Modern Exploitation and Defenses

CS-576 Systems Security

Instructor: Georgios Portokalidis

Fall 2018

Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

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 Hijacking Prone Statements

Statements where the target statement cannot be known a priori

- Indirect control-flow transfers

Indirect calls, returns, and some switches

Calls to virtual functions are indirect calls

```
return;
```

```
return 100;
```

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

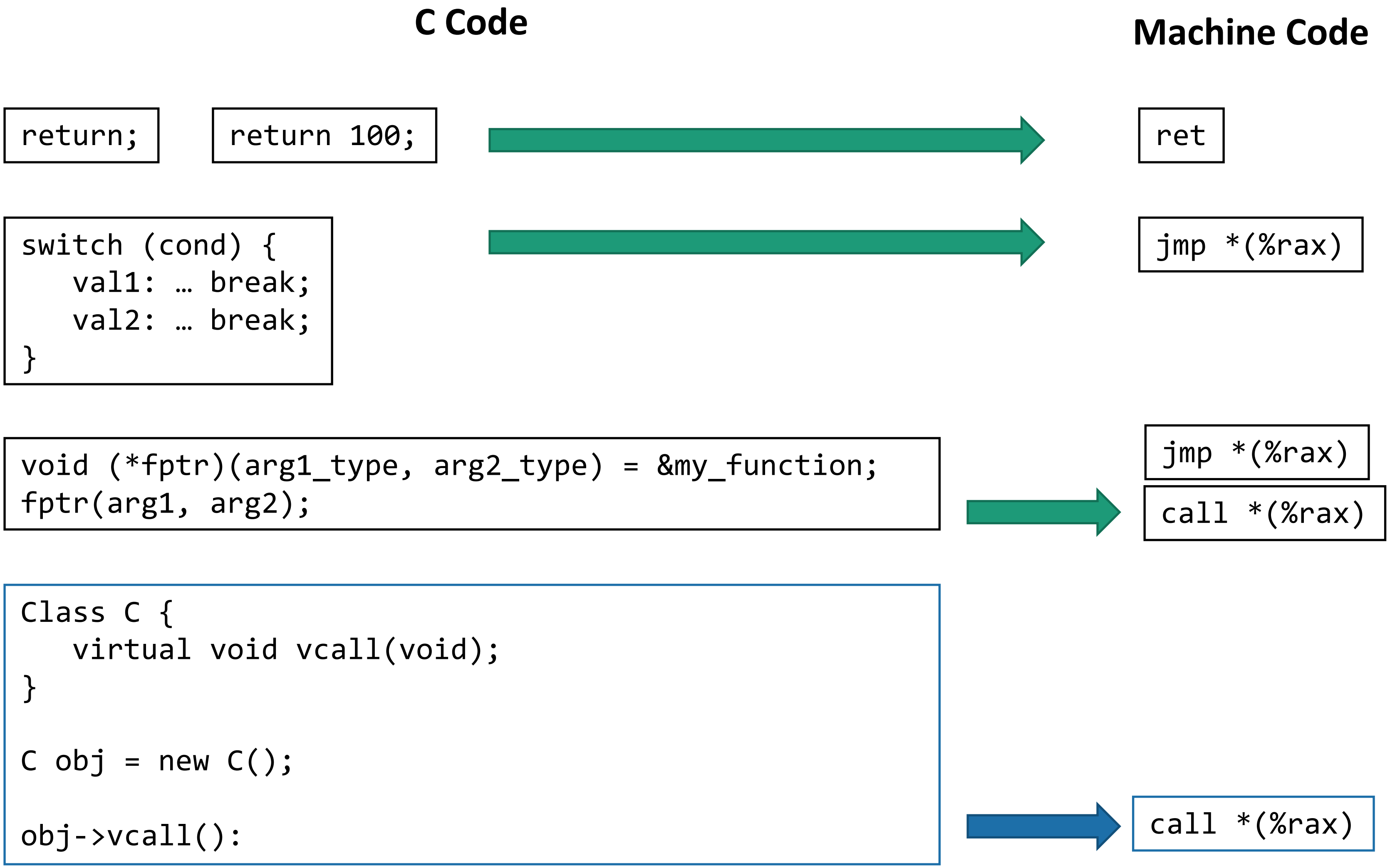
```
void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);
```

```
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)
```

Non-fixed Pointer Arguments

Indirect branch instruction	Pointer location
`ret`	`(%rsp)`

`jmp *%rax`
`call *%rax`
	`%rax`

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

Non-fixed Pointer Arguments

Indirect branch instruction	Pointer location
`ret`	`(%rsp)`

`jmp *%rax`
`call *%rax`
	`%rax`

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

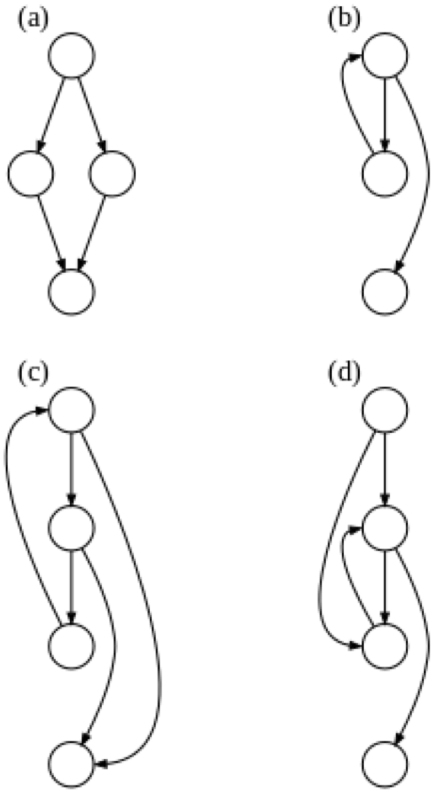
CFI aims to restrict what these instructions can target

How?

CFI → Enforce the Control-flow Graph

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

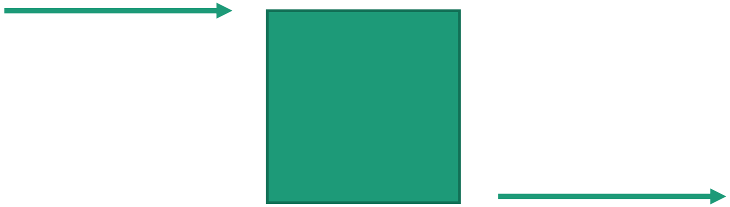
Nodes are **basic blocks (bbl)**



Basic Blocks

In this case a bbl is a sequence of instructions with a single entry and single exit

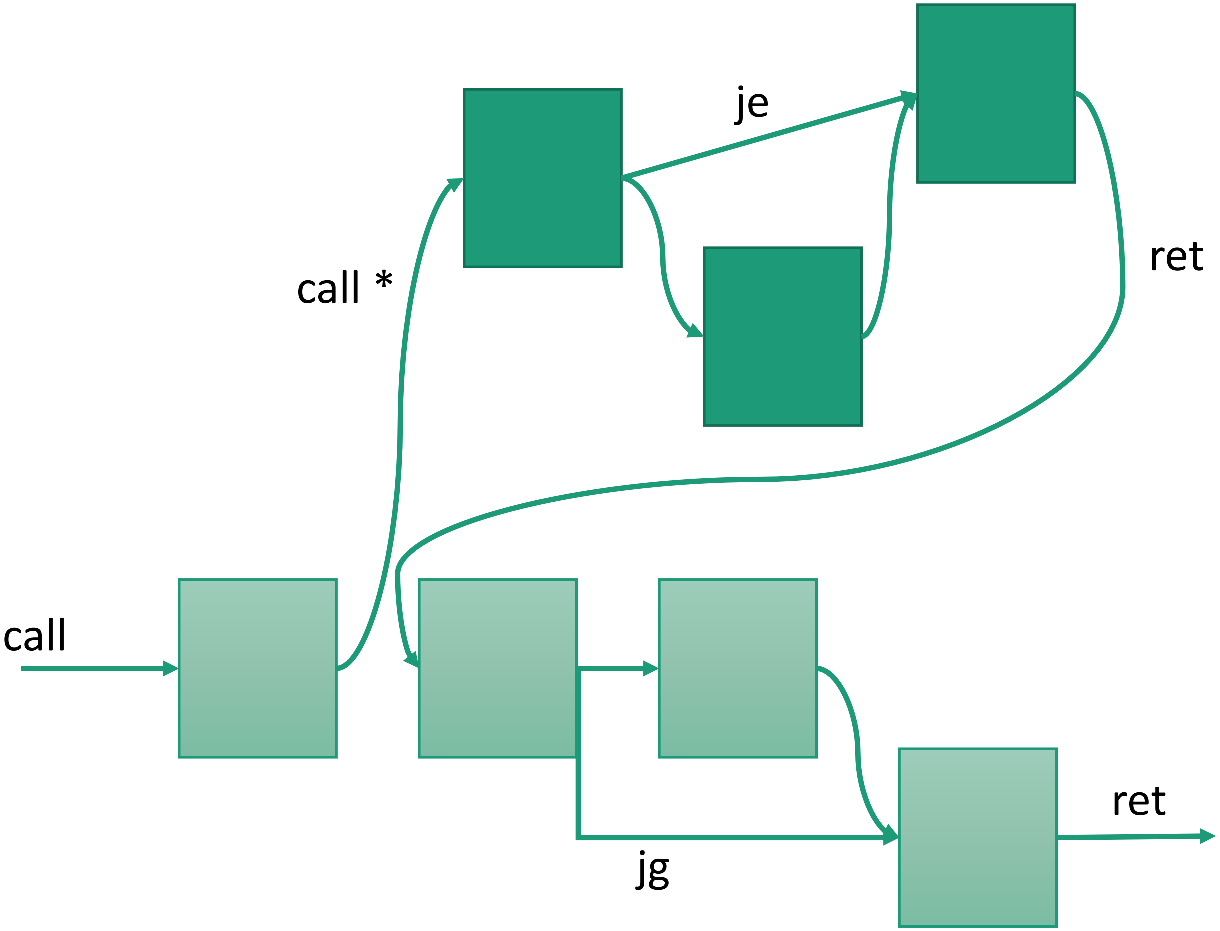
Execution can enter the bbl at the first instruction



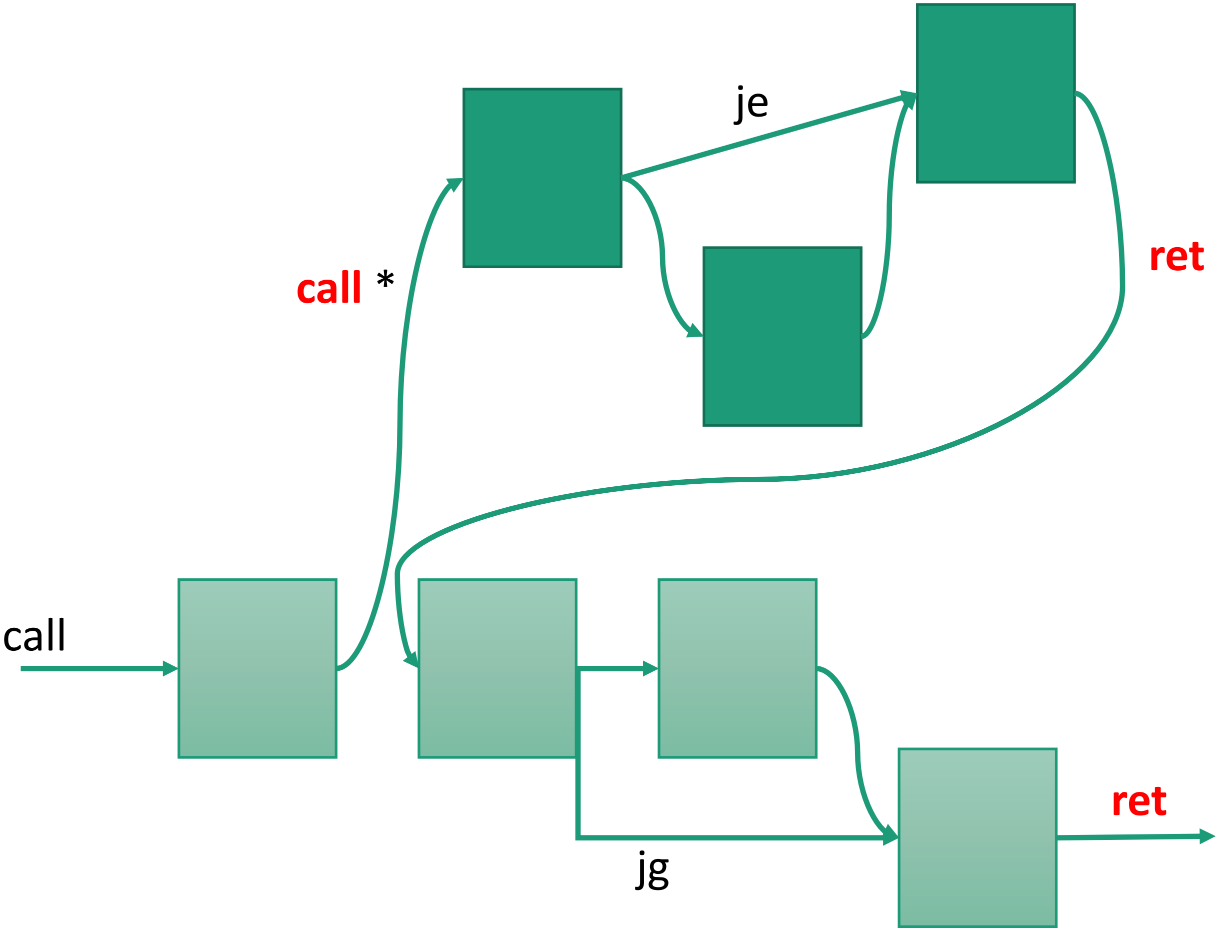
Execution can leave the bbl at the last instruction

Note: asynchronous events (e.g., signal) can temporarily transfer control flow elsewhere

CFG Example



CFG Example



Extracting the CFG

With source code

- More reliable
- Cannot be fully reconstructed
- Resolving pointers is hard

```
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);
        }
}
```

Pointer aliasing. In computer programming, **aliasing** refers to the situation where the same memory location can be accessed using different names. For instance, if a function takes two **pointers** A and B which have the same value, then the name A[0] aliases the name B[0] .

