Hardware Support for Memory Safety

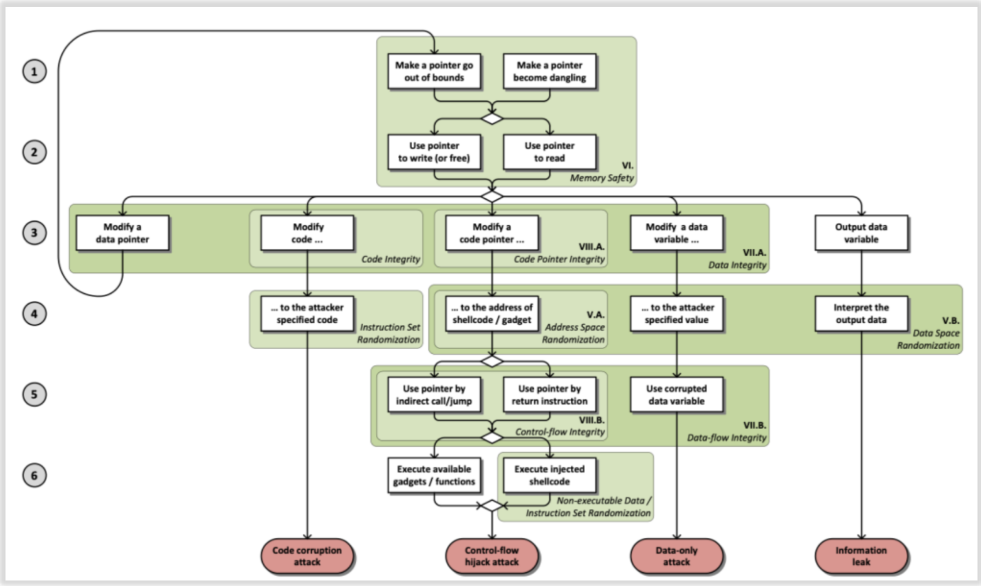
Mengjia Yan

Spring 2025

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 memory location where variable is stored
- It is programmers' responsibility to do **pointer check**, e.g. NULL, out-of-bound, use-after-free

Buffer Overflow Example

DEMO

```
char * buffer1 = (char*)malloc(buffer1_size);
char * buffer2 = (char*)malloc(buffer2_size);

// fill up buffer2 with 'B'
for (int i=0; i<buffer2_size; i++){
   buffer2[i] = 'B';
 }

// fill up buffer1 with 'A'
for (int i=0; i< 2 * buffer1_size ; i++){
   buffer1[i] = 'A';
 }
```

Use-after-free Example

```
char * buffer1 = (char*)malloc(buffer1_size);
char * buffer2 = (char*)malloc(buffer2_size);
// fill buffer1 and buffer2 with ‘A’ and ‘B’ respectively

 free(buffer1);
 char *buffer3 = (char*)malloc(buffer3_size);
 for (int i=0; i<buffer3_size; i++){
   buffer3[i] = 'C';
 }
```

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
- 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

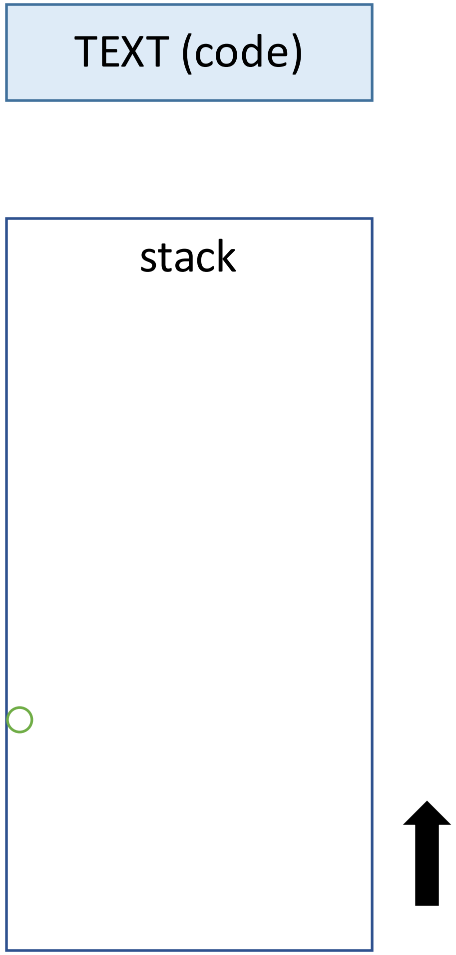
From software bugs to attacks?

Recap Stack Operations

```
int hello(){
    int a = 100;
    return a;
}

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

ra

How will the stack look like during the execution?



Stack Smash

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

int main() {
  …
  func (input);
  …
}
```

ra

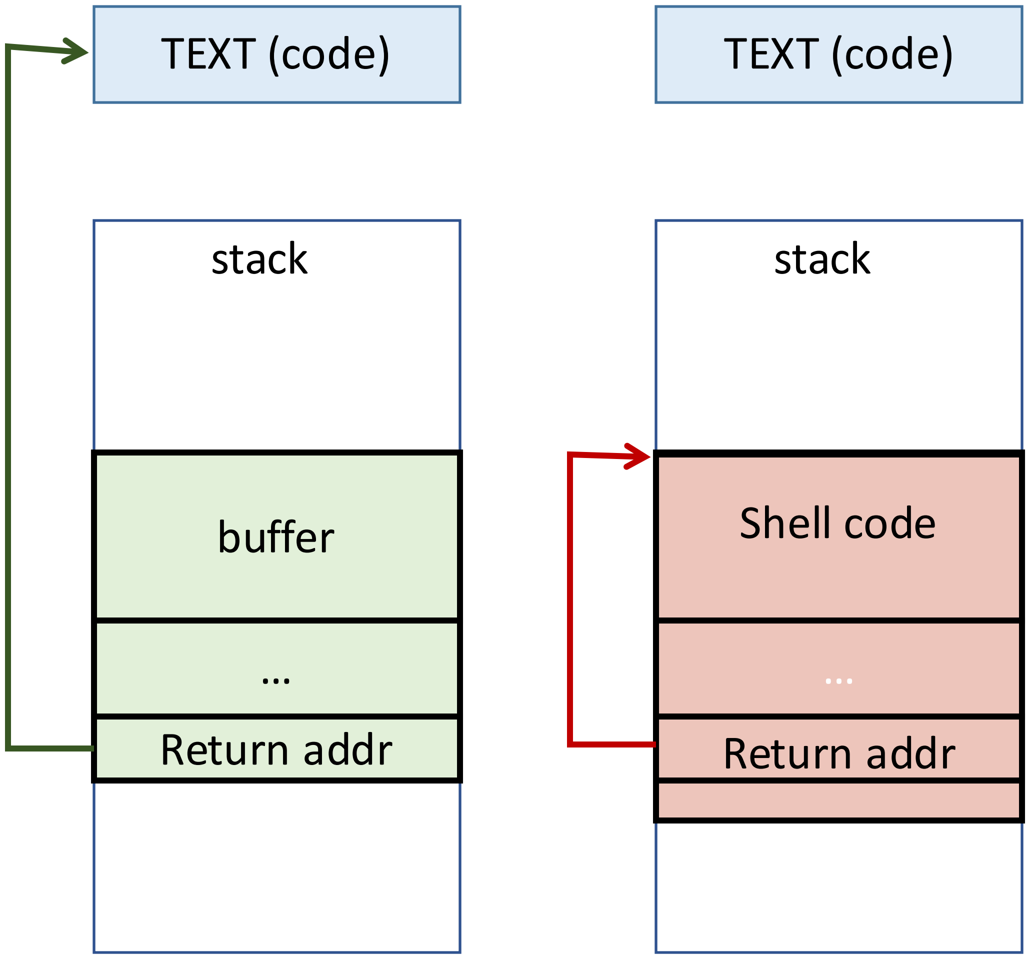
Shell code:

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

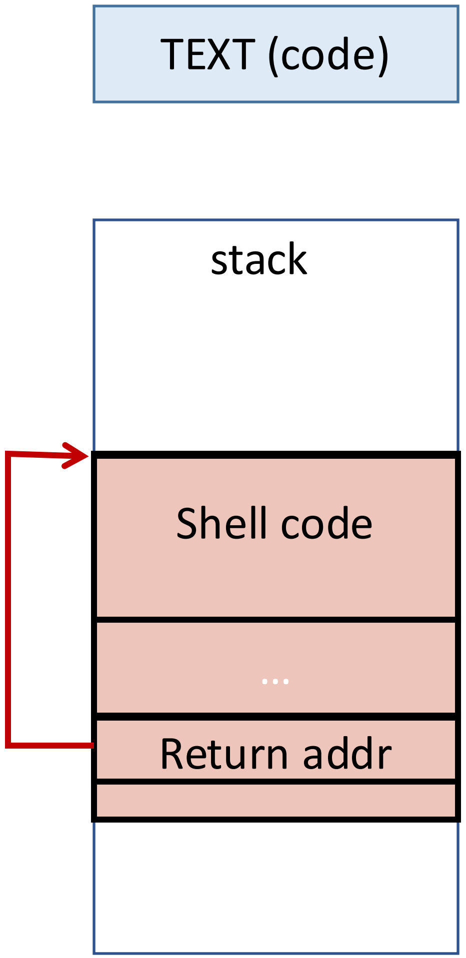
Input str:

