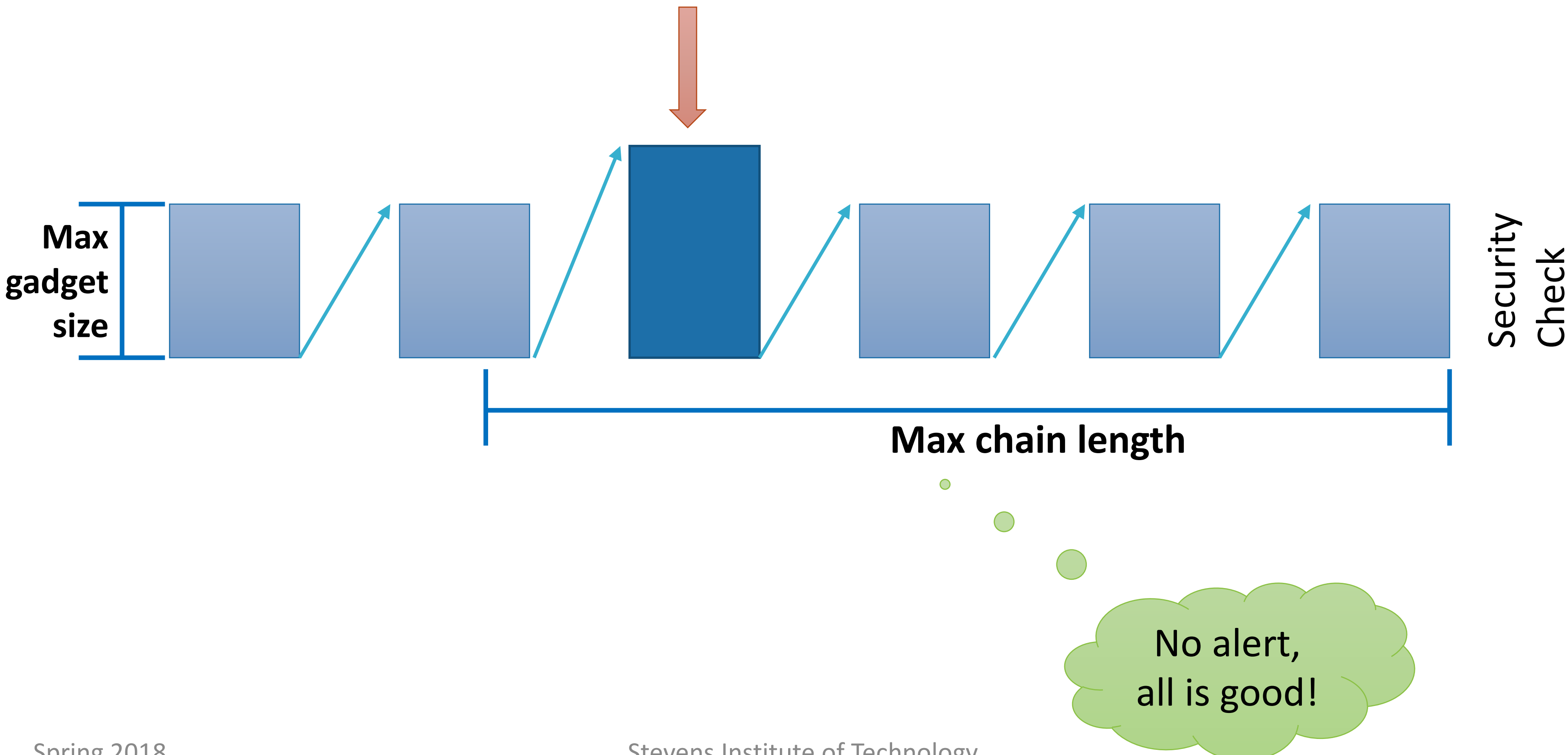


Why Picking Parameters Is Hard



How to Avoid Detection?

Interpose longer gadgets in the exploit



Using Long Gadgets

Long gadgets frequently:

- Use a high number of registers
- Leave used registers dirty at exit
- Require memory preparations to avoid crashing
- Have whacky code sequences

```
mov eax, ebx
mov ecx, edx
add esi, edi

mov esi, [0x1234]
cmp esi, 10
jg  X

mov ecx, 0x2321
div ecx
mov [eax], edi

mov ecx, 0x5678
and edi, ecx
xor eax, edi
retn
```