Extracting the CFG

With source code

- More reliable
- Cannot be fully reconstructed
- Resolving pointers is hard

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);
        }
}
```

Working with an Imperfect CFG

Lets assume that we know/can learn

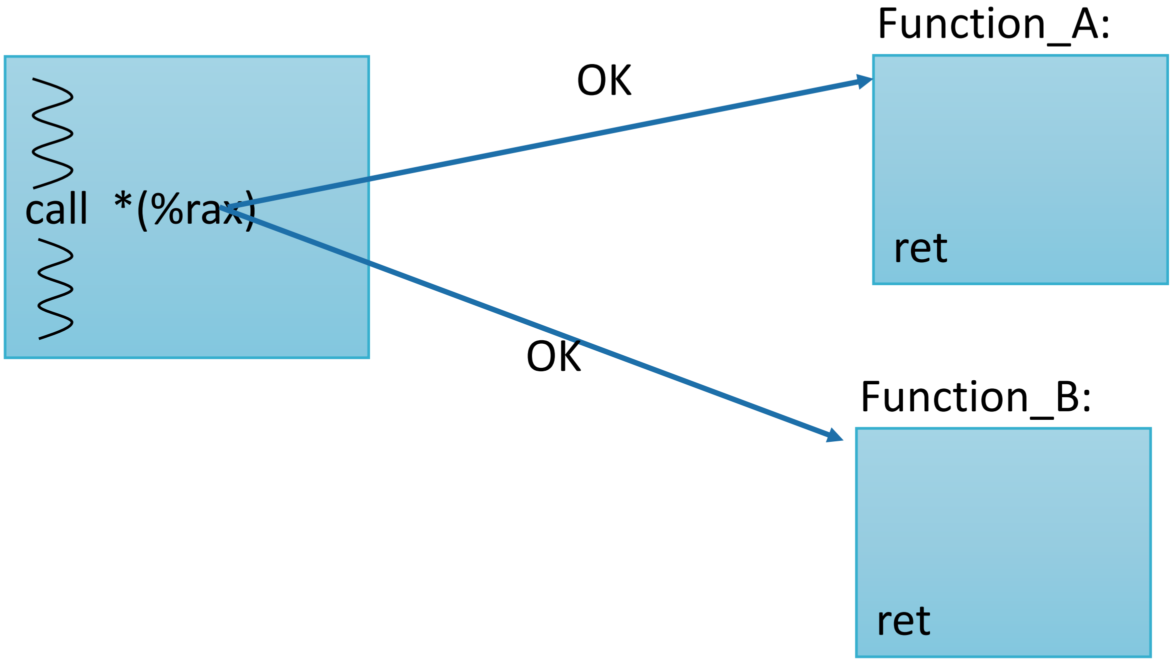
- The location of every function
- The location of every indirect branch instruction

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
- More permissive than the actual (potentially unknown) CFG but better than before

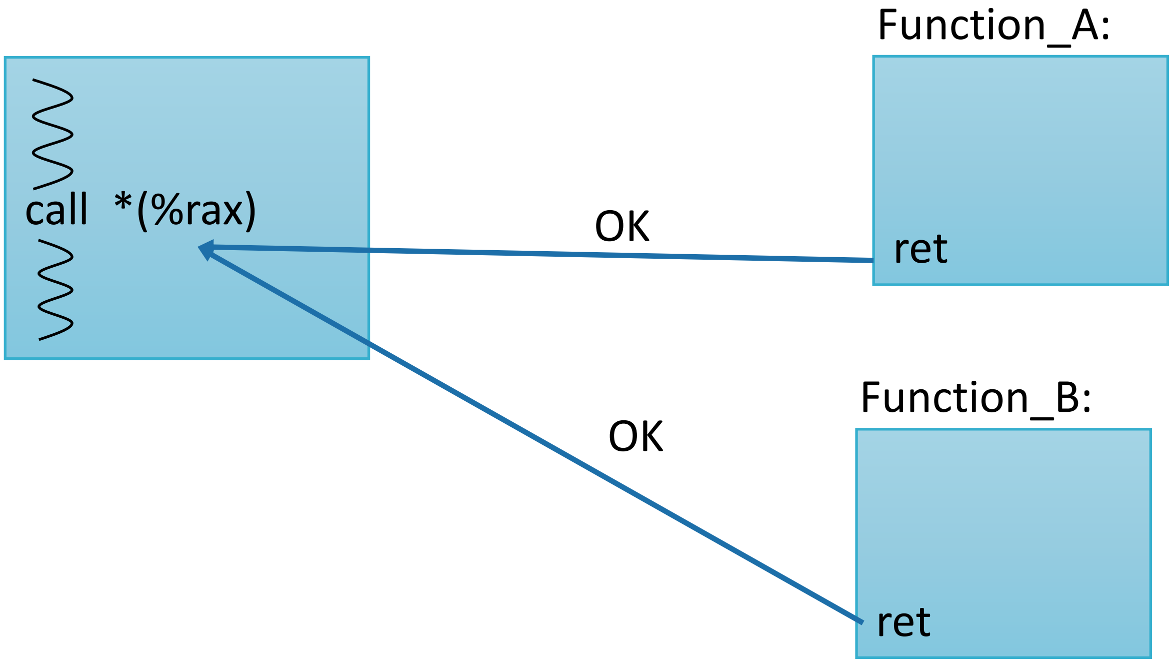
What is Allowed

Indirect calls should only transfer control to functions



What is Allowed

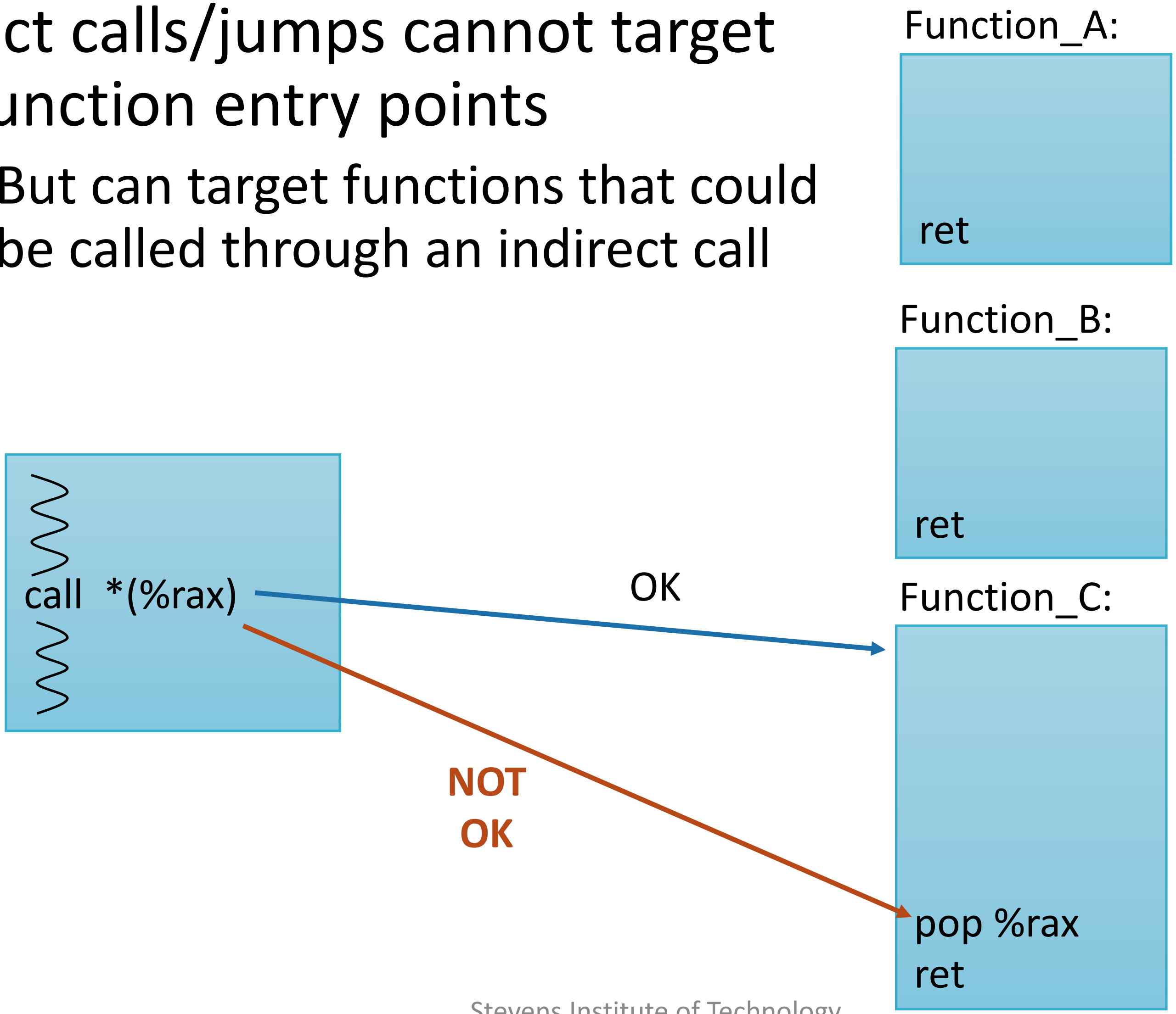
Returns should only transfer control to instructions following a indirect call or jump



What is Not Allowed

Indirect calls/jumps cannot target non function entry points

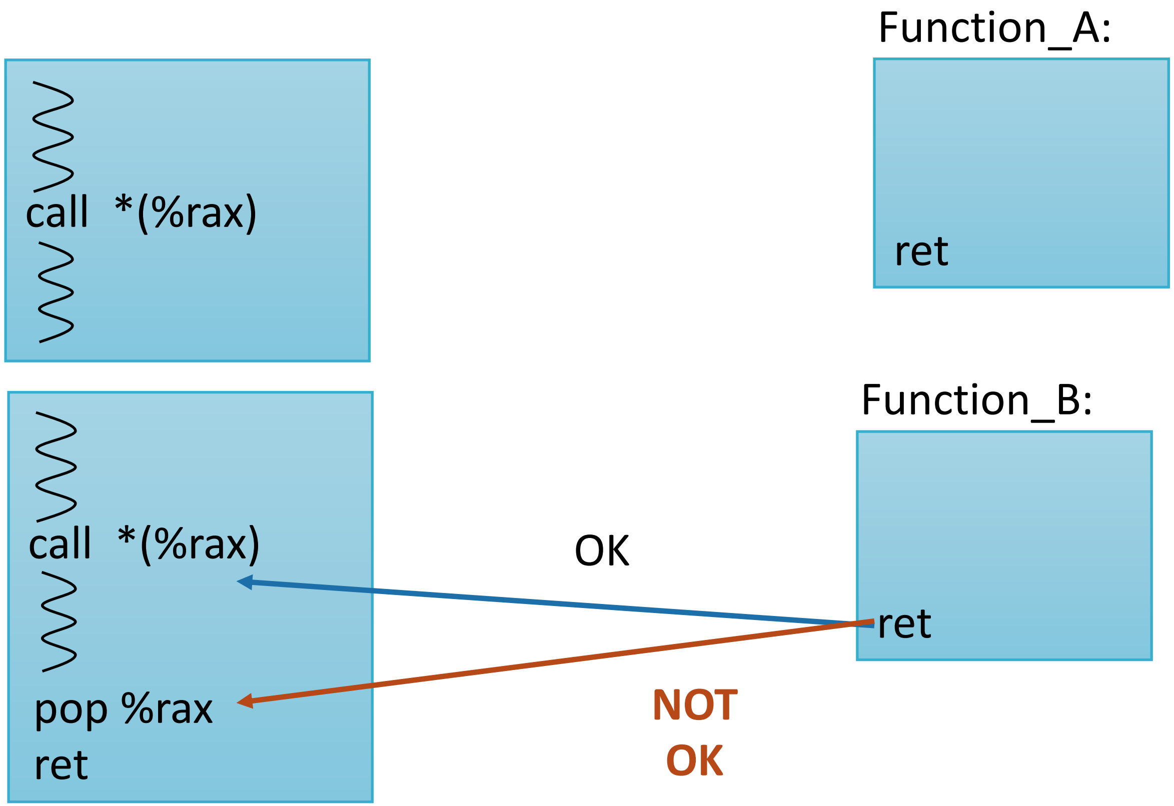
- But can target functions that could be called through an indirect call



What is Not Allowed

Returns cannot target bytes not following a call/jump

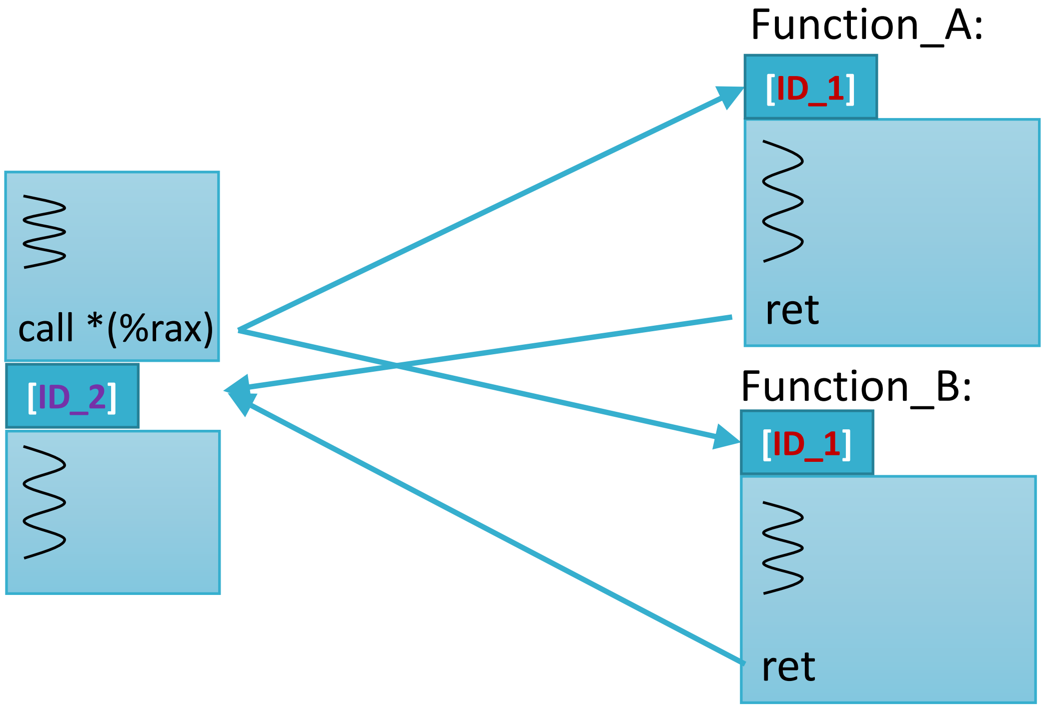
- But can target valid bytes in functions that may have not called them



Enforcing Through Embedded IDs

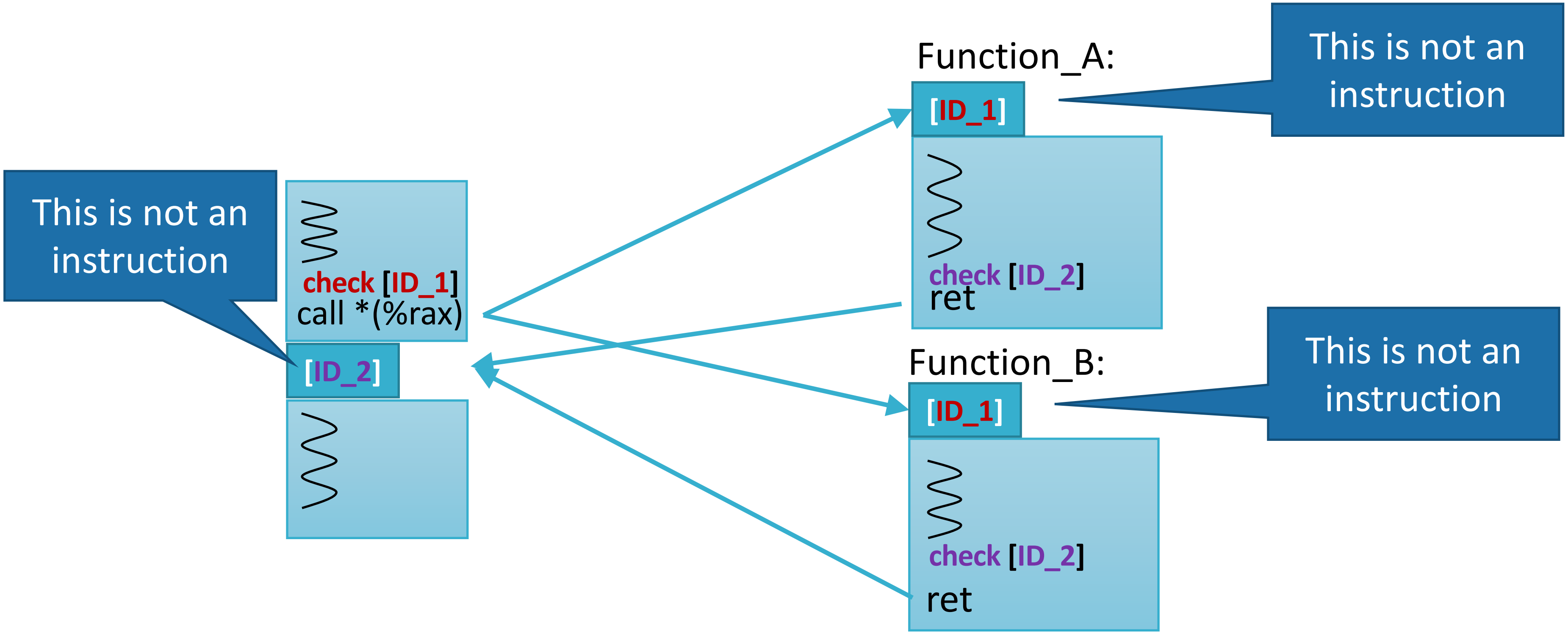
ID codes are embedded into the binary program to identify acceptable targets

- 2-ID policy



Enforcing Through Embedded IDs

Checks are introduced right before the control transfer



This is not an instruction

check [ID_1]
call *(%rax)

[ID_2]

Function_A:

[ID_1]

check [ID_2]
ret

This is not an instruction

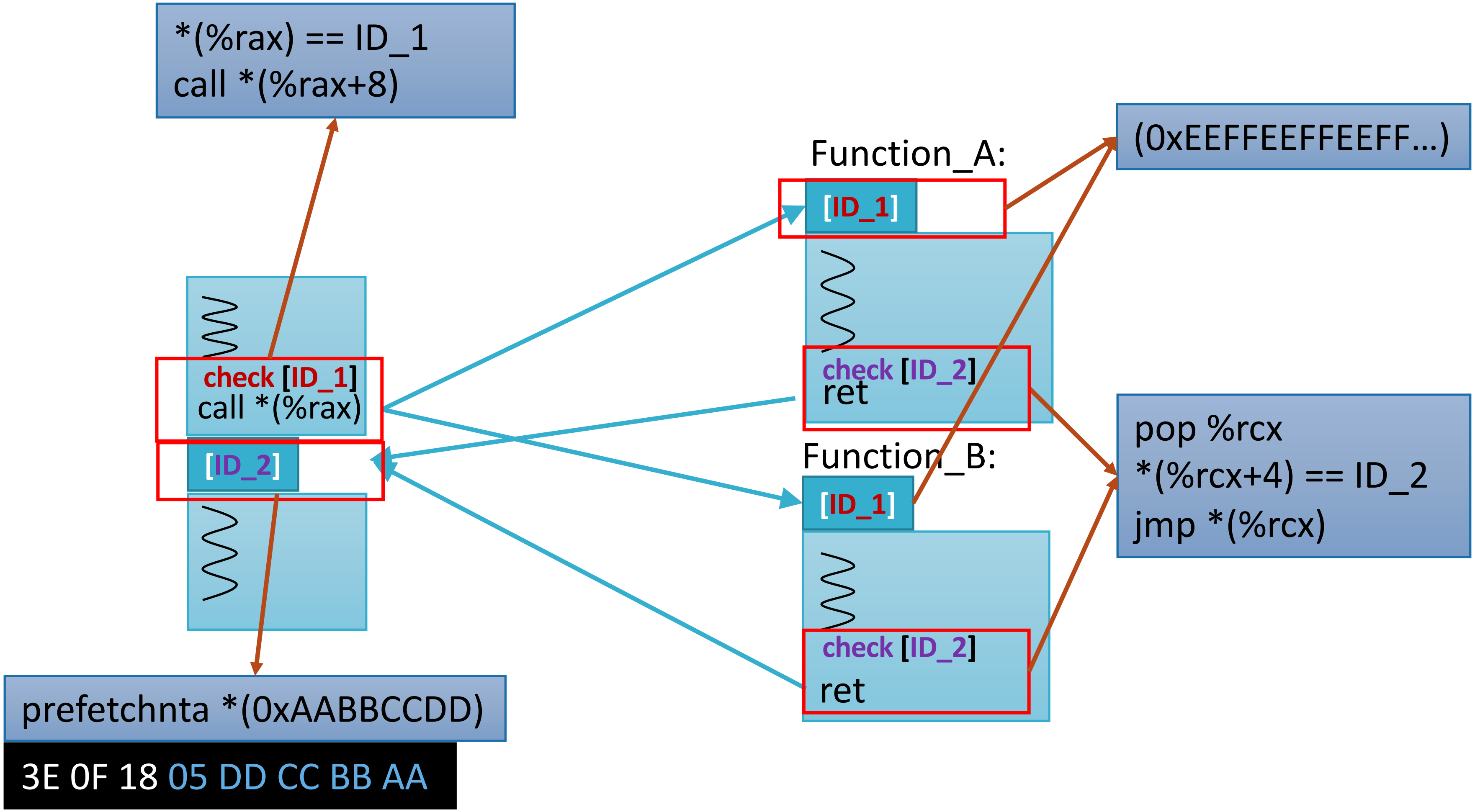
Function_B:

[ID_1]

check [ID_2]
ret

This is not an instruction

Modifications for CFI Enforcement



*(%rax) == ID_1
call *(%rax+8)

Function_A:

[ID_1]

check [ID_2]
ret

(0xEEFFEEFFEEFF…)

check [ID_1]
call *(%rax)

[ID_2]

Function_B:

[ID_1]

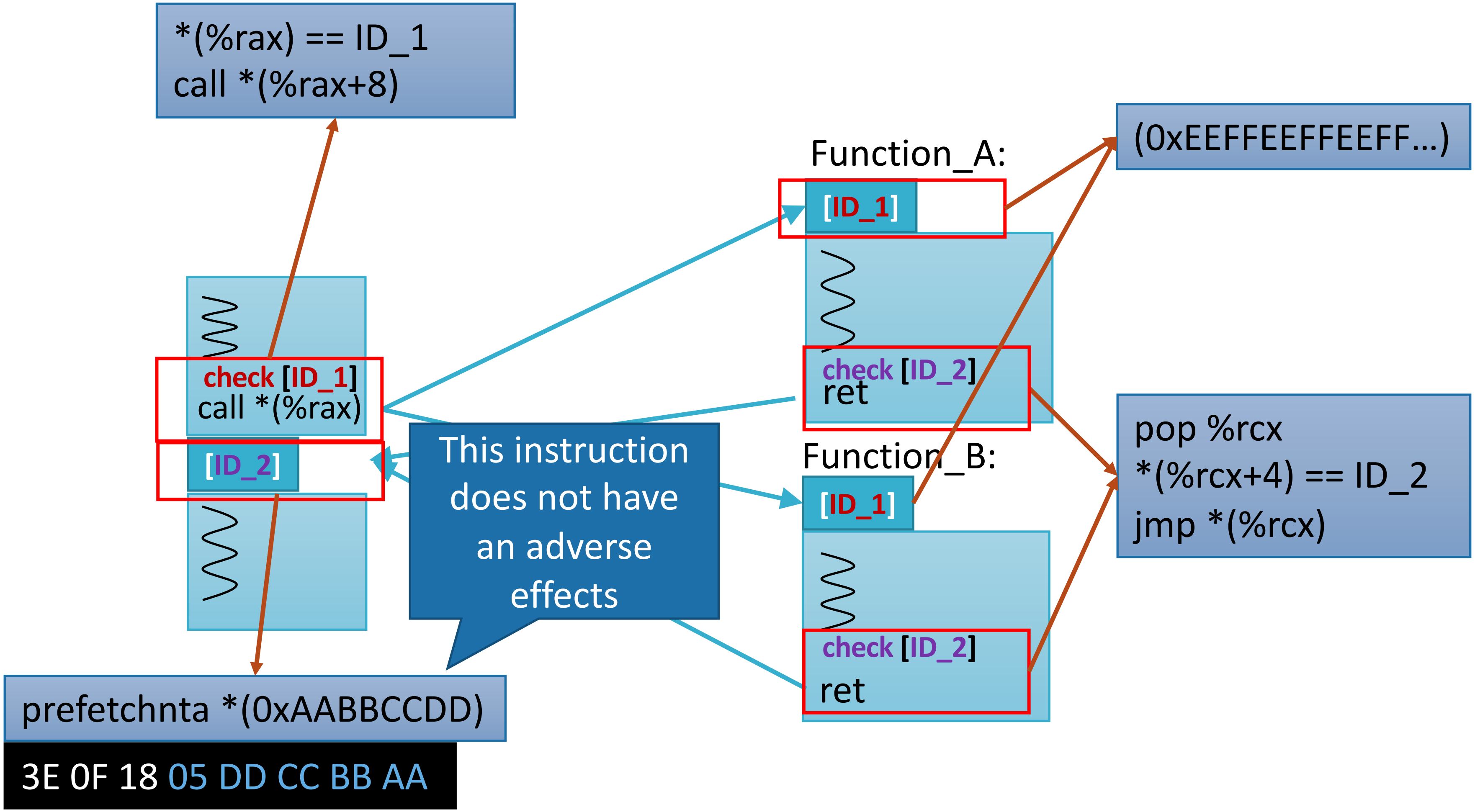
check [ID_2]
ret

pop %rcx
*(%rcx+4) == ID_2
jmp *(%rcx)

prefetchnta *(0xAABBCCDD)

3E 0F 18 05 DD CC BB AA

Modifications for CFI Enforcement



*(%rax) == ID_1
call *(%rax+8)

(0xEEFFEEFFEEFF…)

Function_A:

[ID_1]

check [ID_2]
ret

Function_B:

[ID_1]

check [ID_2]
ret

pop %rcx
*(%rcx+4) == ID_2
jmp *(%rcx)

check [ID_1]
call *(%rax)

[ID_2]

This instruction does not have an adverse effects

prefetchnta *(0xAABBCCDD)

3E 0F 18 05 DD CC BB AA

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

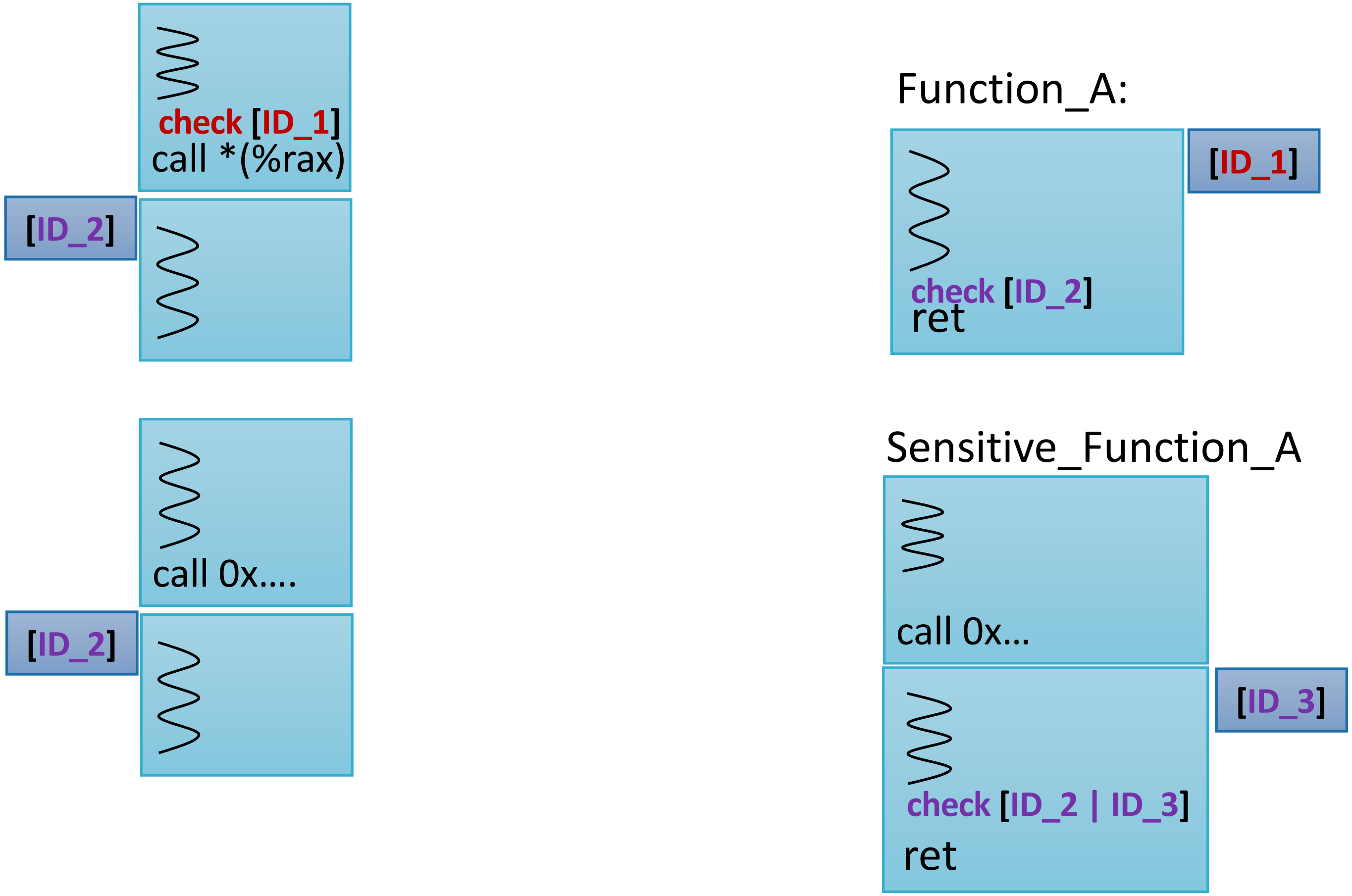
Chao Zhang
Tao Wei
Zhaofeng Chen
Lei Duan
Laszlo Szekeres
Stephen McCamant
Dawn Song
Wei Zou

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



check [ID_1]
call *(%rax)

[ID_2]

call 0x….

[ID_2]

Function_A:

[ID_1]

check [ID_2]
ret

Sensitive_Function_A

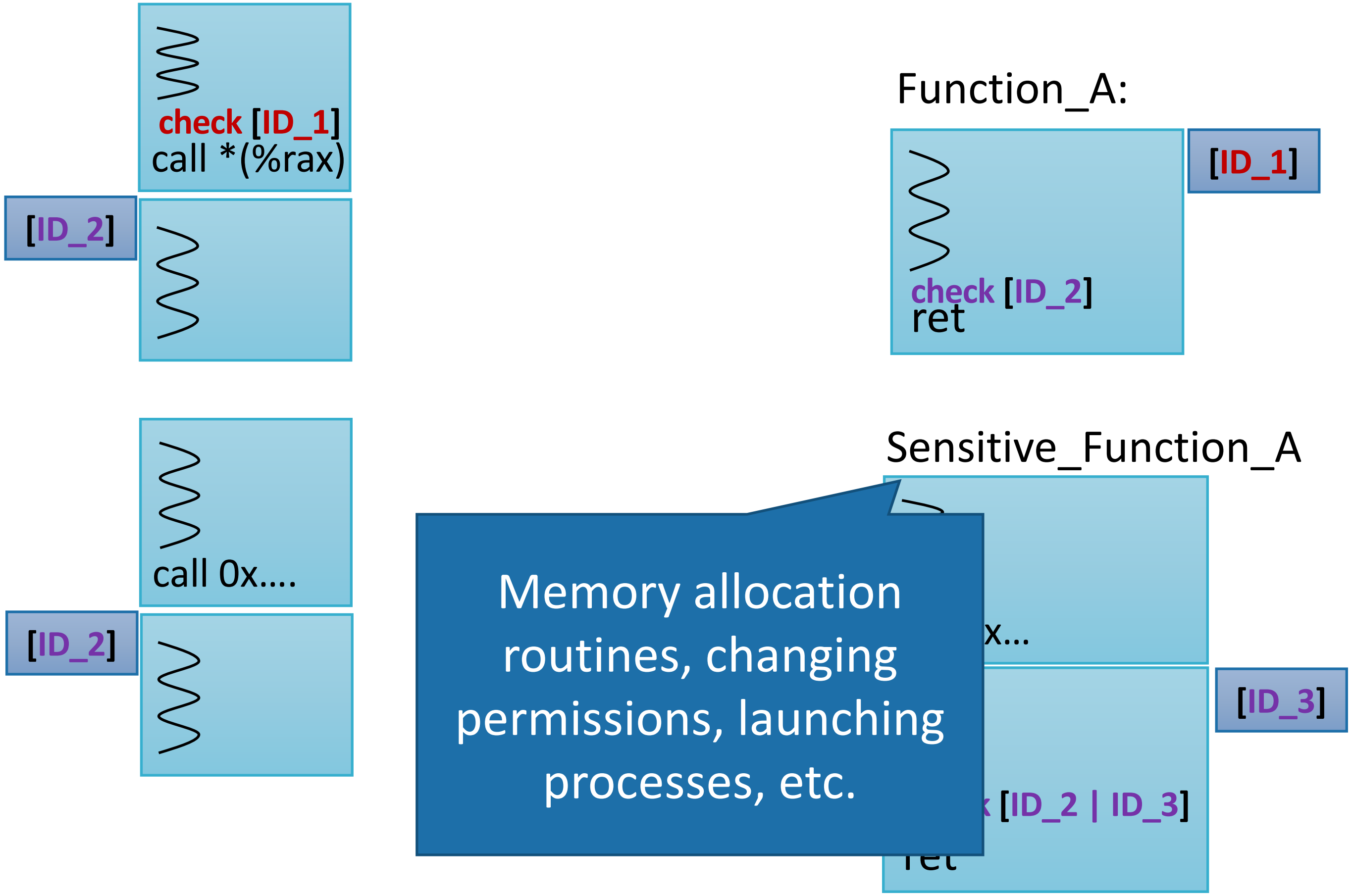
call 0x…

[ID_3]

check [ID_2 | ID_3]
ret

CCFIR

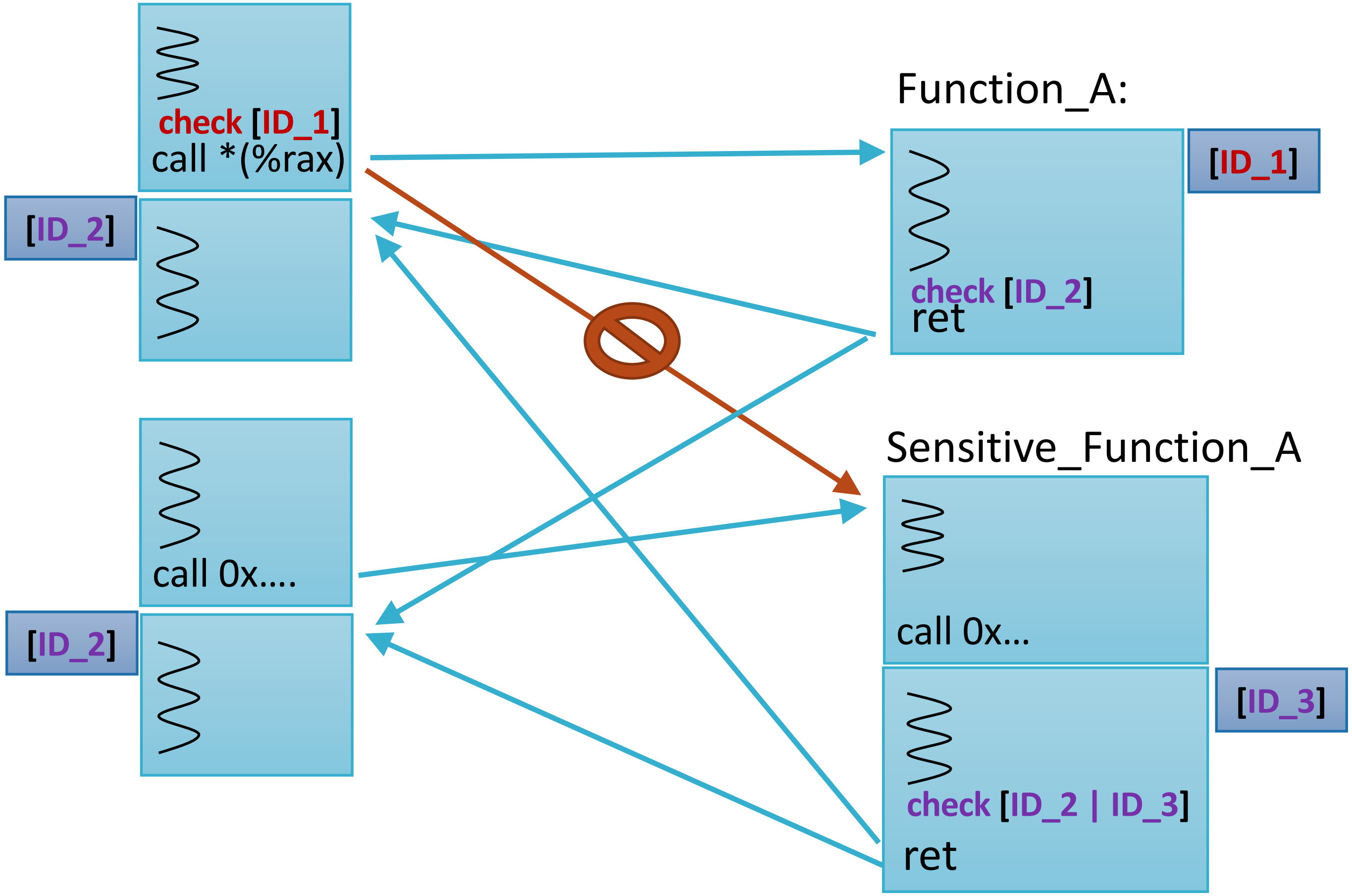
Three IDs are used to restrict control flow



Memory allocation routines, changing permissions, launching processes, etc.

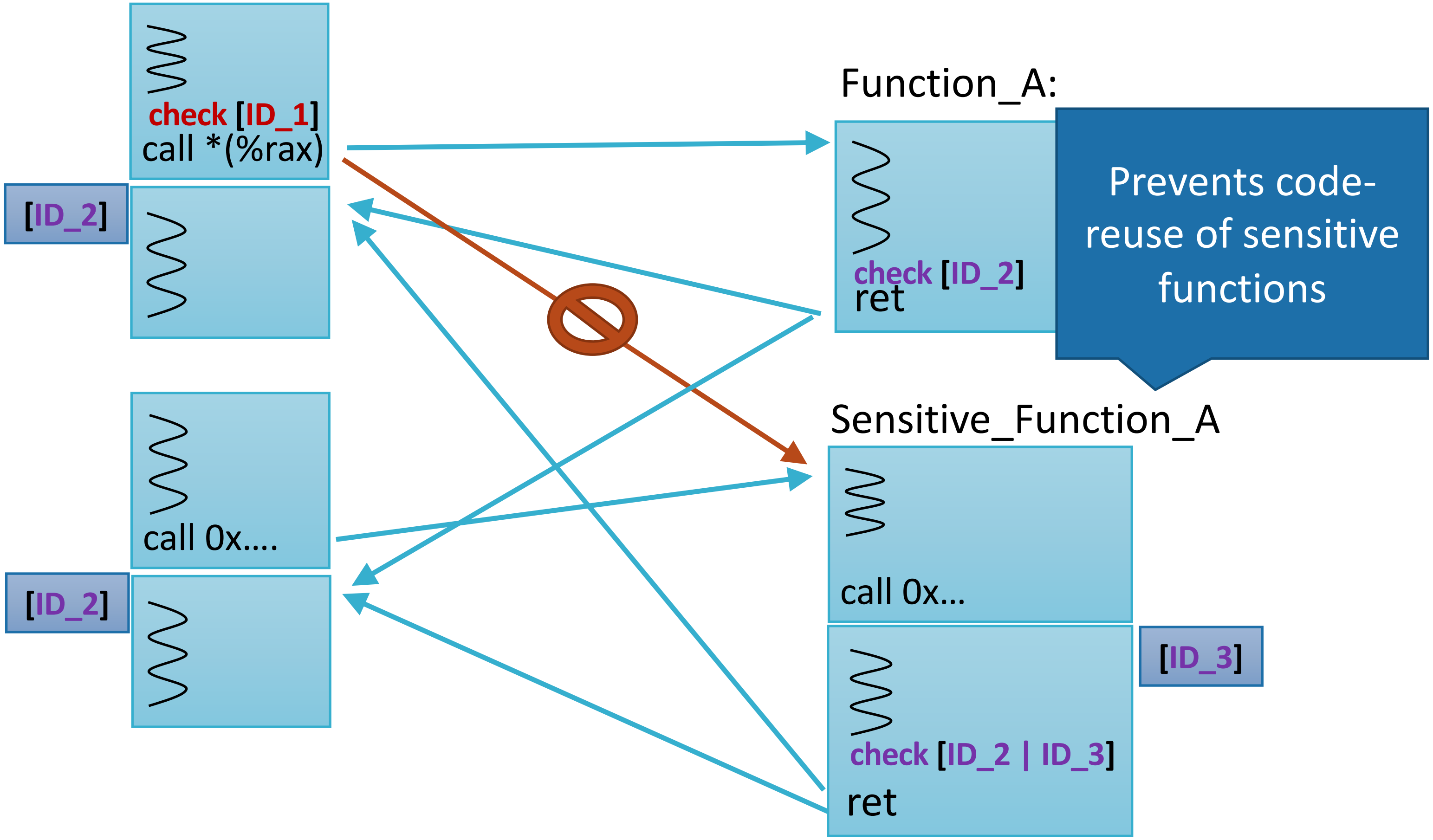
CCFIR

Three IDs are used to restrict control flow

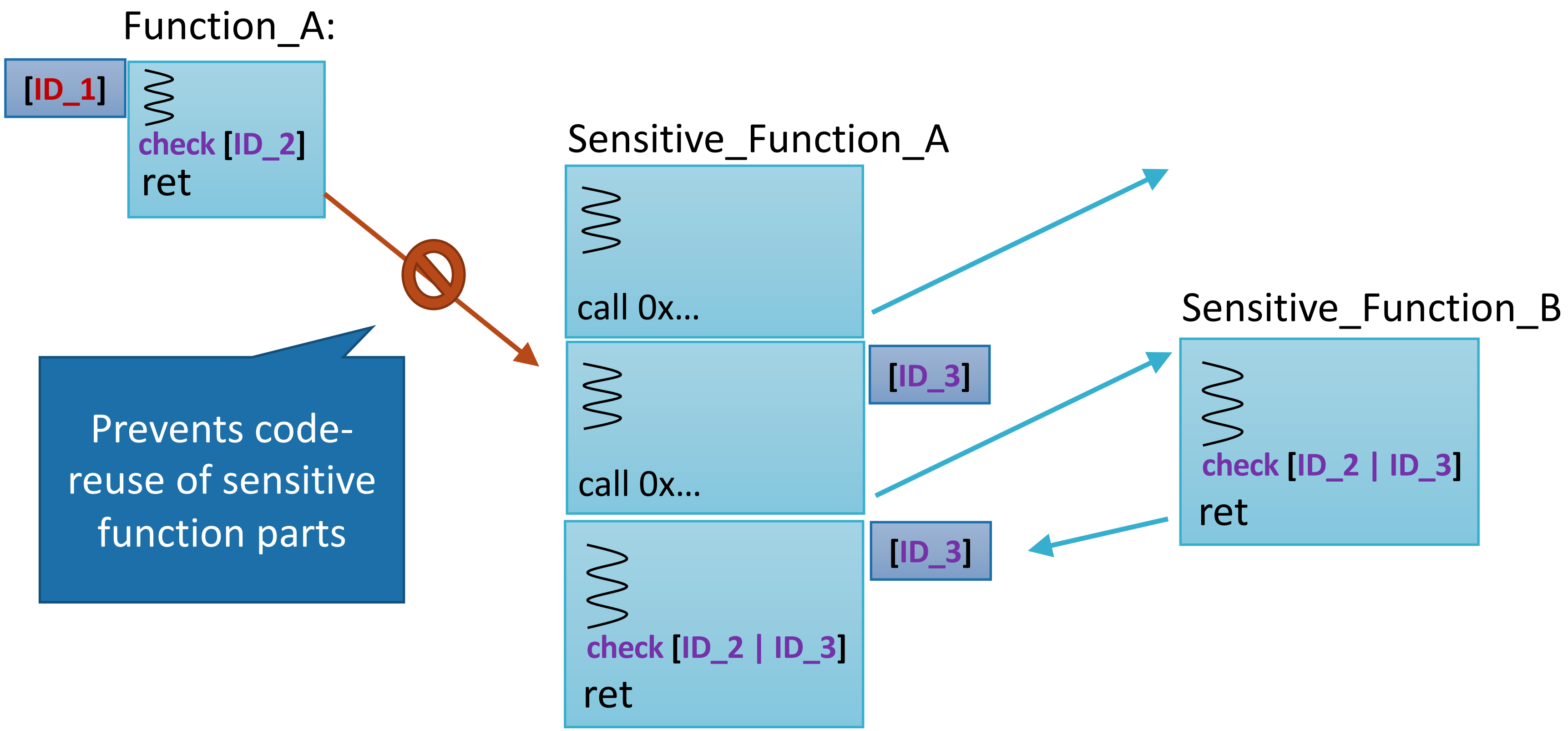


CCFIR

Three IDs are used to restrict control flow

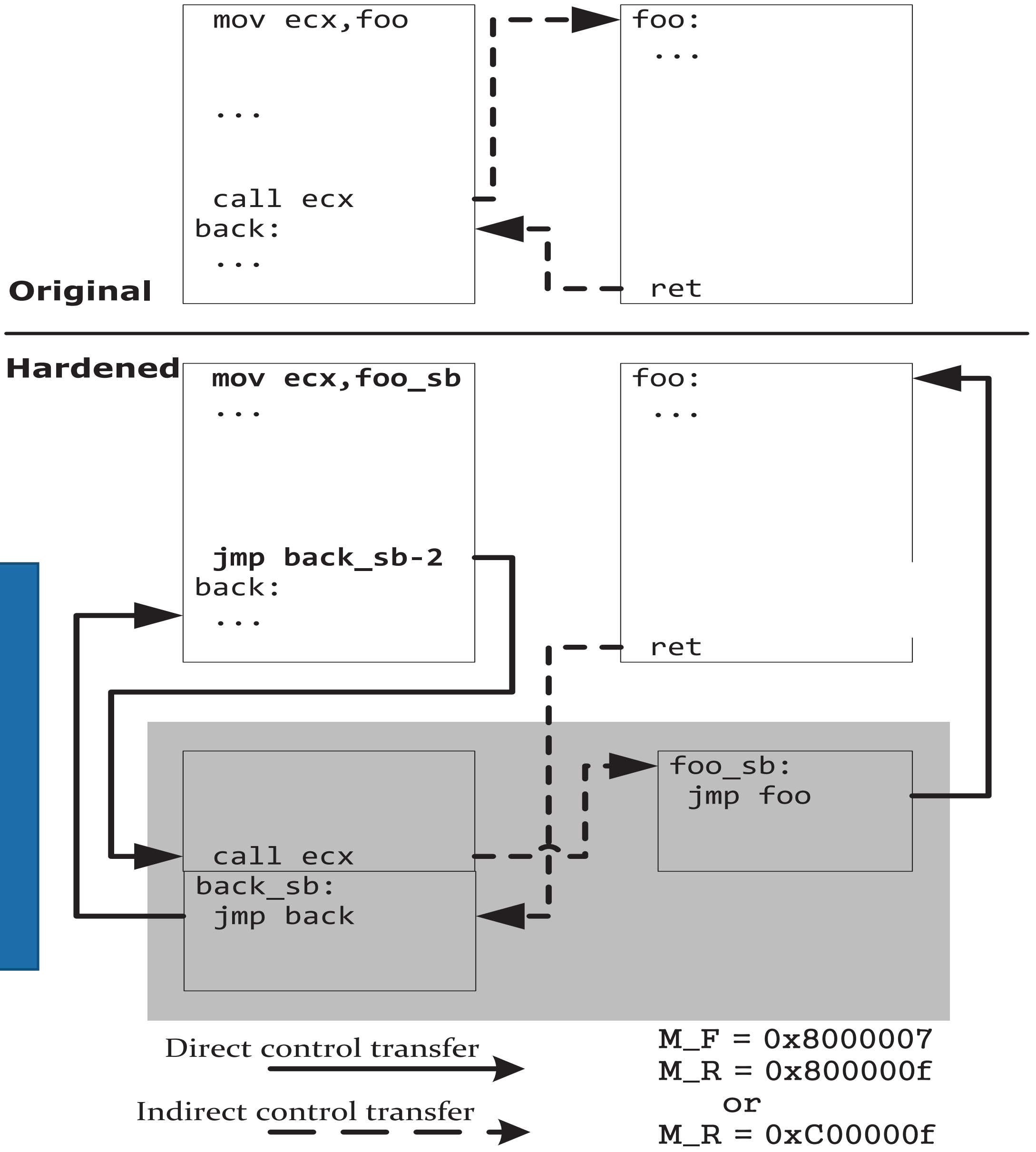


Sensitive Functions Heuristic



Prevents code-reuse of sensitive function parts

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



Original