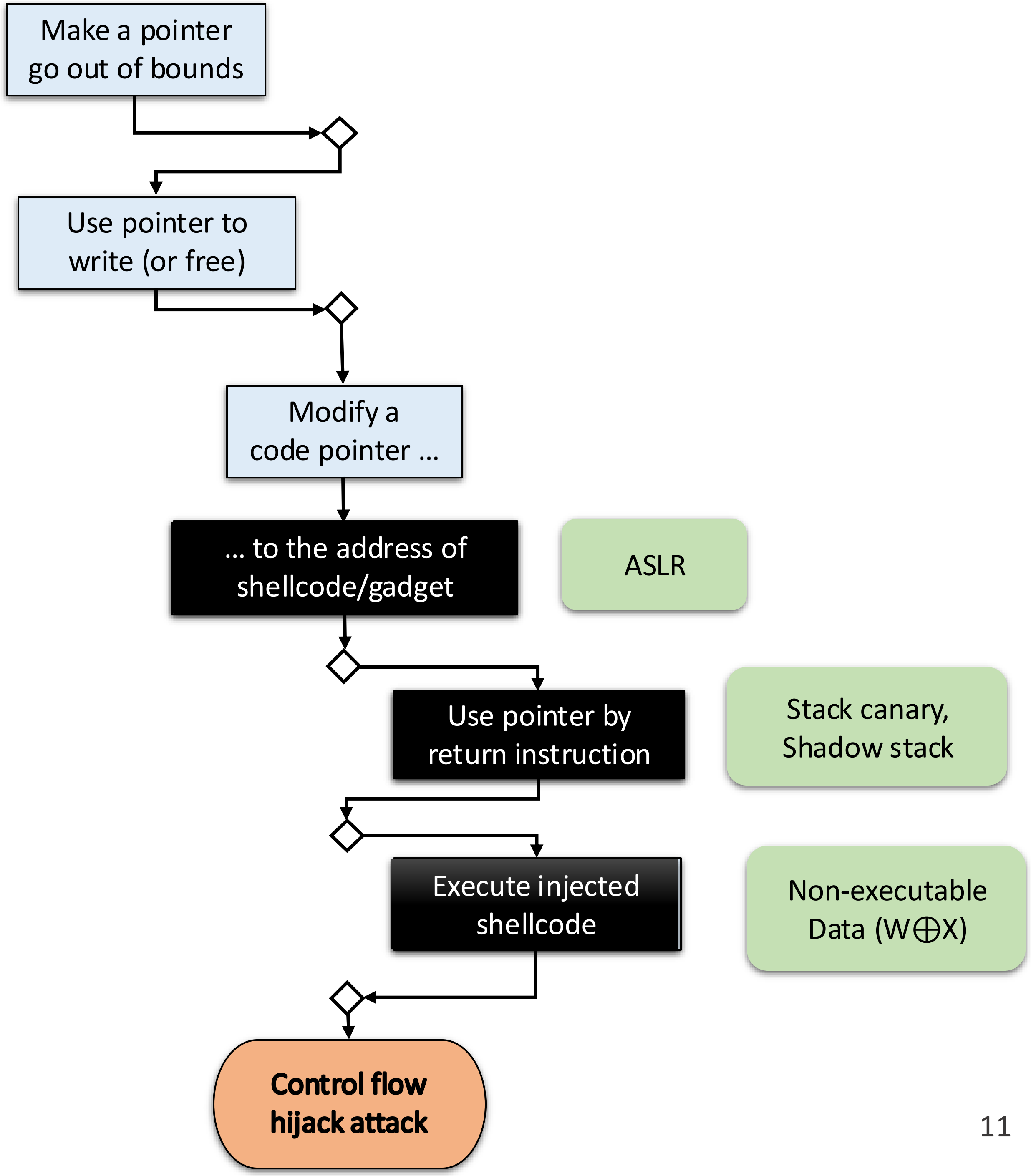
```
Shell code
.. Some padding..
Address of buffer
```

