

Hardware Support for Memory Safety

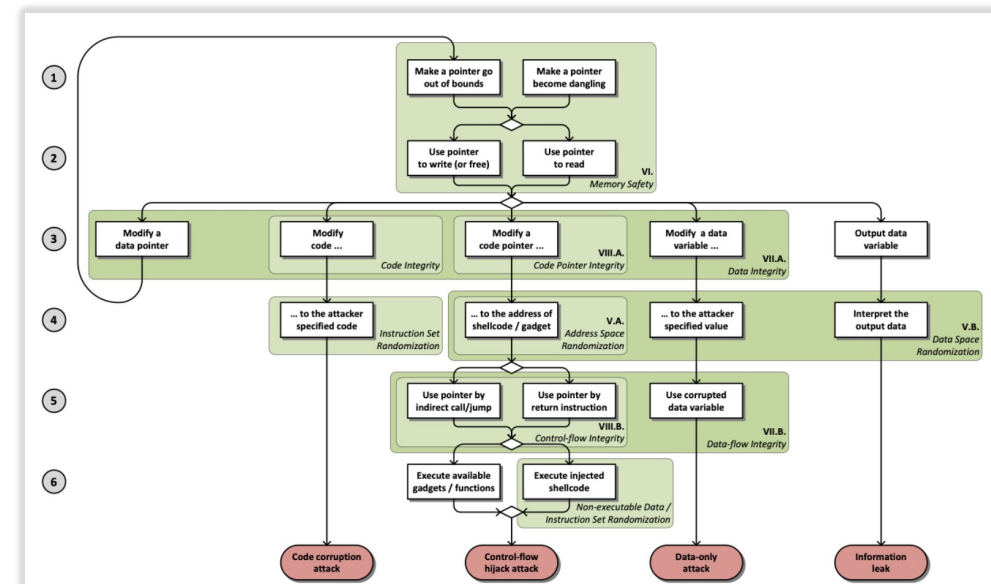
Mengjia Yan

Spring 2024



Overview

- What attacker can do with software bugs?
 - Demos, variations, real-world examples
- Hardware mitigations: what are the design tradeoffs?



The Problem: Software Bugs

- Low-level Language Basics (C/C++/Assembly)
 - + Efficient, programmers have more control
 - Bugs
 - Programming productivity
- Widely used in production systems and legacy systems
 - Operating systems, web browsers, etc.
 - Large CVE numbers every year



The Problems of Using Pointers

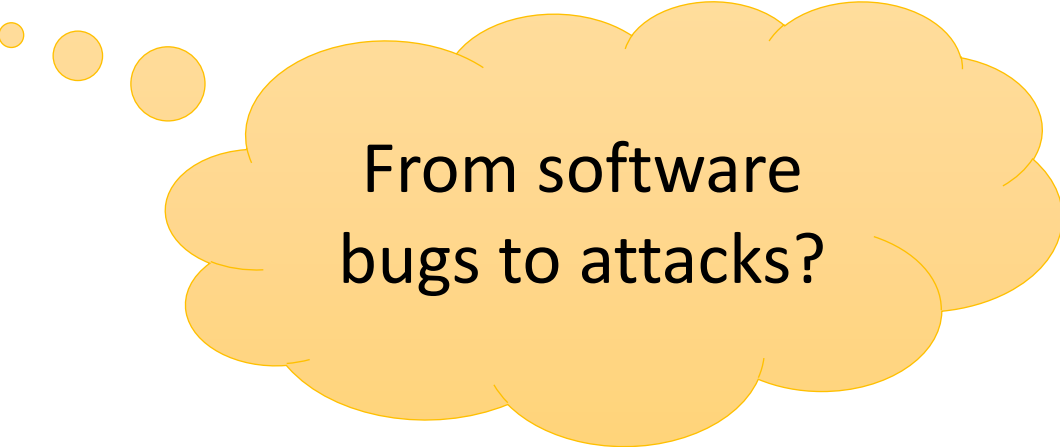
- Pointer = Address of variables:
 - An 64-bit integer to indicate the index of memory location where variable is stored
- It is programmers' responsibility to do **pointer check**, e.g. NULL, out-of-bound, use-after-free



- Why Python (and other high-level programming language) does not have these problems?
 - out-of-bound access => emit runtime checks
 - use-after-free => garbage collection

Memory Corruption Vulnerabilities

- Spatial safety:
 - out-of-bound (inter-object, intra-object)
 - Can happen on heap and stack
- Temporal safety:
 - Use-after-free
 - Use before initialization

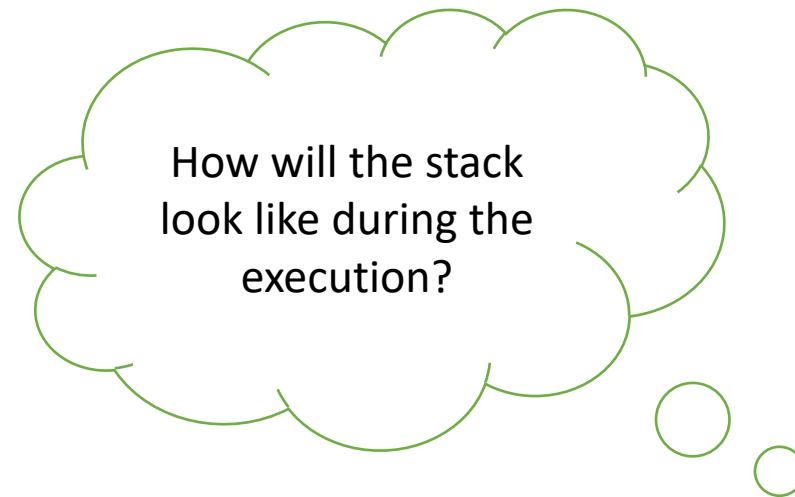


From software bugs to attacks?

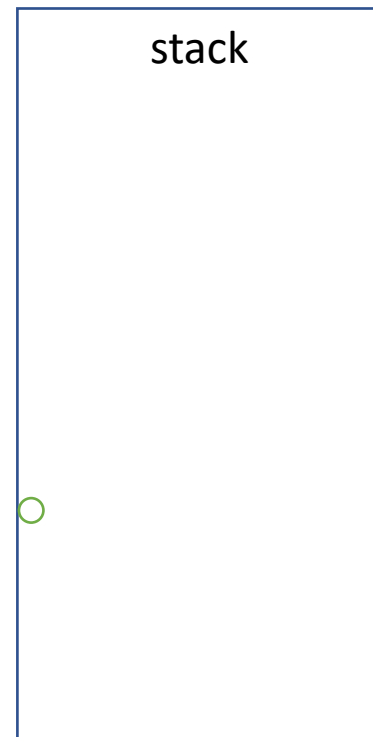
Stack and Stack Smash

```
int hello() {  
    int a = 100;  
    return a;  
}  
int main() {  
    int a;  
    int b = -3;  
    int c = 12345;  
    int *p = &b;  
    int d = hello();  
    return 0;  
}
```

ra →



TEXT (code)



Stack Smash

```
int func (char *str) {  
    char buffer[12];  
    strncpy(buffer, str, len(str));  
    return 1;  
}  
  
int main() {  
    ...  
    func (input);  
    ...  
}
```

ra



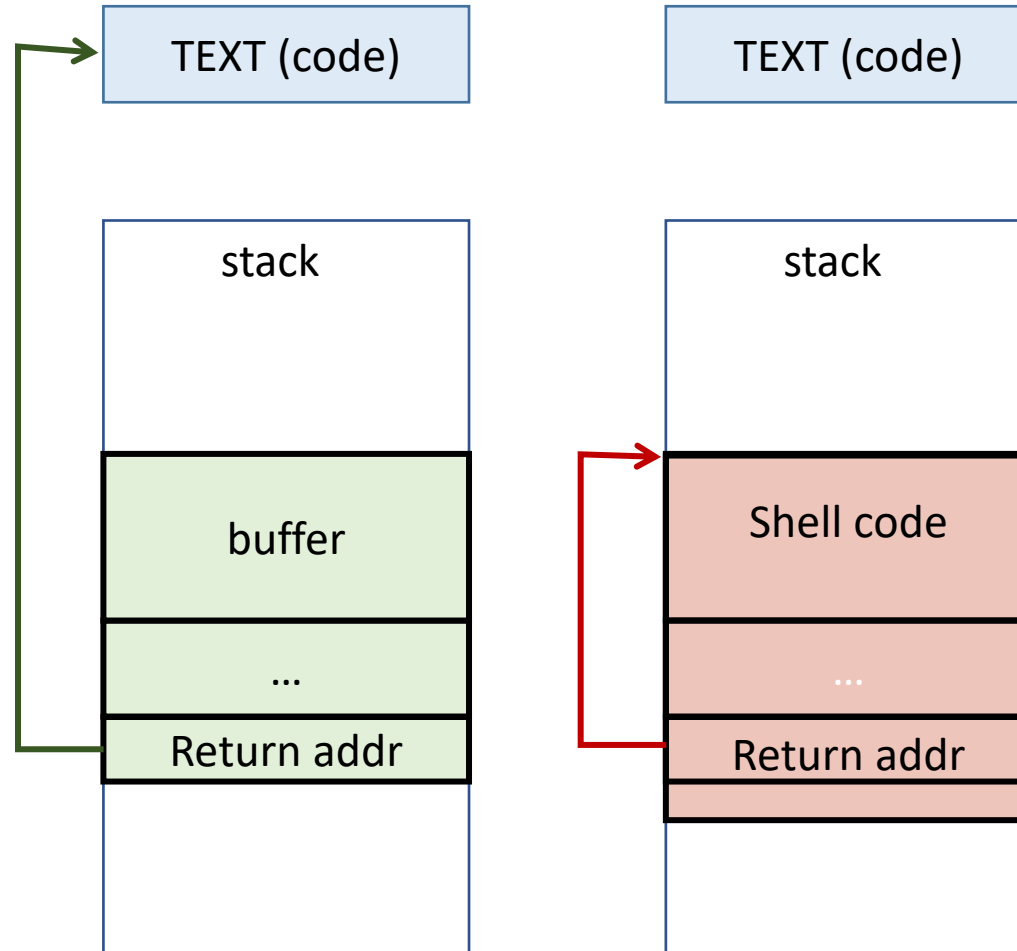
Shell code:

```
PUSH "/bin/sh"  
CALL system
```