```
mov ecx,foo
...
call ecx
back:
...
```

```
foo:
...
ret
```

Hardened

```
mov ecx,foo_sb
...
jmp back_sb-2
back:
...
```

```
foo:
...
ret
```

```
call ecx
back_sb:
jmp back
```

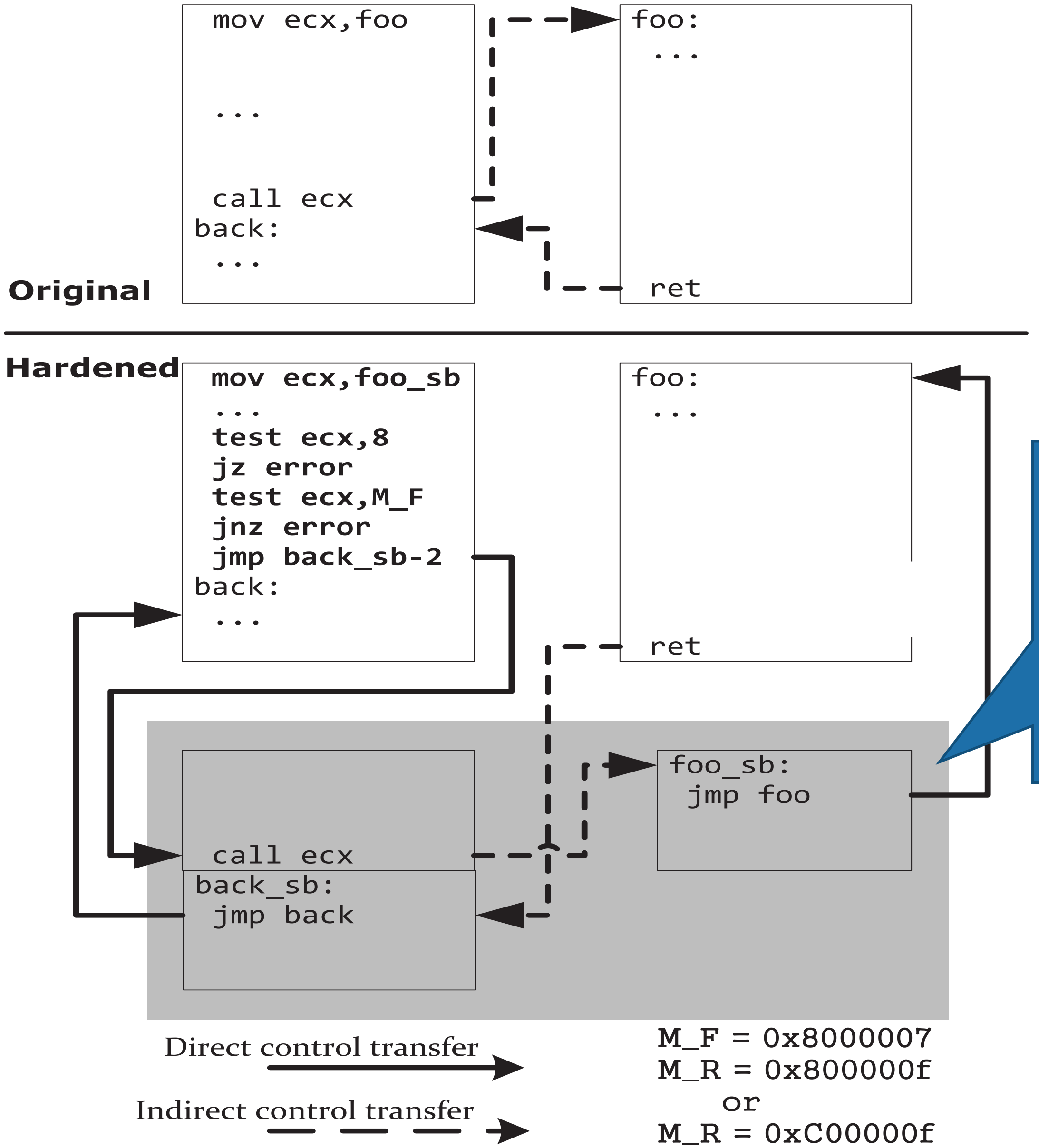
```
foo_sb:
jmp foo
```

Direct control transfer

Indirect control transfer

```
M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f
```

Targeted functions are called indirectly through another trampoline



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:
 ...

 ret
```

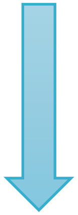
```
 call ecx
back_sb:
 jmp back
```

```
foo_sb:
 jmp foo
```

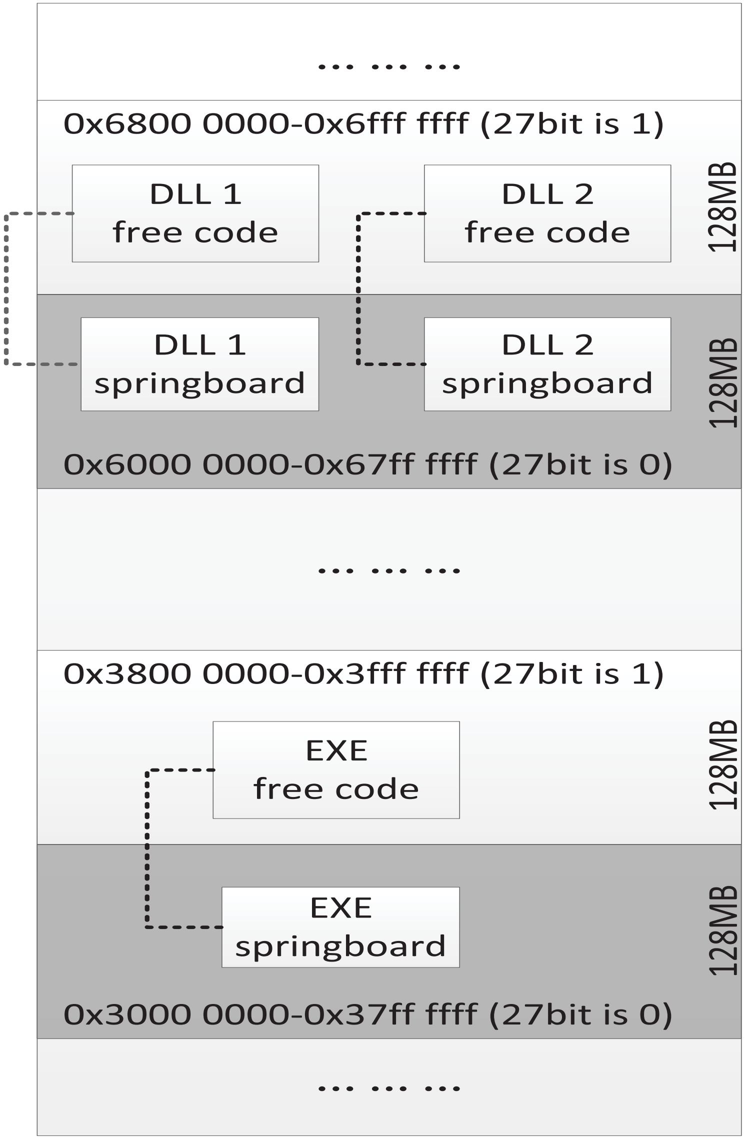
```
M_F = 0x8000007
M_R = 0x800000f
      or
M_R = 0xC00000f
```

Function stubs are carefully to aligned to easily perform checks

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



```
0
```



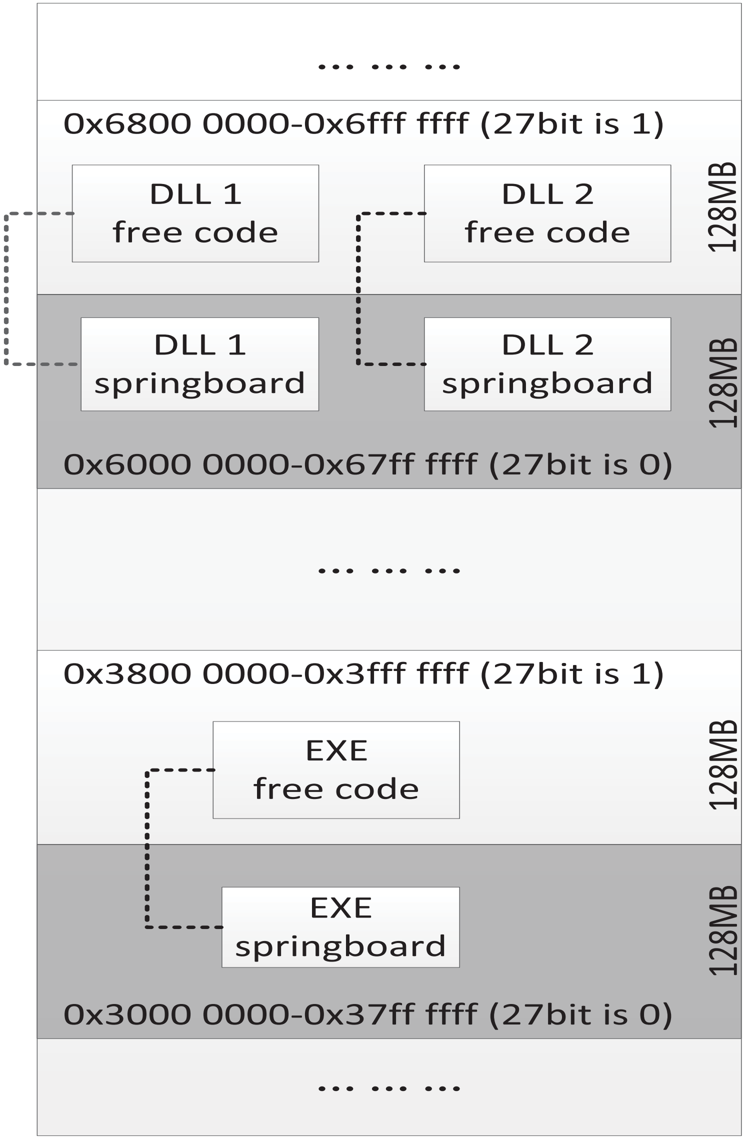
… … …
0x6800 0000-0x6fff ffff (27bit is 1)
DLL 1 free code
DLL 2 free code
128MB
DLL 1 springboard
DLL 2 springboard
128MB
0x6000 0000-0x67ff ffff (27bit is 0)
… … …
0x3800 0000-0x3fff ffff (27bit is 1)
EXE free code
128MB
EXE springboard
128MB
0x3000 0000-0x37ff ffff (27bit is 0)
… … …

128MB segments

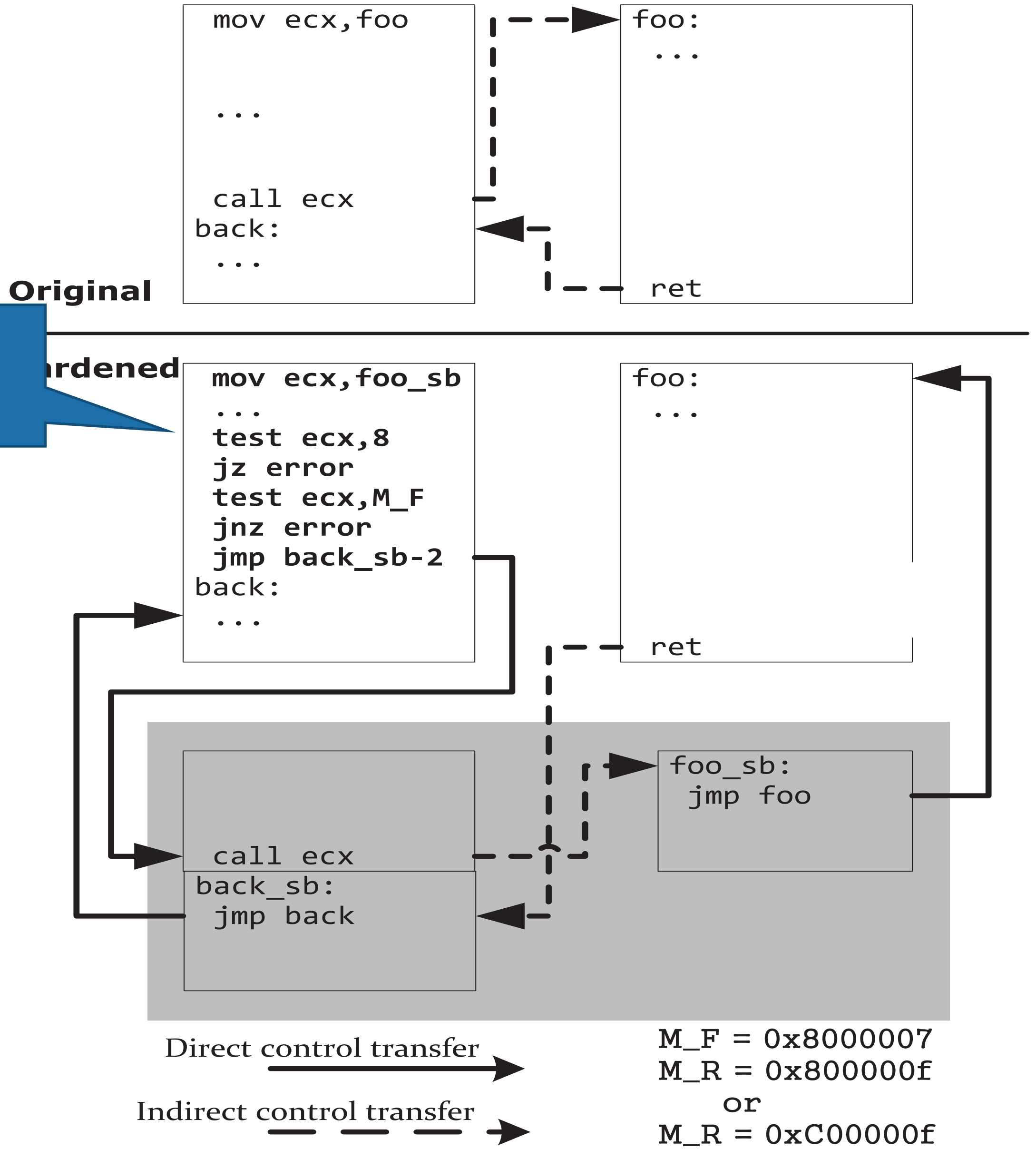
```
Function stu  address
         AN
    M_F = 0x8000007

          0
```

8-byte aligned slots


… … …
0x6800 0000-0x6fff ffff (27bit is 1)
DLL 1 free code
DLL 2 free code
128MB
DLL 1 springboard
DLL 2 springboard
128MB
0x6000 0000-0x67ff ffff (27bit is 0)
… … …
0x3800 0000-0x3fff ffff (27bit is 1)
EXE free code
128MB
EXE springboard
128MB
0x3000 0000-0x37ff ffff (27bit is 0)
… … …