Such Defenses Are Also Vulnerable

http://www.cs.stevens.edu/~gportoka/files/sizematters_usenixsec14.pdf

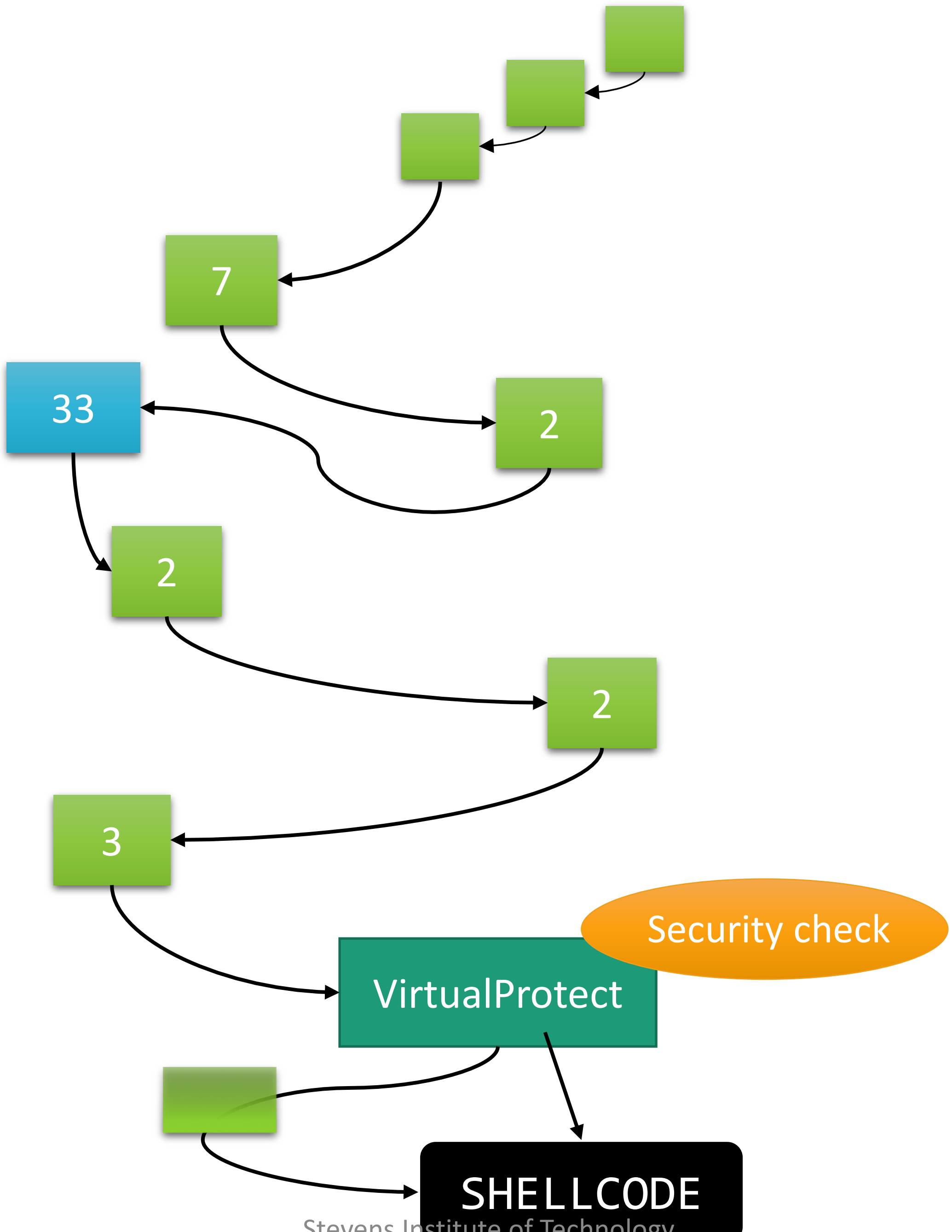
Exploiting **Internet Explorer 8** similar to CFI attack