Stack Smash / Code Injection Attack

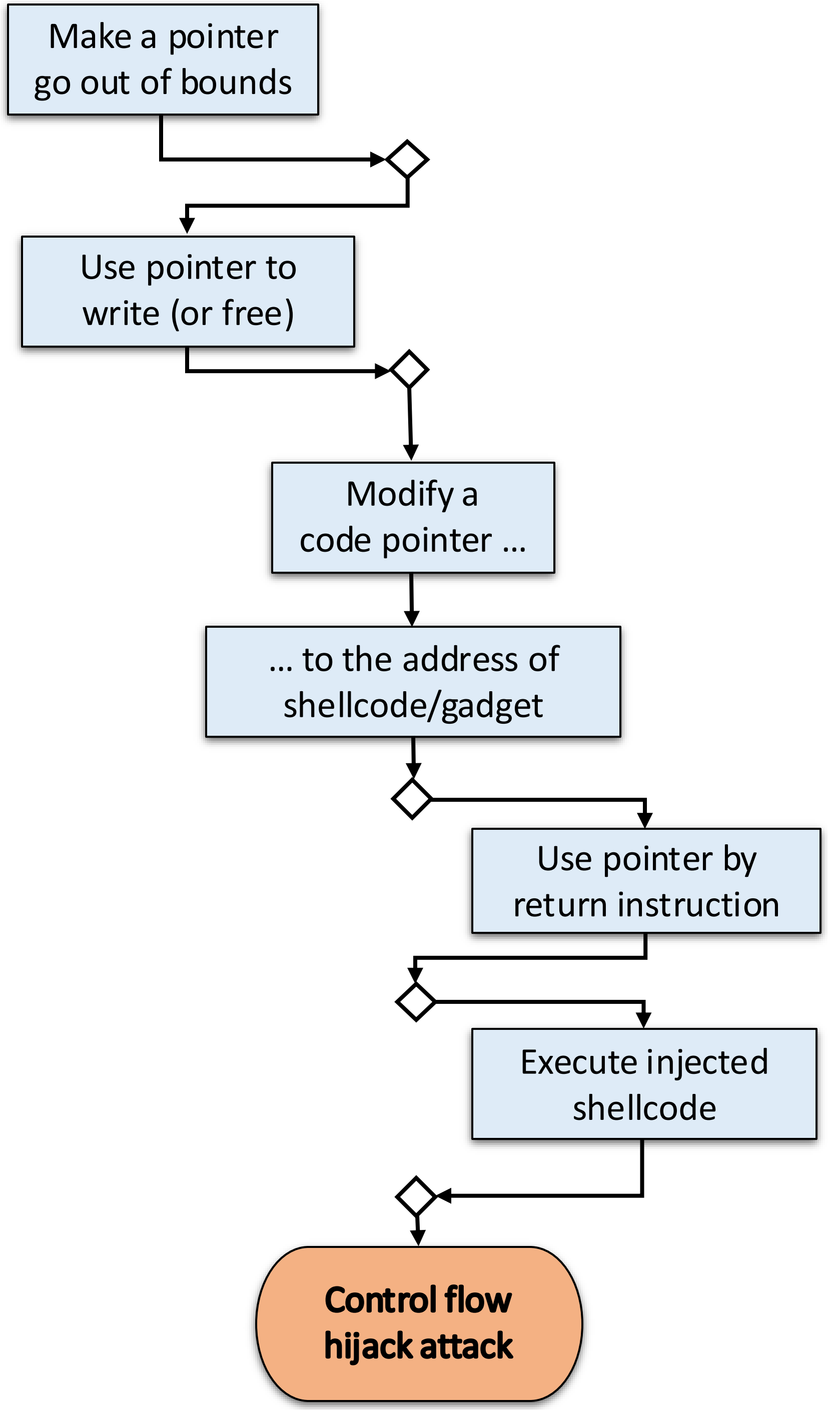


Mitigations



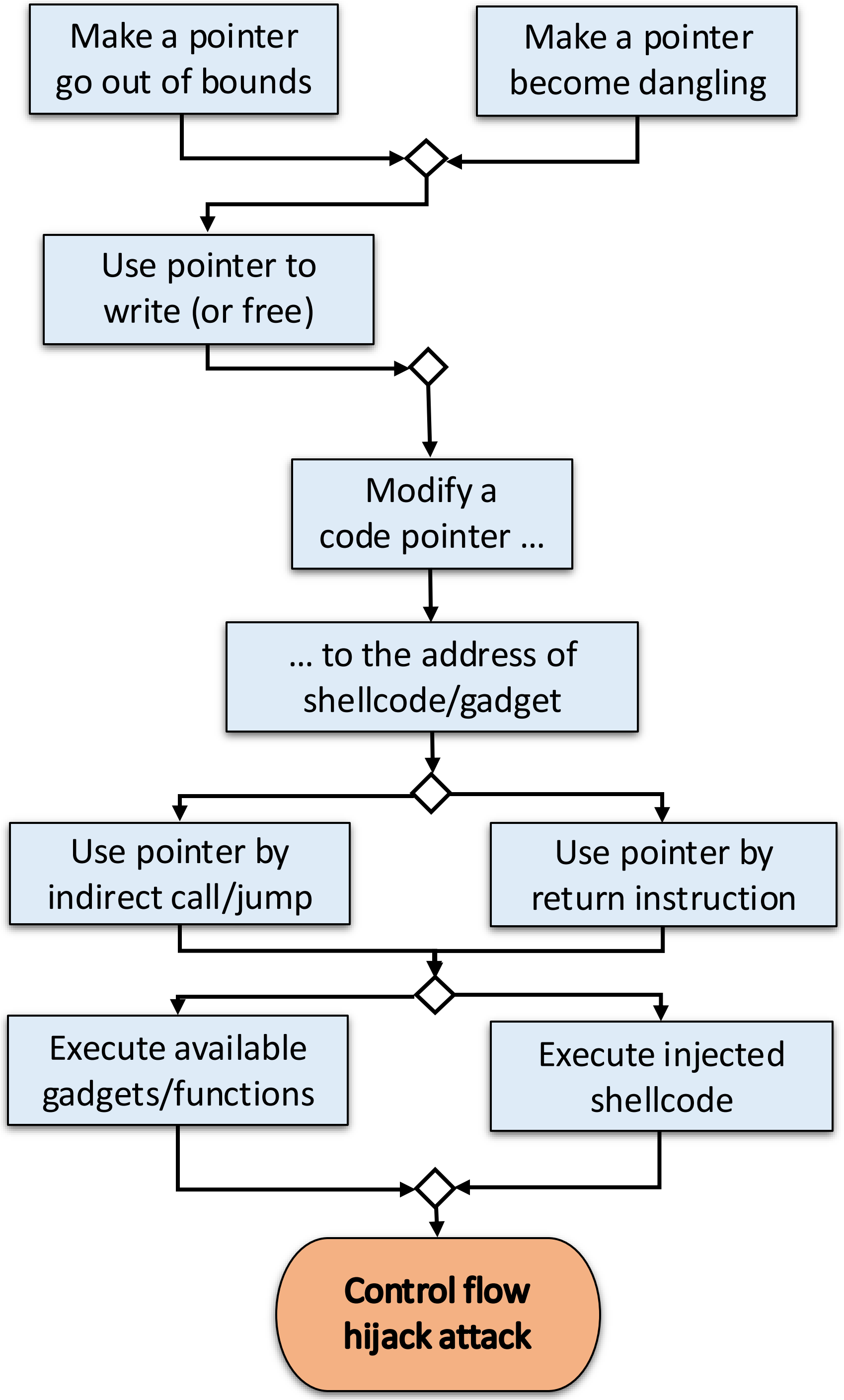


Attack Variations

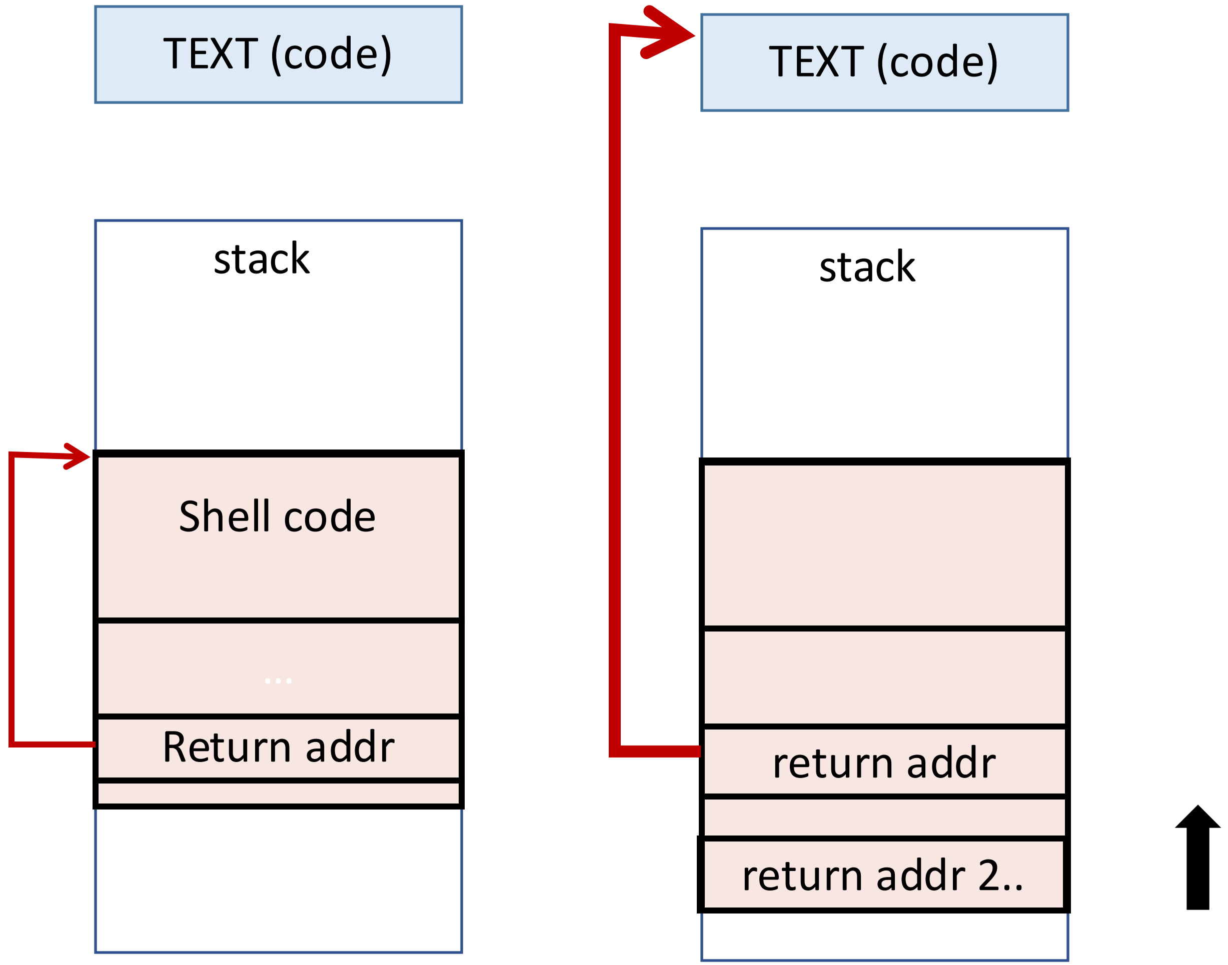


L. Szekeres, M. Payer, T. Wei and D. Song, "SoK: Eternal War in Memory," S&P'2013

Attack Variations



Return-Oriented Programming (ROP)