Fast checks



Original

```
 mov ecx,foo

 ...

 call ecx
back:
 ...
```

```
foo:
 ...

 ret
```

rdened

```
 mov ecx,foo_sb
 ...
 test ecx,8
 jz error
 test ecx,M_F
 jnz error
 jmp back_sb-2
back:
 ...
```

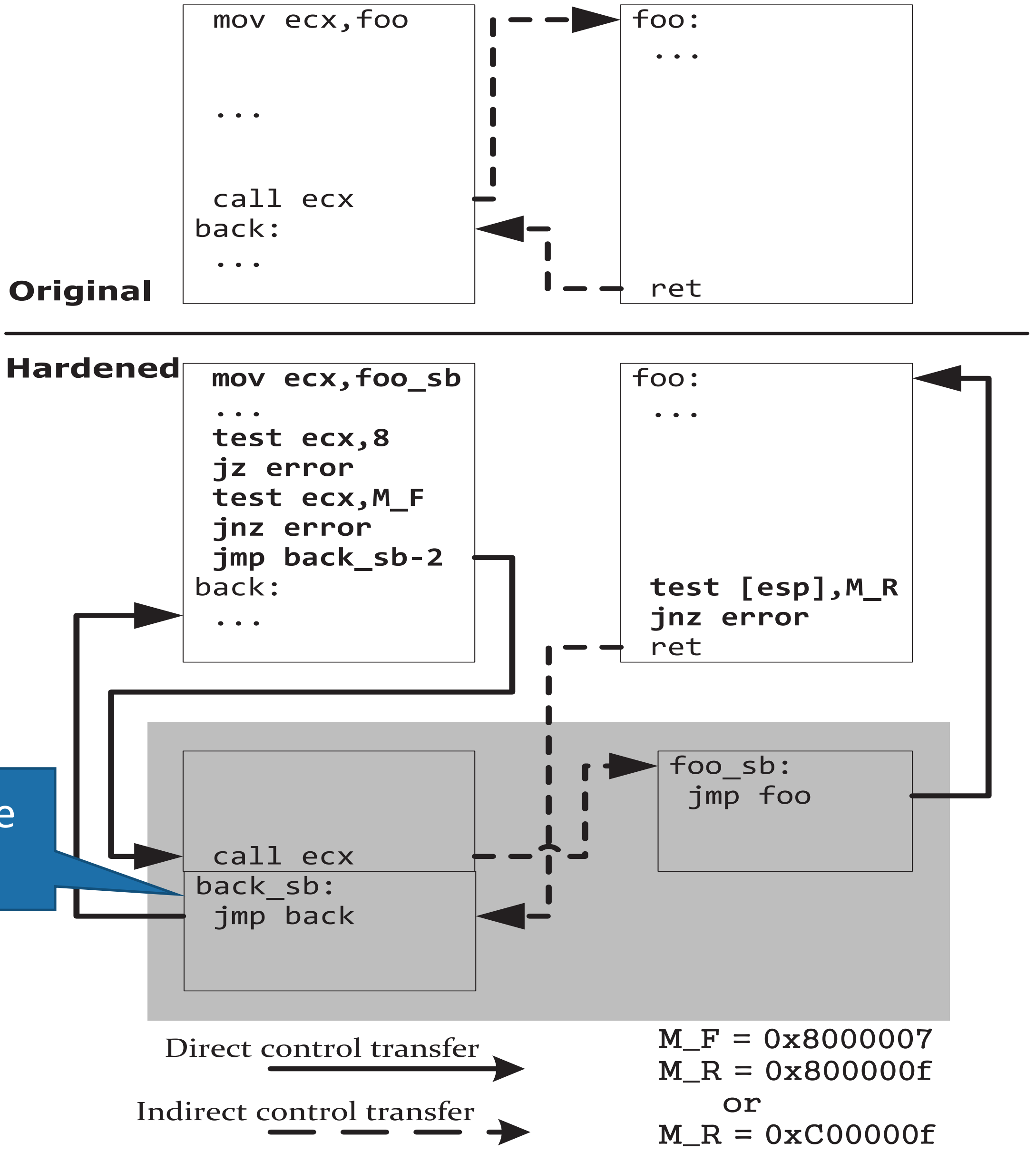
```
foo:
 ...

 ret
```

```
 call ecx
back_sb:
 jmp back
```

```
foo_sb:
 jmp foo
```

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:
...
```

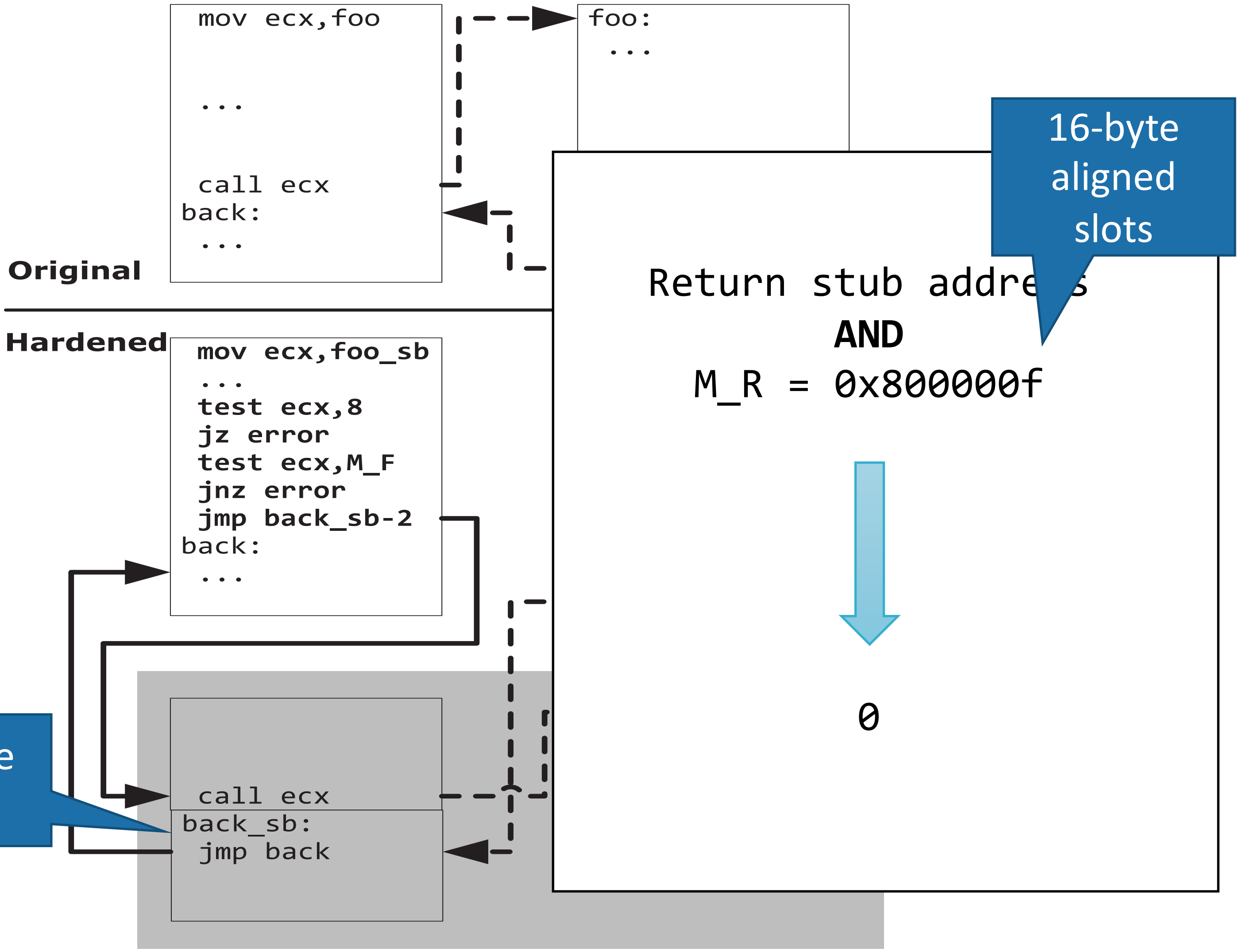
```
foo:
...
test [esp],M_R
jnz error
ret
```

```
call ecx
back_sb:
 jmp back
```

```
foo_sb:
 jmp foo
```

```
M_F = 0x8000007
M_R = 0x800000f
     or
M_R = 0xC00000f
```

Return stubs are also aligned



```
mov ecx,foo
...
call ecx
back:
...
```

```
foo:
 ...
```

```
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 stub addre s
AND
M_R = 0x800000f

0

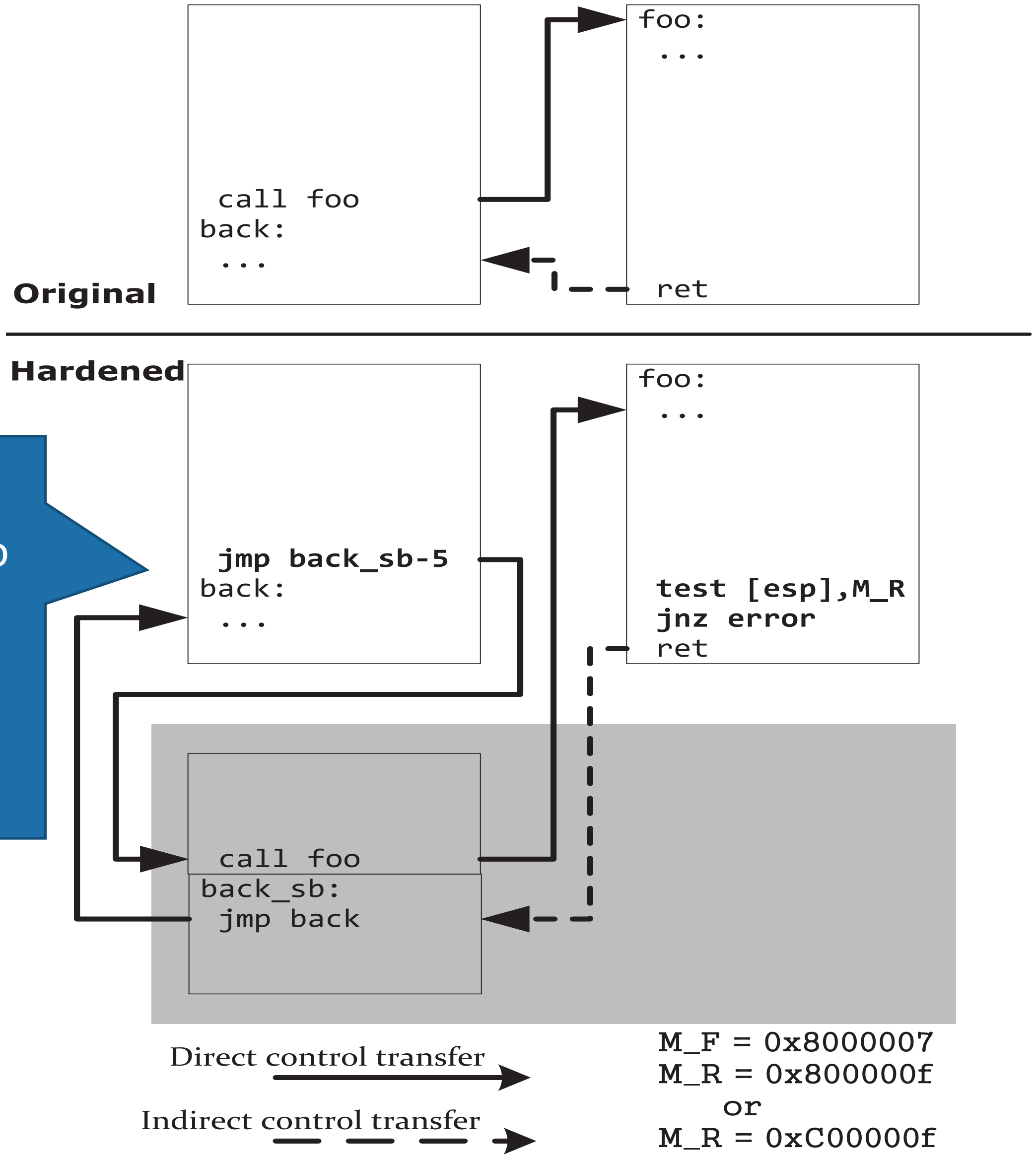
16-byte aligned slots

Return stubs are also aligned

Direct control transfer

Indirect control transfer

M_F = 0x8000007
M_R = 0x800000f
or
M_R = 0xC00000f



```
 call foo
back:
 ...
```

```
foo:
 ...

 ret
```

```
 jmp back_sb-5
back:
 ...
```

```
foo:
 ...

 test [esp],M_R
 jnz error
 ret
```

```
 call foo
back_sb:
 jmp back
```

```
M_F = 0x8000007
M_R = 0x800000f
    or
M_R = 0xC00000f
```

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

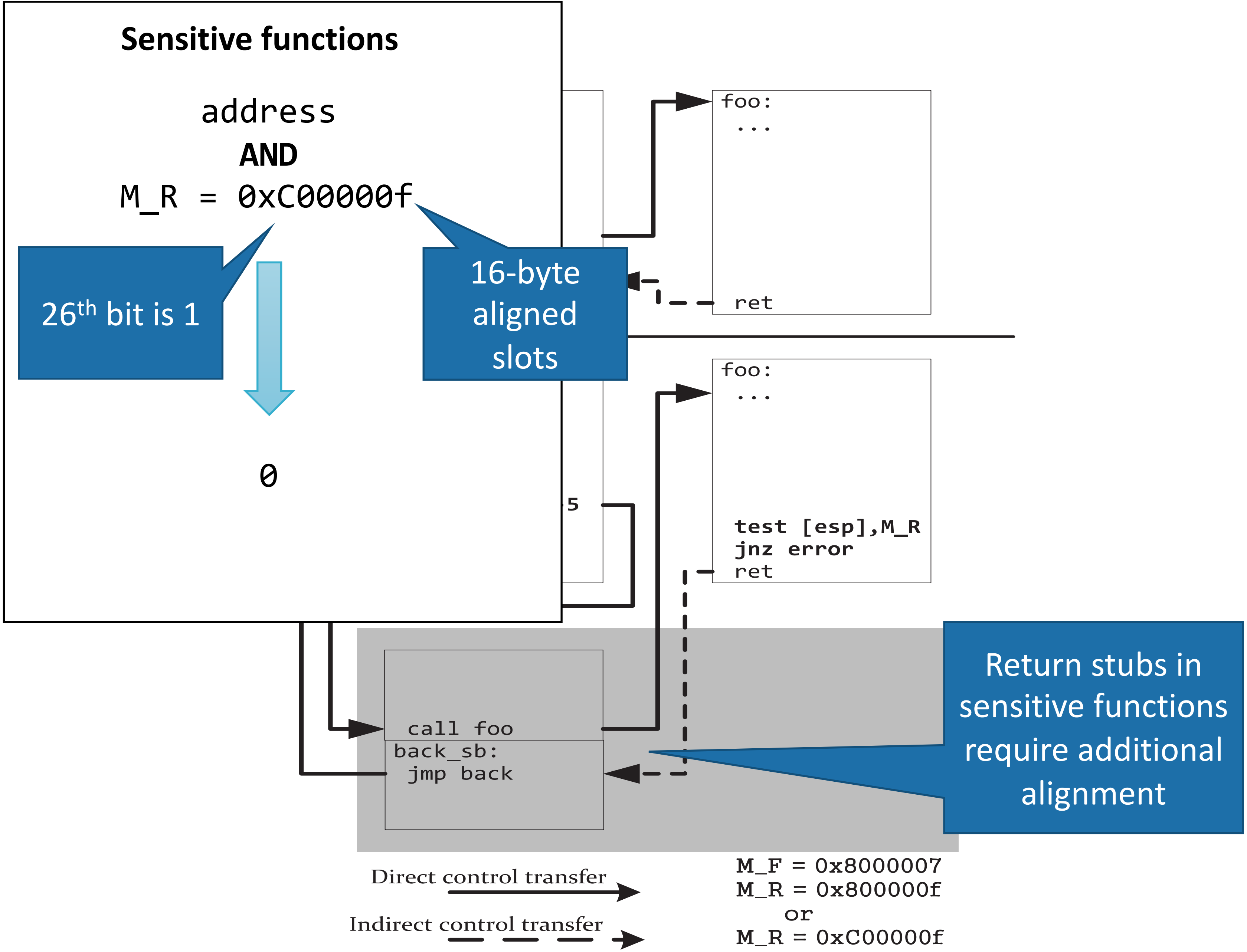
Sensitive functions

```
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

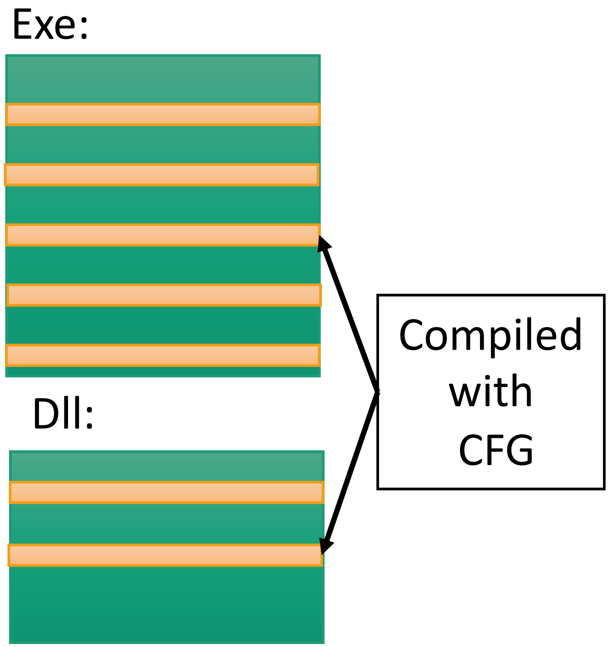
```
M_F = 0x8000007
M_R = 0x800000f
     or
M_R = 0xC00000f
```

Microsoft's Control-Flow Guard

Included in MS Visual Studio

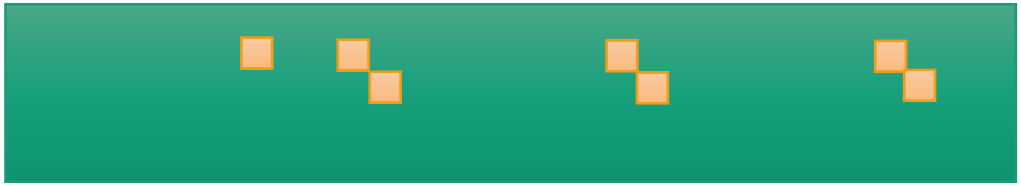
Inserts control-flow checks before indirect calls during compilation

A bitmap marks the allowed targets



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

bitmap:



1 bit per 8 or 16-byte slot

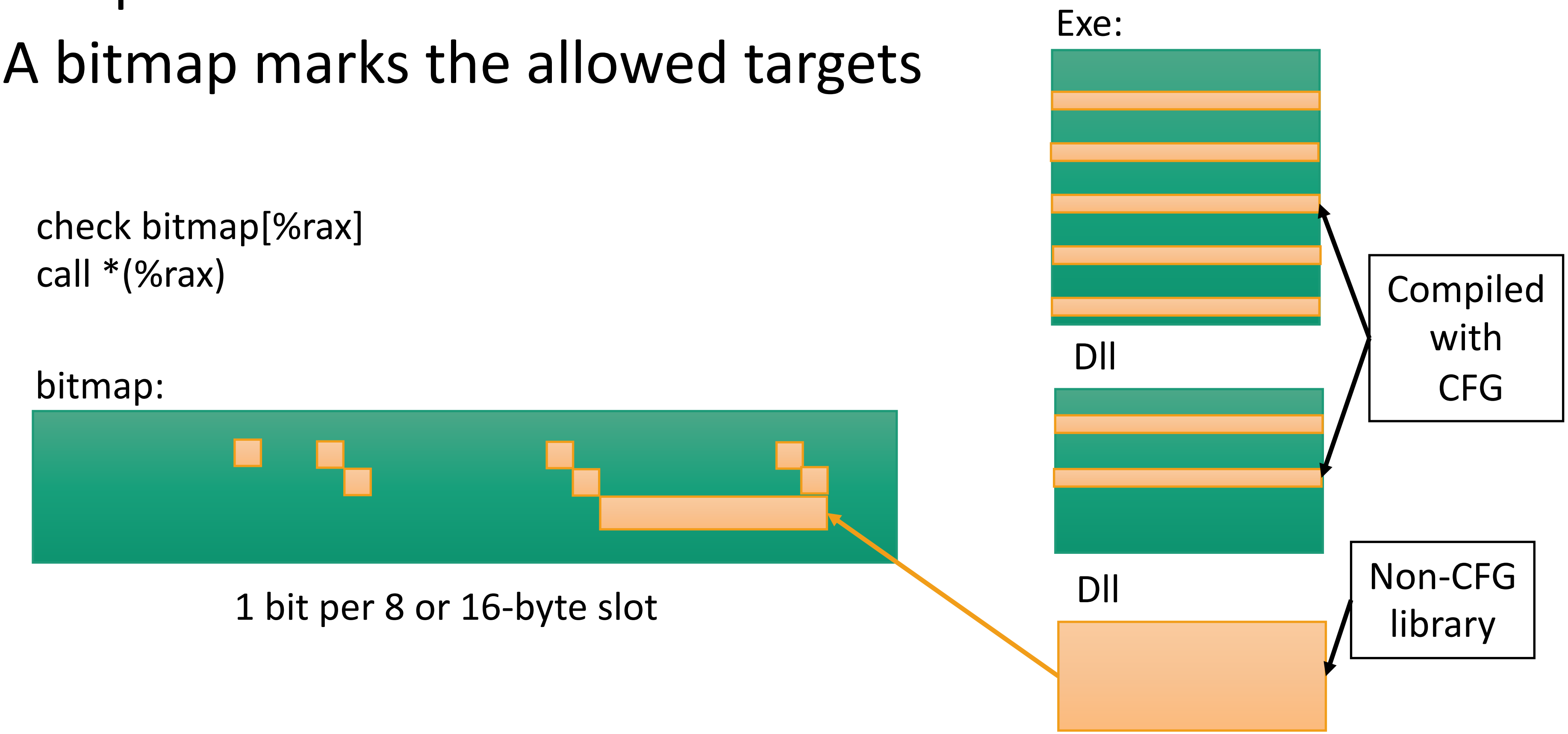
Microsoft's Control-Flow Guard

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)
```



Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

Fine-grained code randomization

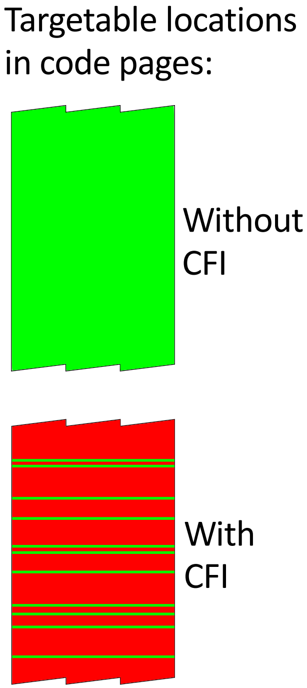
JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

Reachable Targets Under CFI

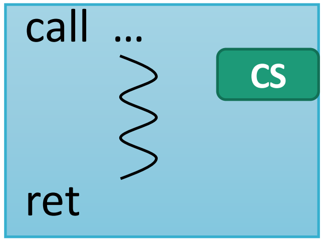
Most instructions cannot be targeted **(> 98%)**



What is Left

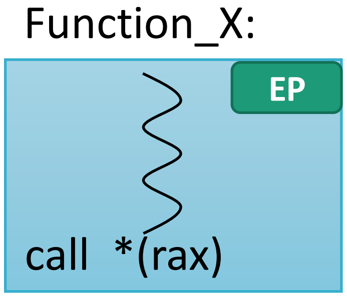
Call Sites (**CS**)