Assumes **kBouncer** is in place

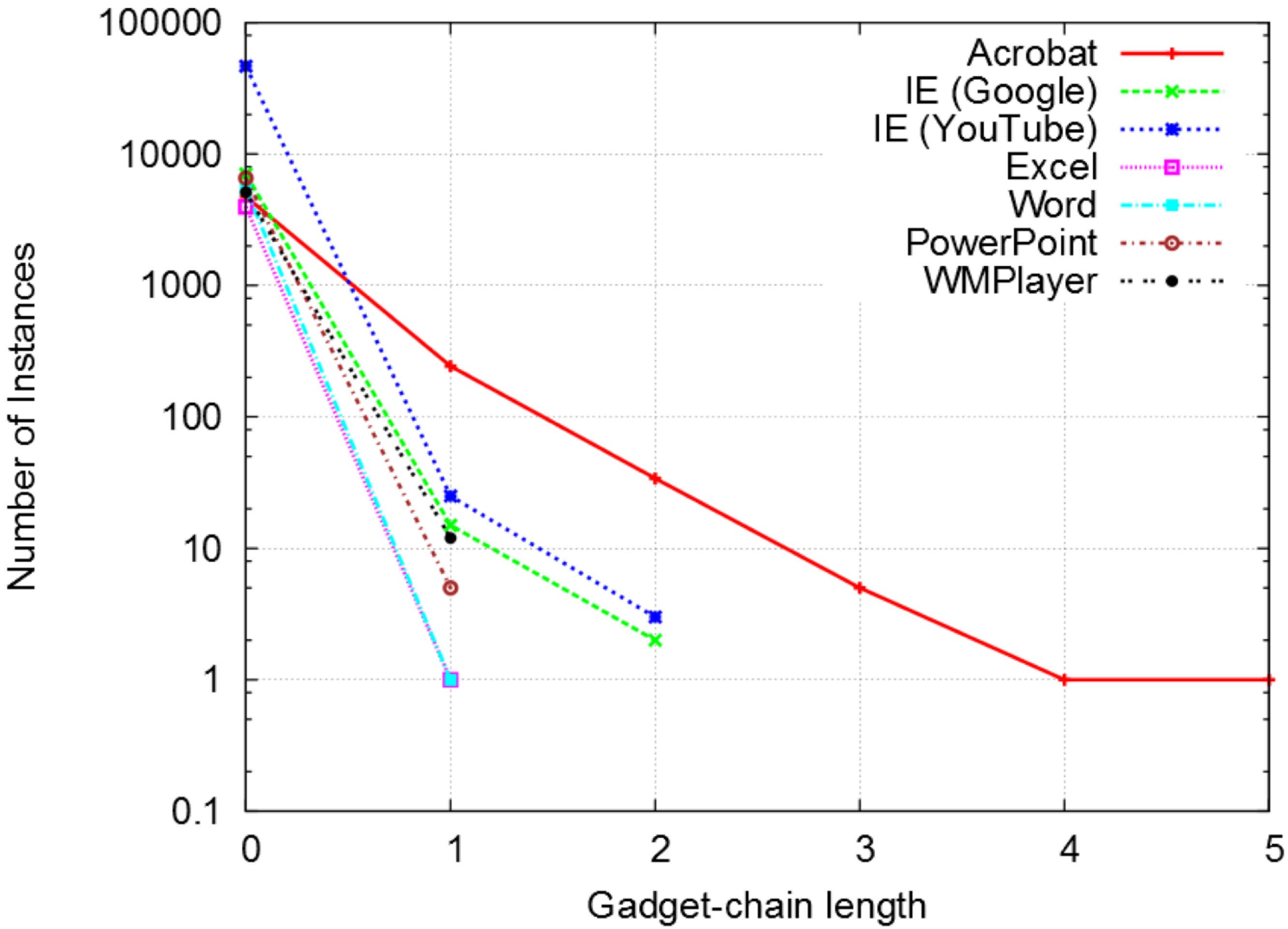
- Also applies to similar defenses like ROPecker [NDSS '13]

Multiple payloads

- kBouncer thresholds: $\mathbf{T_C=6, T_G=20}$
- Stricter thresholds: $\mathbf{T_C=2, T_G=27}$