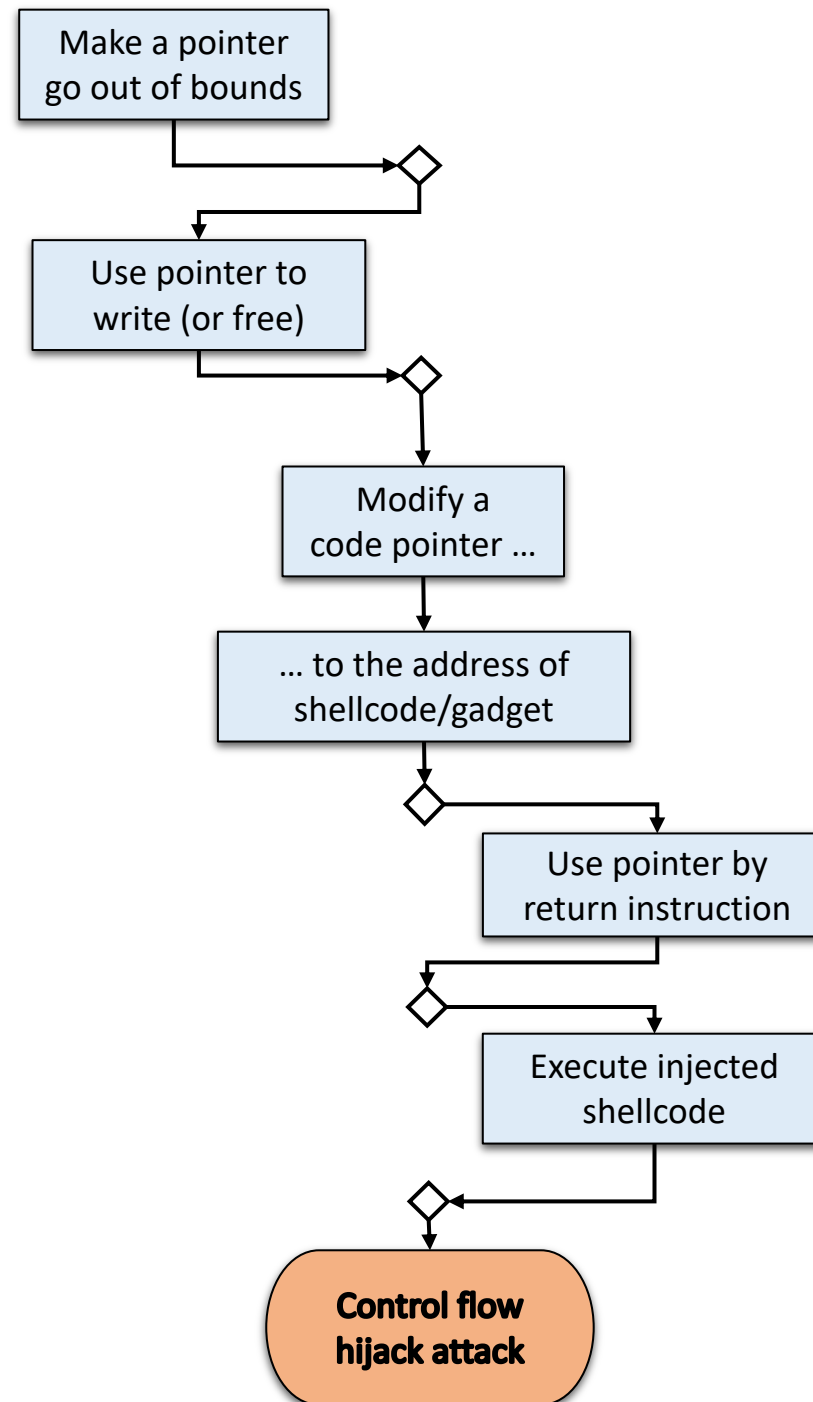
Input str:

```
Shell code  
.. Some padding..  
Address of buffer
```

Stack Smash / Code Injection Attack

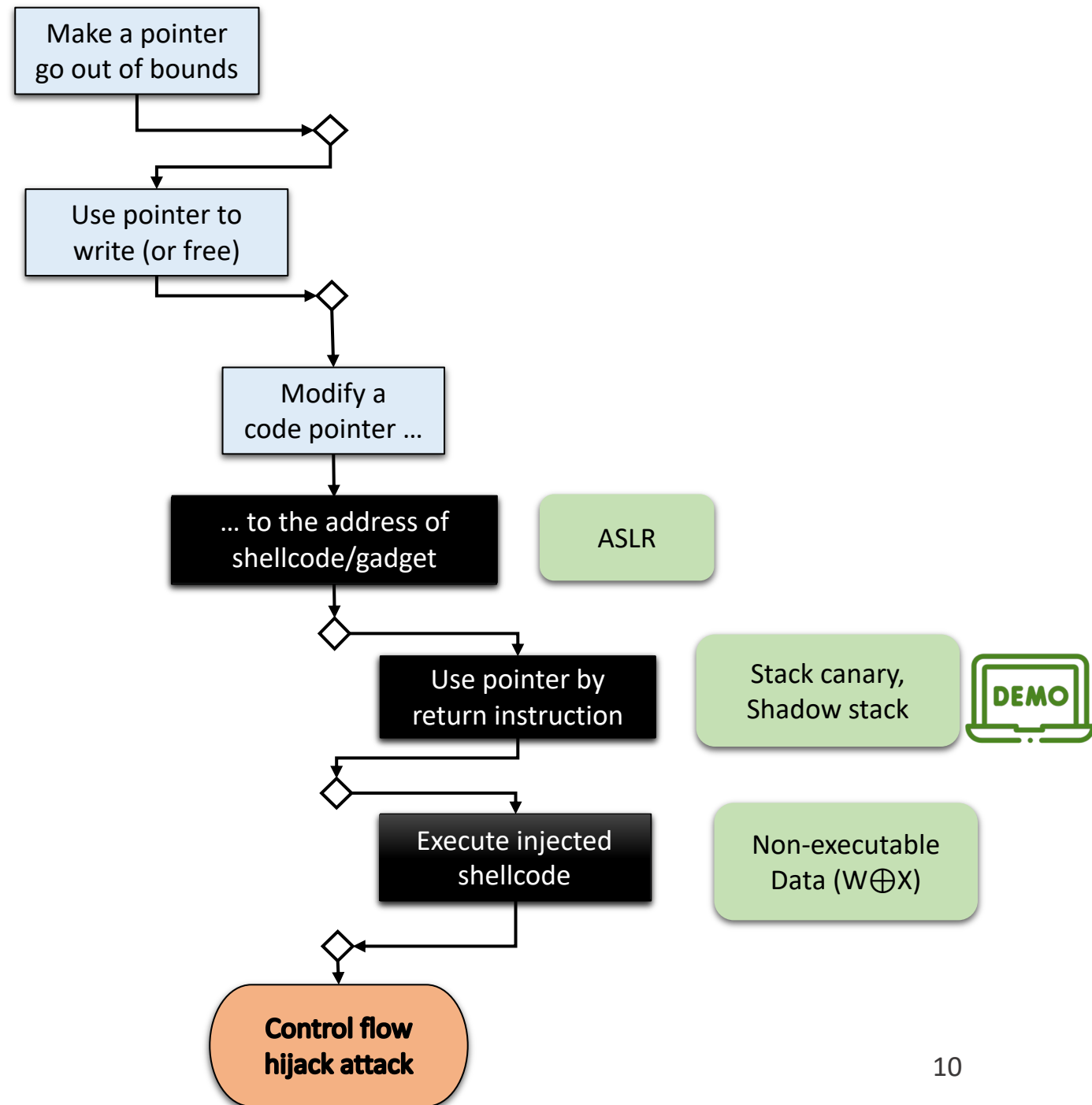
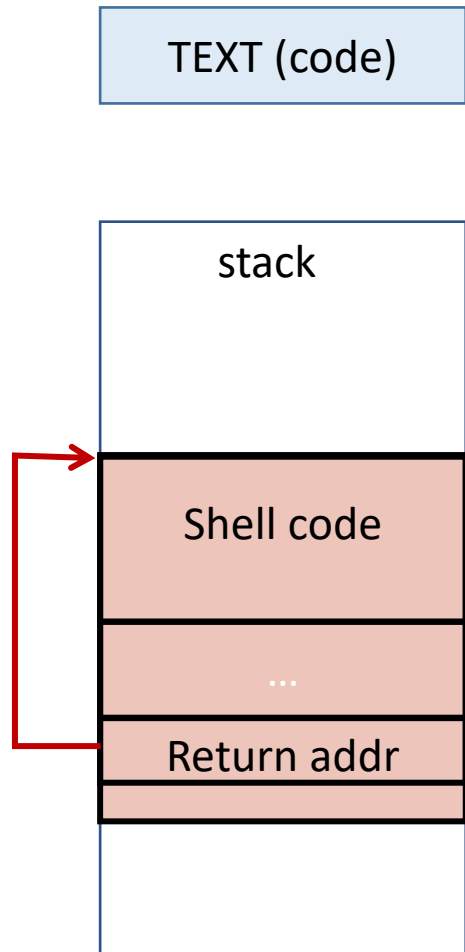