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

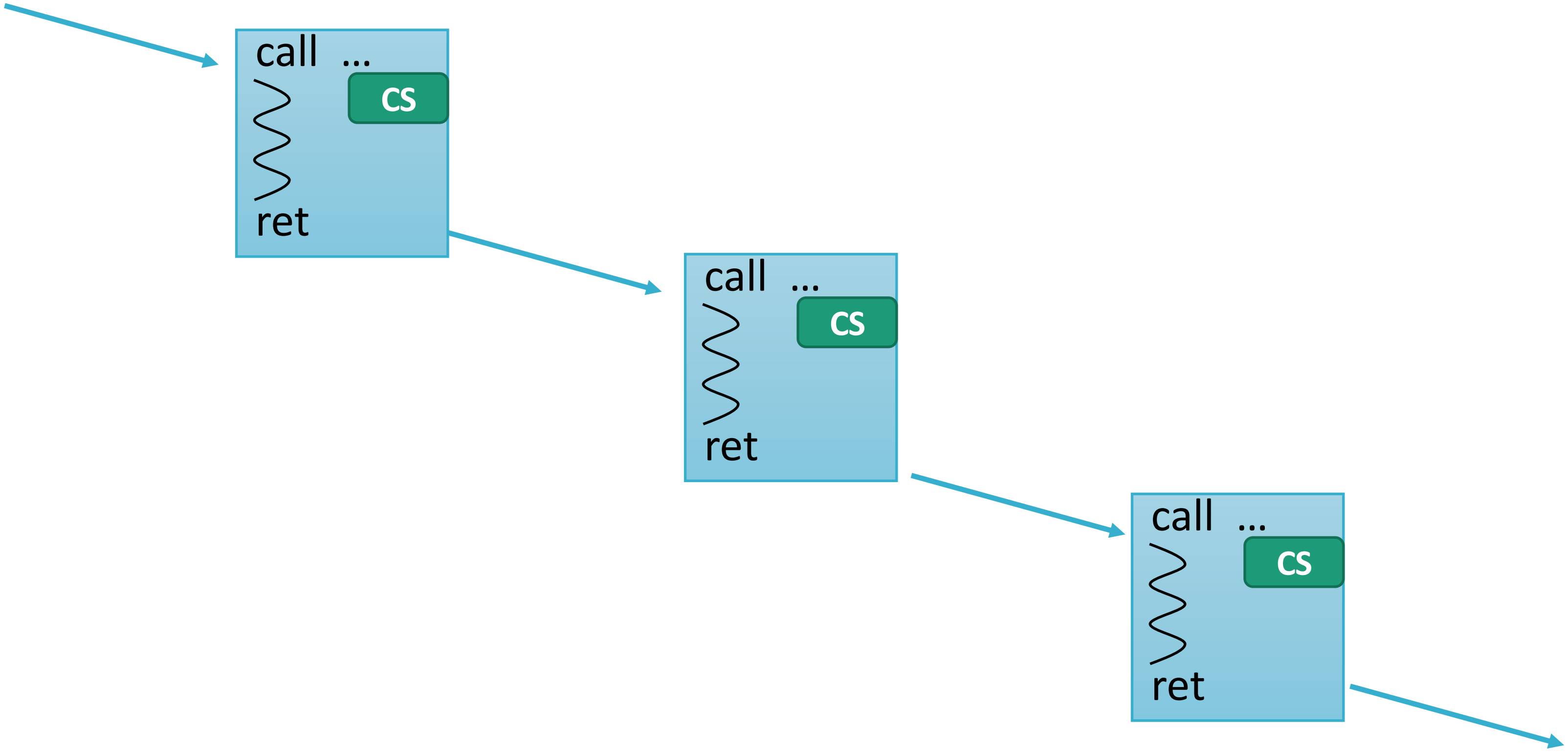


Function Entry Points (**EP)**

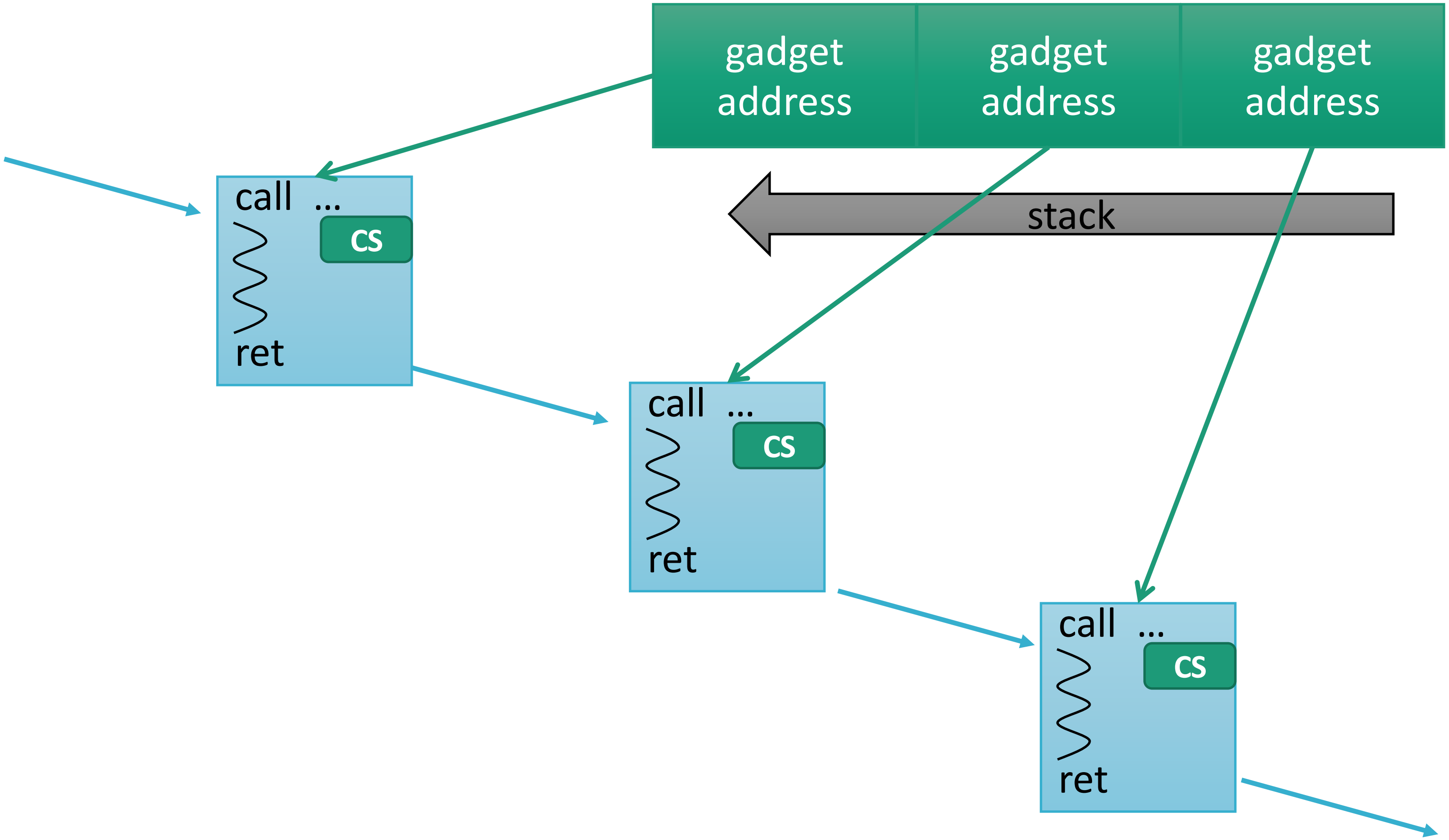
- Targetable by **indirect call** and **indirect jump** instructions
- EP gadgets
- Call Oriented Programming (COP)