7
33
2
2
2
3
Security check
VirtualProtect
SHELLCODE

Per Application Thresholds



What if We Had the Perfect CFG

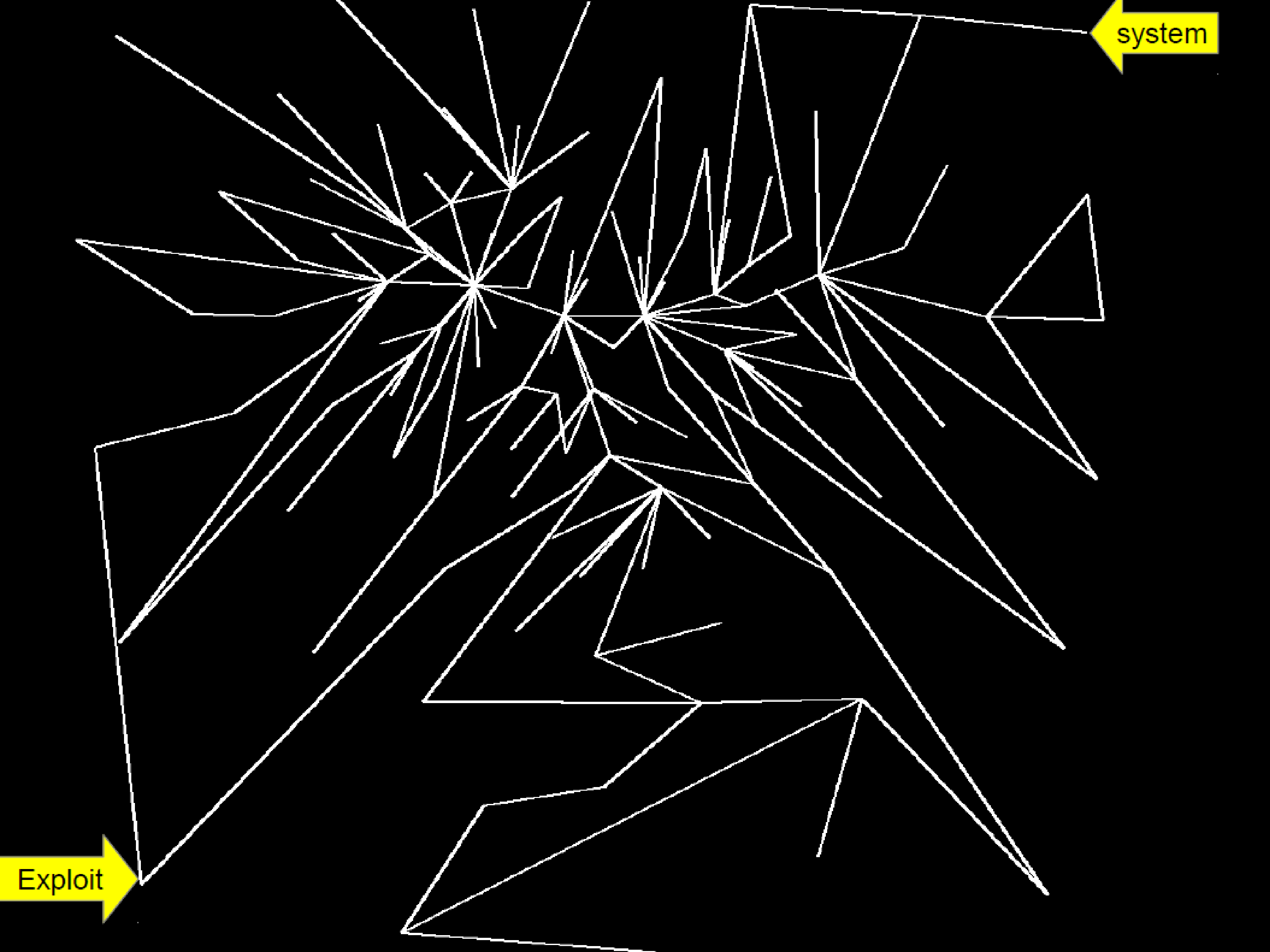
We know exactly which functions are called from an indirect call