Attack Variations

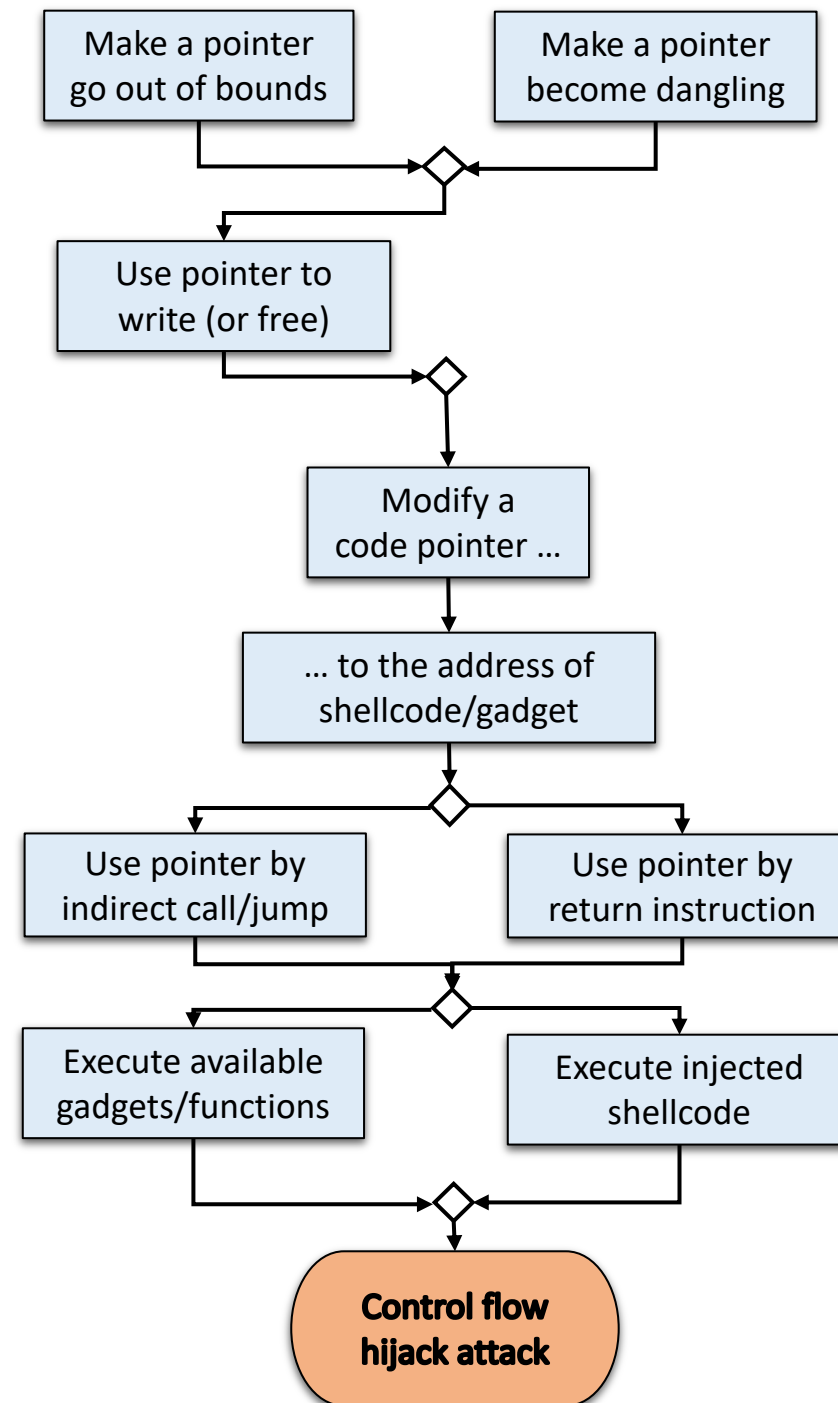


L. Szekeres, M. Payer, T. Wei and D. Song, "SoK: Eternal War in Memory," S&P'2013

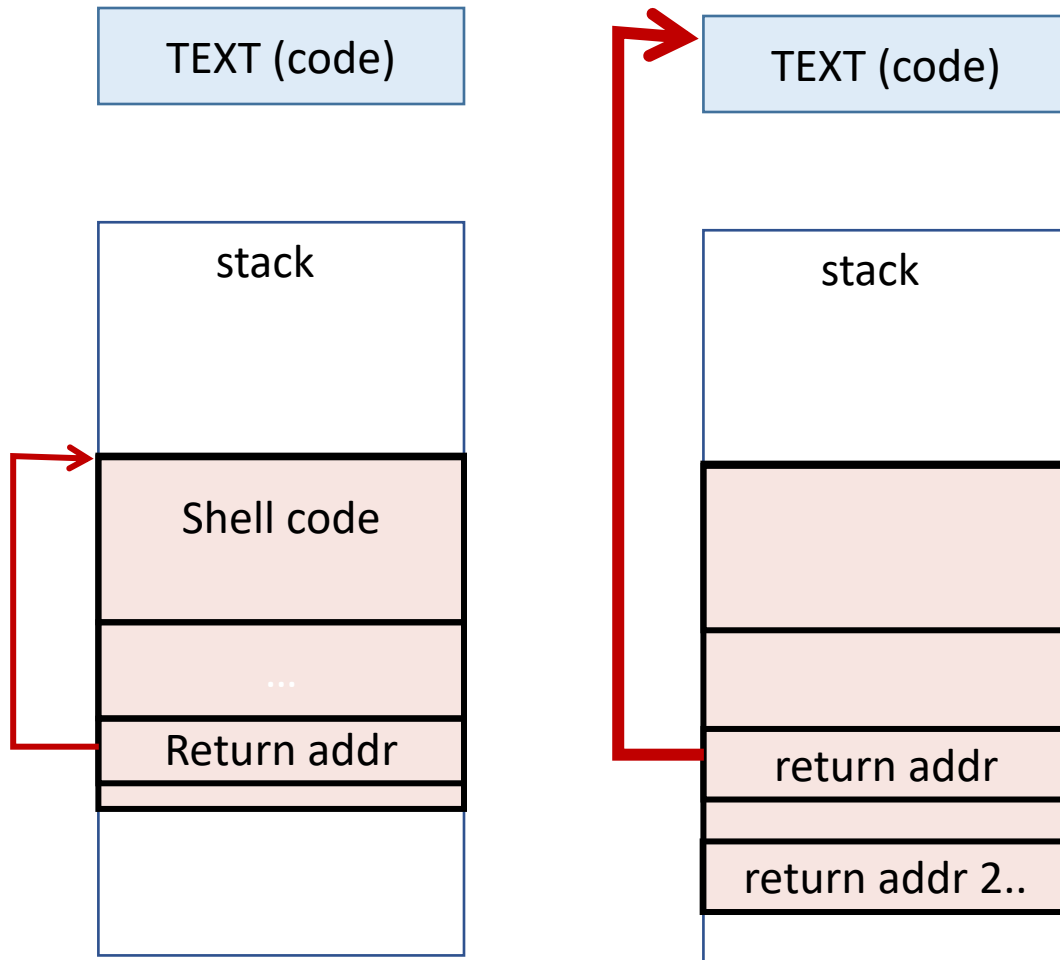
Mitigations



Attack Variations



Return-Oriented Programming (ROP)