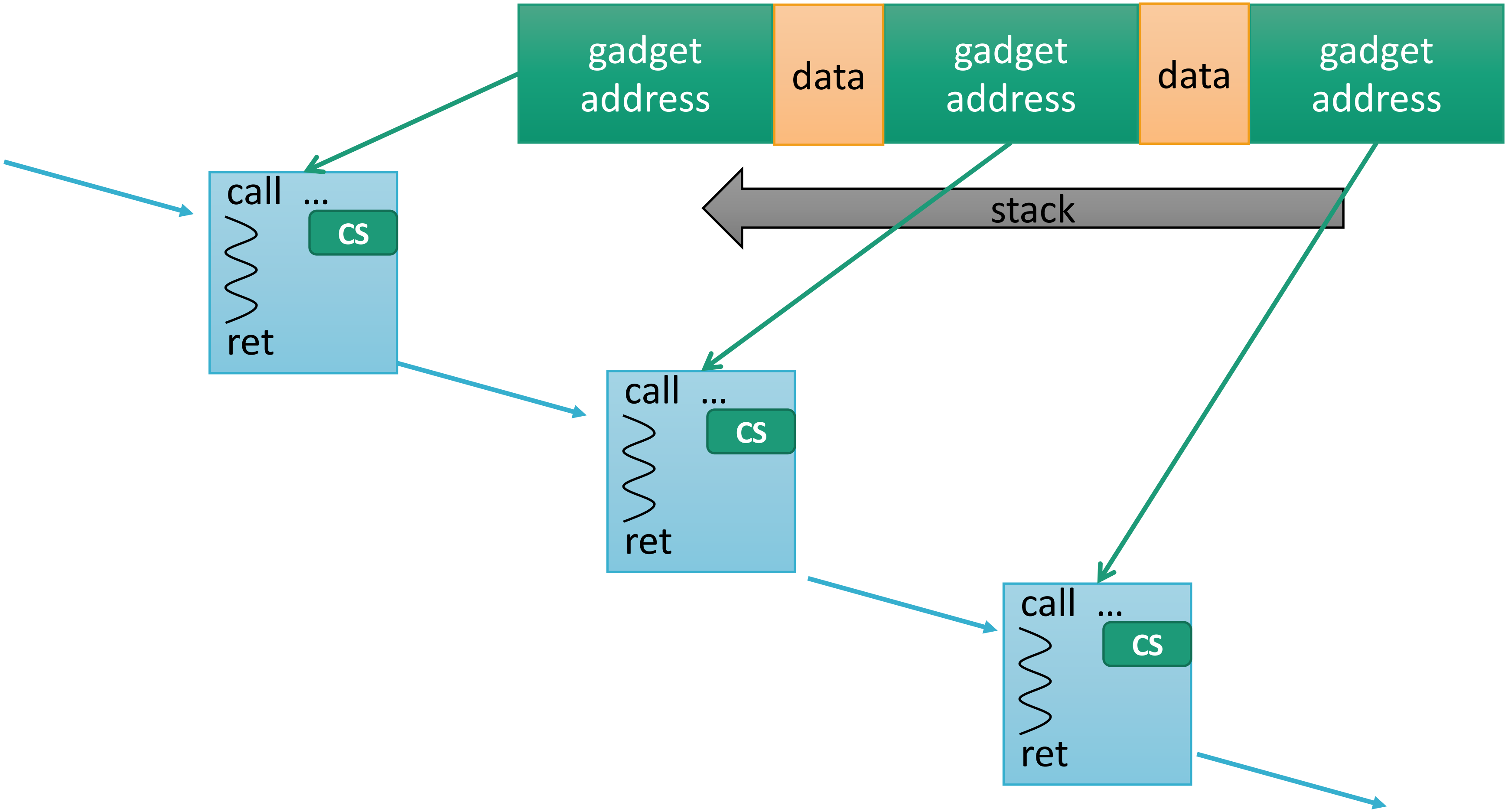
CS gadgets: Linking



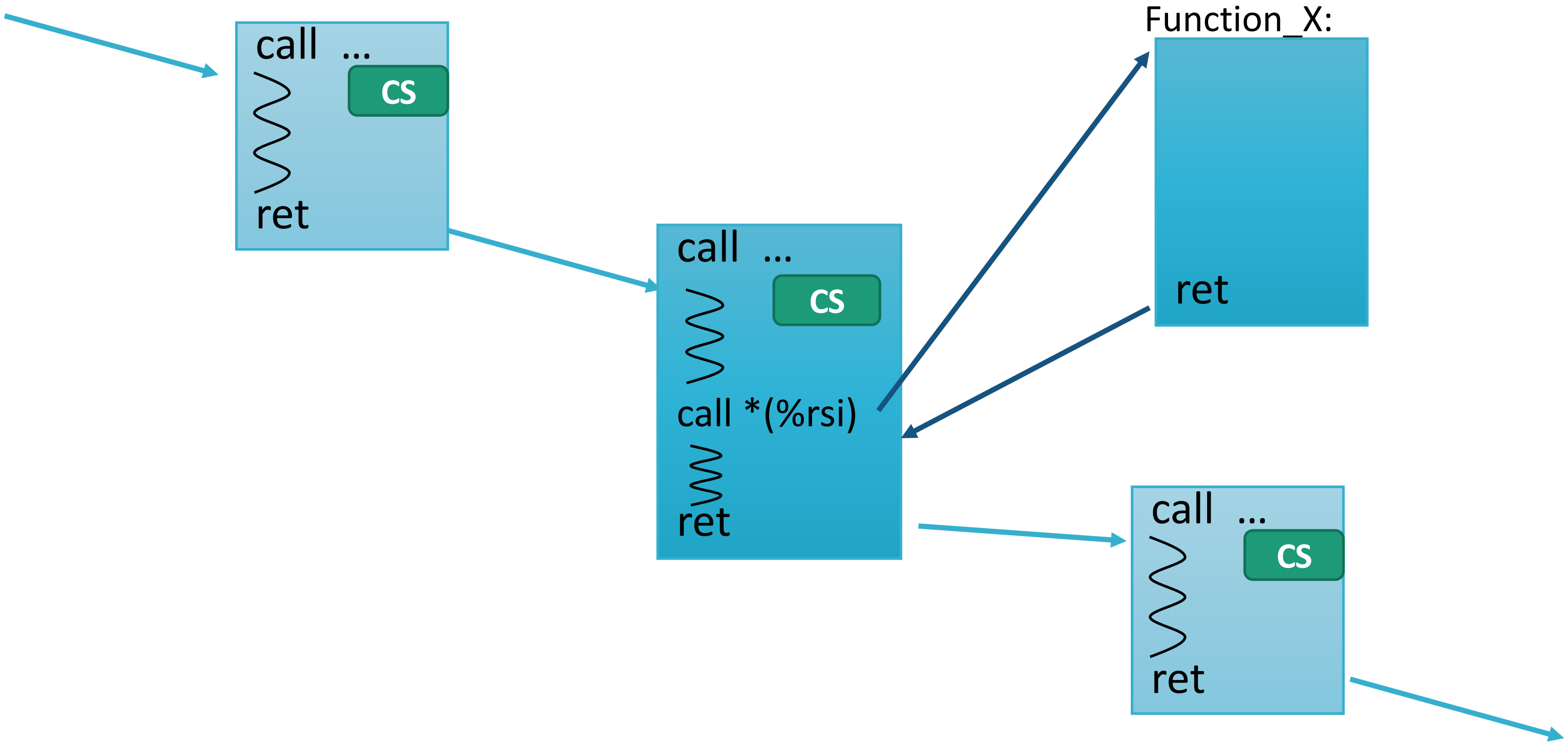
CS gadgets: Linking



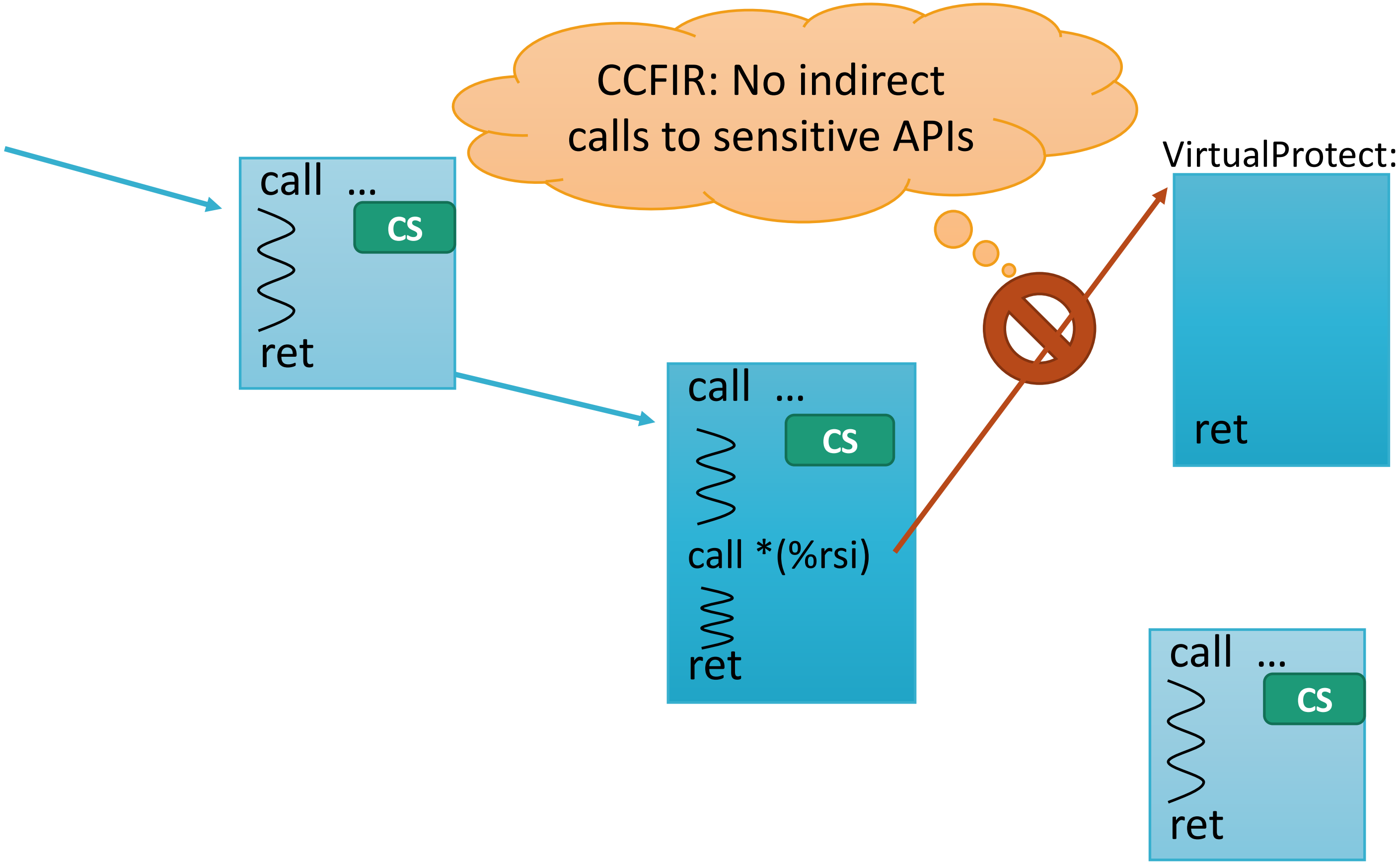
CS gadgets: Linking



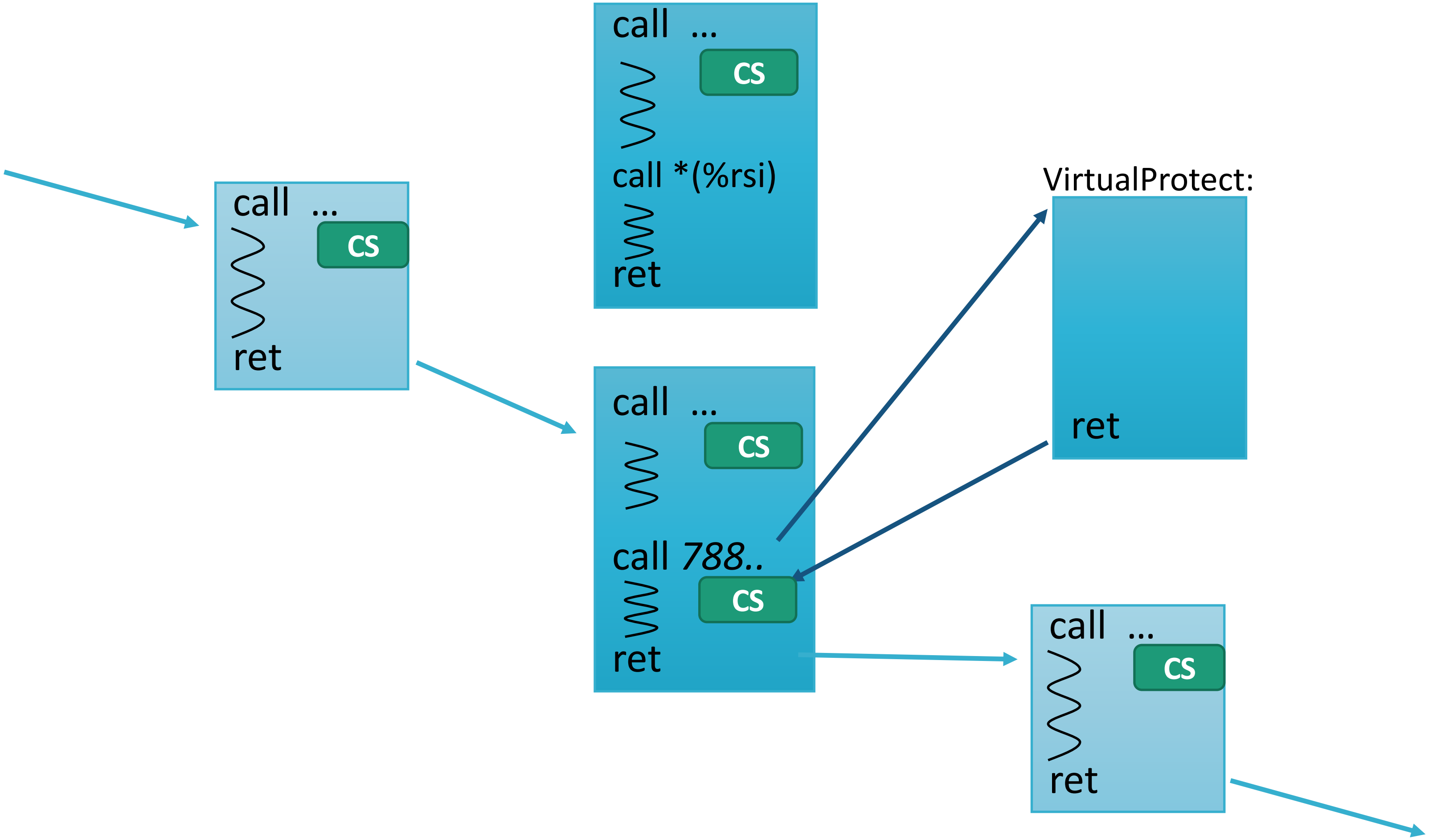
CS gadgets: Calling Functions



CS gadgets: Calling Sensitive Functions

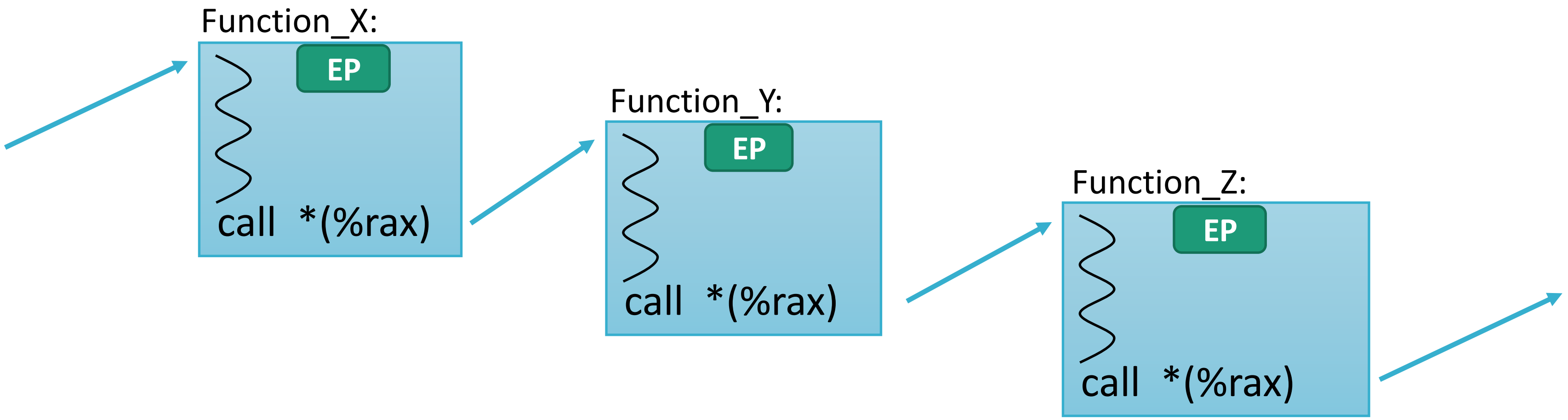


CS gadgets: Calling Sensitive Functions

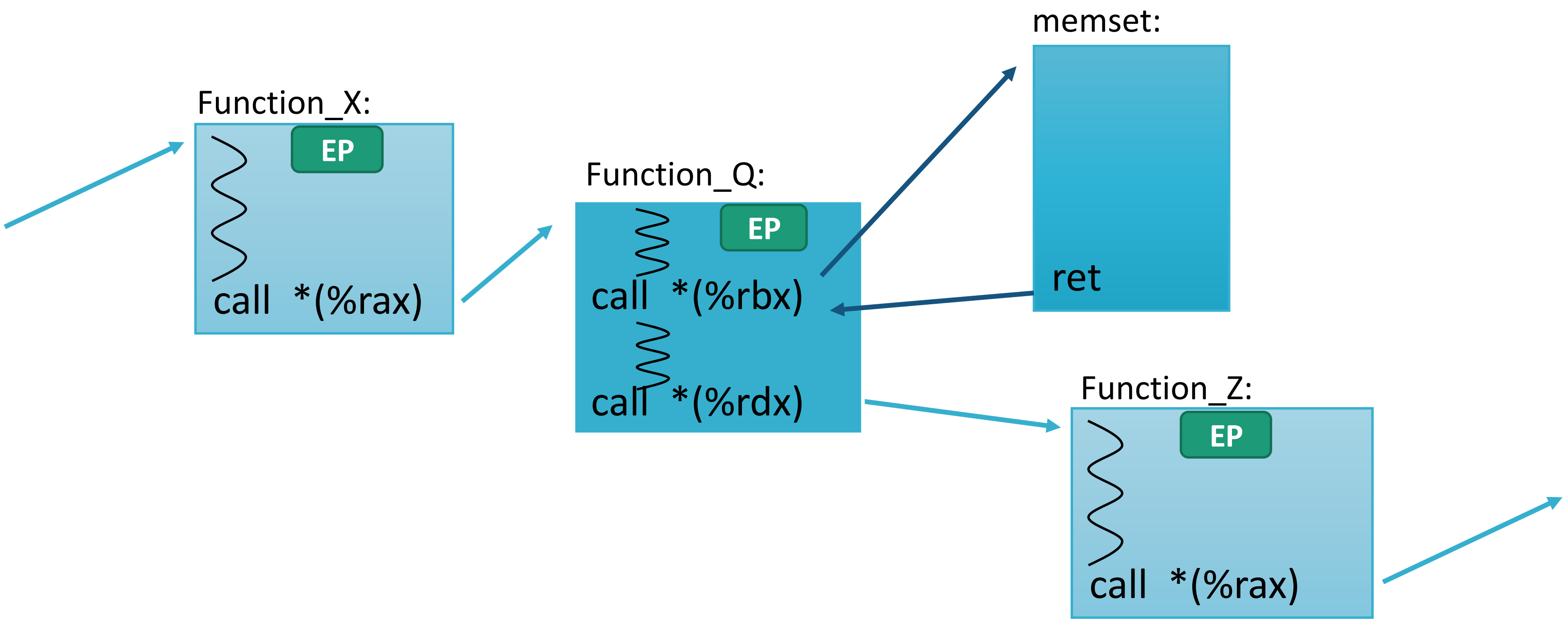


EP gadgets: Linking

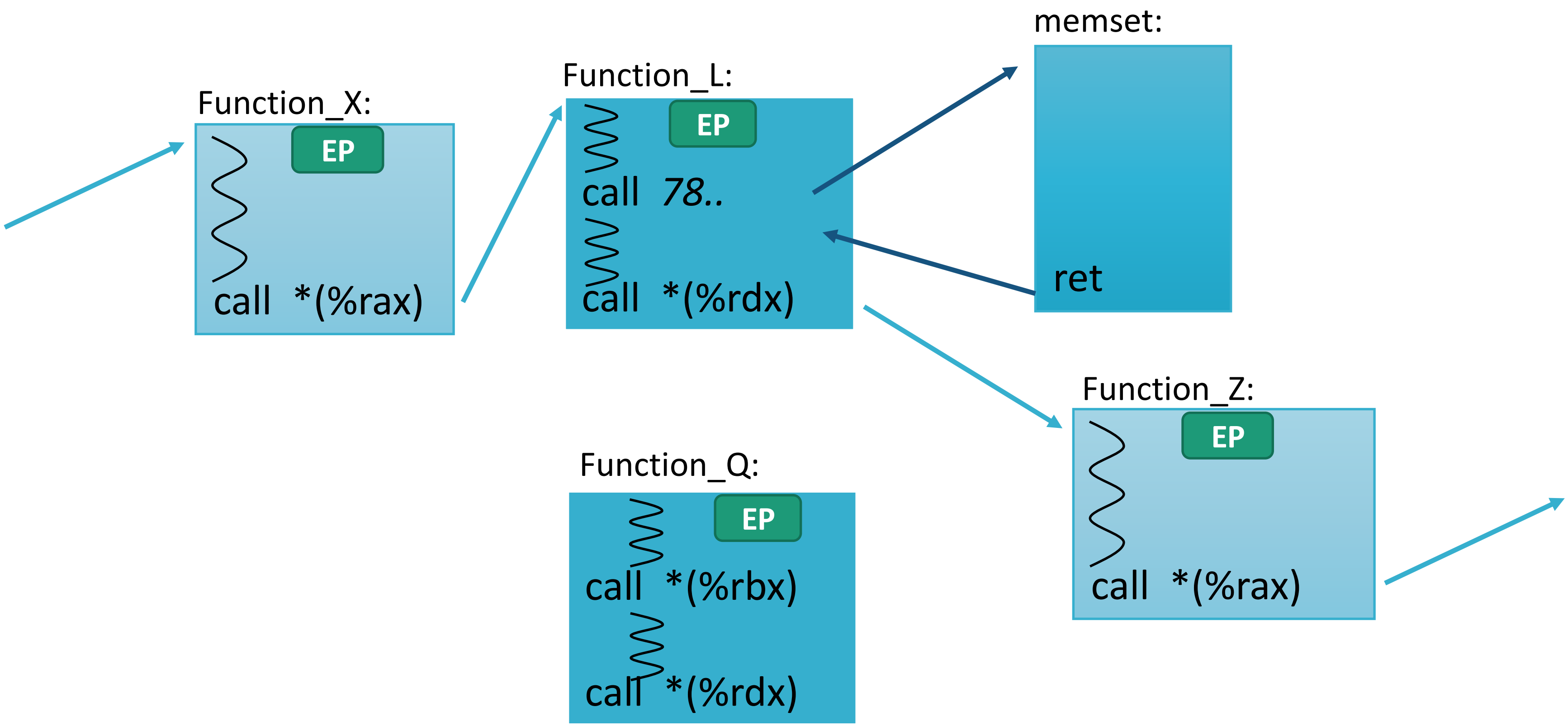
Chaining is significantly harder



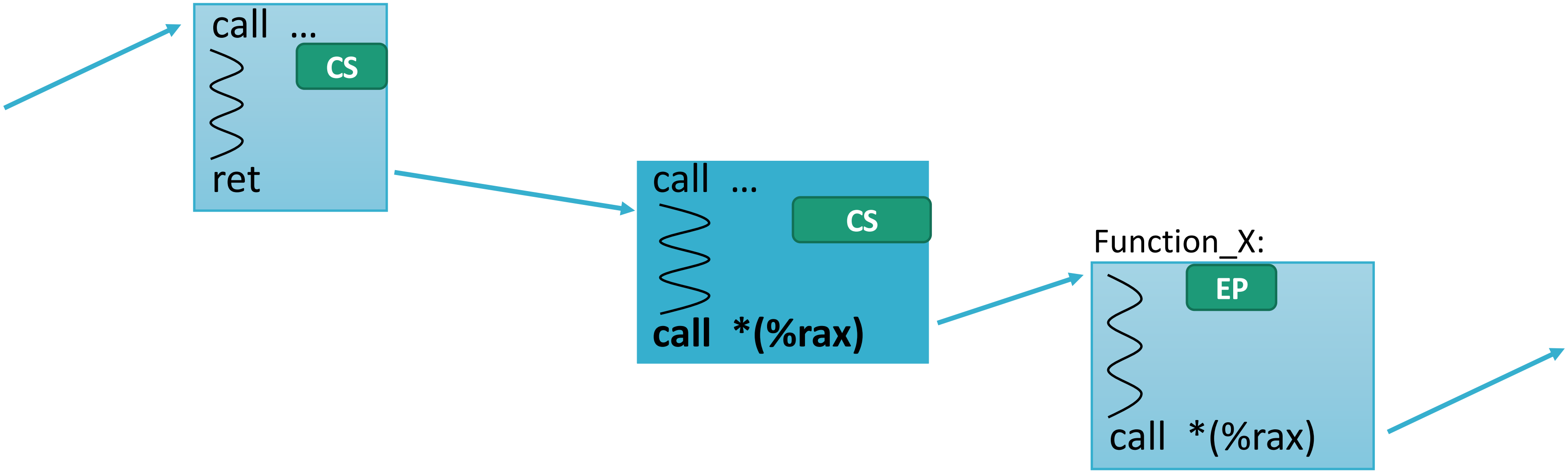
EP gadgets: Calling Functions



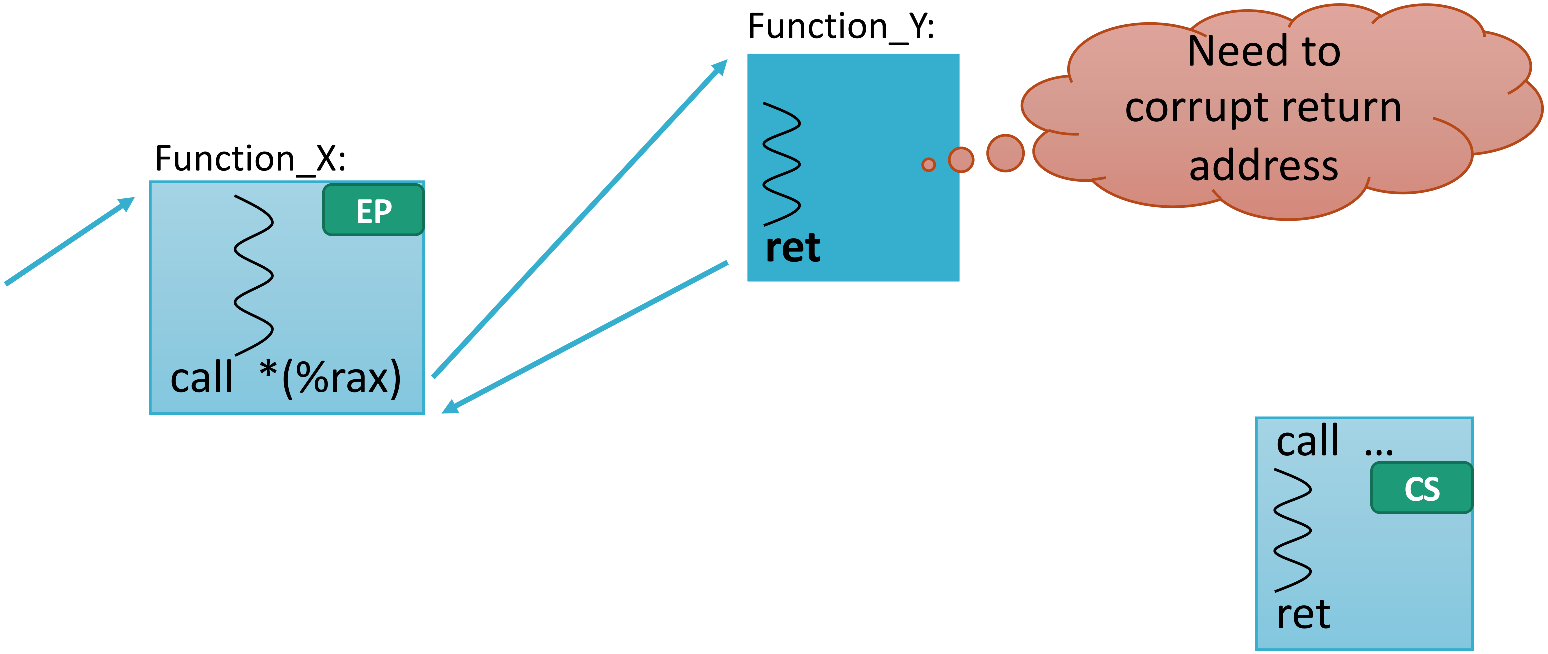
EP gadgets: Calling Functions



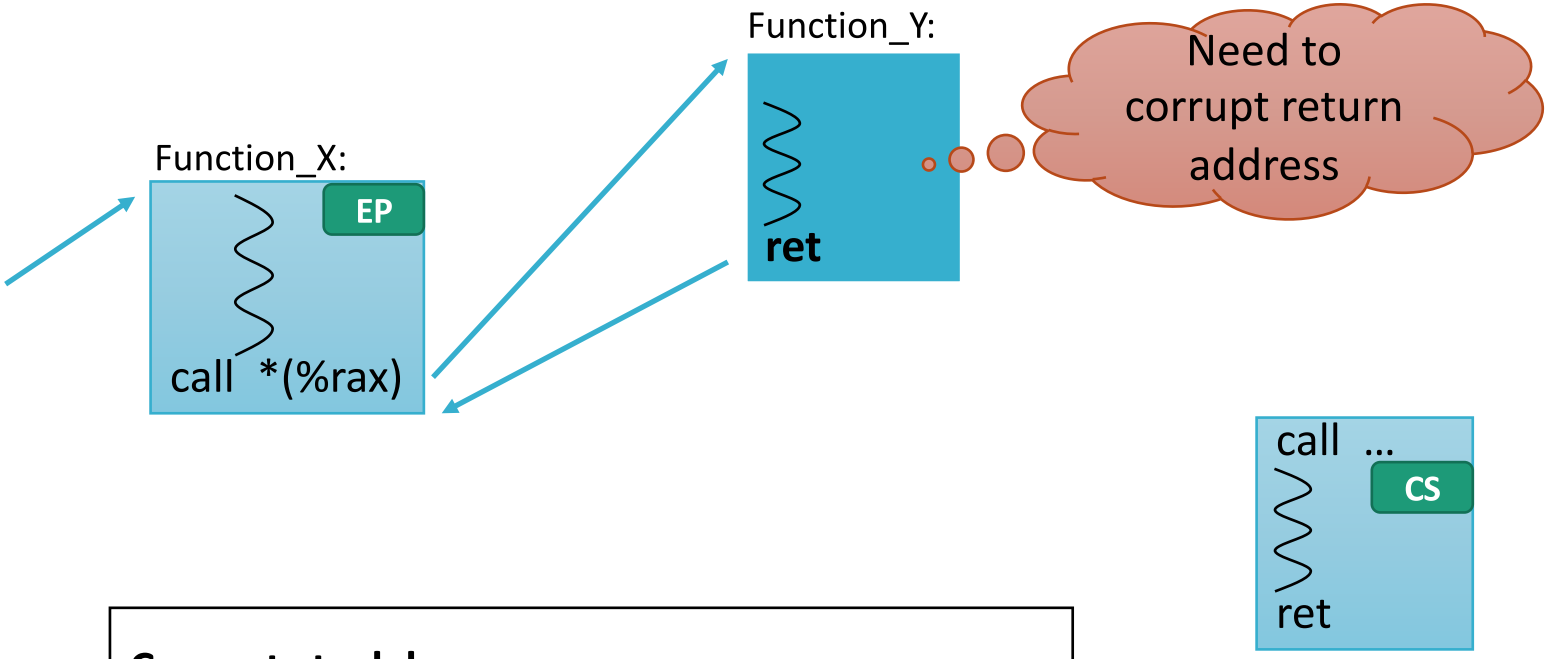
Switch Control: CS → EP



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

Finer-Grained CFI

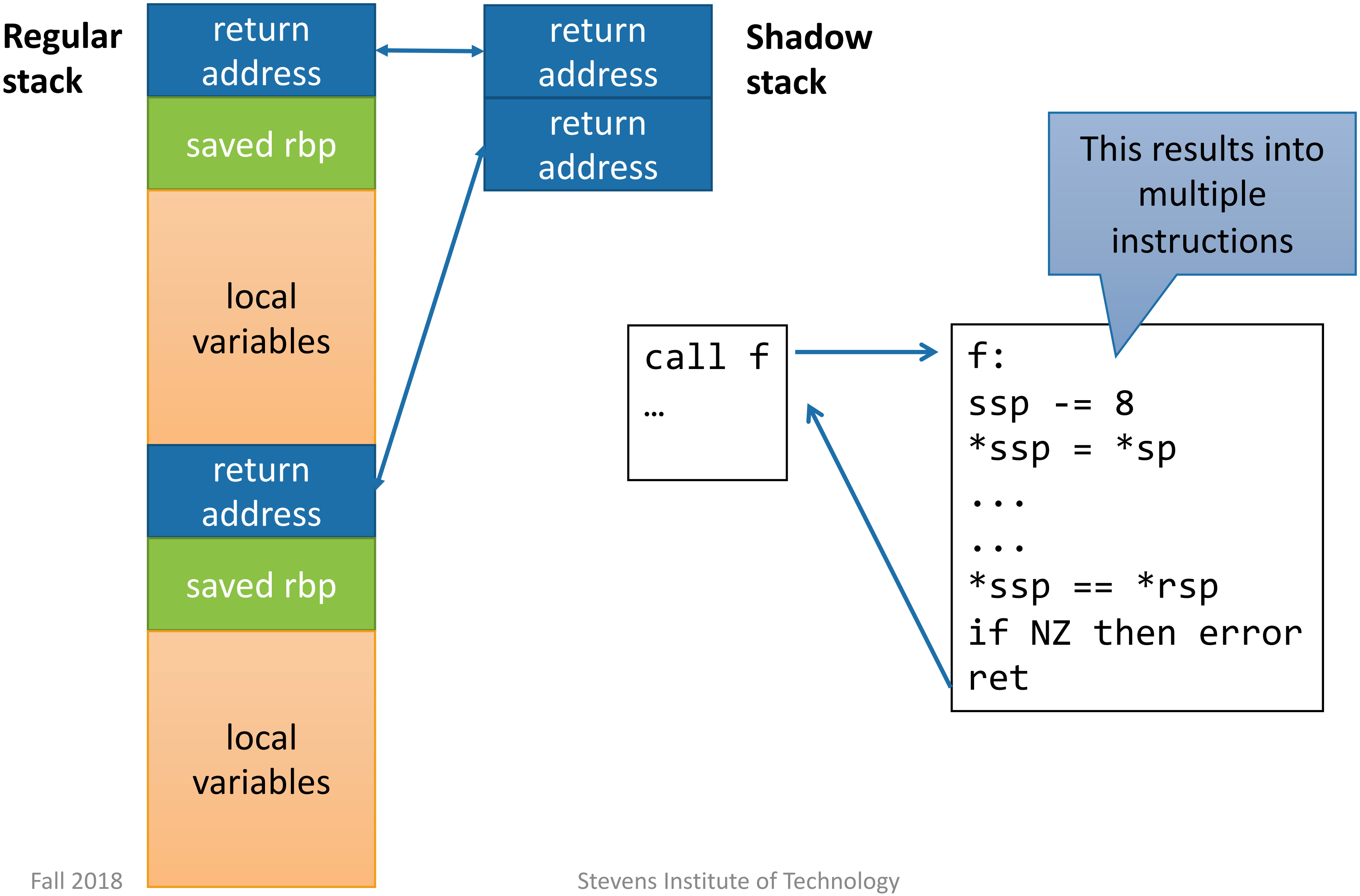
Various approaches to improve CFI

- 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

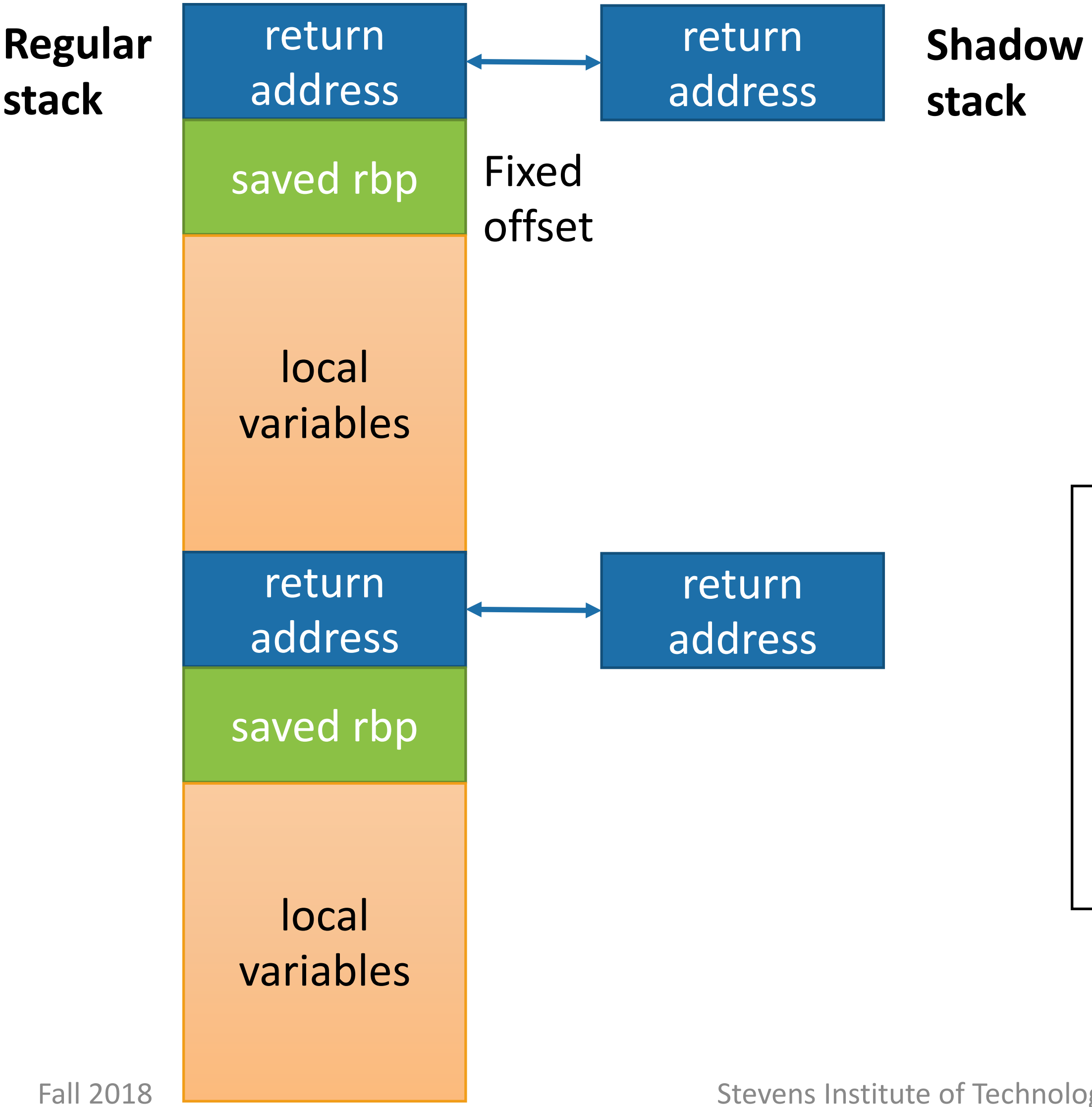
Shadow Stacks



```
call f
…
```

```
f:
ssp -= 8
*ssp = *sp
...
...
*ssp == *rsp
if NZ then error
ret
```

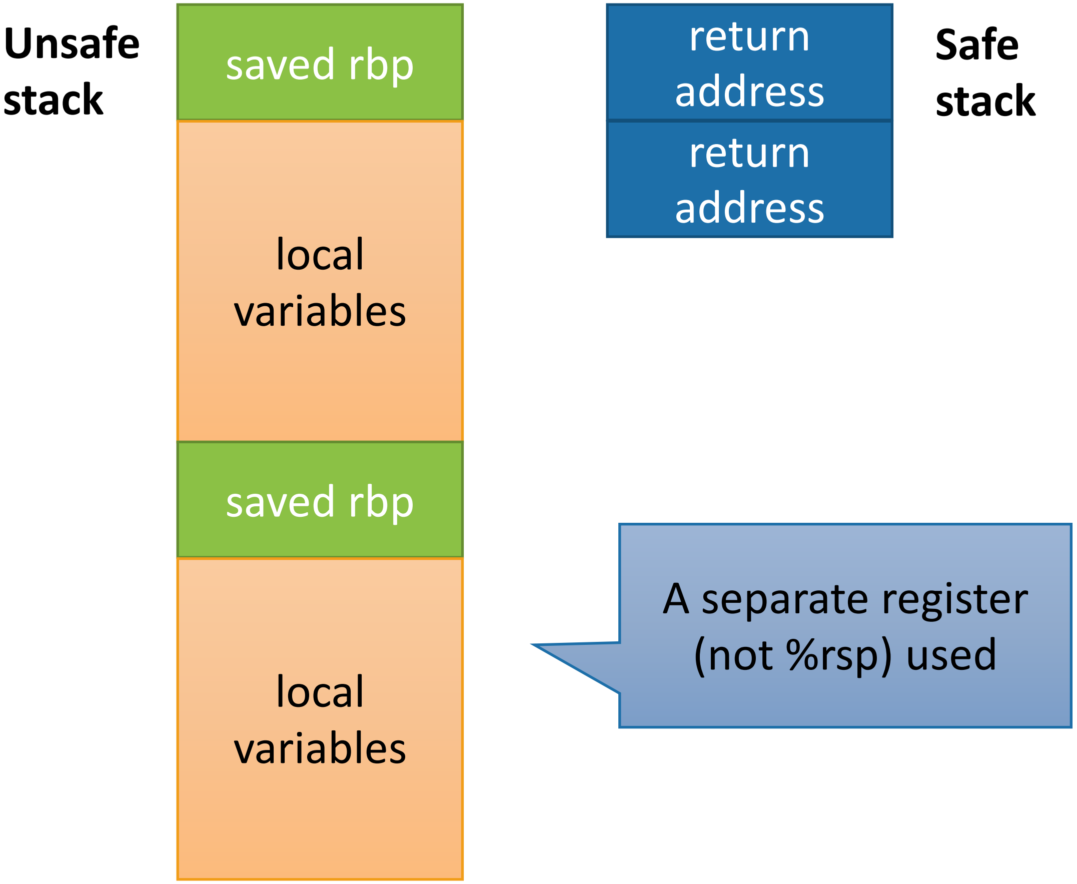
Shadow Stacks



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

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

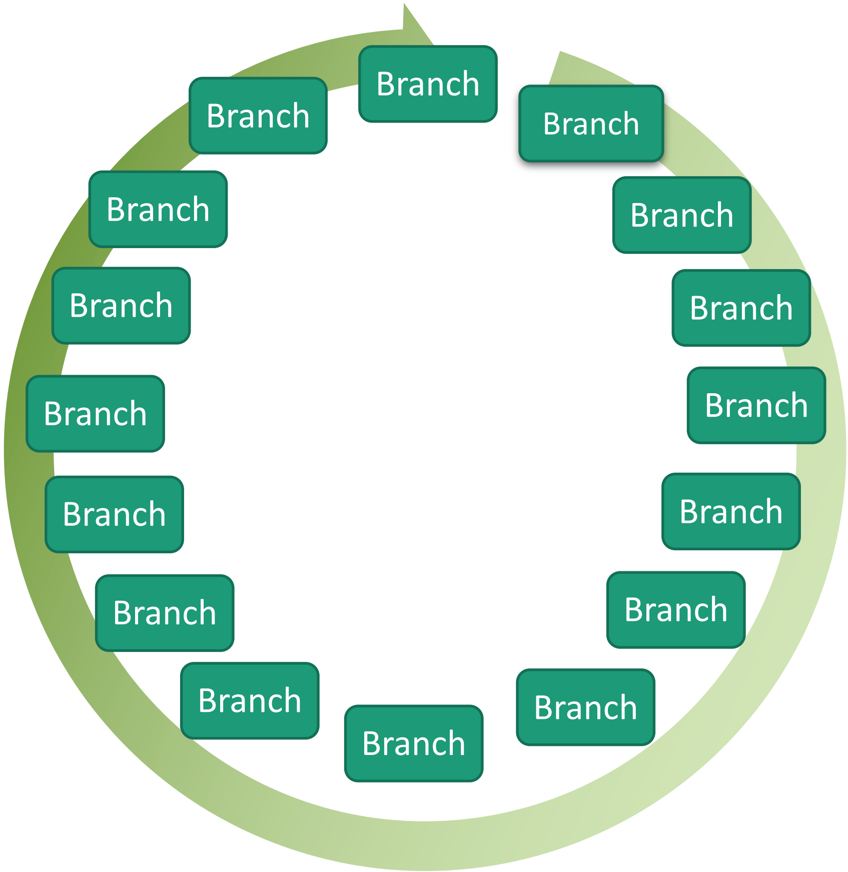
- Ring-buffer structure

Holds last 16 entries

- Store source:destination

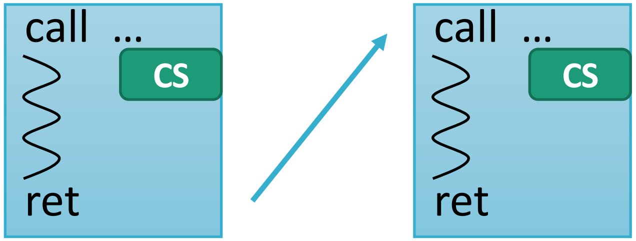
Configurable

- Example: Store only indirect calls

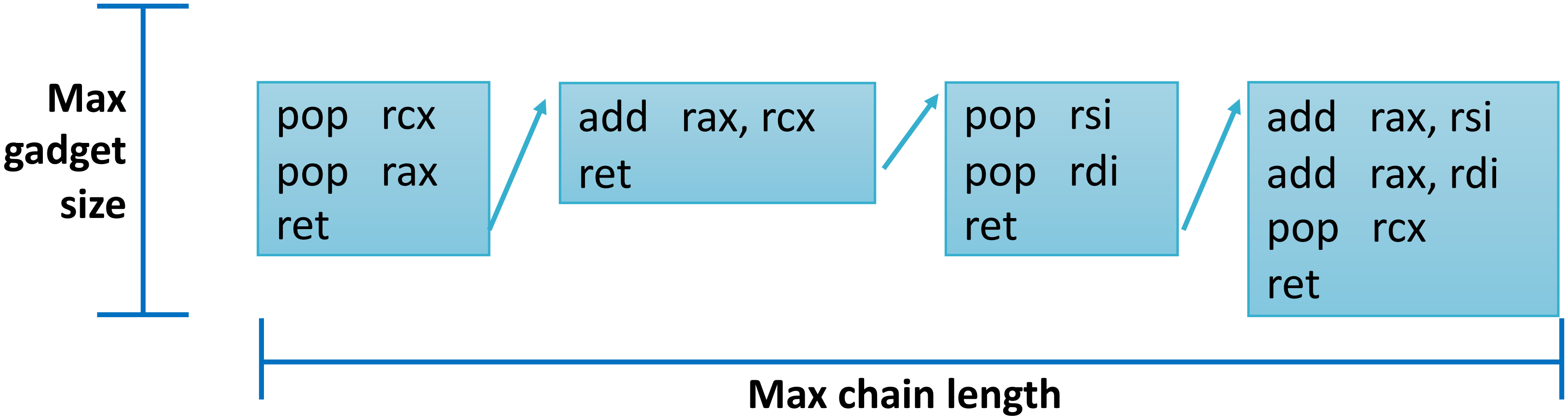


Detection Approach

1. Returns must target call sites



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



Max gadget size

```
pop   rcx
pop   rax
ret
```

```
add   rax, rcx
ret
```

```
pop   rsi
pop   rdi
ret
```

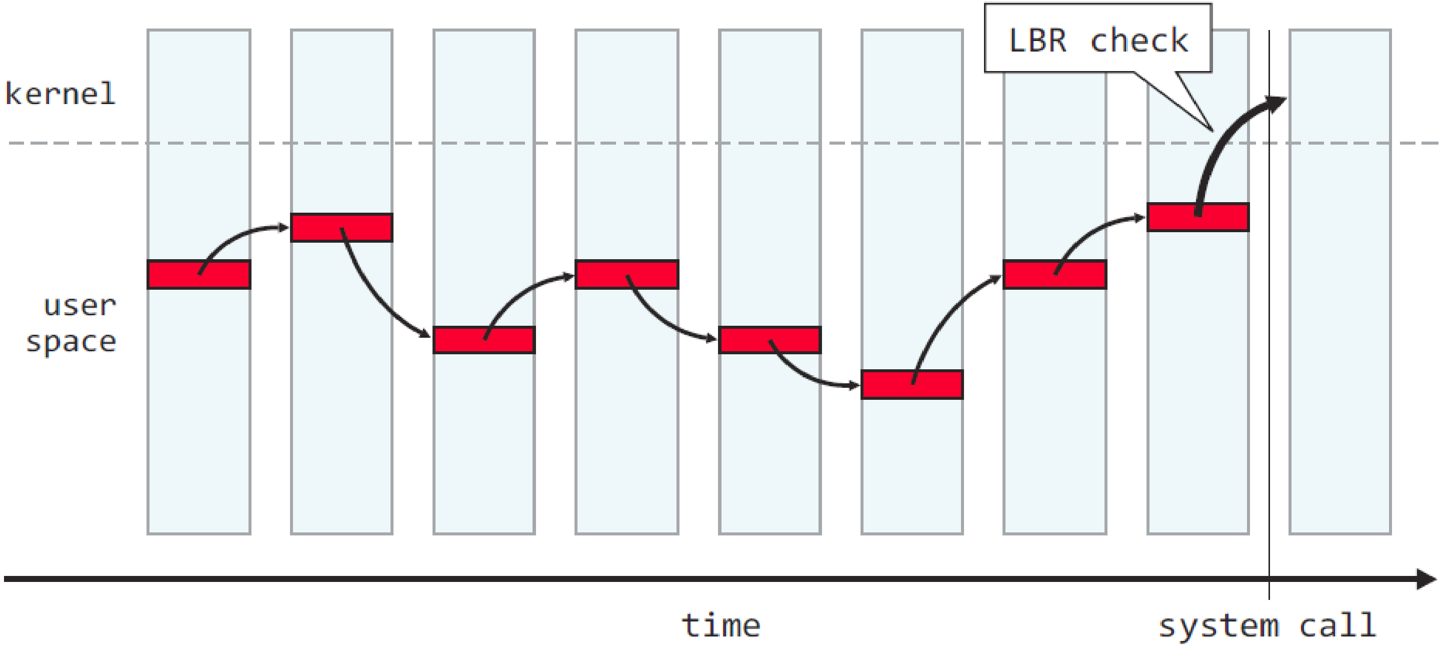
```
add   rax, rsi
add   rax, rdi
pop   rcx
ret
```

Max chain length

Fast Checks

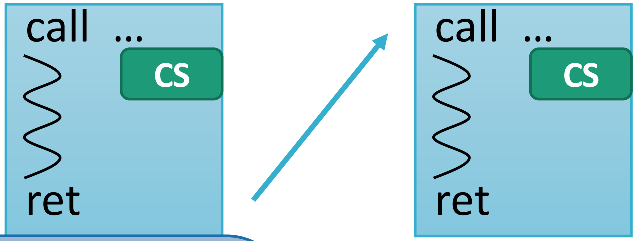
The payload will eventually interact with the OS through system calls

- Check for abnormal control transfers on system call entry



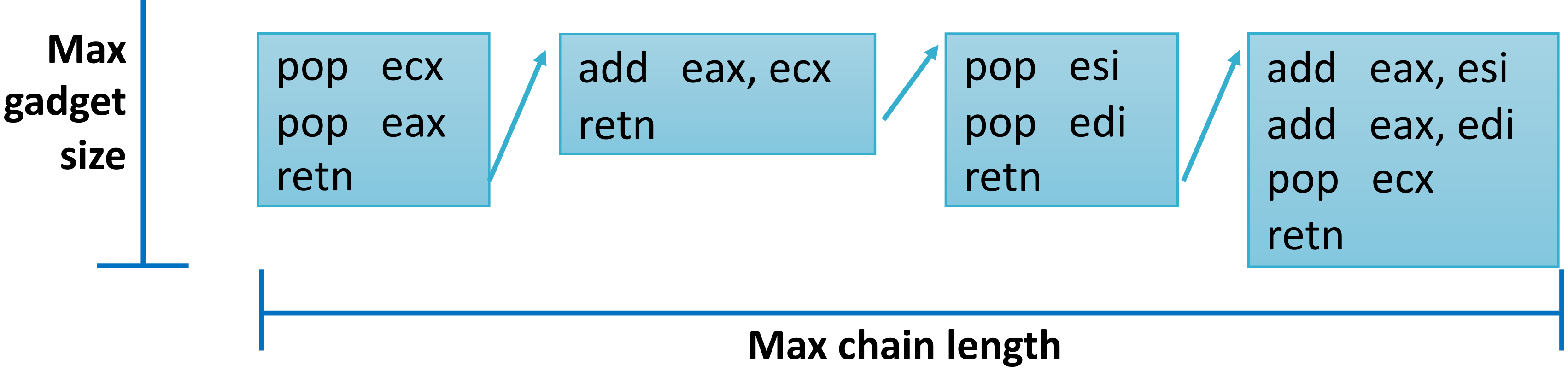
Detection Approach

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?**



```
pop   ecx
pop   eax
retn
```

```
add   eax, ecx
retn
```

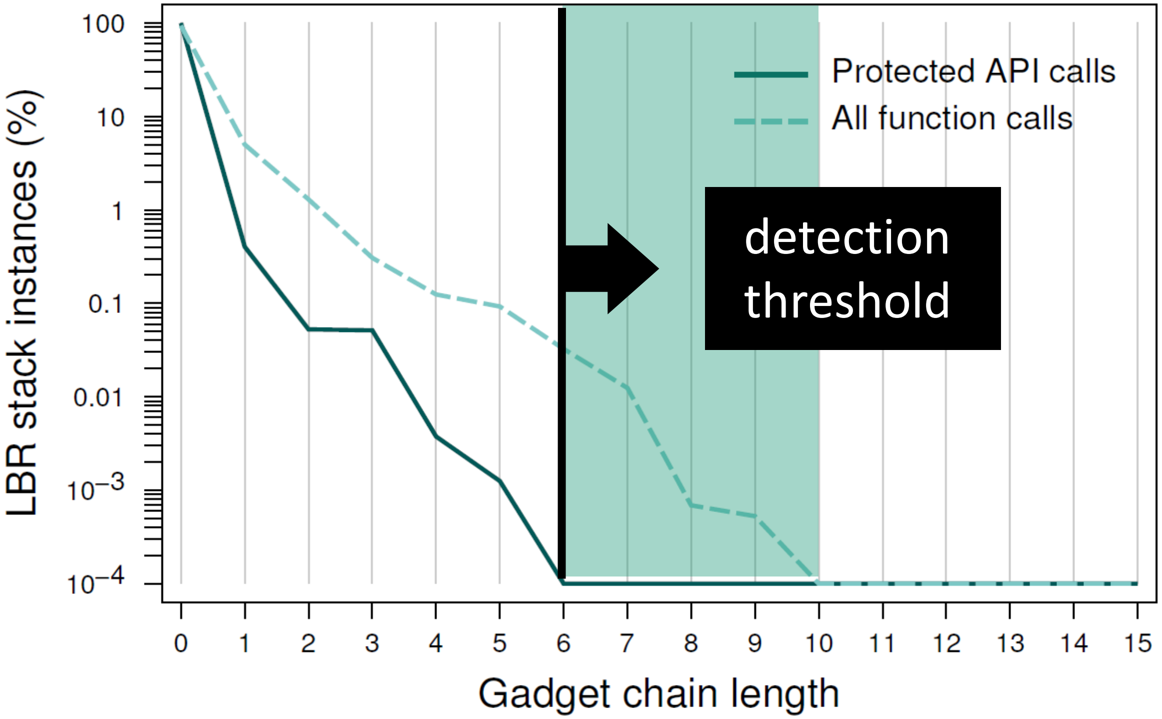
```
pop   esi
pop   edi
retn
```

```
add   eax, esi
add   eax, edi
pop   ecx
retn
```

Establishing The Parameters

Set max gadget size to 19 (<20)

Evaluate max chain length **experimentally**



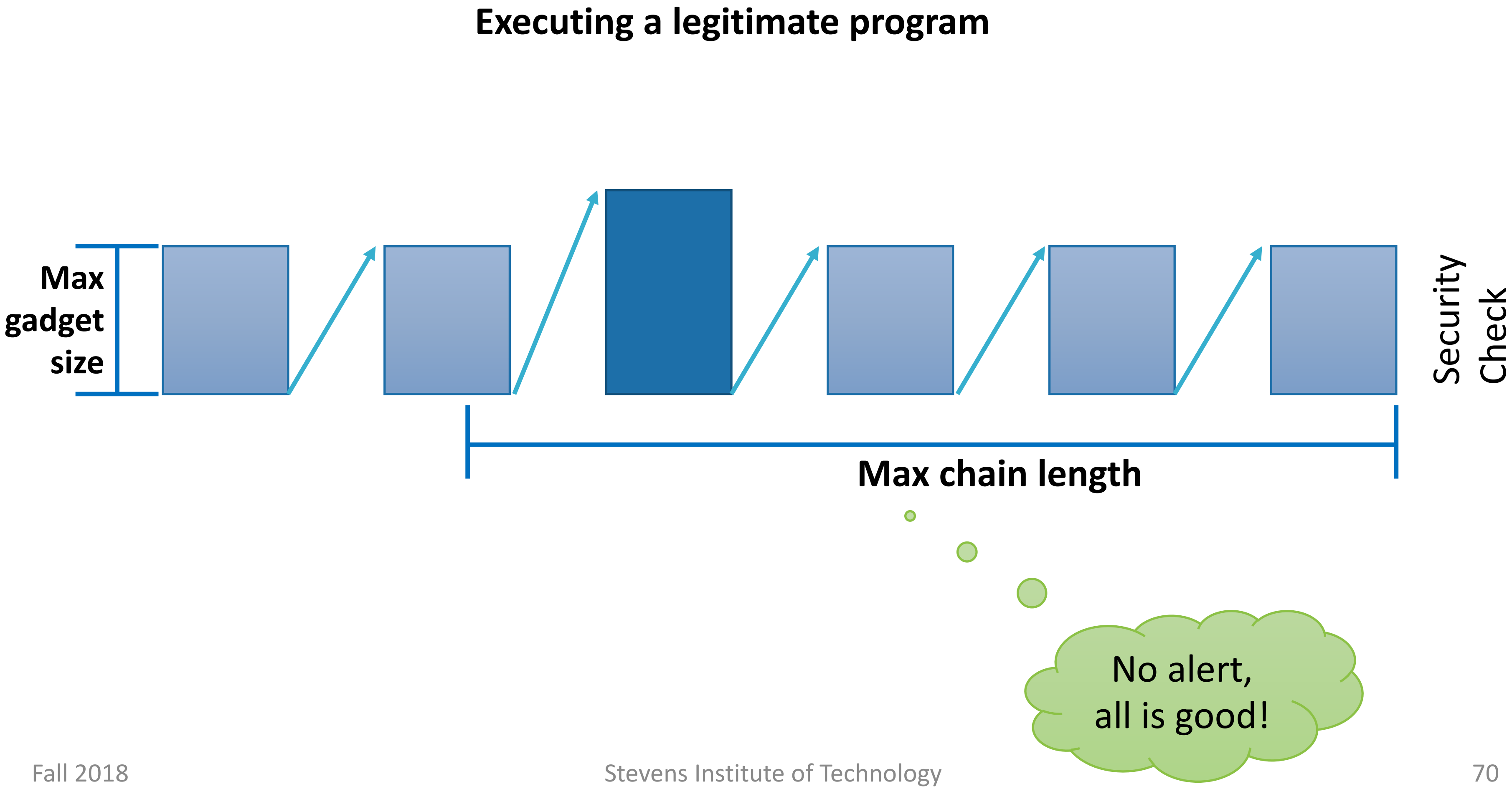
Dataset: Internet Explorer, Adobe Reader, Flash Player, Microsoft Office (Word, Excel, Powerpoint)

Chosen Parameters

Approach similar to kBouncer

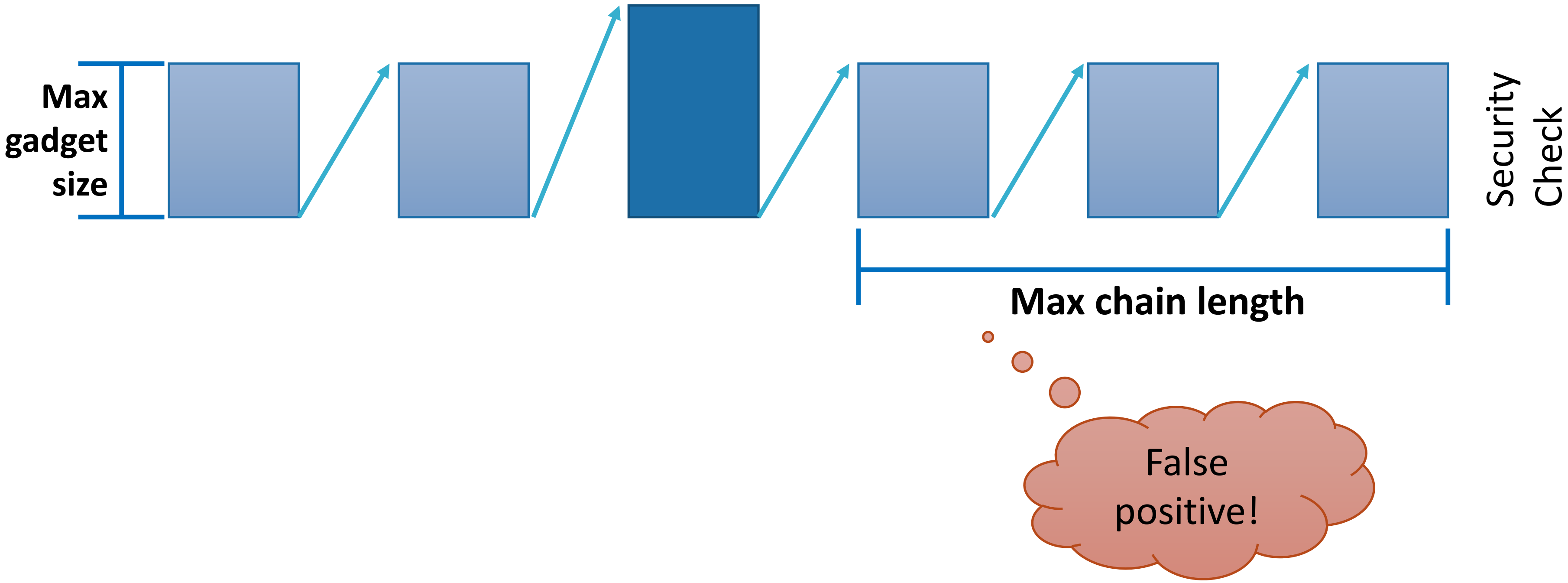
	kBouncer	ROPecker
Time-of-Check	Entry of Sensitive API	Entry of Sensitive API + during execution
Gadget Length	**20** instructions	**6** instructions
Inspect BH instances	Detected max "benign" gadget chain length: **5**	Detected max "benign" gadget chain length: **10**
Gadget Chain Length	**8** gadgets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

Why Picking Parameters Is Hard

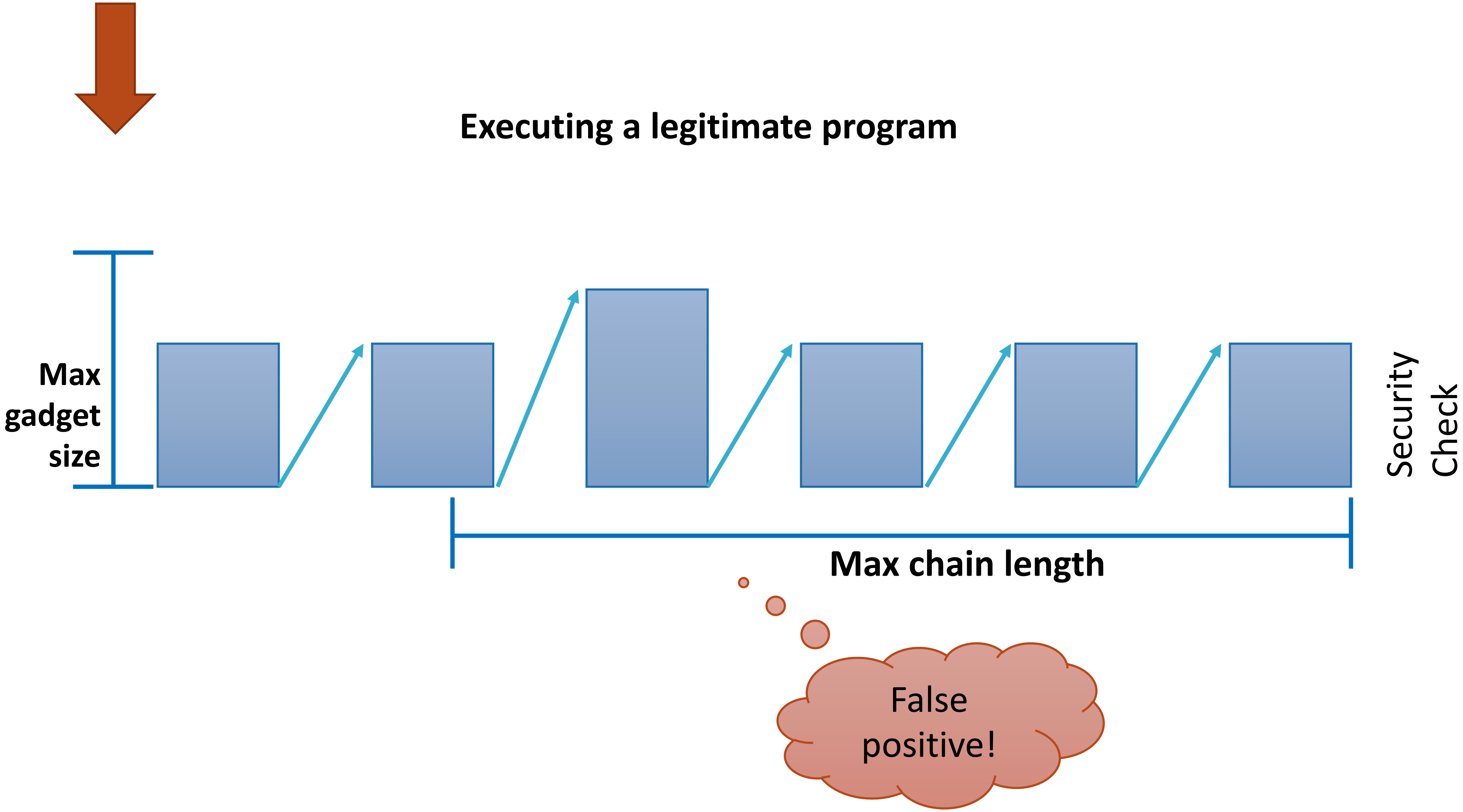


Why Picking Parameters Is Hard

Executing a legitimate program

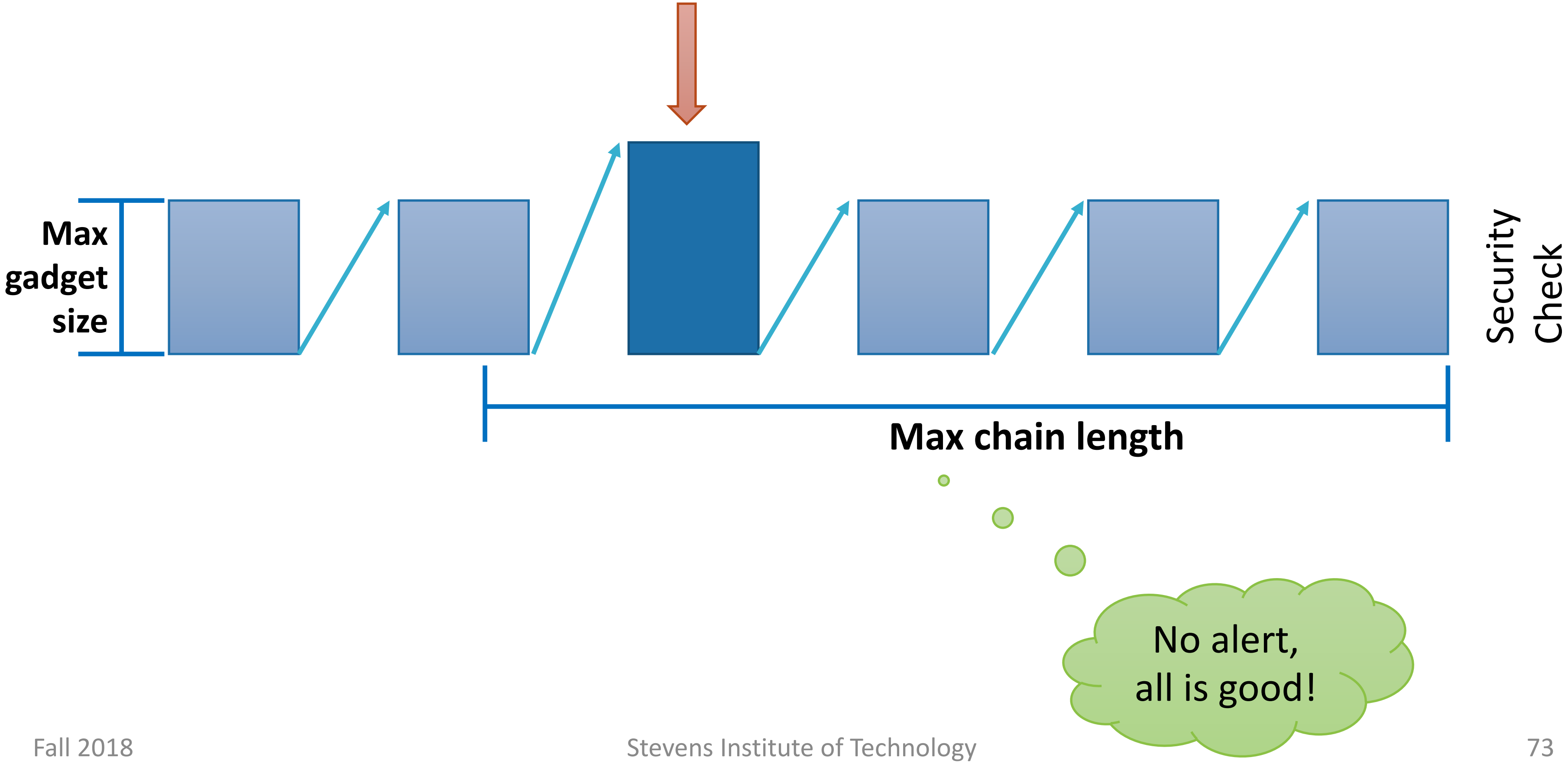


Why Picking Parameters Is Hard



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

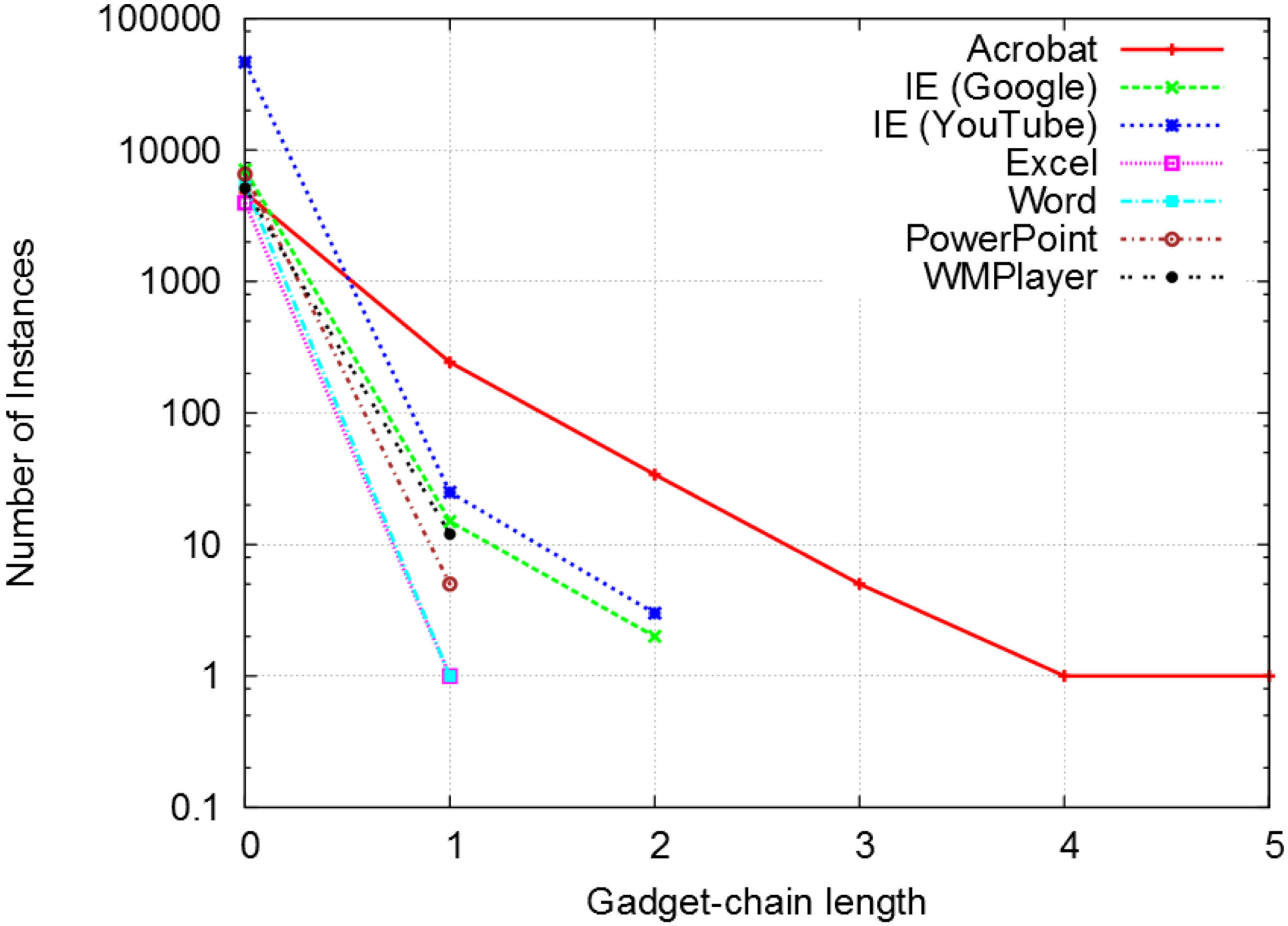
Assumes **kBouncer** is in place

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

Multiple payloads

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

Per Application Thresholds



What if We Had the Perfect CFG

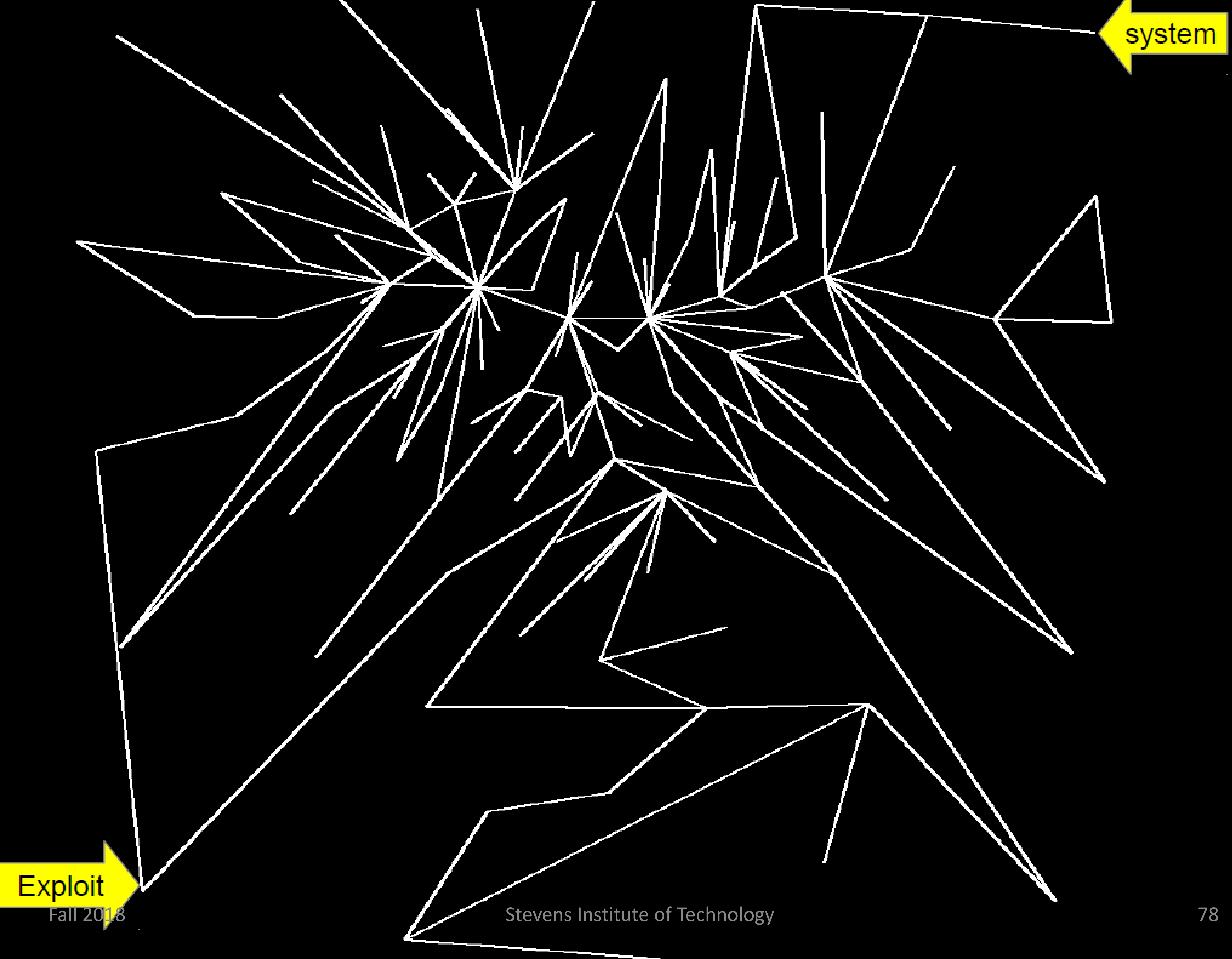
We know exactly which functions are called from an indirect call

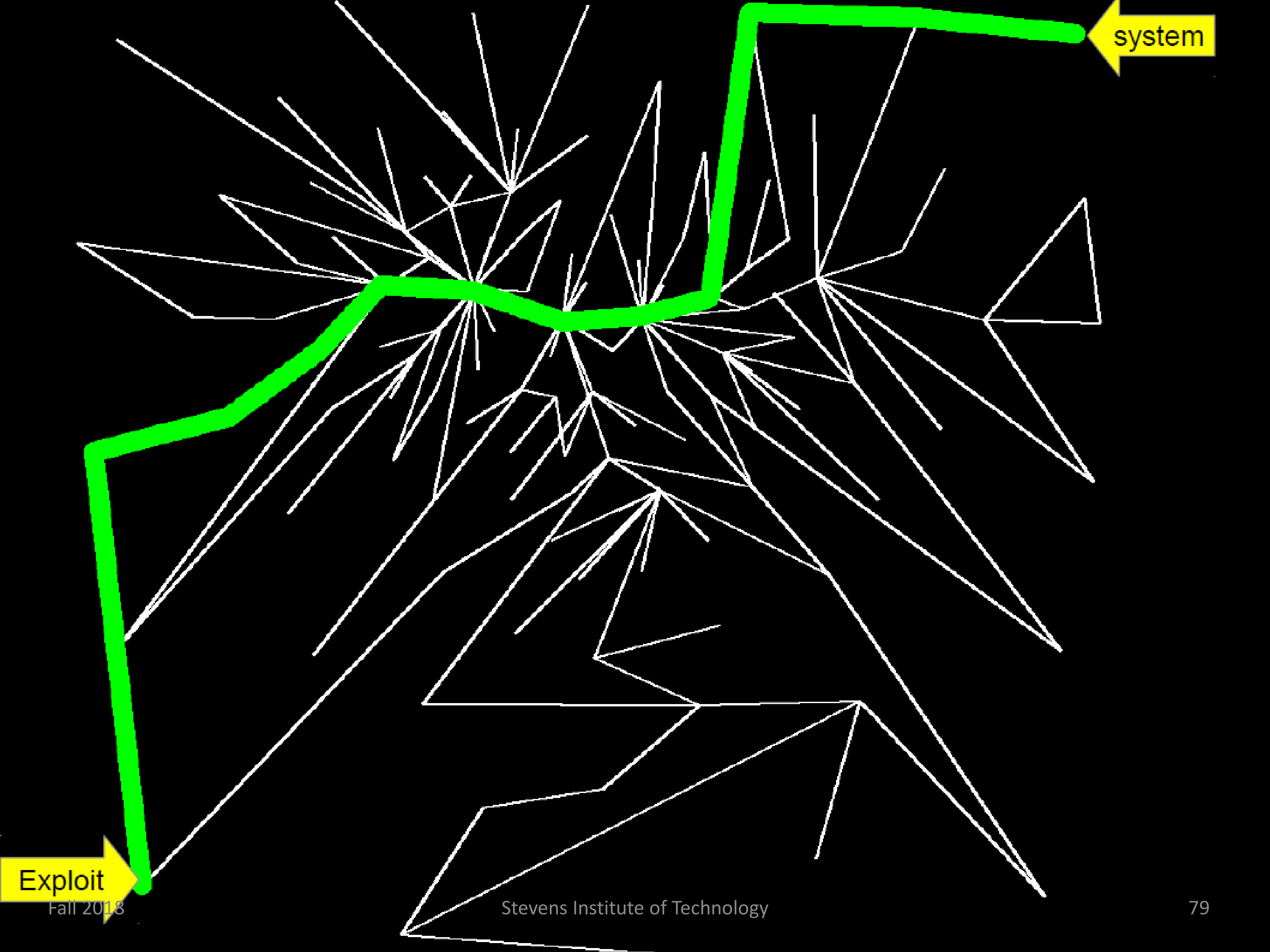
We know exactly the call sites where a function’s return is supposed to return

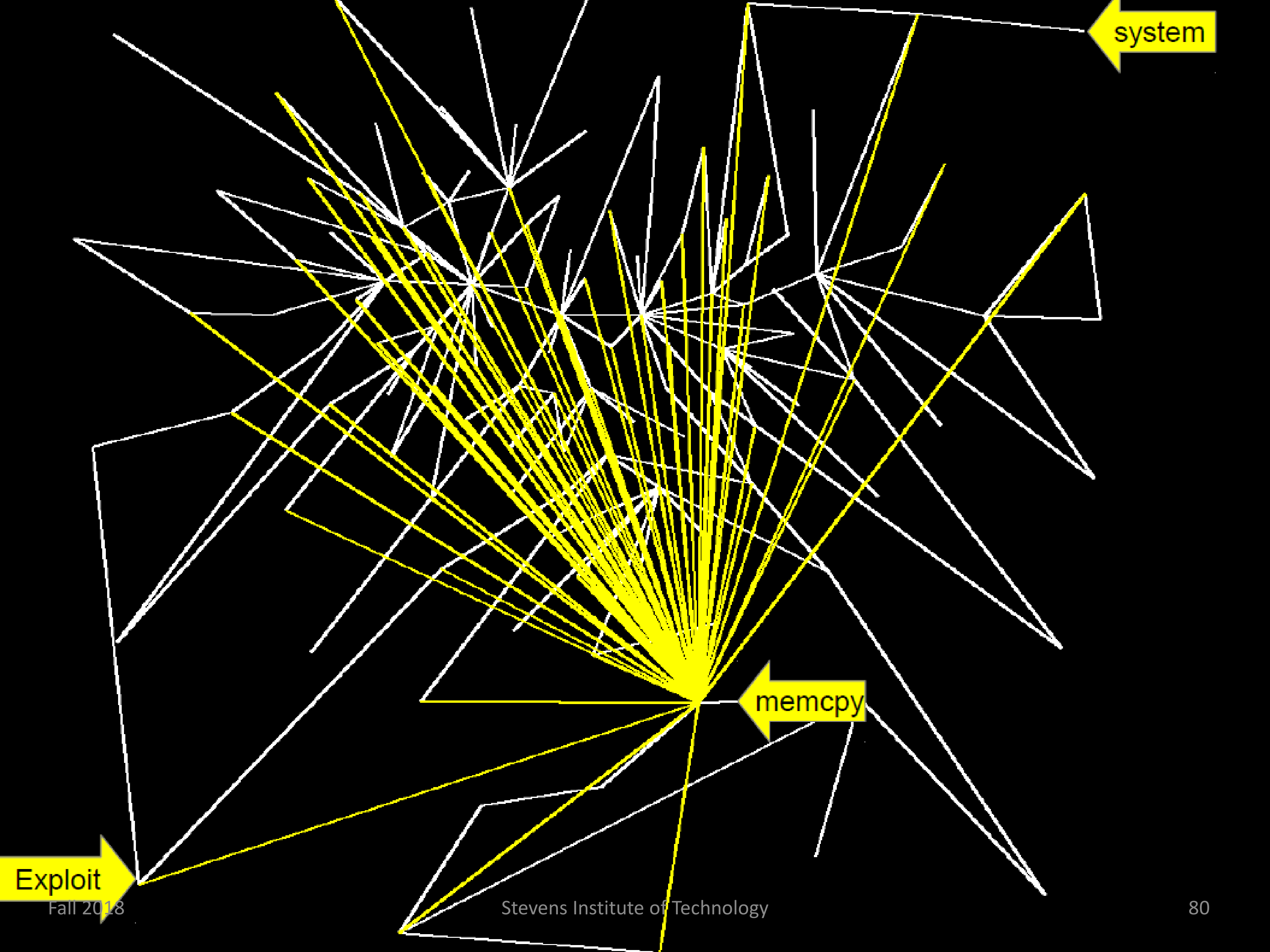
But we still do not have a shadow stack

Control Flow Bending

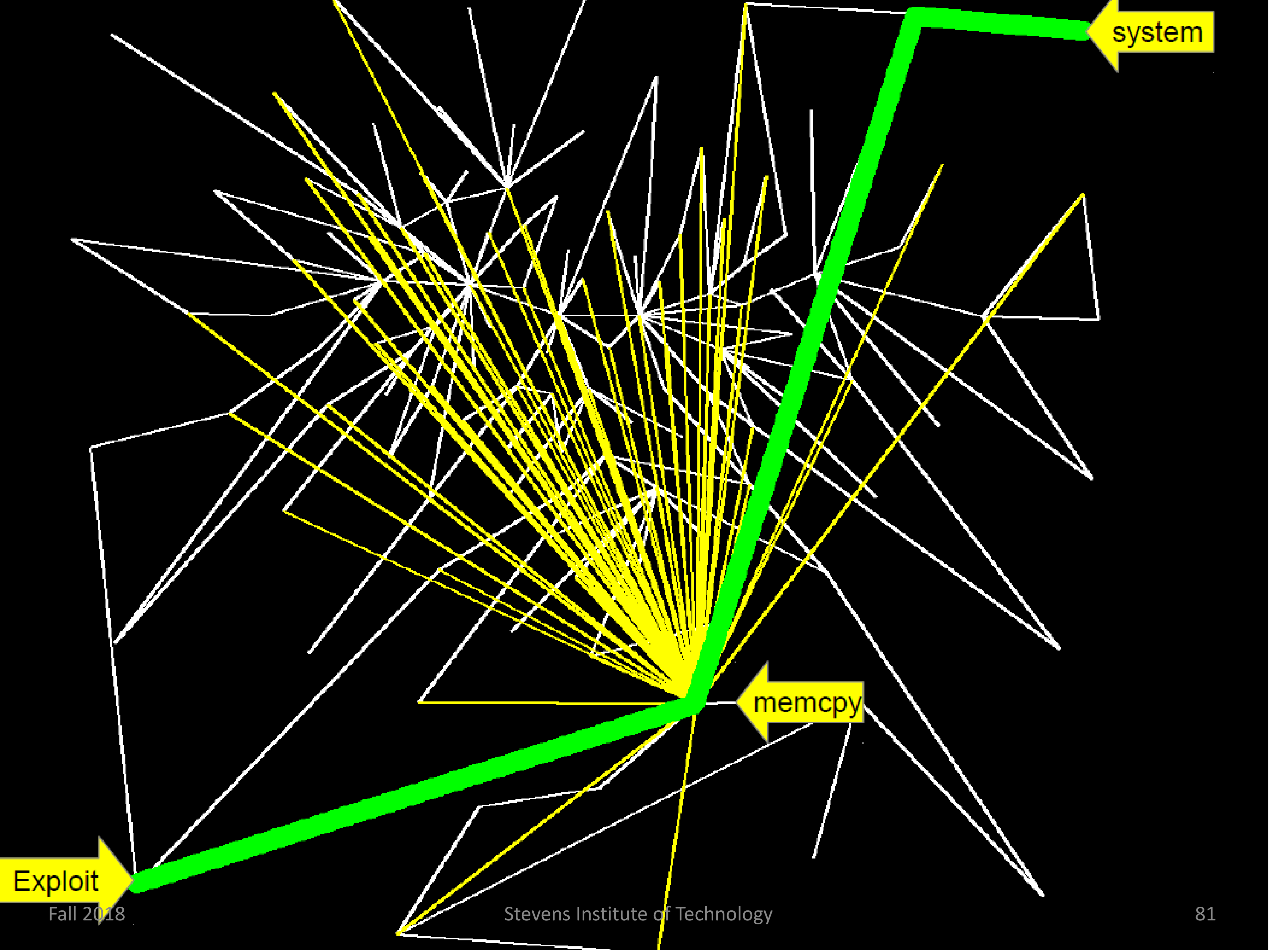
https://www.usenix.org/sites/default/files/conference/protected-files/sec15_slides_carlini.pdf


system
Exploit

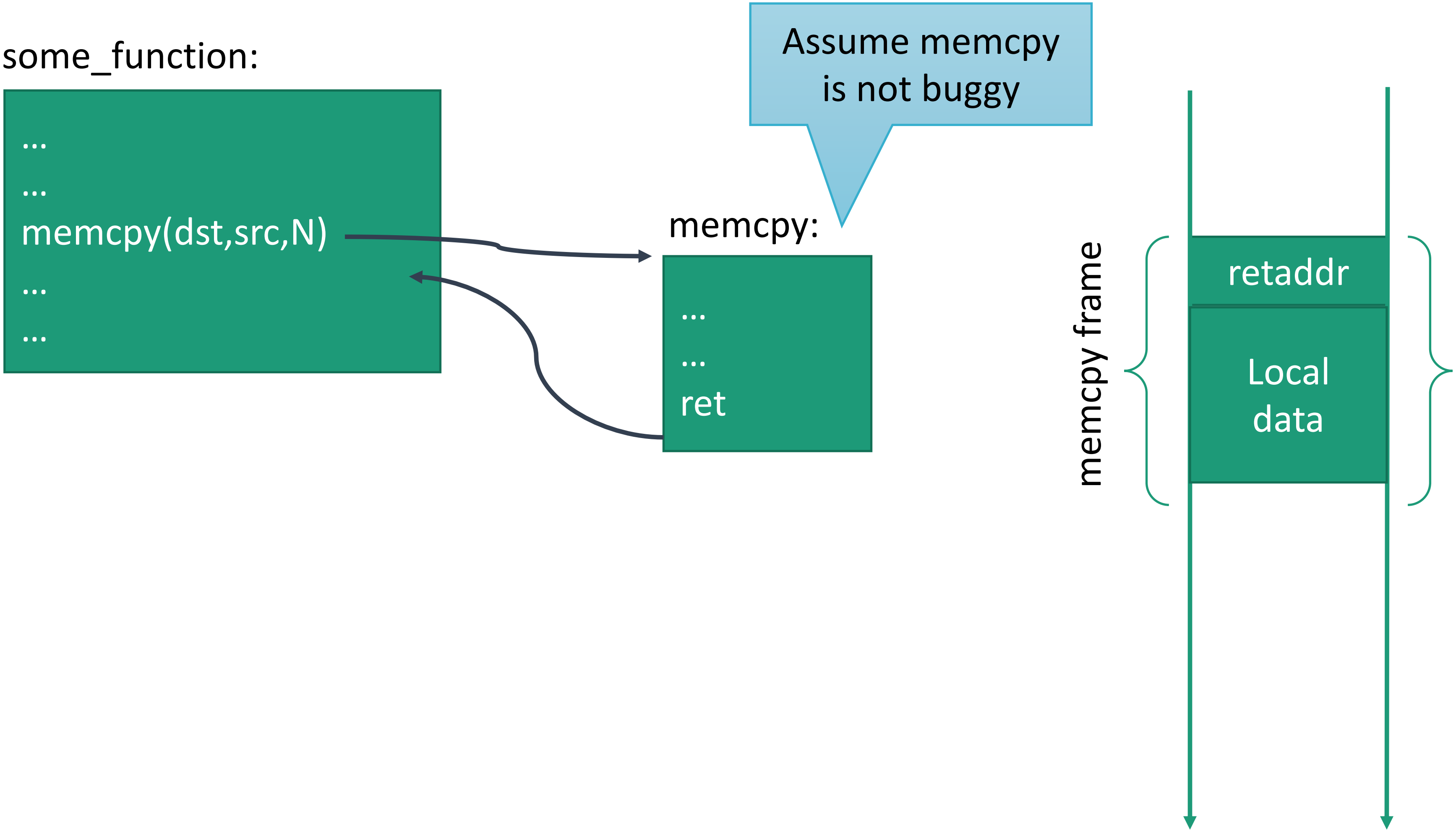

system
Exploit



system
memcpy
Exploit


system
memcpy
Exploit

How to Exploit the memcpy() Hotspot



some_function:

```
…
…
memcpy(dst,src,N)
…
…
```

Assume memcpy is not buggy

memcpy:

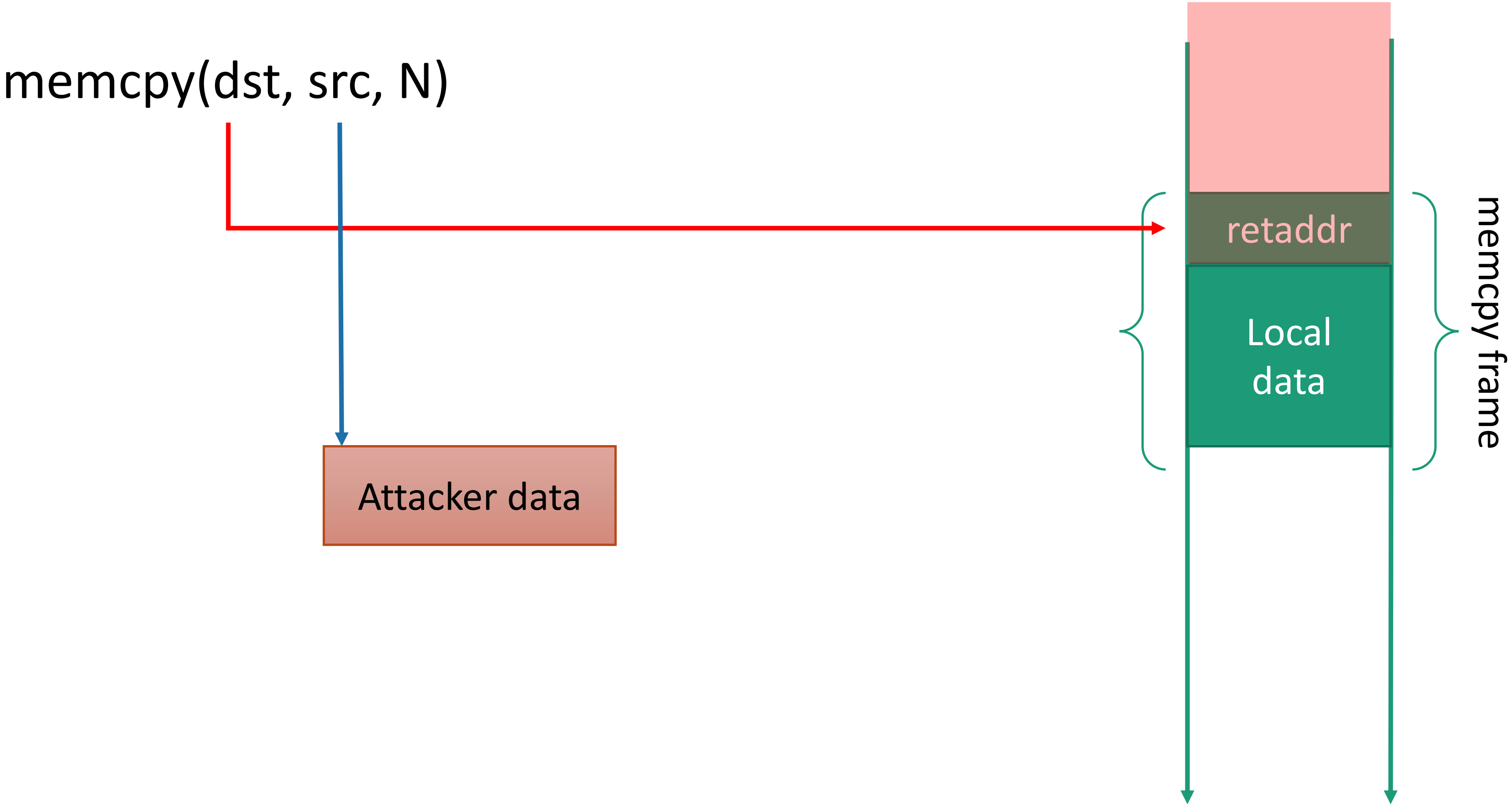
```
…
…
ret
```

memcpy frame

retaddr

Local data

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