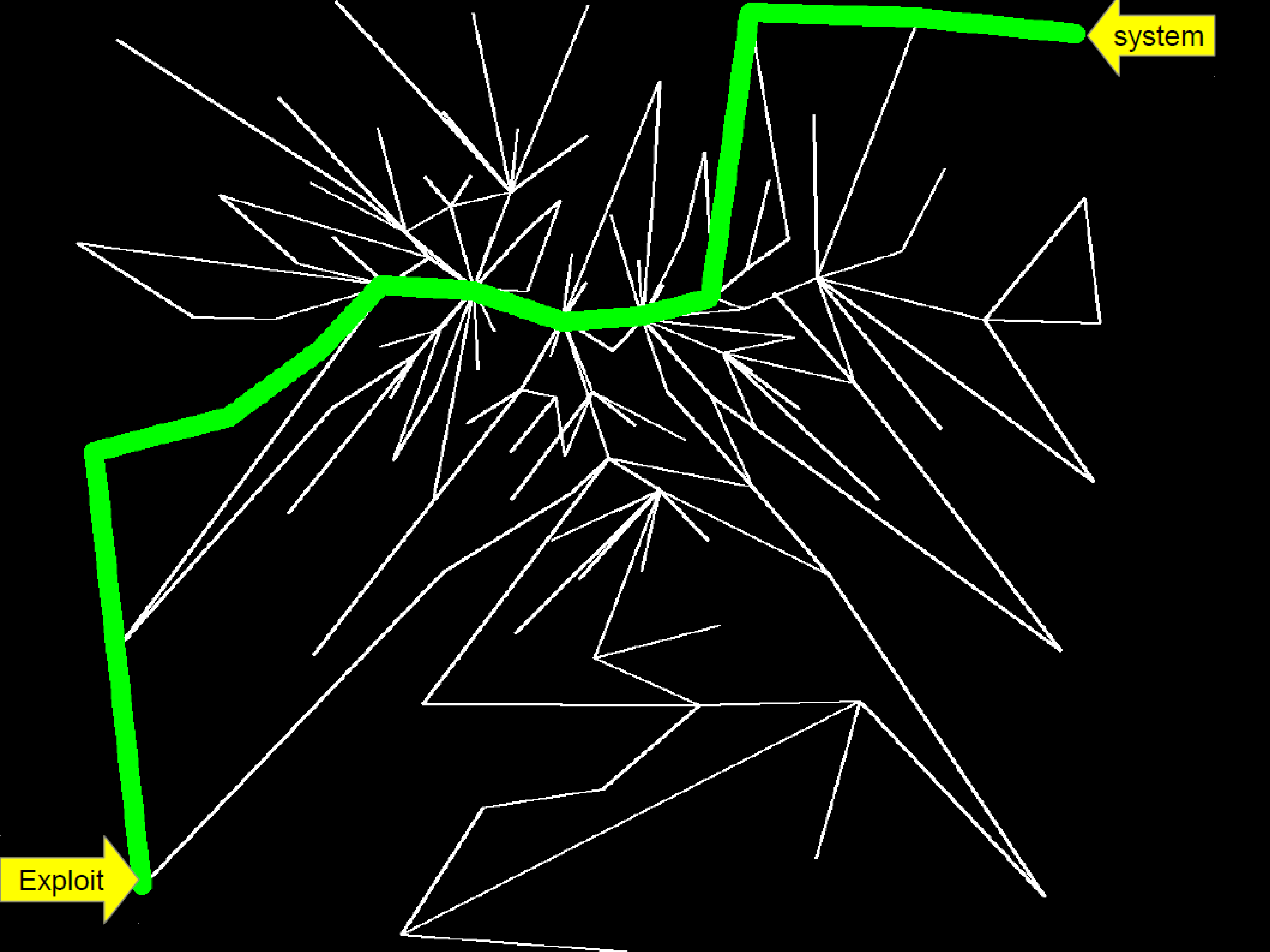
We know exactly the call sites where a function’s return is supposed to return