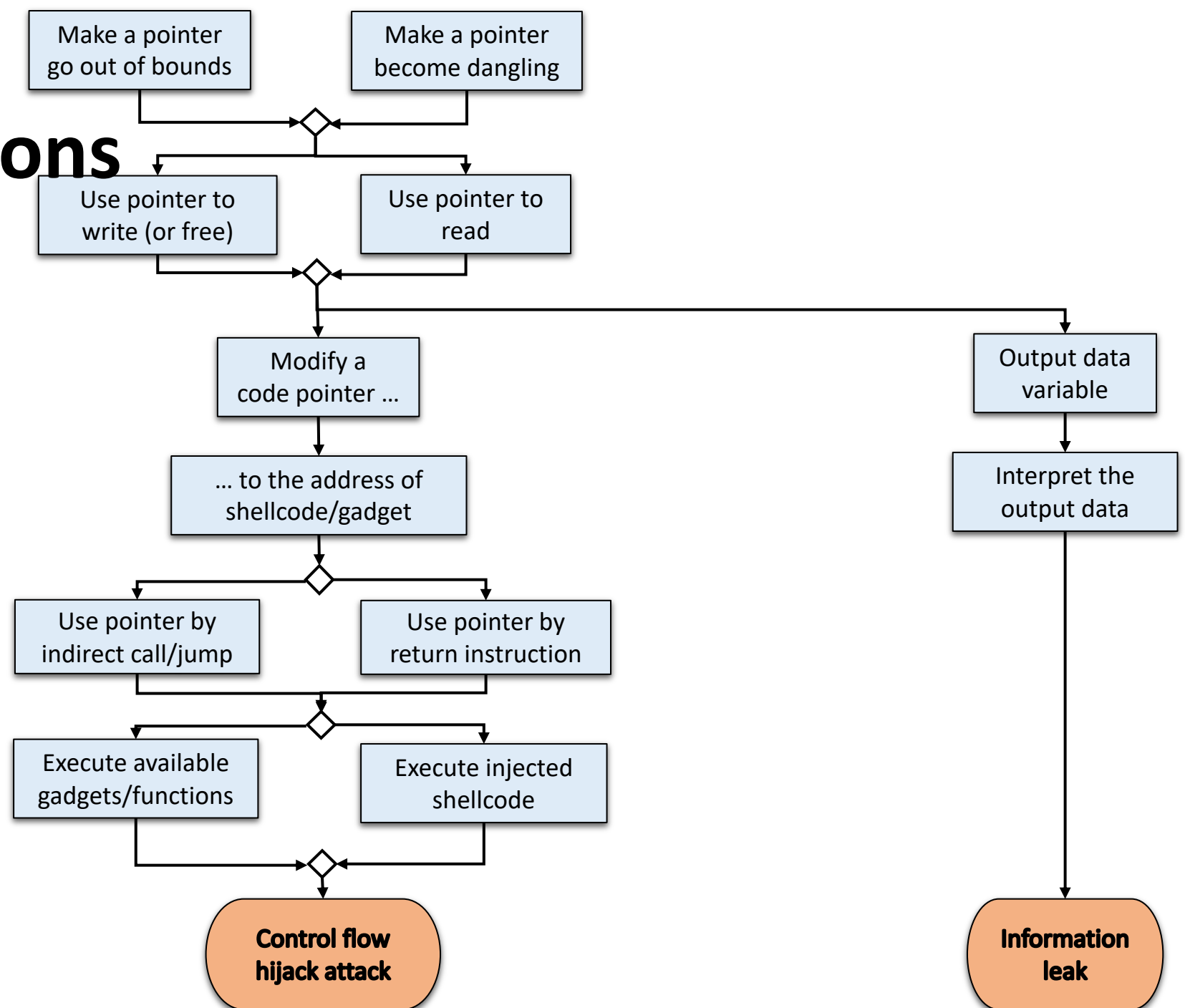
Gadget example:

```
pop rdi  
ret
```



Jump-oriented programming

Attack Variations

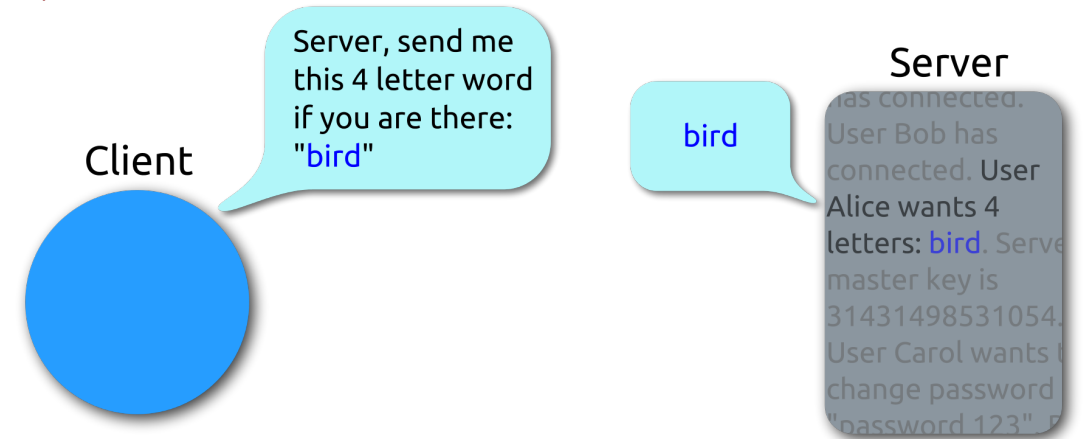


HeartBleed Vulnerability

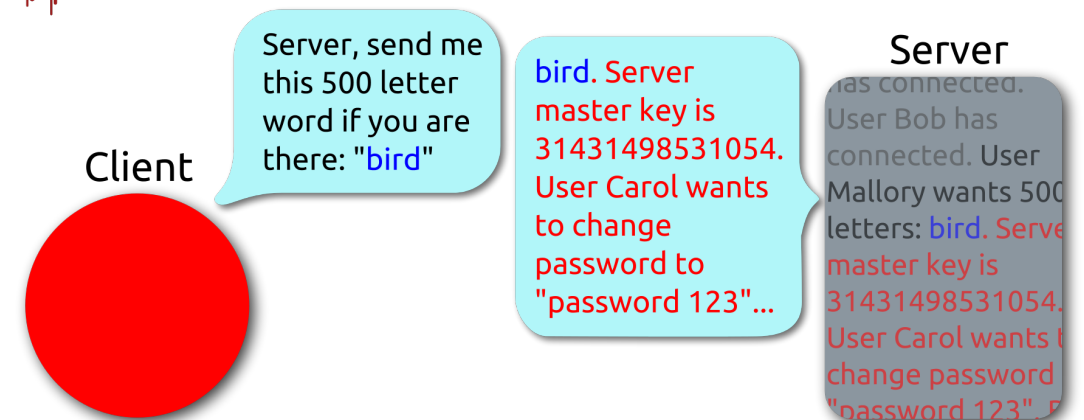
- Publicly disclosed in April 2014
- Missing a bound check
- Bug in the OpenSSL cryptographic software library heartbeat extension

<https://heartbleed.com/>

Heartbeat – Normal usage



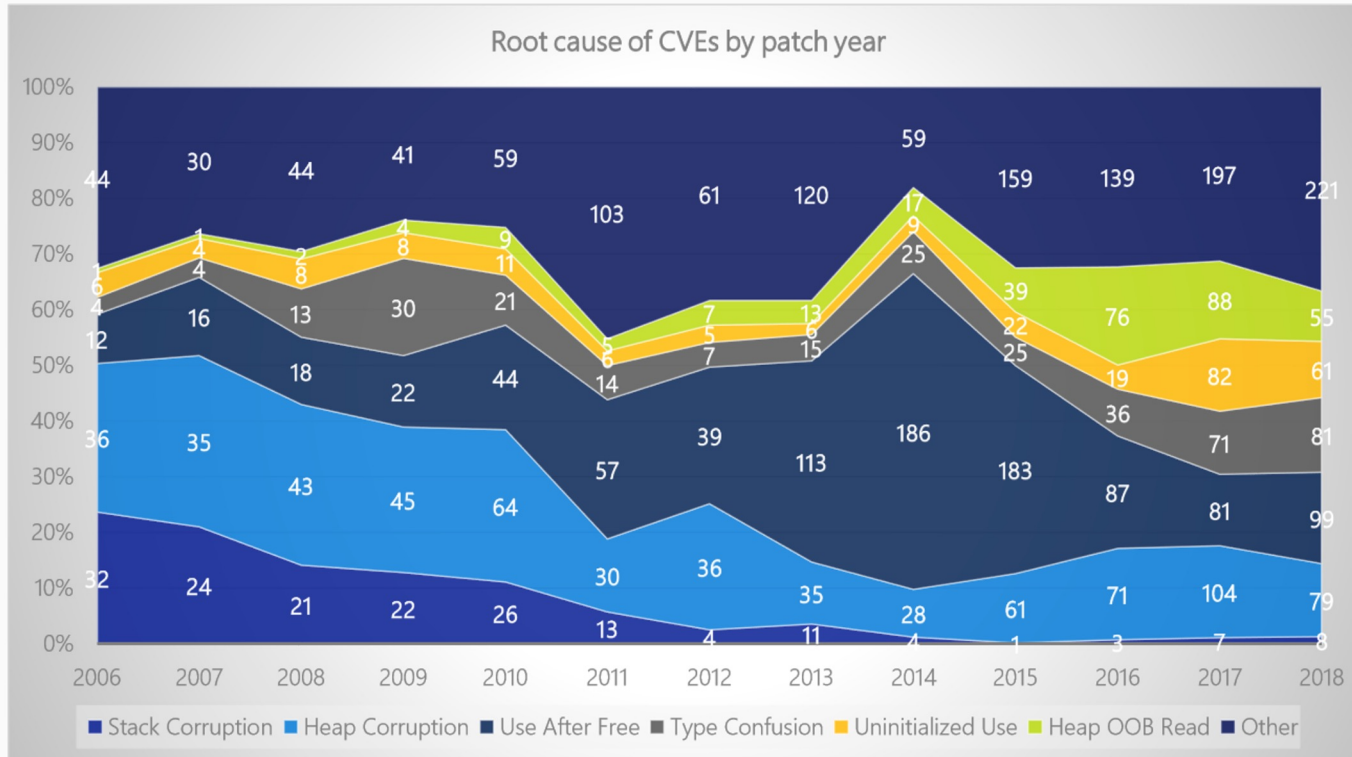
Heartbeat – Malicious usage



Trend reported by Microsoft

https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

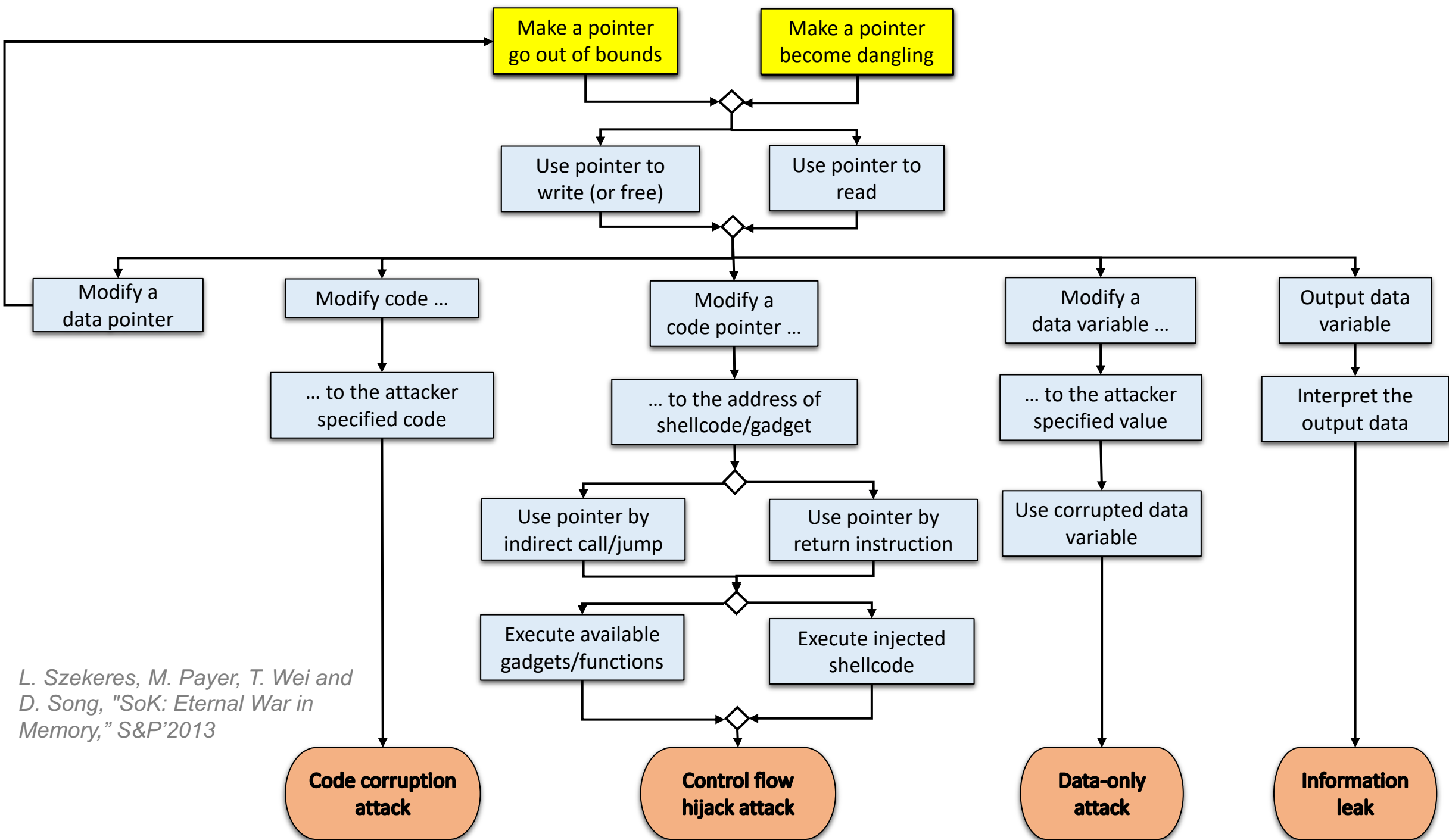
#1: heap out-of-bounds

#2: use after free

#3: type confusion

#4: uninitialized use

Hardware Supported Mitigations



L. Szekeres, M. Payer, T. Wei and D. Song, "SoK: Eternal War in Memory," S&P'2013

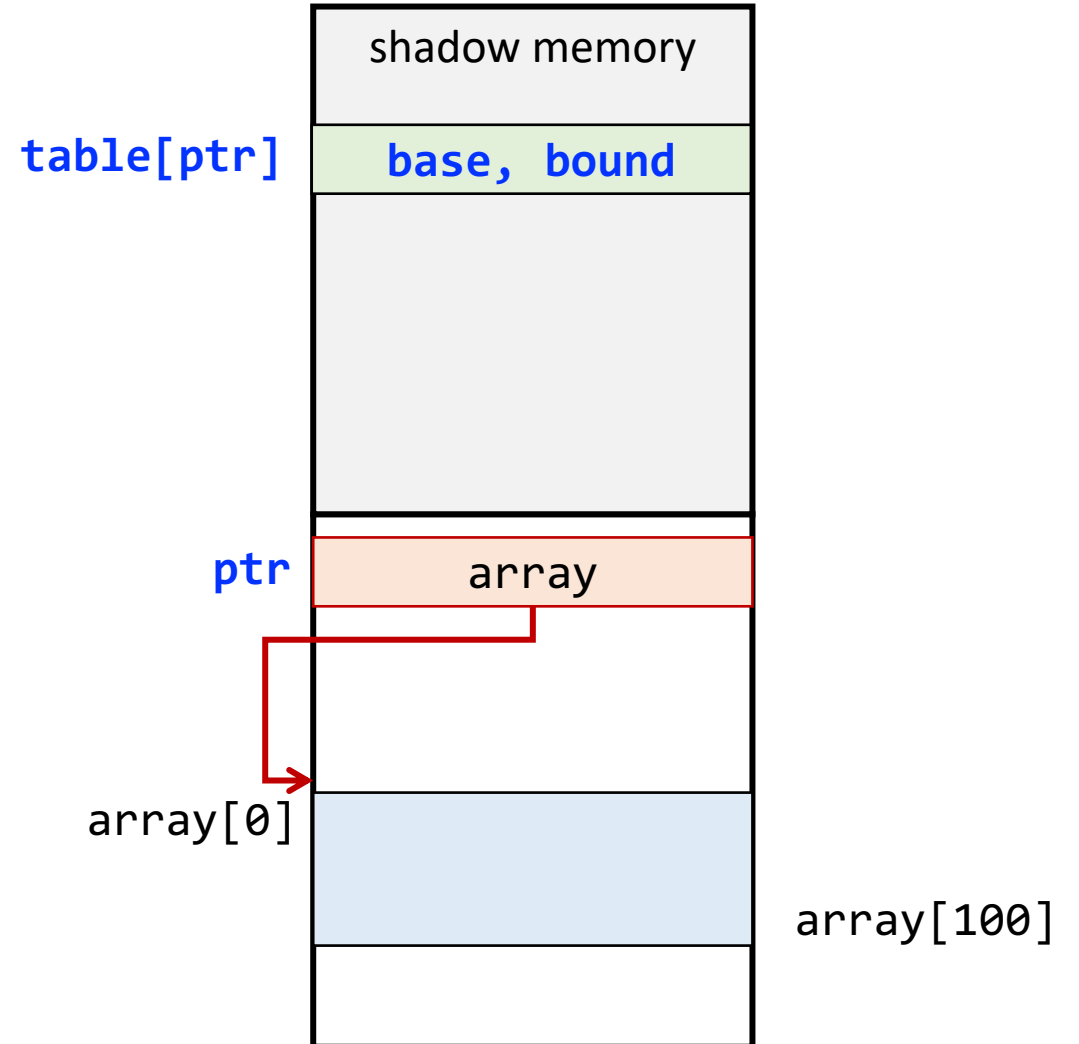
Memory Safety

- Strongest security property that tries to address the problem at the root.
- Idea: include metadata and perform security checks at runtime
 - Spatial safety (bound information)
 - Temporal safety (allocation/de-allocation information)
- Software solutions
 - Problem #1: performance overhead, extra instructions to perform the check
 - Problem #2: where to store metadata? -> in shadow memory

SoftBound

Creating a pointer:

```
int array[100];  
ptr = &array;  
ptr_base = &array[0];  
ptr_bound = &array[100];  
table[ptr] = {base, bound};
```



SoftBound

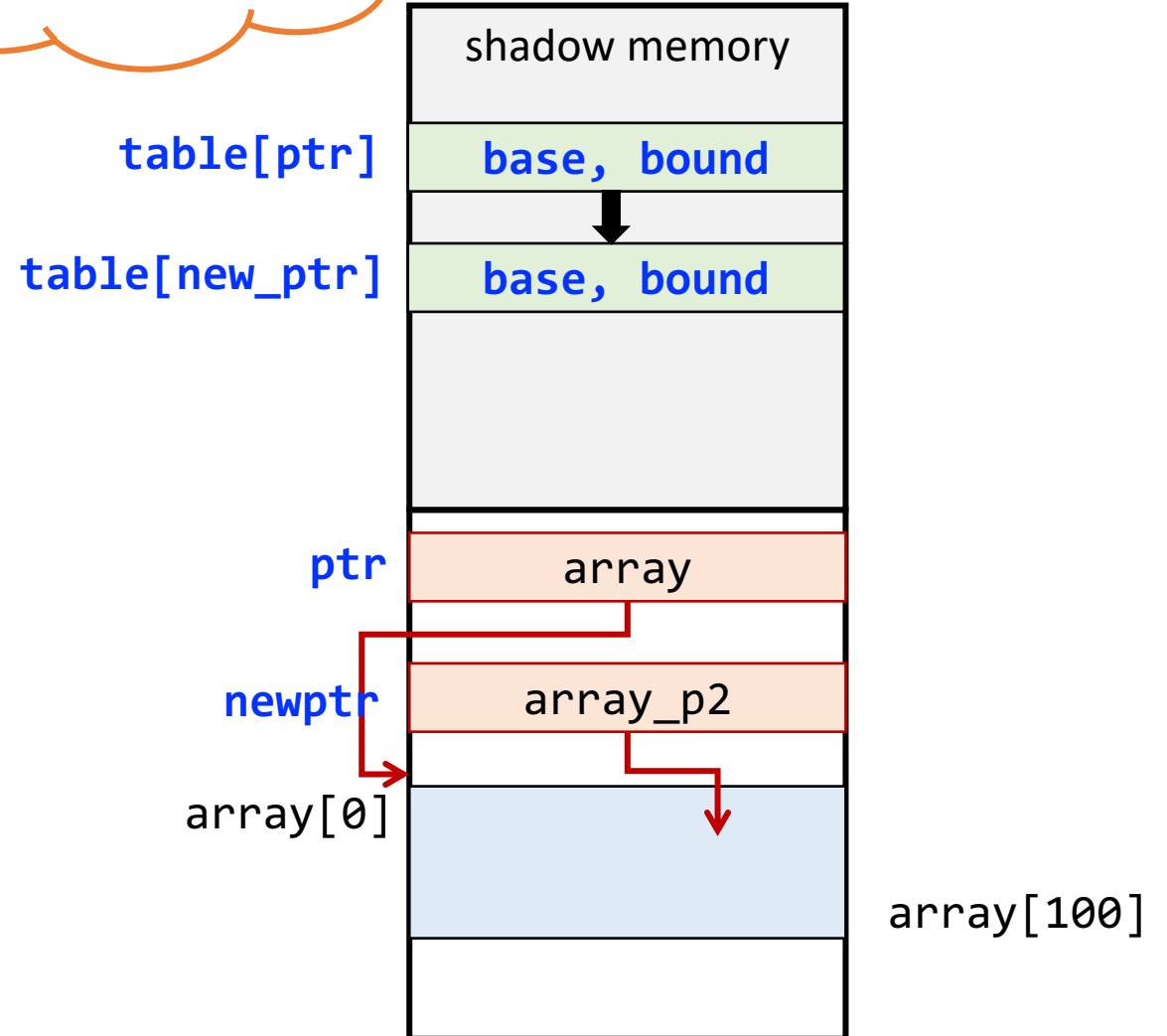
Compare number of memory accesses?

Creating a pointer:

```
int array[100];  
ptr = &array;  
ptr_base = &array[0];  
ptr_bound = &array[100];  
table[ptr] = {base, bound};
```

Pointer arithmetic:

```
int* array_p2 = &array[10];  
newptr_base = table[ptr].base;  
newptr_bound = table[ptr].bound;  
table[newptr] = {base, bound};
```



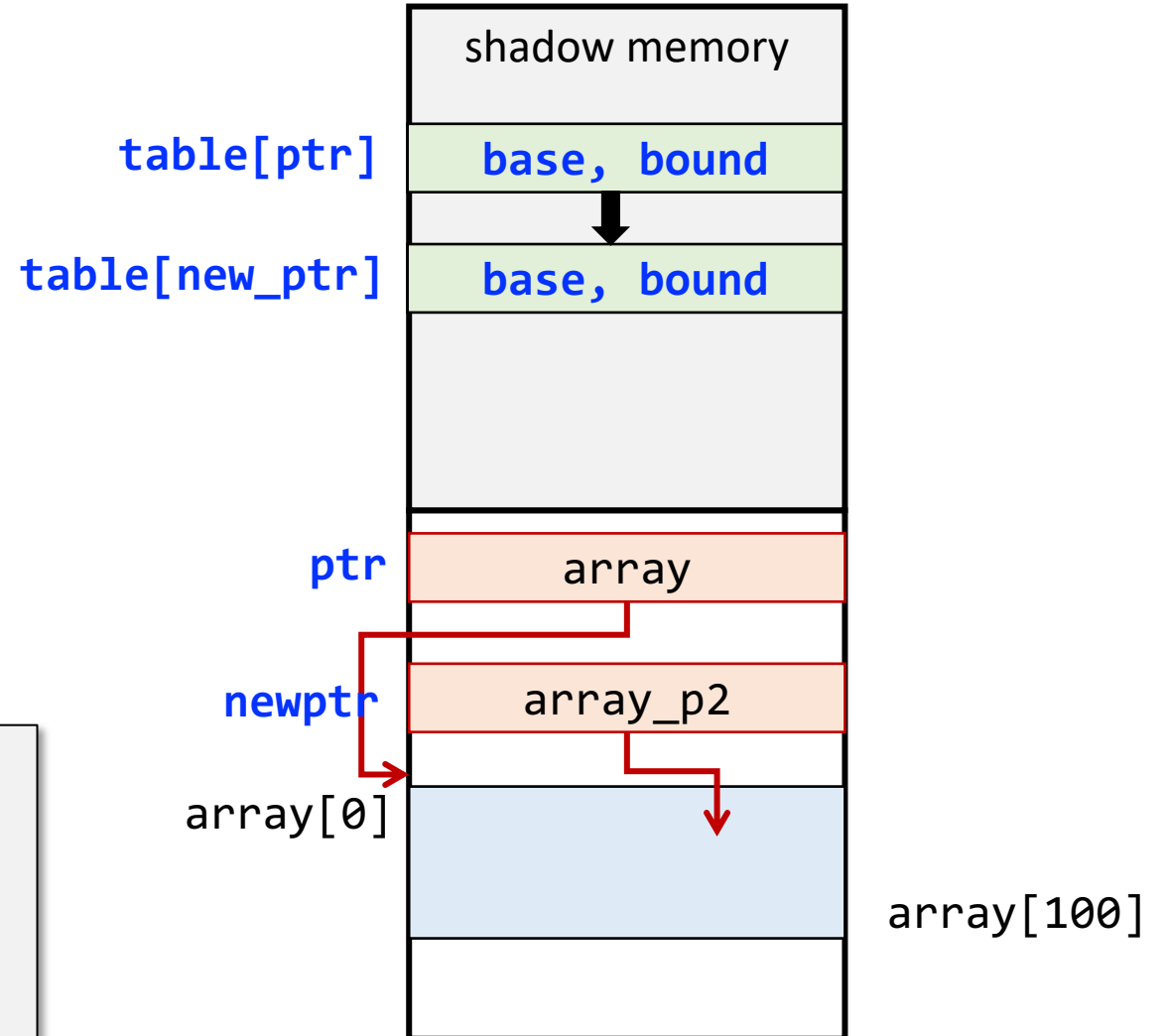
SoftBound

Creating a pointer:

```
int array[100];  
ptr = &array;  
ptr_base = &array[0];  
ptr_bound = &array[100];  
table[ptr]={base, bound};
```

Check a pointer:

```
newptr = &array_p2;  
{base, bound} = table[newptr];  
if (base > array_p2 || bound ...)  
    go to err;  
int* array_p2 = 0xFF;
```



HW Support for Memory Safety

A lot of work. The key is to understand the design trade-offs.

	Intel MPX (Memory Protection Extension)	ARM MTE (Memory Tagging Extension)
History	Announced in 2013, produced in 2015, now not supported anymore.	Introduced in ARM-8.5 in 2018. In 2019, Google announced that it is adopting Arm's MTE in Android. Apple will ship it soon.
Security		
Performance		
Compatibility		

Intel MPX (Memory Protection Extension)

4 bound registers (bnd0-3)

- **Bndmk**: create base and bound metadata
- **Bndldx/bndstx**: load/store metadata from/to bound tables
- **Bndcl/bndcu**: check pointer with lower and upper bounds

Any problem?

Original Program

```
p=malloc(16);  
... // p = p + 4;  
*p = 'a';
```

Instrumented Program

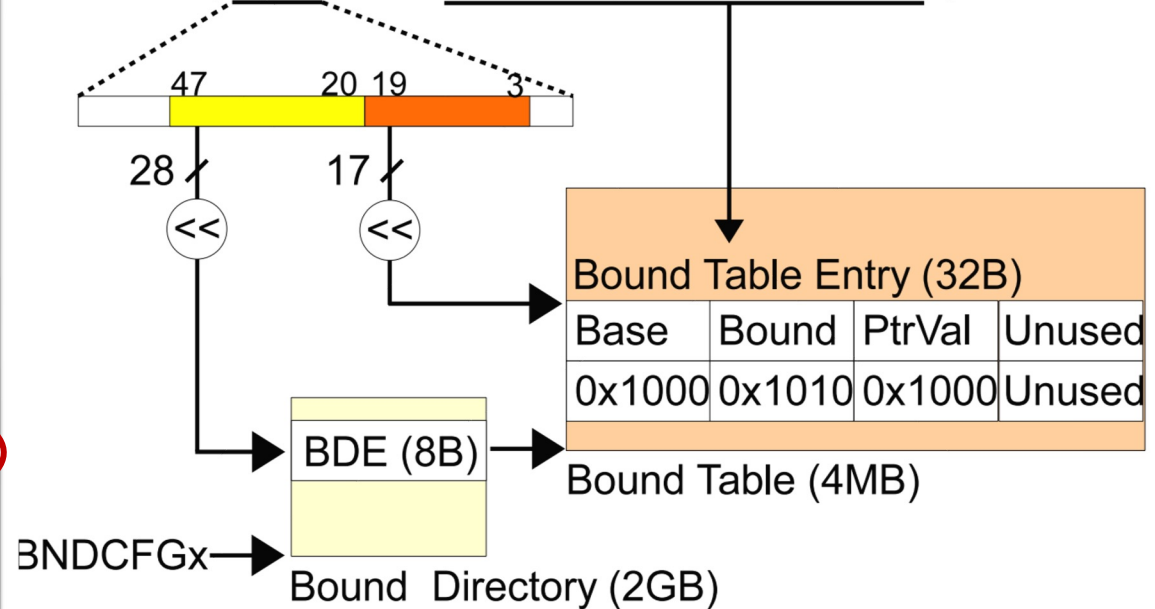
```
p=malloc(16);  
bnd0 = bndmk(p,16);  
bndstx (&p,p,bnd0);  
... // p = p + 4;  
bnd1 = bndldx(&p,p);  
bndcl (&p, bnd1);  
bndcu (&p, bnd1);  
*p = 'a';
```

Store the metadata in a two-level table in hardware



Why two-level?

```
bndstx ( Ptr Addr., Ptr Val., [Base, Bound] );  
bndstx ( &p, 0x1000, [0x1000, 0x1010] );
```



Analysis of Intel MPX

Performance and cost:

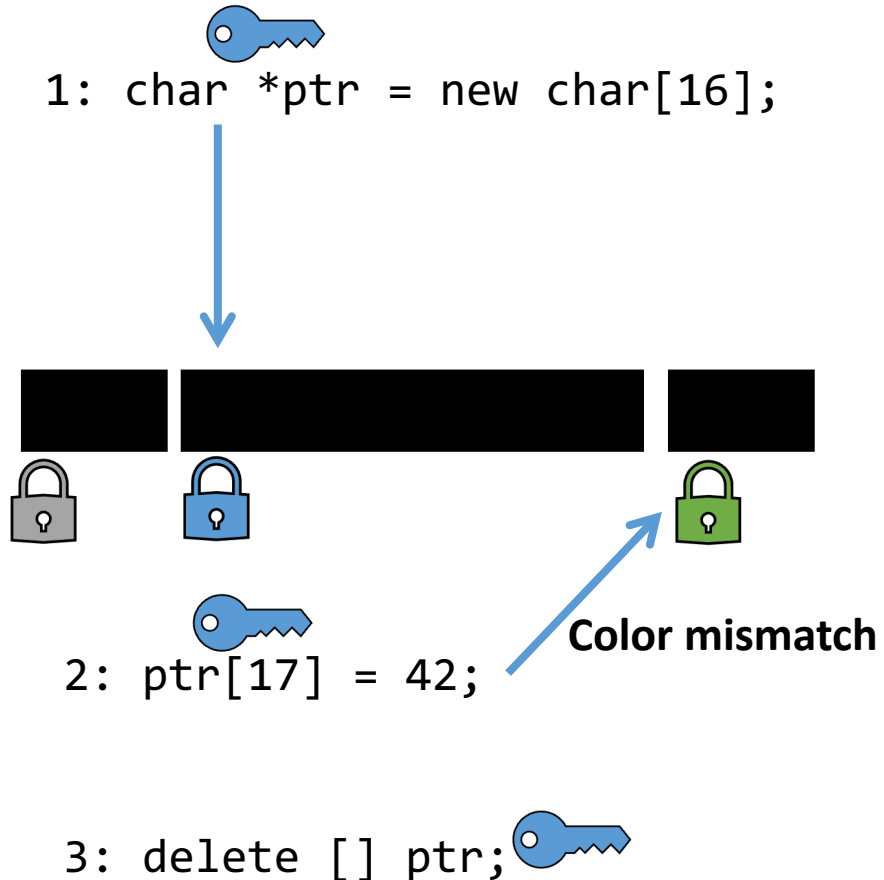
- + Reduce number of instructions, and reduce register pressure
- + No branch instructions, so not pollute the branch predictor
- High overhead: Check is sequential
- + Two-level page table organization should be more area-efficient
- High overhead: loading/storing bounds registers involves two-level table lookup

Compatibility:

- Not straightforward about how to extend the scheme to support temporal safety, etc.
- Does not support multithreading transparently
- All the code need to be rewritten, otherwise either security breaks or correct code broken



ARM MTE (Memory Tagging Extension)



- The concept of keys and locks
- Memory locations are tagged by adding **four bits** of metadata to each **16** bytes of physical memory

ARM MTE (Memory Tagging Extension)



```
1: char *ptr = new char[16];
```



```
2: ptr[17] = 42;
```

```
3: delete [] ptr;
```

- The concept of keys and locks
- Memory locations are tagged by adding **four bits** of metadata to each **16** bytes of physical memory

Analysis of ARM MTE



```
1: char *ptr = new char[16];
```



```
2: ptr[17] = 42;
```

```
3: delete [] ptr;
```

- Where to store tags (key and lock)?
 - Pointer tag is stored in top unused bits inside the pointer (no extra register needed)
 - Physical memory tag is stored in hardware (new hardware needed for both DRAM and cache)
- Limited tag bits
 - Cannot ensure two allocations have different colors
 - But can ensure that the tags of sequential allocations are always different

Analysis of ARM MTE

Security:

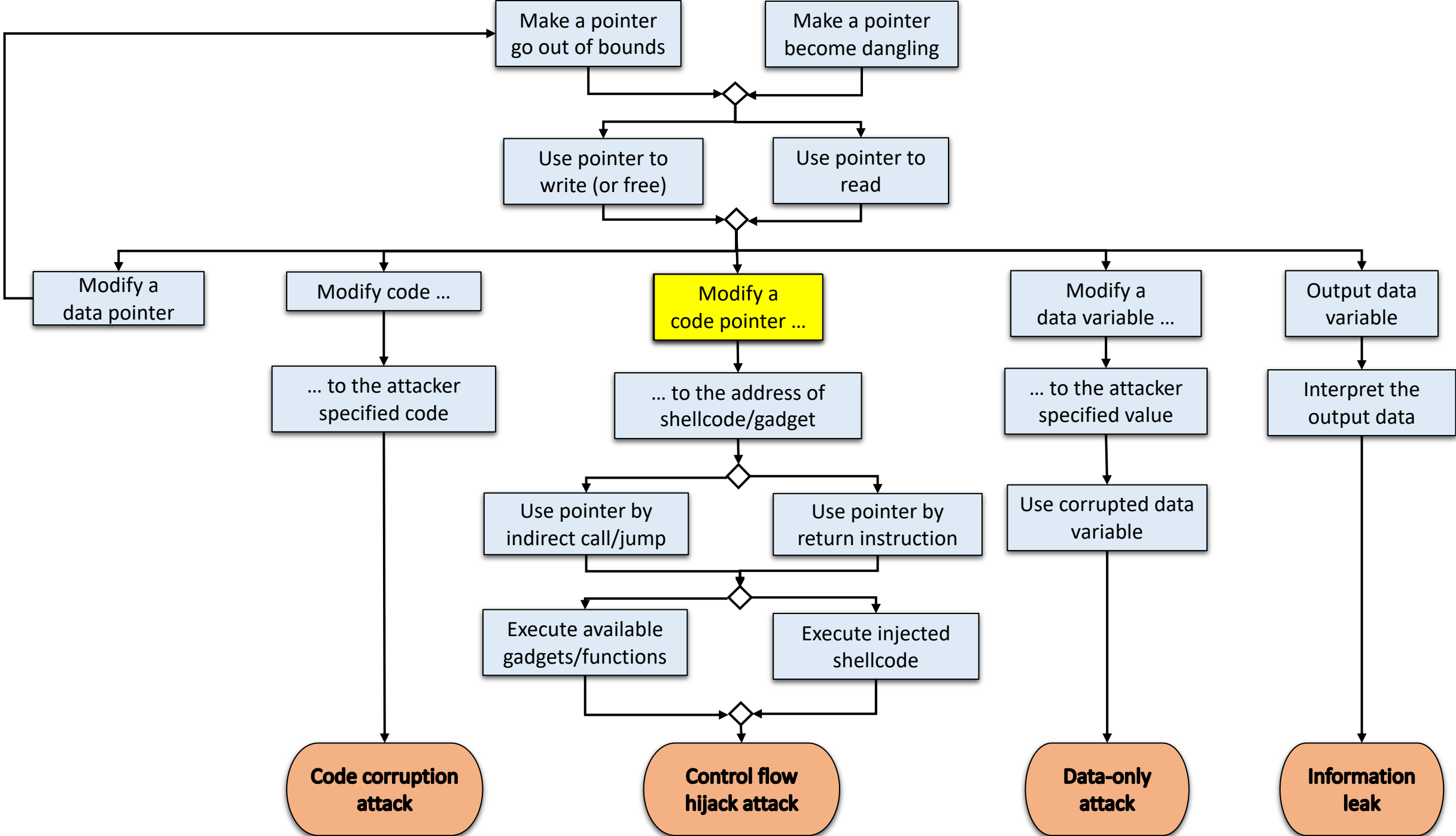
- Coarse-grained spatial safety. Non-sequential violation is detected probabilistically
- + Can support temporal safety similar to spatial safety
- + Other extensions (see HAKC paper)

Performance and other overhead:

- + Storage overhead is ok 4 bits per 64 bytes
- + Performance overhead is low, mostly lies in the allocation and free time, since need to modify tags in bulk

Compatibility:

- + To protect heap, modify libraries to do malloc and free; modify OS to trap on invalid pointer. No extensive code rewritten needed.



Control-flow Integrity

- To maintain code pointer integrity
- Naïve idea:
 - Make pointer immutable (read-only)
 - Only work for global offset table and virtual function tables
- How about other pointers?
 - Return address?
 - Programmer-defined function pointers
 - Change function pointers after changing vtable pointer

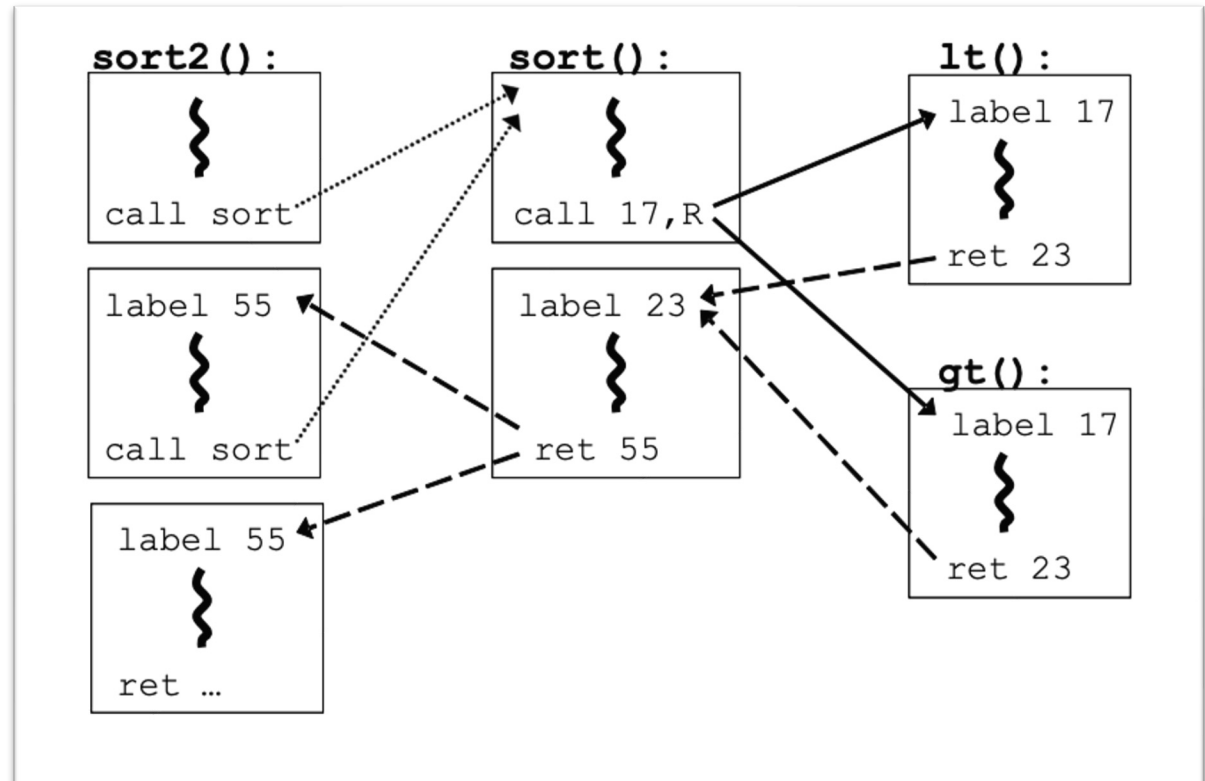
Control Flow Integrity (CFI)

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

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

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

sort(int x[], int len, fun_ptr)
{
    for(int i=0; ...)
        for (int j=i; ...)
            if (fun_ptr(x[i], x[j]))
                ... //swap x[i] and x[j]
}
```



Intel® Control-Flow Enforcement Technology (Intel CET)

INTEL
CET

=

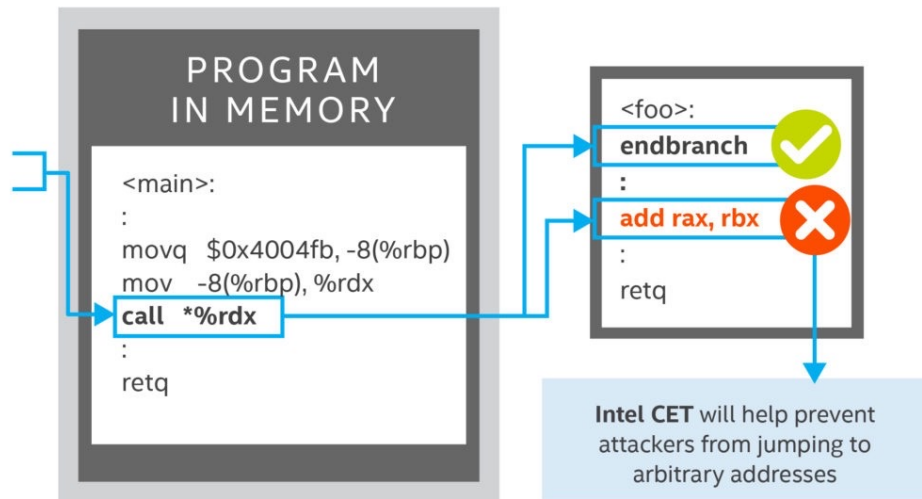
INDIRECT BRANCH
TRACKING (IBT)

+

SHADOW
STACK (SS)

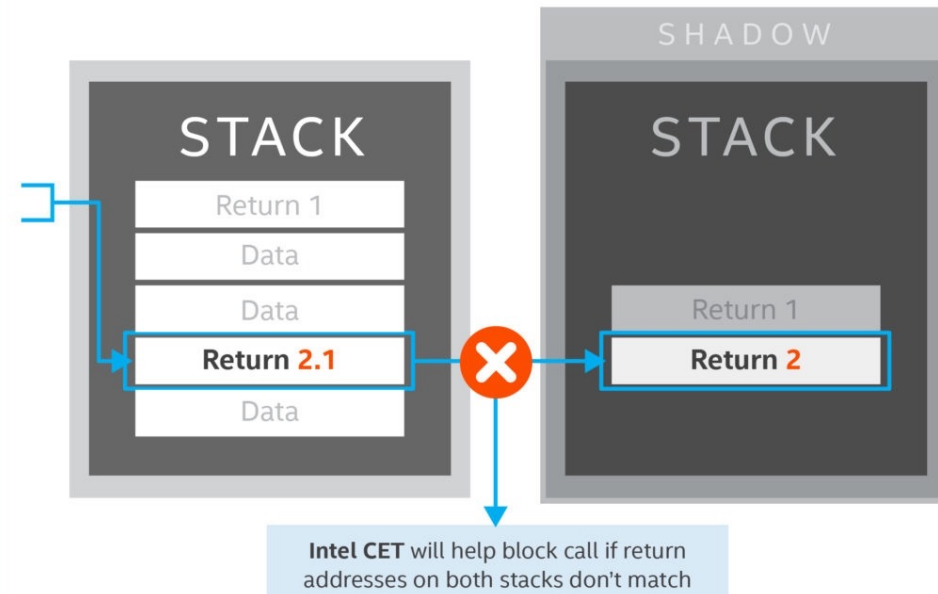
INDIRECT BRANCH TRACKING (IBT)

IBT delivers indirect branch protection to defend against jump/call oriented programming (JOP/COP) attack methods.



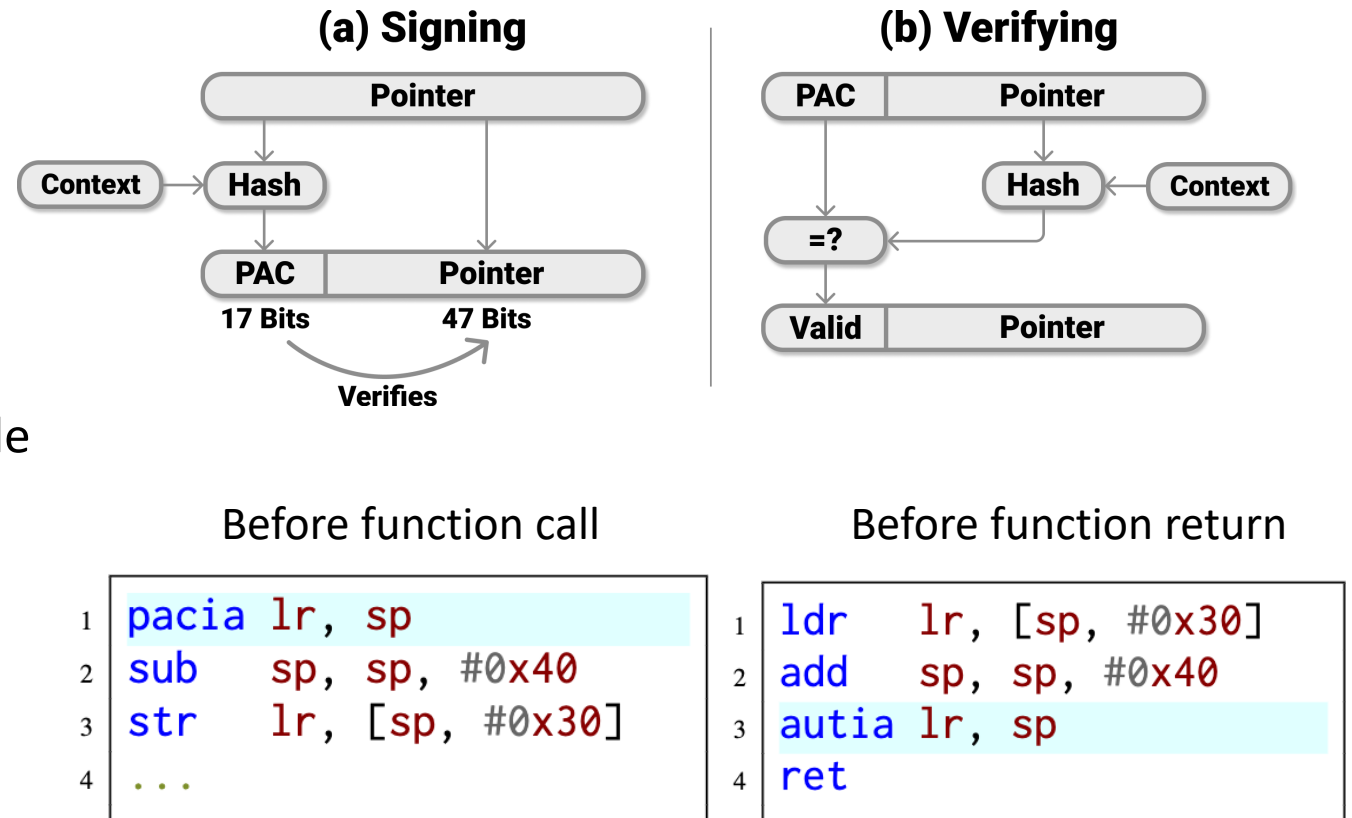
SHADOW STACK (SS)

SS delivers return address protection to defend against return-oriented programming (ROP) attack methods.



ARM PA (Pointer Authentication)

- Widely used in Apple processors
- Motivation:
 - 64-bit pointer, but 48-bit virtual address space
 - Unused high bits
- Hash:
 - A tweakable message authentication code (MAC)
 - ARM calls it **PAC (pointer authentication code)**
- Context:
 - secret key
 - salt (could be the stack pointer)



Summary

- Memory corruption problems: An eternal war
- Attack variations and mitigations
- Trade-off in hardware support