Gadget example:

```
pop rdi
ret
```

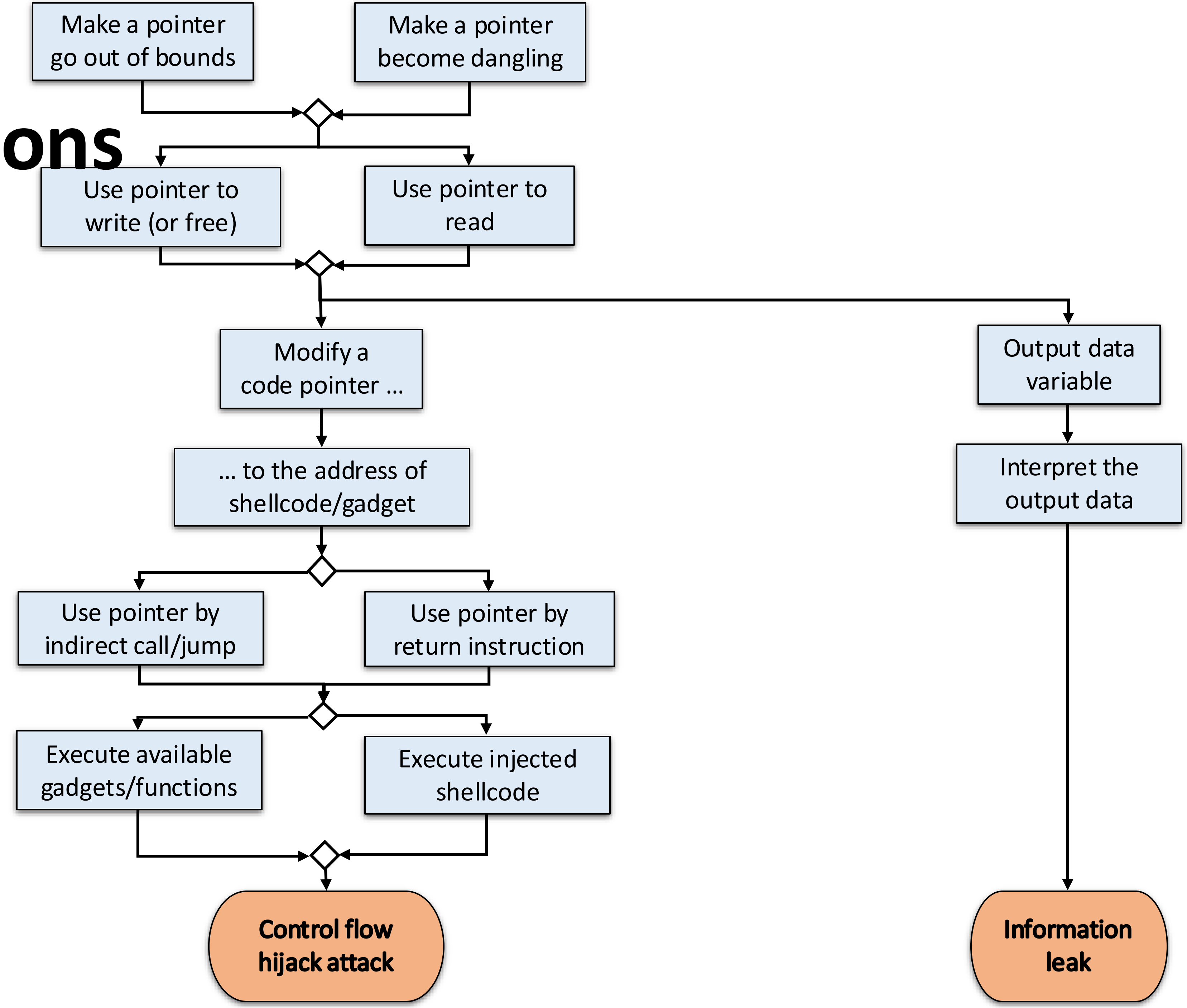
An Attack Puzzle

Jump-oriented programming

```
0x4011b6: int func0(u_int64_t s);
……
0x4011f2: int func1(u_int64_t s);
……
0x40122e: int hack(u_int64_t s);
……
```

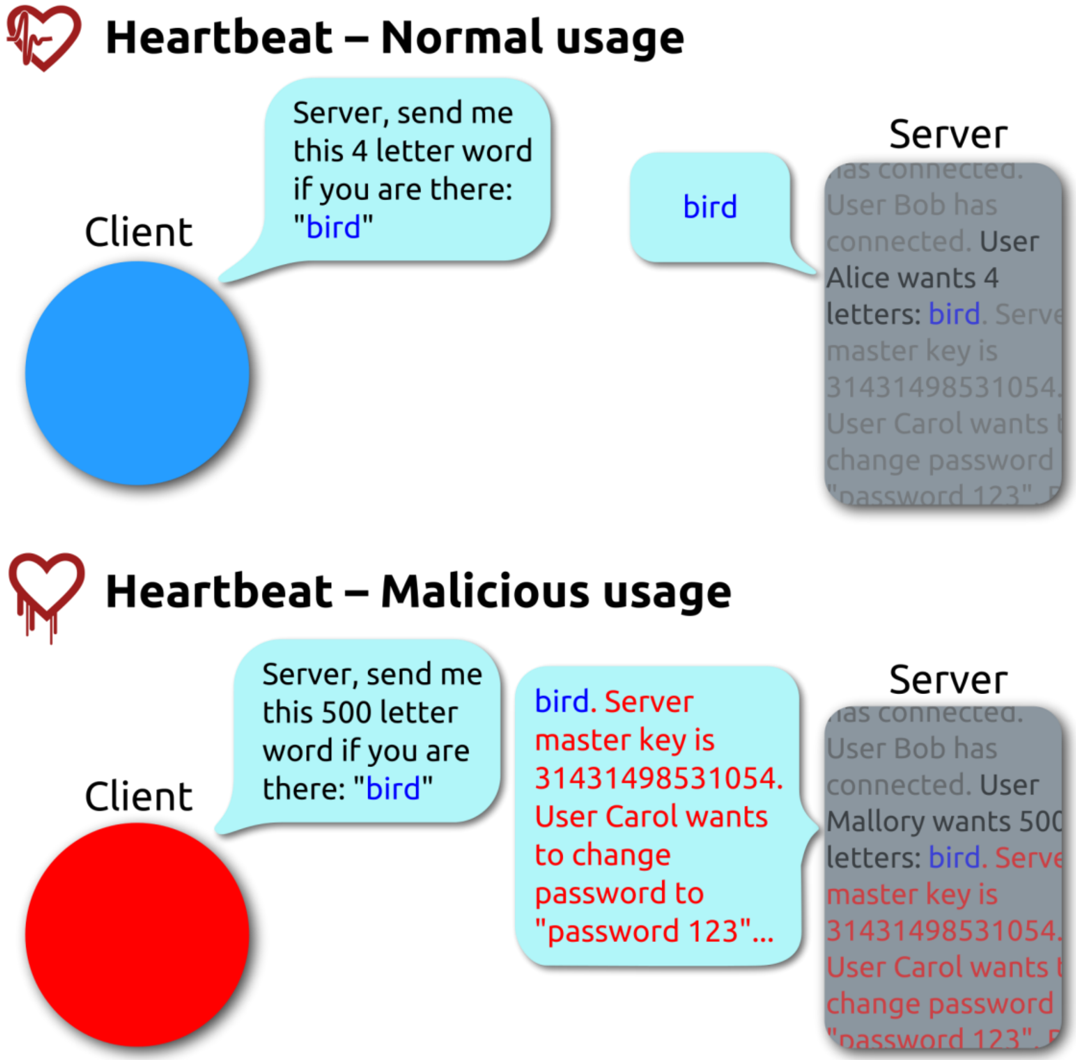
```
int main(int argc, char *argv[]){
  // an array with two function pointers
  // to emulate a vtable
  int (*func_ptr[2])(u_int64_t) = {&func0, &func1};
  u_int64_t val;
  int choice = atoi(argv[1]);

  // obtain ”val” from input; Attacker-controlled
  val = strtol(argv[2], NULL, 16);
  // index into the array to make a function call
  (*func_ptr[choice])(val);
}
```

Attack Variations



HeartBleed Vulnerability

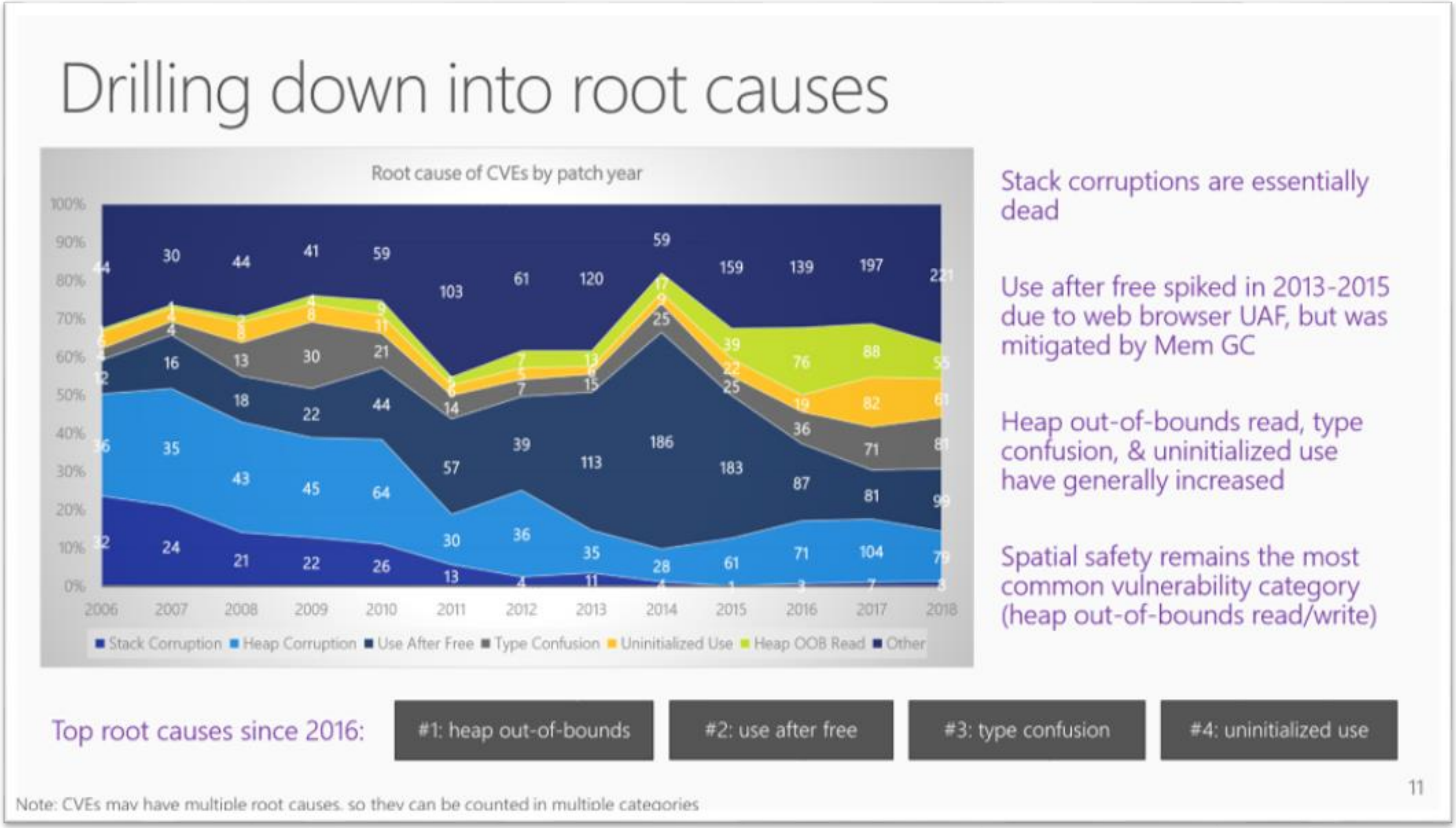
- Publicly disclosed in April 2014
- Missing a bound check
- Bug in the OpenSSL cryptographic software library heartbeat extension



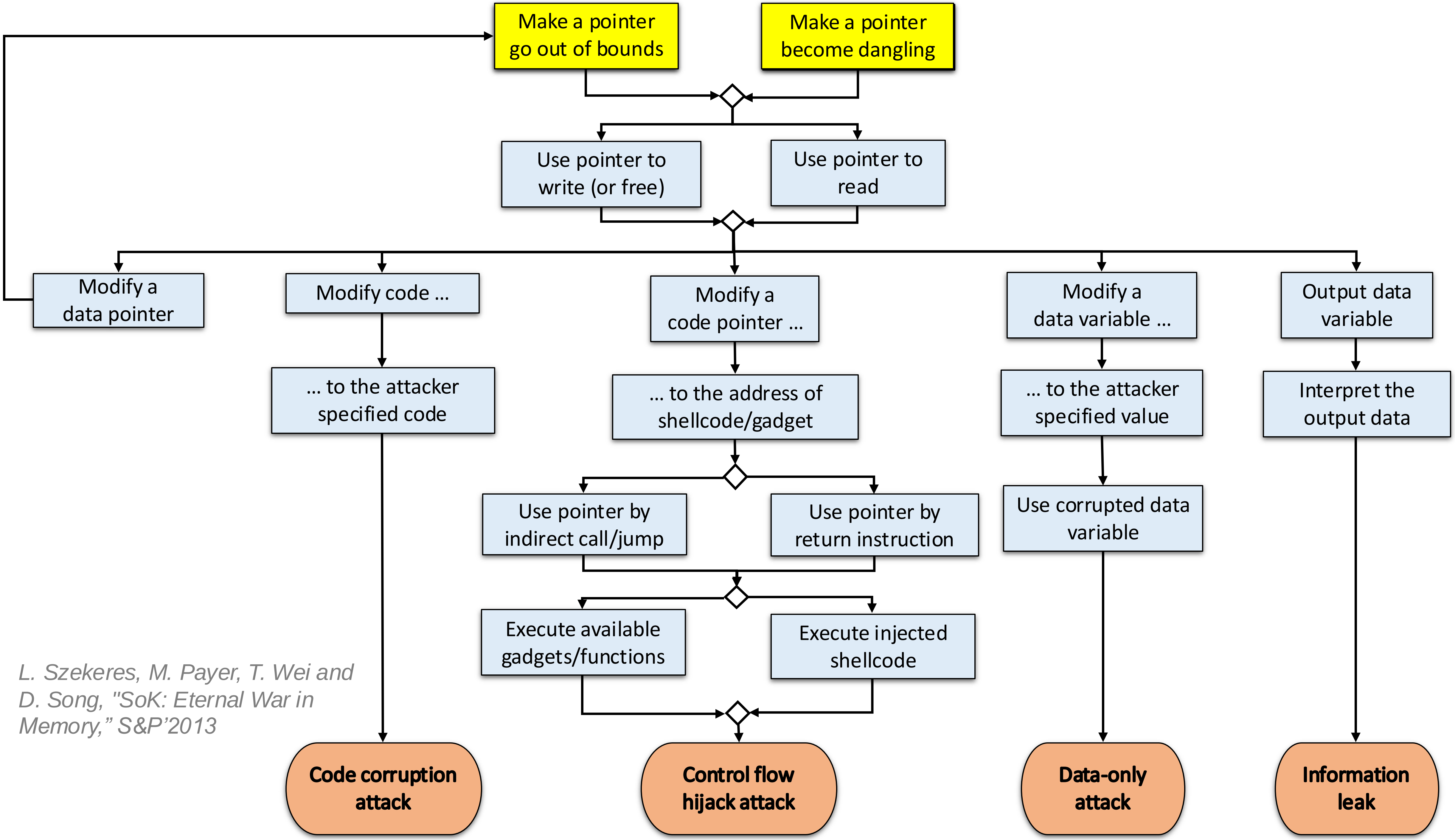
https://heartbleed.com/

Trend reported by Microsoft

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



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)

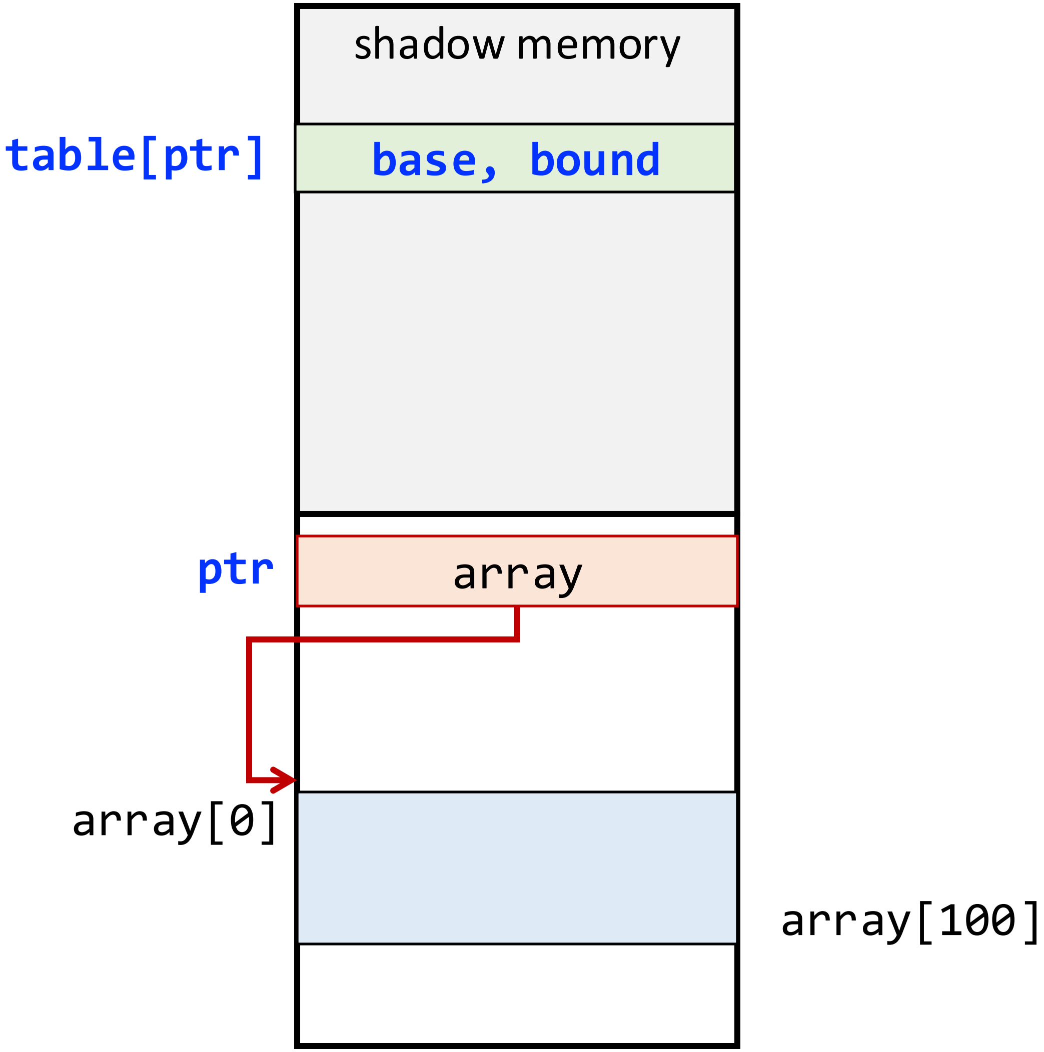
- Software solutions
  - Problem #1: performance overhead, extra instructions to perform the check
  - Problem #2: where to store metadata? -> in shadow memory

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C; Nagarakatte et al; PLDI'09

SoftBound

Creating a pointer:

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



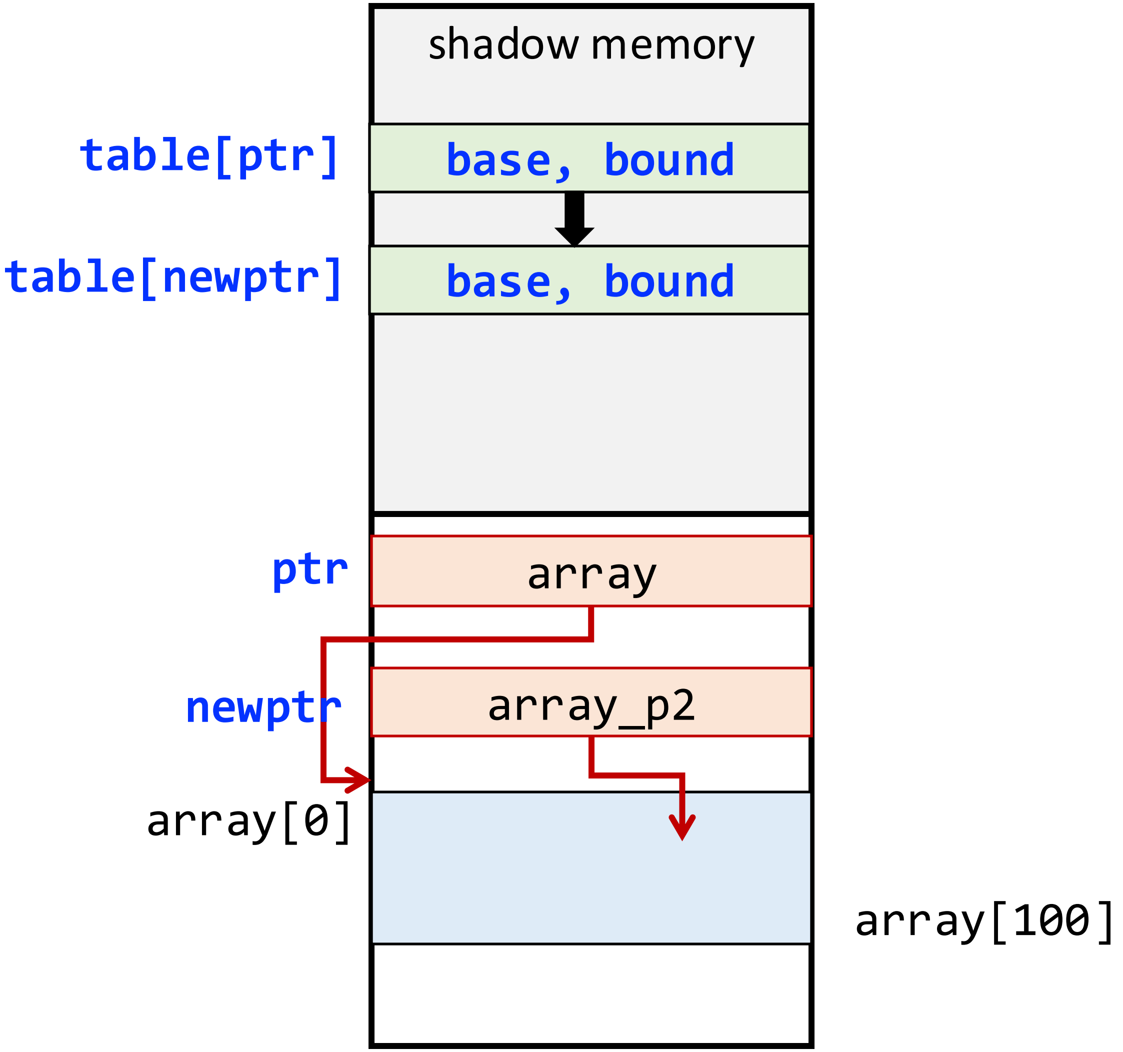
SoftBound

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

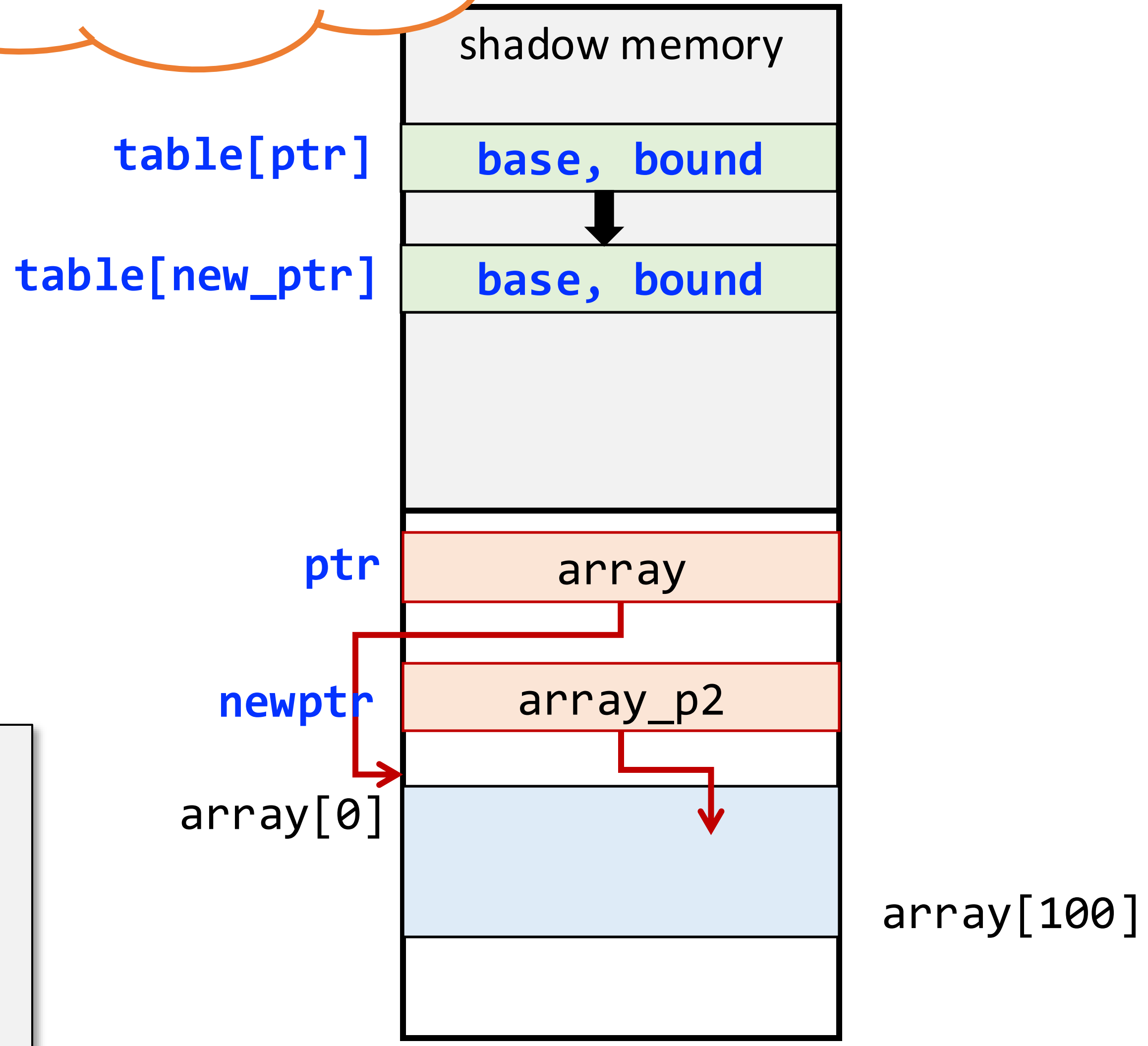
How many memory accesses do we need to do one array access?

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

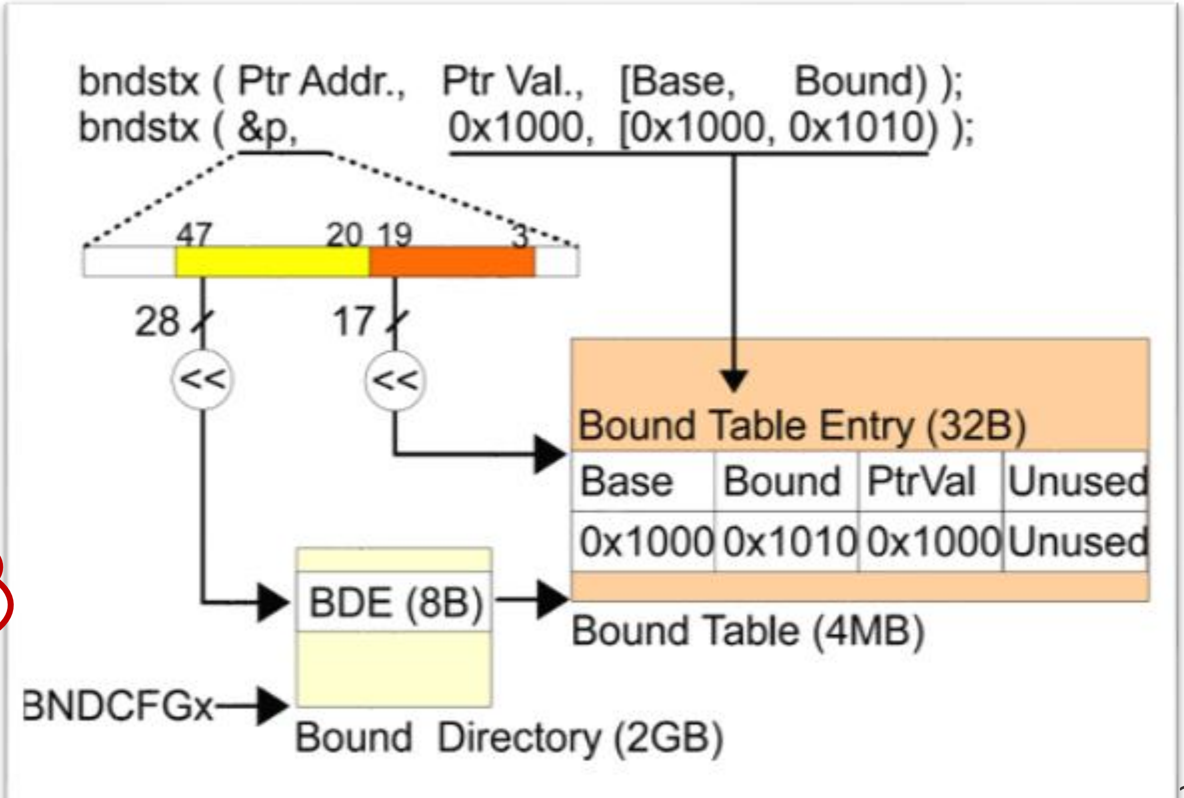
```
p=malloc(16);
... // p = p + 4;
*p = 'a';
```

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?



BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free; Zhang et al; ASPLOS'19

Analysis of Intel MPX

Performance and cost:

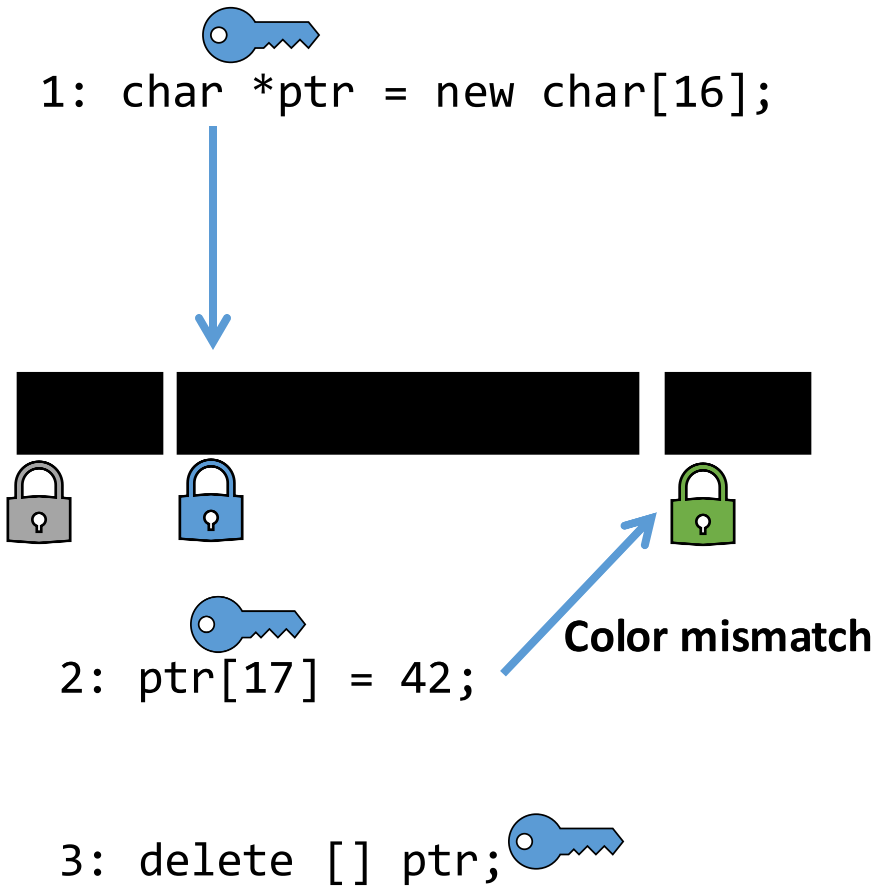
+ Reduce number of instructions, and reduce register pressure
+ No branch instructions, so not pollute the branch predictor
- High overhead: Check is sequential
+ Two-level page table organization should be more area-efficient
- High overhead: loading/storing bounds registers involves two-level table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

Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack; OLEKSENKO et al; SIGMETRICS'18

ARM MTE (Memory Tagging Extension)

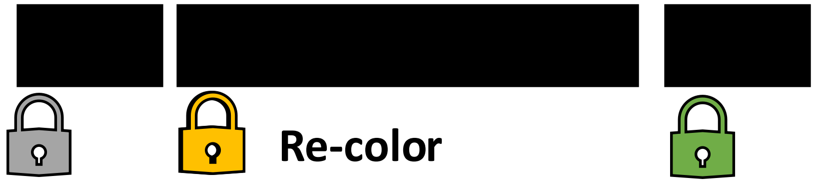


- The concept of keys and locks
- Memory locations are tagged by adding four bits of metadata to each 16 bytes of physical memory

Armv8.5-A Memory Tagging Extension White paper
https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html

ARM MTE (Memory Tagging Extension)

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




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

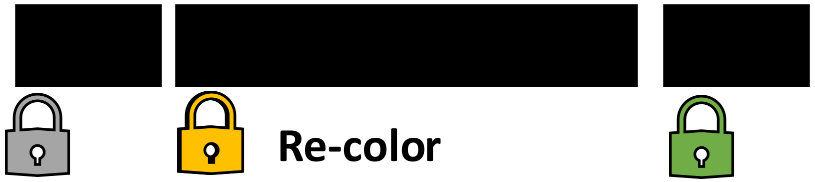
```
3: delete [] ptr;
```

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

Armv8.5-A Memory Tagging Extension White paper
https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html

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

Armv8.5-A Memory Tagging Extension White paper
https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html

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)

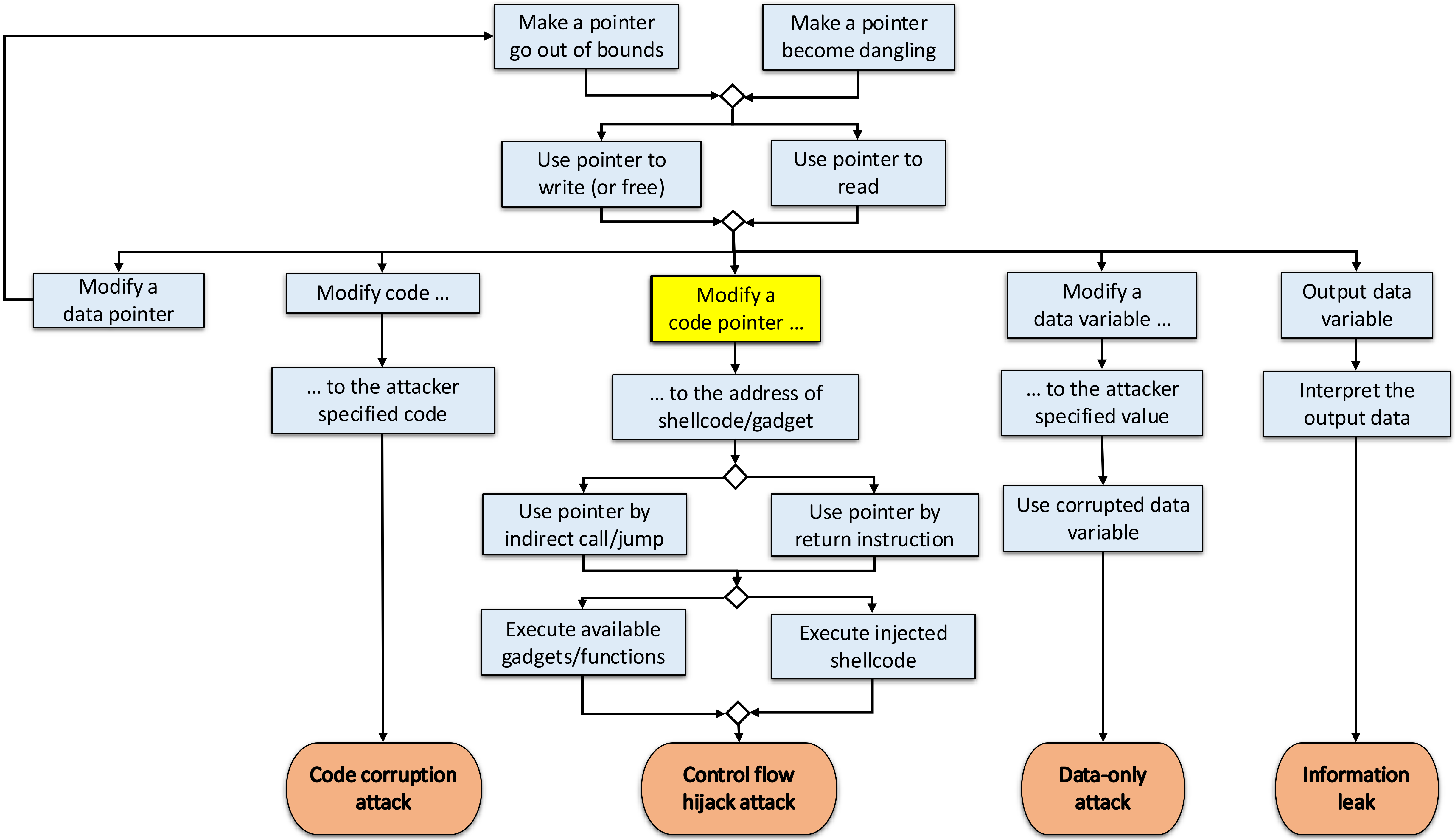
Performance and other overhead:

+ 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.


Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
Modify a data pointer
Modify code ...
Modify a code pointer ...
Modify a data variable ...
Output data variable
... to the attacker specified code
... to the address of shellcode/gadget
... to the attacker specified value
Interpret the output data
Use pointer by indirect call/jump
Use pointer by return instruction
Use corrupted data variable
Execute available gadgets/functions
Execute injected shellcode
Code corruption attack
Control flow hijack attack
Data-only attack
Information leak

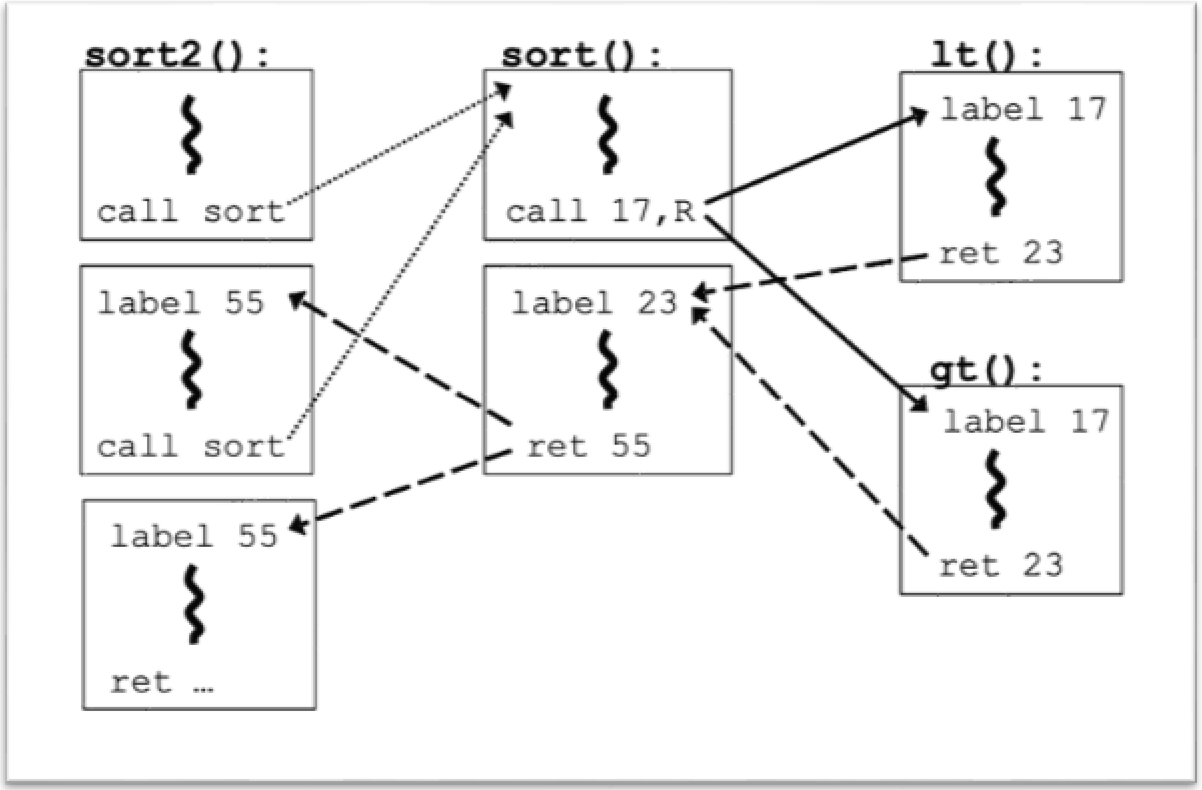
Control-flow Integrity

- 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]
}
```



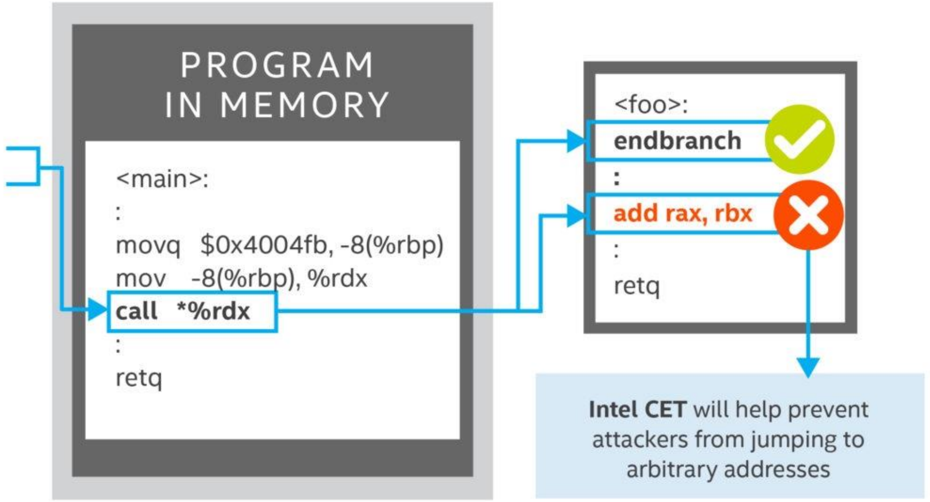
Control-Flow Integrity Principles, Implementations, and Applications; Mart´ın Abadi, et al. CCS'05

Intel® Control-Flow Enforcement Technology (Intel CET)

INTEL CET = **INDIRECT BRANCH TRACKING (IBT)** + **SHADOW STACK (SS)**

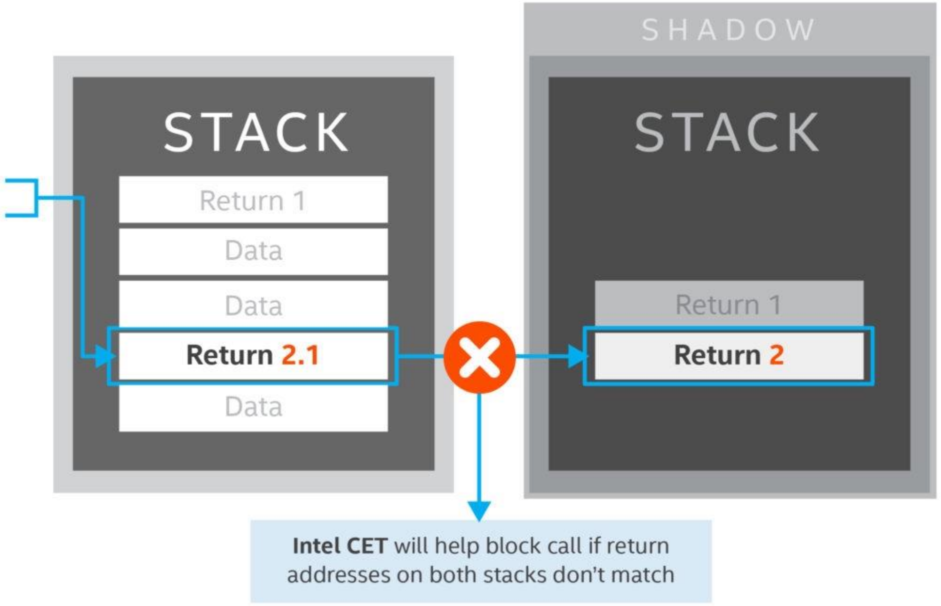
INDIRECT BRANCH TRACKING (IBT)

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



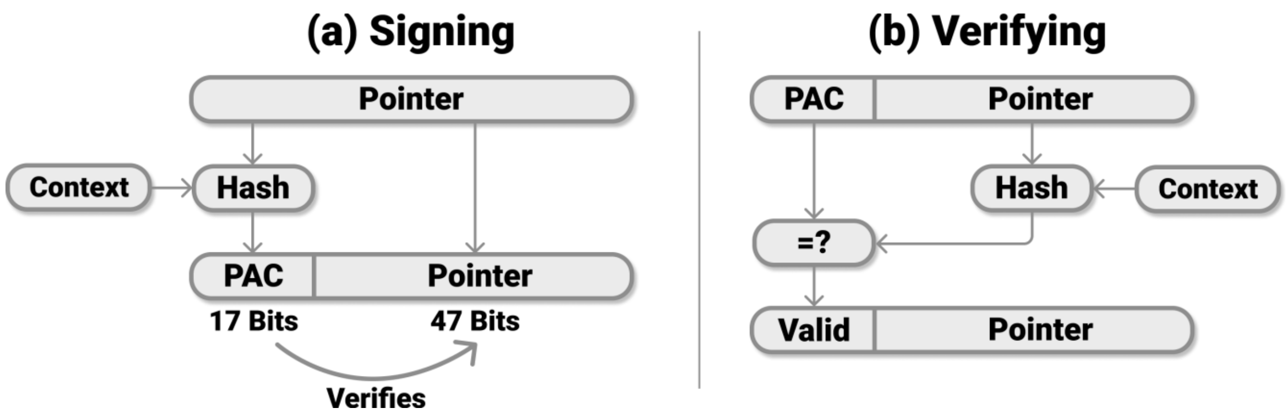
SHADOW STACK (SS)

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)



Before function call

```
pacia lr, sp
sub   sp, sp, #0x40
str   lr, [sp, #0x30]
...
```

Before function return

```
ldr   lr, [sp, #0x30]
add   sp, sp, #0x40
autia lr, sp
ret
```

Summary

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