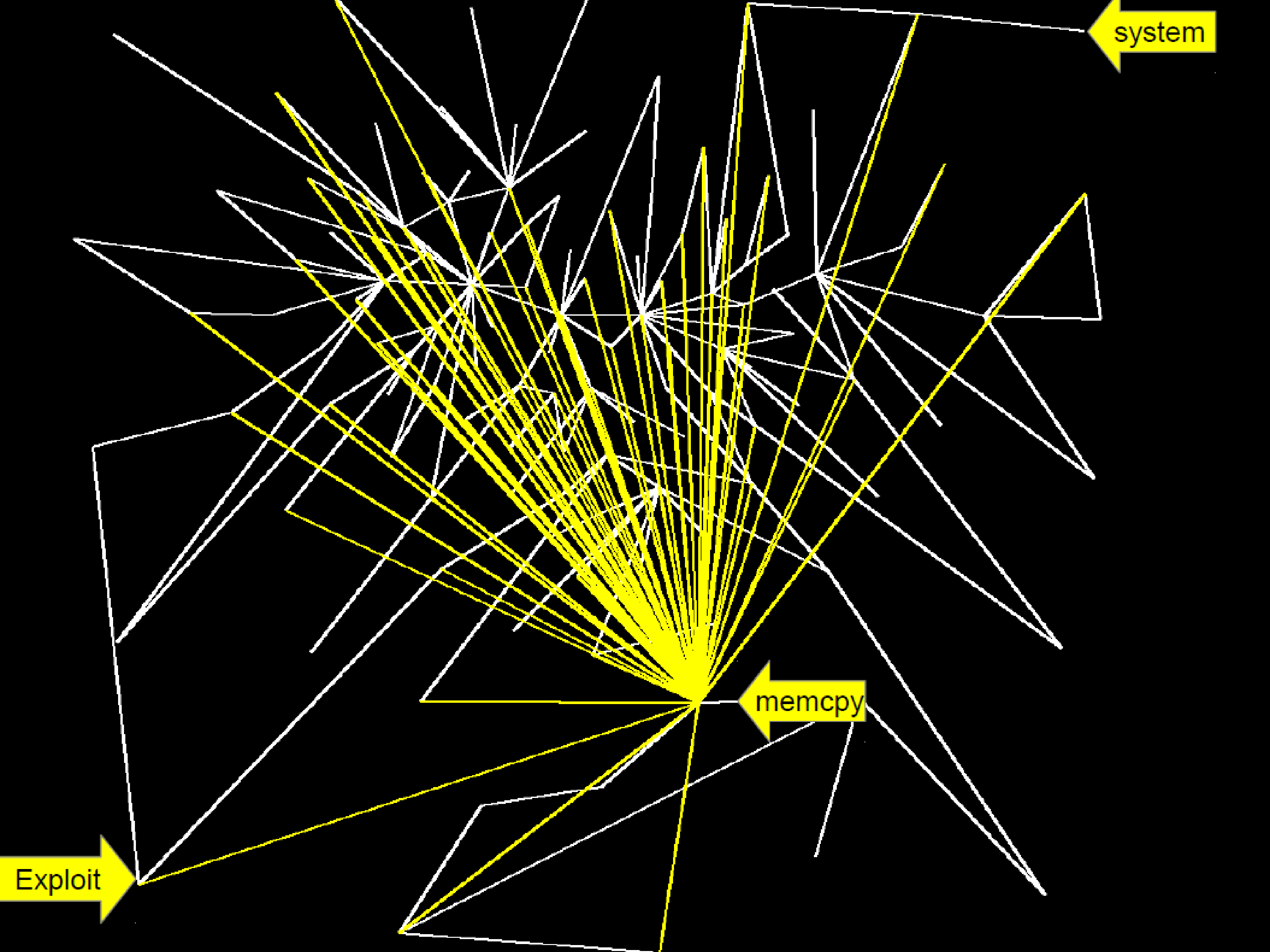
But we still do not have a shadow stack

Control Flow Bending

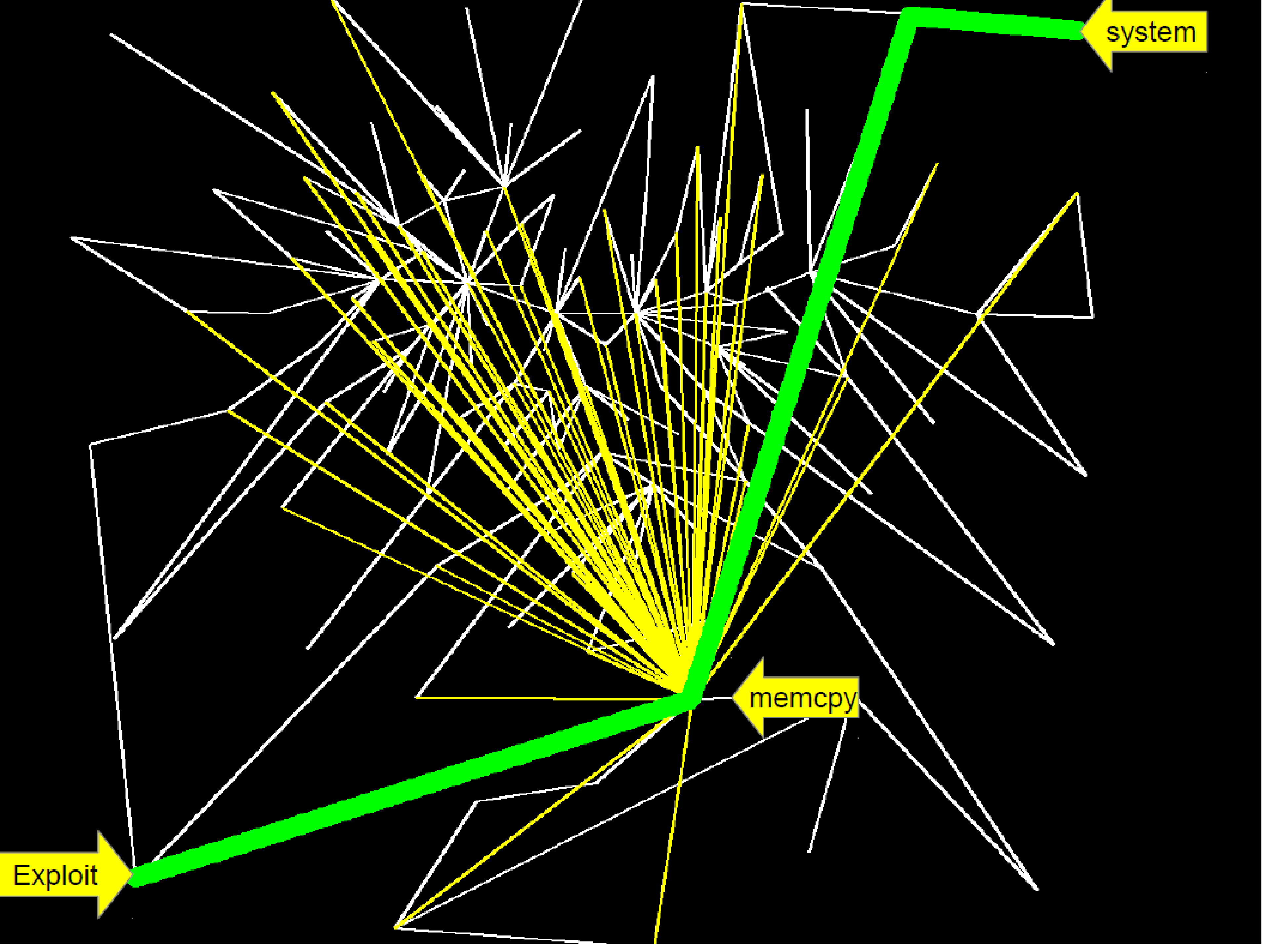
https://www.usenix.org/sites/default/files/conference/protected-files/sec15_slides_carlini.pdf


system
Exploit

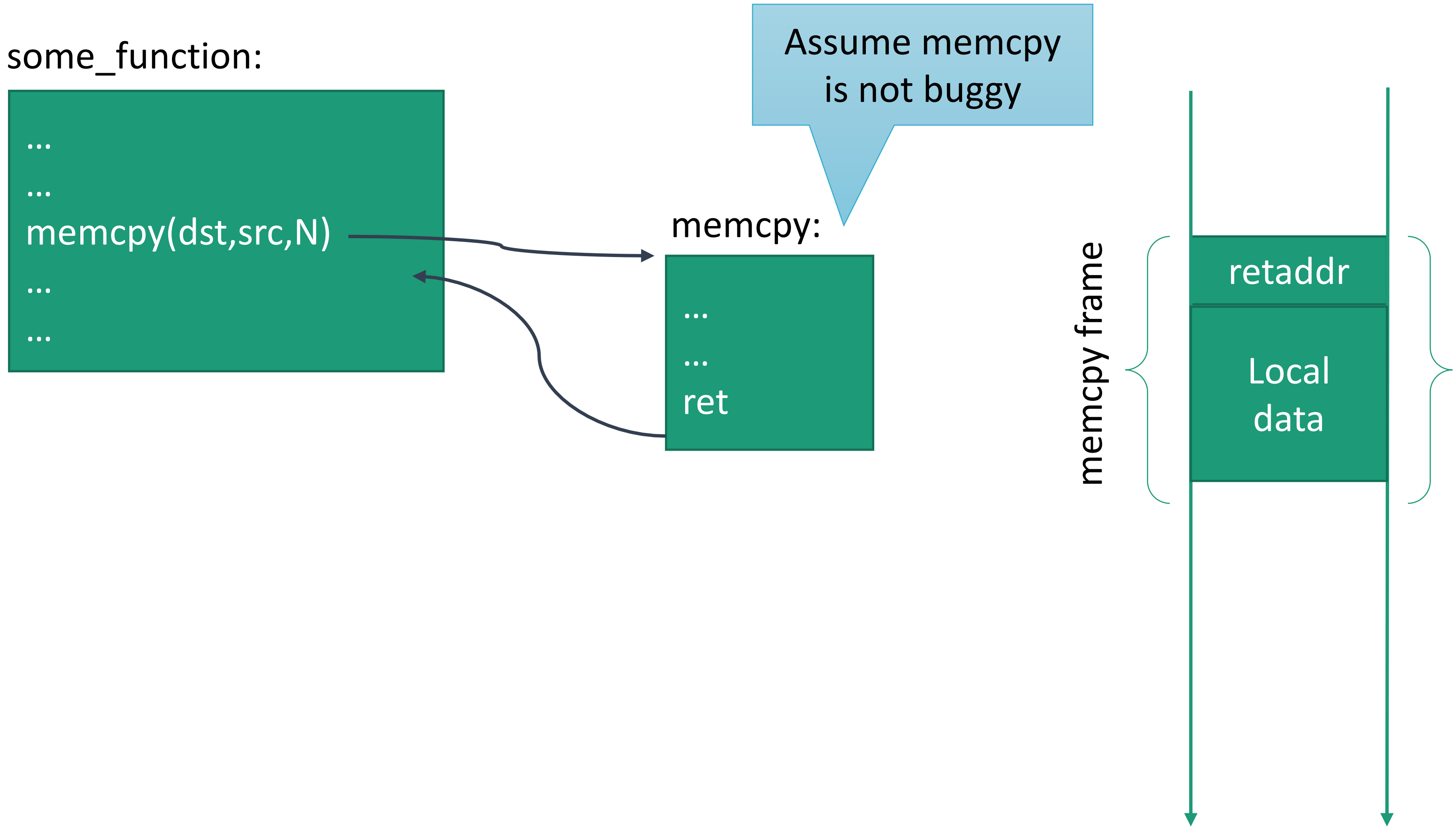

system
Exploit



system
memcpy
Exploit

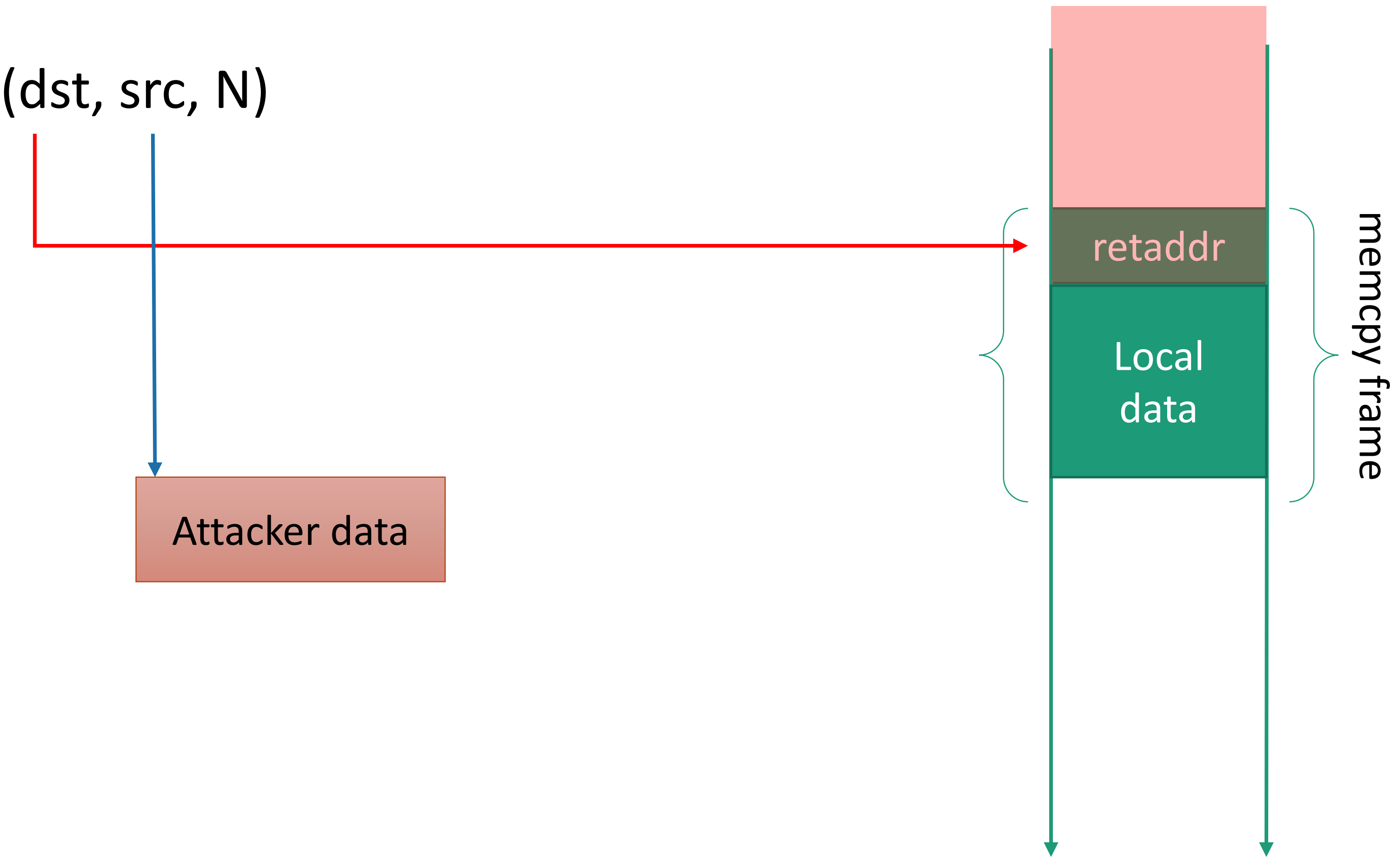

system
memcpy
Exploit

How to Exploit the memcpy() Hotspot



How to Exploit the memcpy() Hotspot

memcpy(dst, src, N)



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

Reading

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

Chained return-to-libc
https://sploitfun.wordpress.com/2015/05/08/bypassing-nx-bit-using-chained-return-to-libc/

Practical return-oriented programming
https://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf

The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)
https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf

Heap feng-shui

https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf