“SoK: Eternal War in Memory”

Presented by Mengjia Yan

MIT 6.888 Fall 2020

Overview

- Discuss the paper “SoK: Eternal War in Memory” with concrete examples

- Recent progress on memory safety
  - With a focus on hardware/architecture

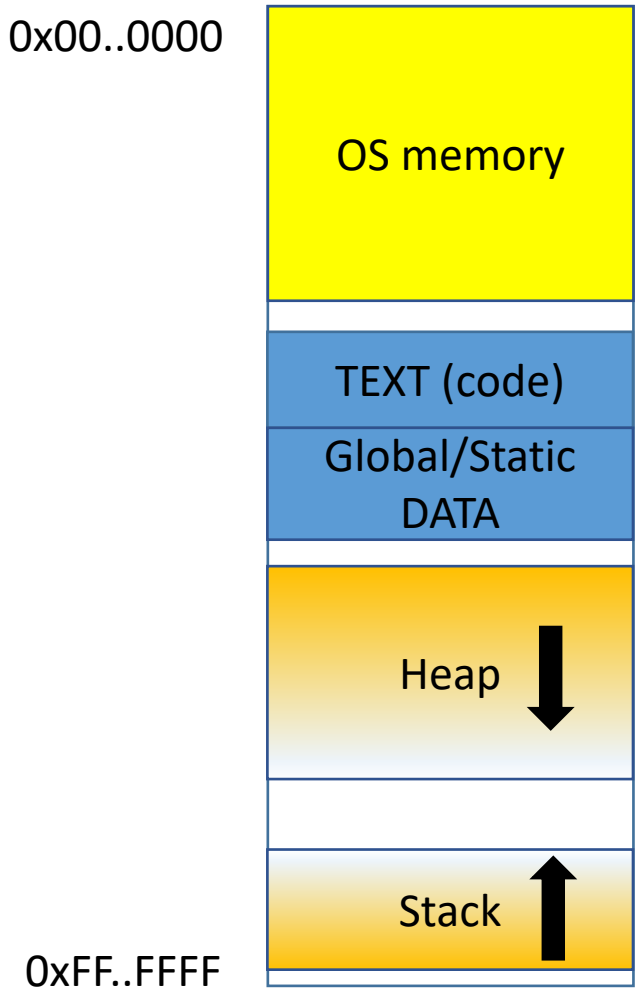
Motivation

- C/C++ is unsafe
- Everybody runs C/C++ code
- They surely have exploitable vulnerabilities

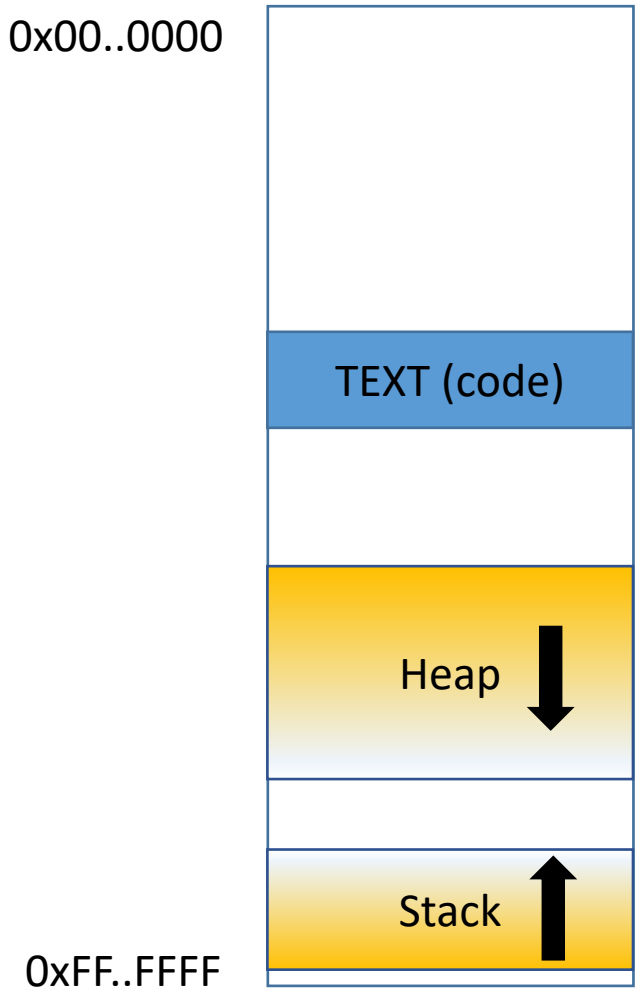
Low-level Language Basics (C/C++/Assembly)

- Programmers have more control
  - + Efficient
  - - Bugs
  - - Programming productivity

- **Pointers**
  - Address of variables (uint64): index of memory location where variable is stored
  - Programmers need to do pointer check, e.g. NULL, out-of-bound, use-after-free

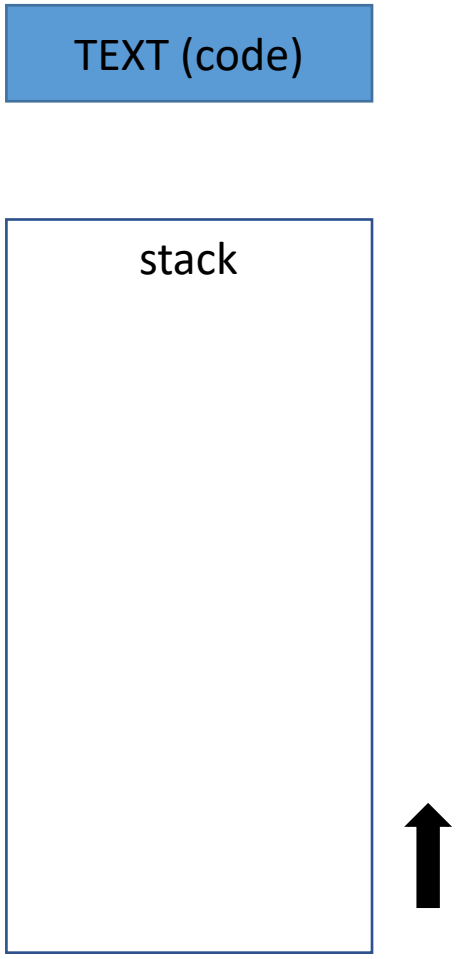


Low-level Language Basics



Low-level Language Basics

```c
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;
}
```



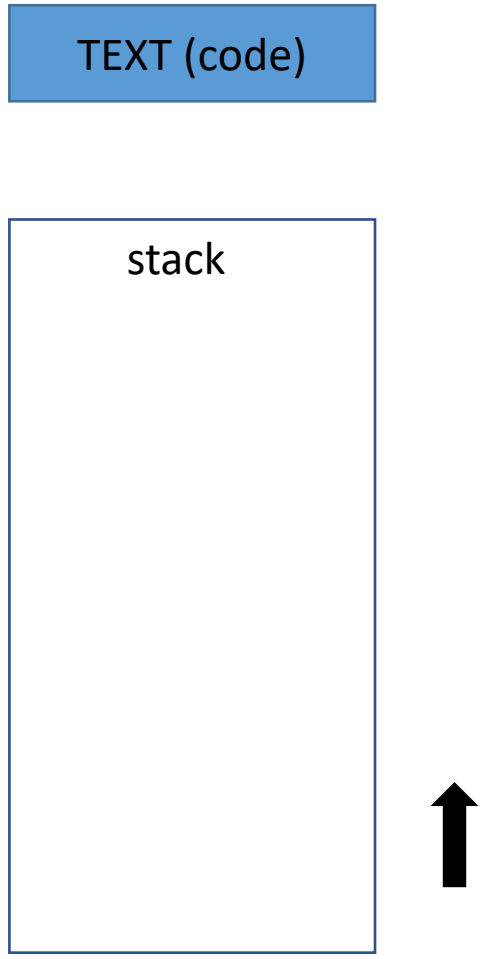
Attacks

Code Injection Attack Example

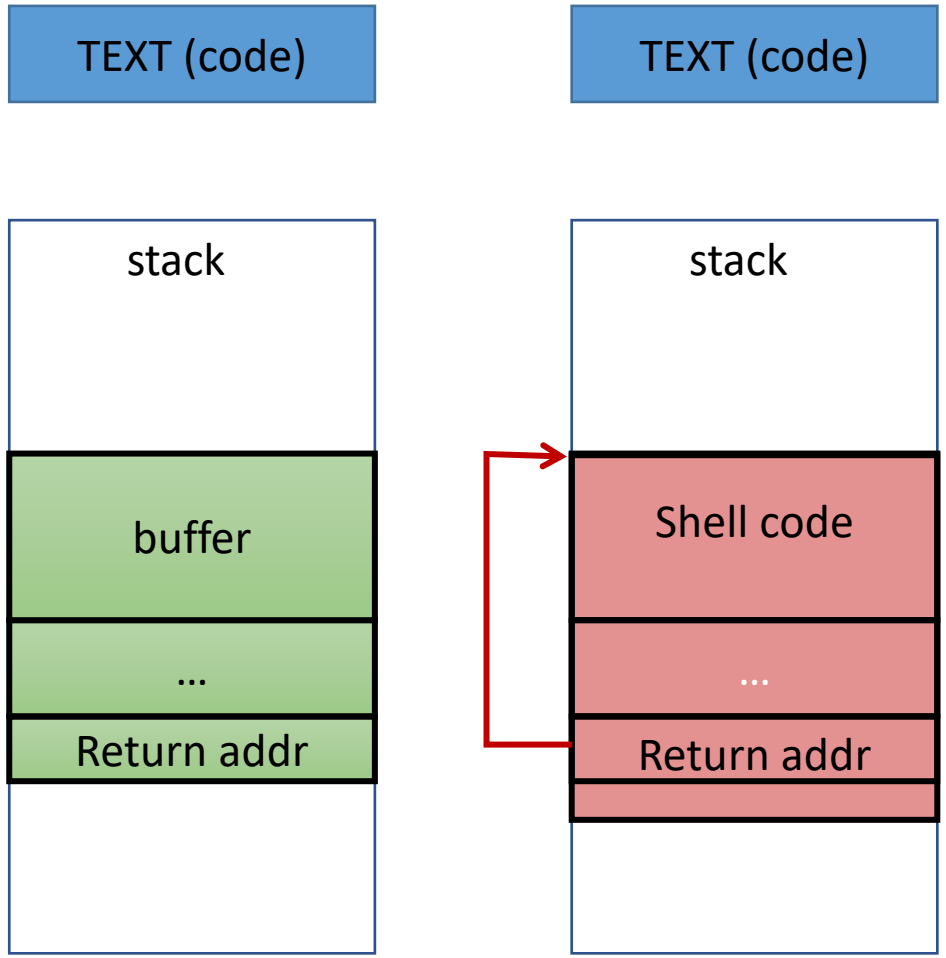
```
int func (char *str) {
        char buffer[12];
        strncpy(buffer, str, len(str));
        return 1;
}

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

Shell code:

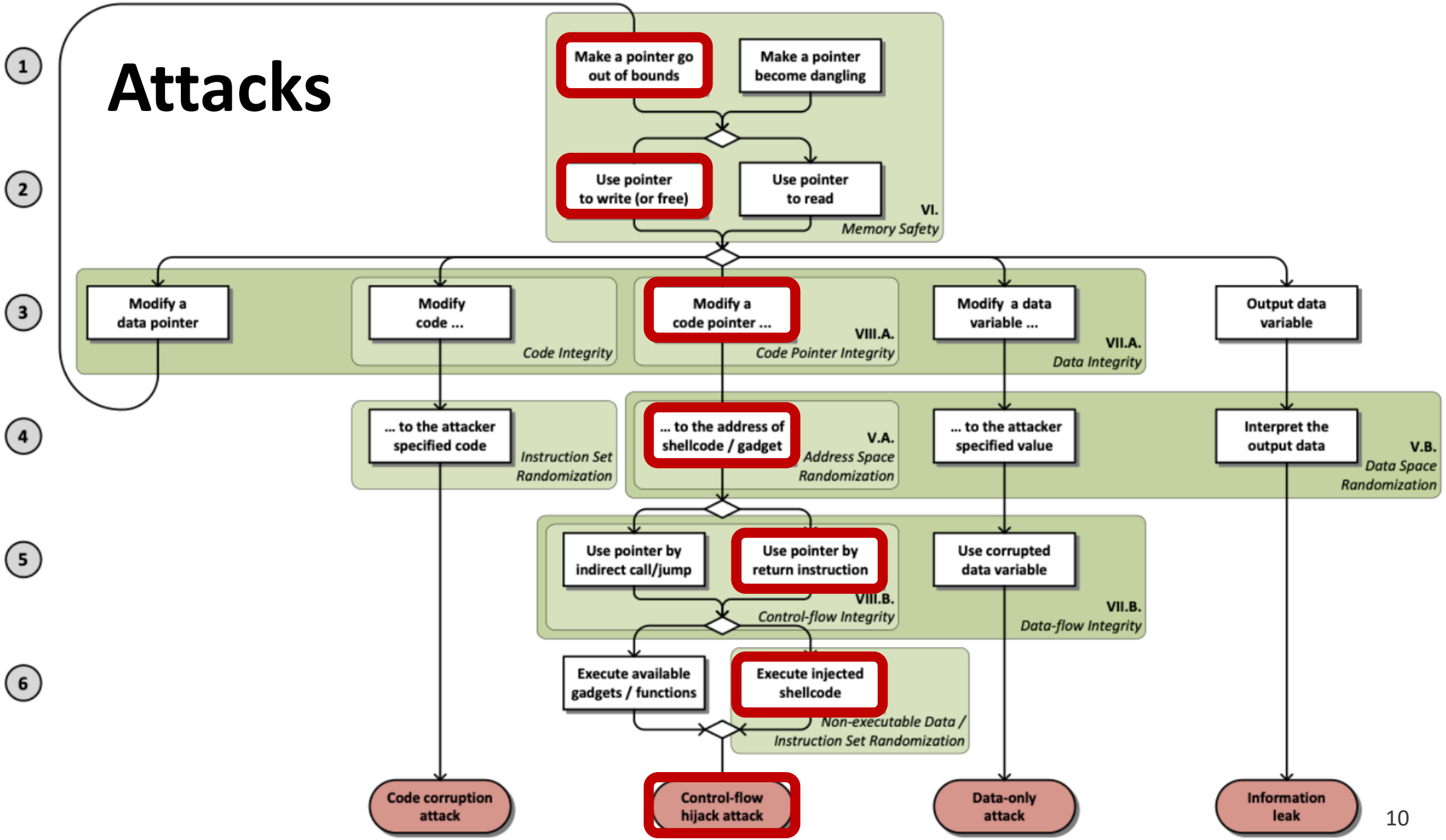
PUSH “/bin/sh”
CALL system



Code Injection Attack

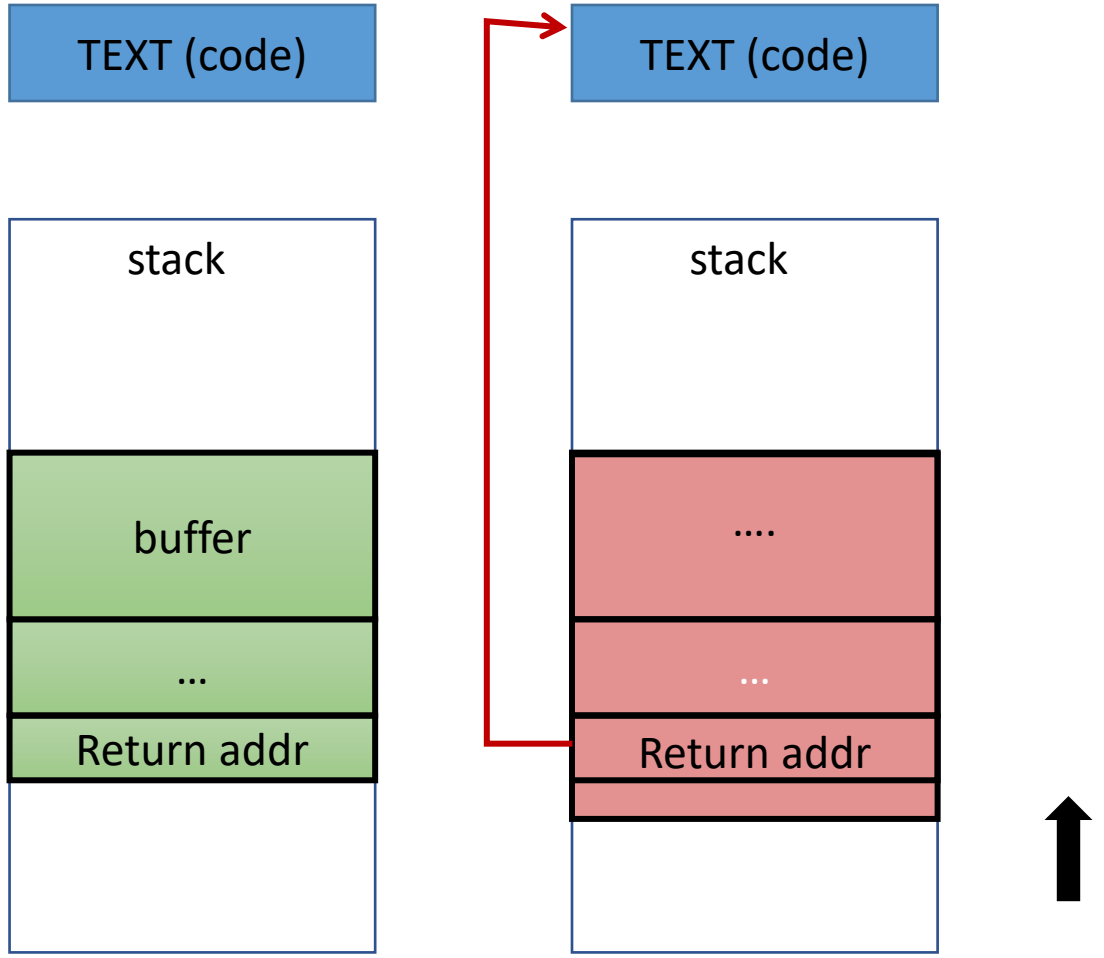


Attacks

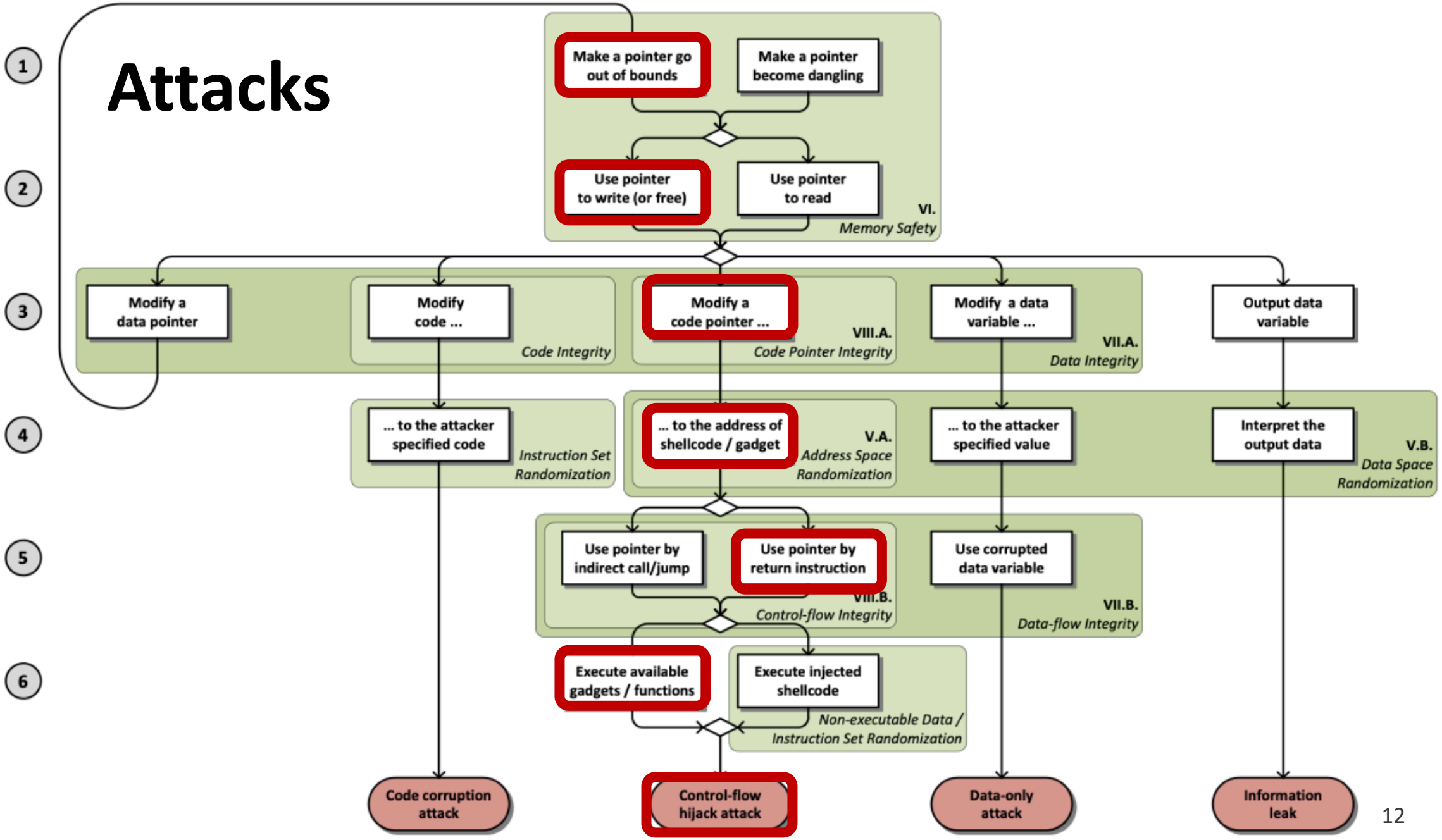

1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI. Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A. Code Pointer Integrity
Modify a data variable ...
VII.A. Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A. Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B. Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B. Control-flow Integrity
Use corrupted data variable
VII.B. Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

Return-Oriented Programming (ROP)

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

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

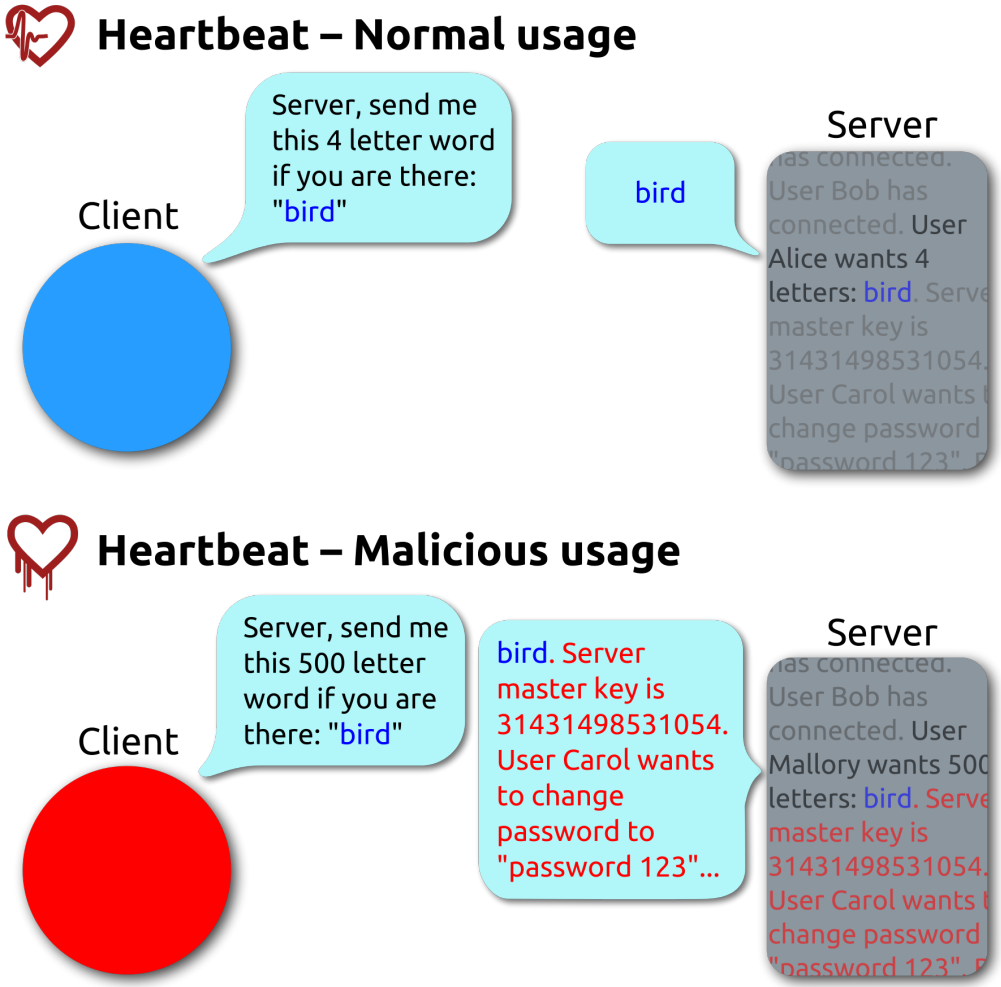


Attacks

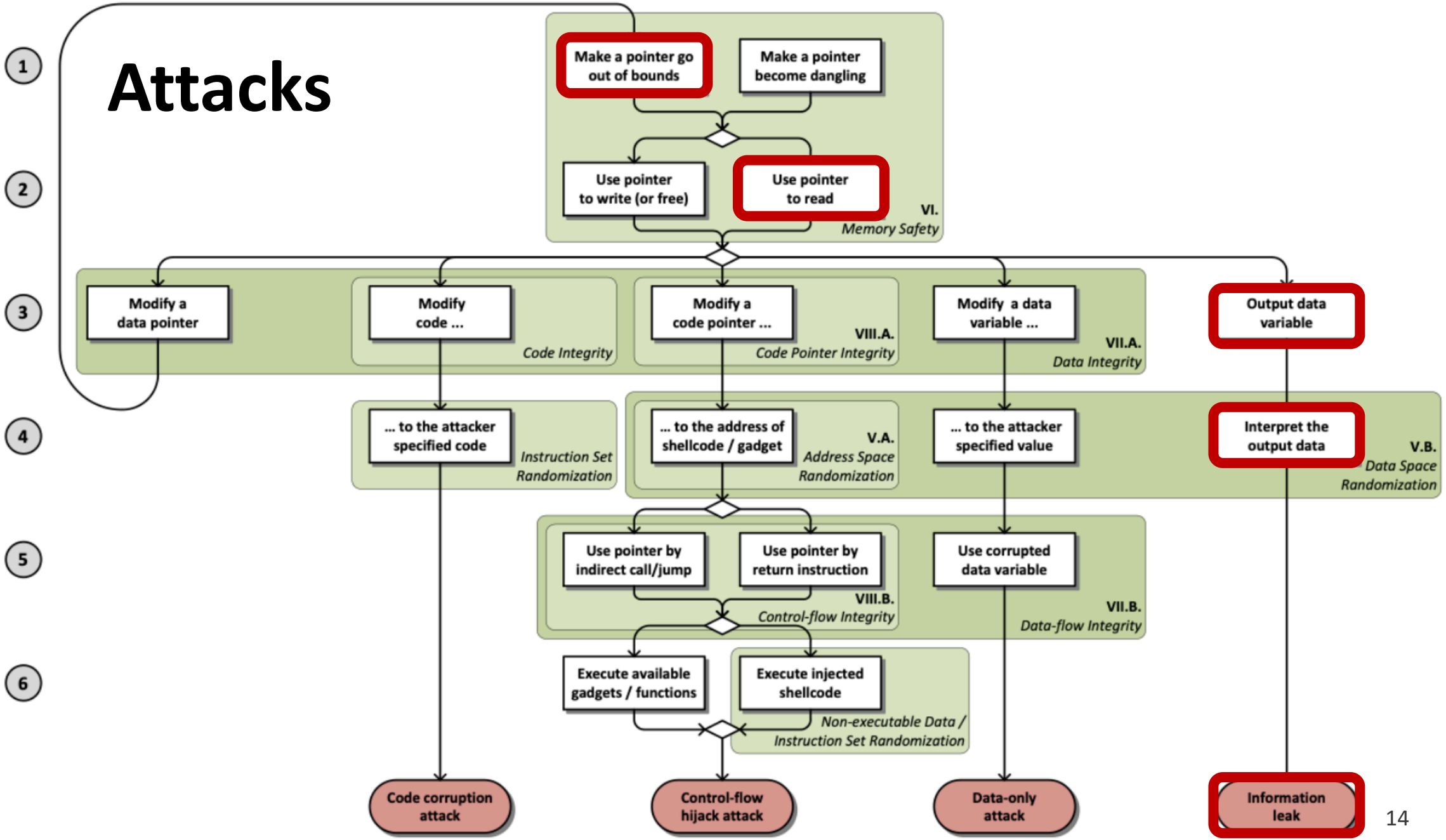

1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI. Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A. Code Pointer Integrity
Modify a data variable ...
VII.A. Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A. Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B. Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B. Control-flow Integrity
Use corrupted data variable
VII.B. Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

HeartBleed Vulnerability

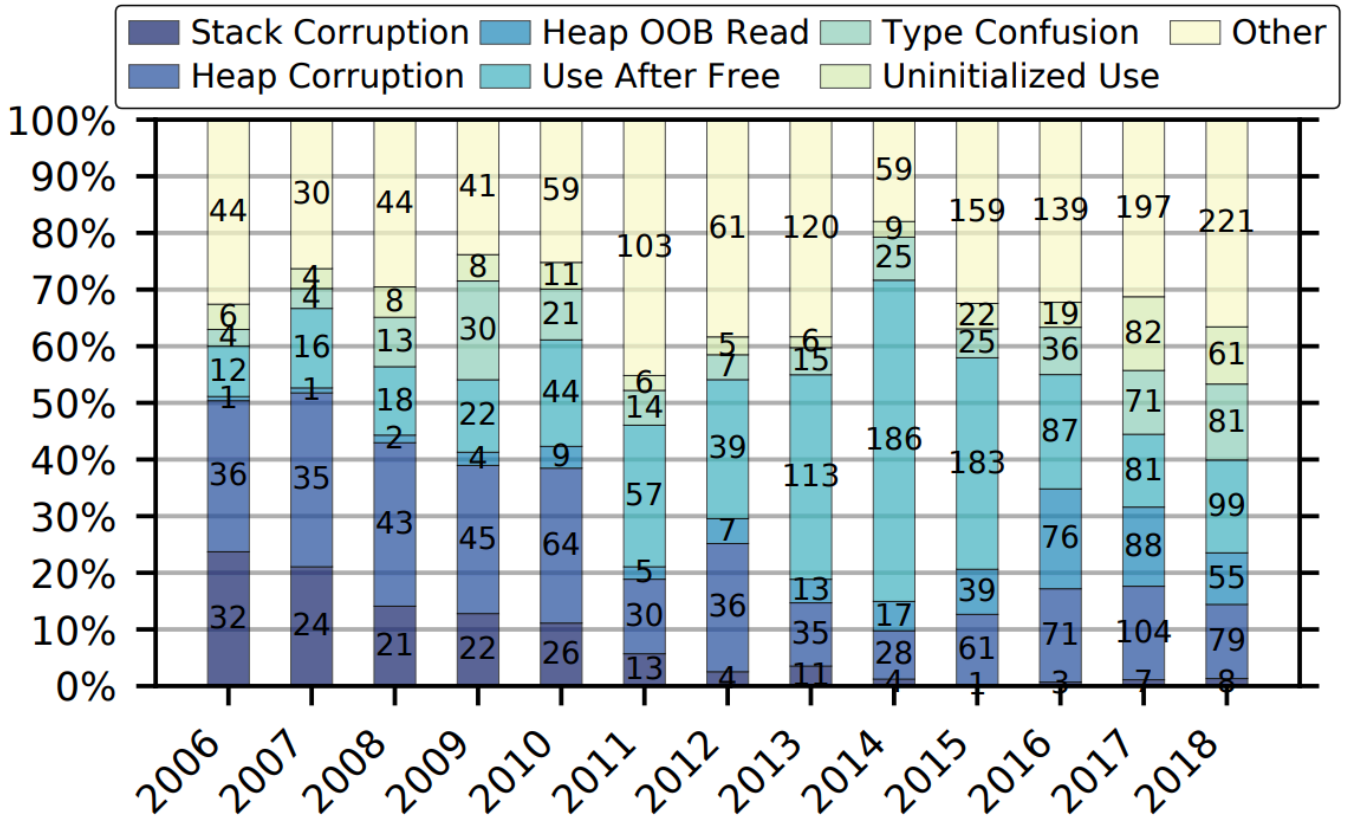
- Publicly disclosed in April 2014
- Bug in the OpenSSL cryptographic software library heartbeat extension
- Missing a bound check



Attacks


1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI. Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A. Code Pointer Integrity
Modify a data variable ...
VII.A. Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A. Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B. Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B. Control-flow Integrity
Use corrupted data variable
VII.B. Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

Recent Memory Safety Issues



A root cause trend of memory safety vulnerabilities reported by Microsoft

From Hardware-based Always-On Heap Memory Safety; Kim et al; MICRO'20

Why Not High-level Language

Why not High-level Language?

- Benefits:
  - Easier to program
  - Simpler concurrency with GC
  - Prevents classes of kernel bugs
- Downsides (performance):
  - Safety tax: Bounds, cast, nil-pointer checks
  - Garbage collection: CPU and memory overhead, pause time
  - Feasibility?

The benefits and costs of writing a POSIX kernel in a high-level language; Cody Cutler, M. Frans Kaashoek, and Robert T. Morris, MIT CSAIL (OSDI'18)

BISCUIT: new x86-64 Go kernel

No fundamental challenges due to HLL

But many implementation puzzles

- Interrupts
- Kernel threads are lightweight
- Runtime on bare-metal

...

Surprising puzzle: heap exhaustion

Why not Rust (no GC)?

Rust compiler analyzes the program to partially automate freeing of memory.
This approach can make sharing data among multiple threads or closures awkward

BISCUIT: Performance Evaluation

	BISCUIT ops/s	Linux ops/s	Ratio
CMailbench (mem)	15,862	17,034	1.07
NGINX	88,592	94,492	1.07
Redis	711,792	775,317	1.09

Test case: pipe ping-pong (systems calls, context switching)

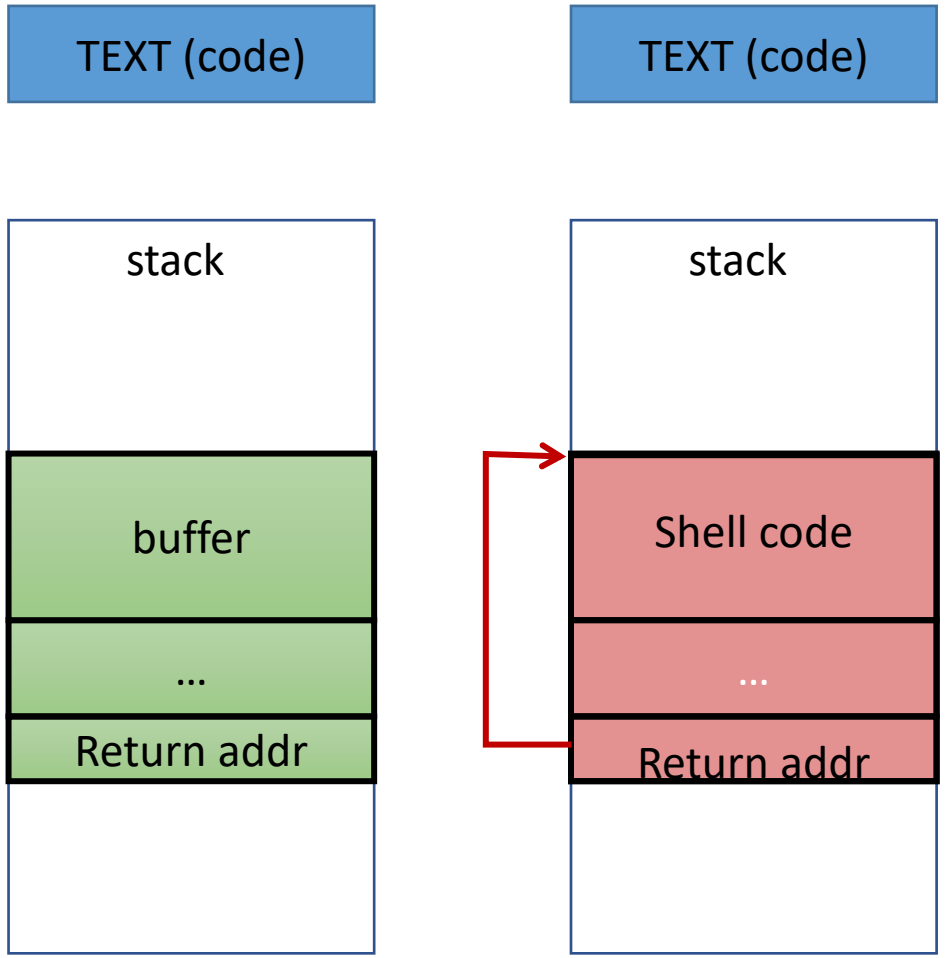
C (ops/s)	Go (ops/s)	Ratio
536,193	465,811	1.15

Prologue/safety-checks ⇒ 16% more instructions

- Conclusion:
  - The HLL worked well for kernel development
  - Performance is paramount ⇒ use C (up to 15%)
  - Minimize memory use ⇒ use C (↓ mem. budget, ↑ GC cost)
  - Safety is paramount ⇒ use HLL (40 CVEs stopped)
  - Performance merely important ⇒ use HLL (pay 15%, memory)

Defenses

How do they work?

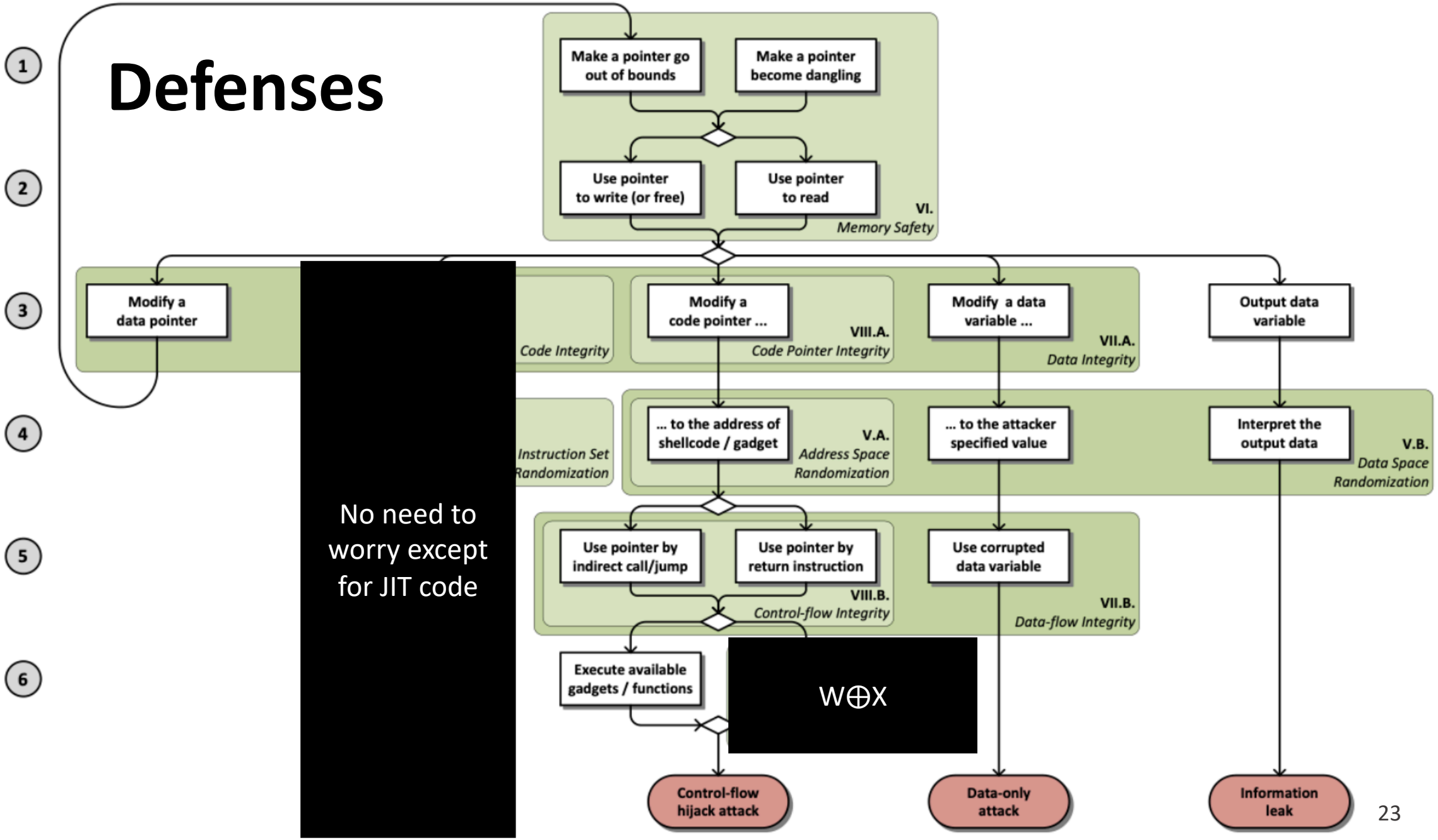


- W⊕X (non-executable data)
- Stack canary (return integrity)
- ASLR

Deployed Defenses

	Policy	Technique	Weakness	Perf.	Comp.
Hijack protection	W⊕X	Page flags	JIT	1x	Good
	Return integrity	Stack cookies	Direct overwrite	1x	Good
	Address space rand.	ASLR	Info-leak.	1.1x	Good

Defenses


1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI. Memory Safety
Modify a data pointer
Code Integrity
Modify a code pointer ...
VIII.A. Code Pointer Integrity
Modify a data variable ...
VII.A. Data Integrity
Output data variable
Instruction Set Randomization
... to the address of shellcode / gadget
V.A. Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B. Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B. Control-flow Integrity
Use corrupted data variable
VII.B. Data-flow Integrity
Execute available gadgets / functions
W⊕X
Control-flow hijack attack
Data-only attack
Information leak
No need to worry except for JIT code

HW Support for Memory Safety

Memory Safety

- Spatial safety (bound information)
- Temporal safety (allocation/de-allocation information)

- Low-level reference monitor
  - SW approach: add checks → performance overhead
  - Execution time: Extra instructions to perform the check
  - Memory: Maintain extra meta data (in shadow memory)

Intel MPX (Memory Protection Extension)

4 128-bit bound registers (bnd0-3)

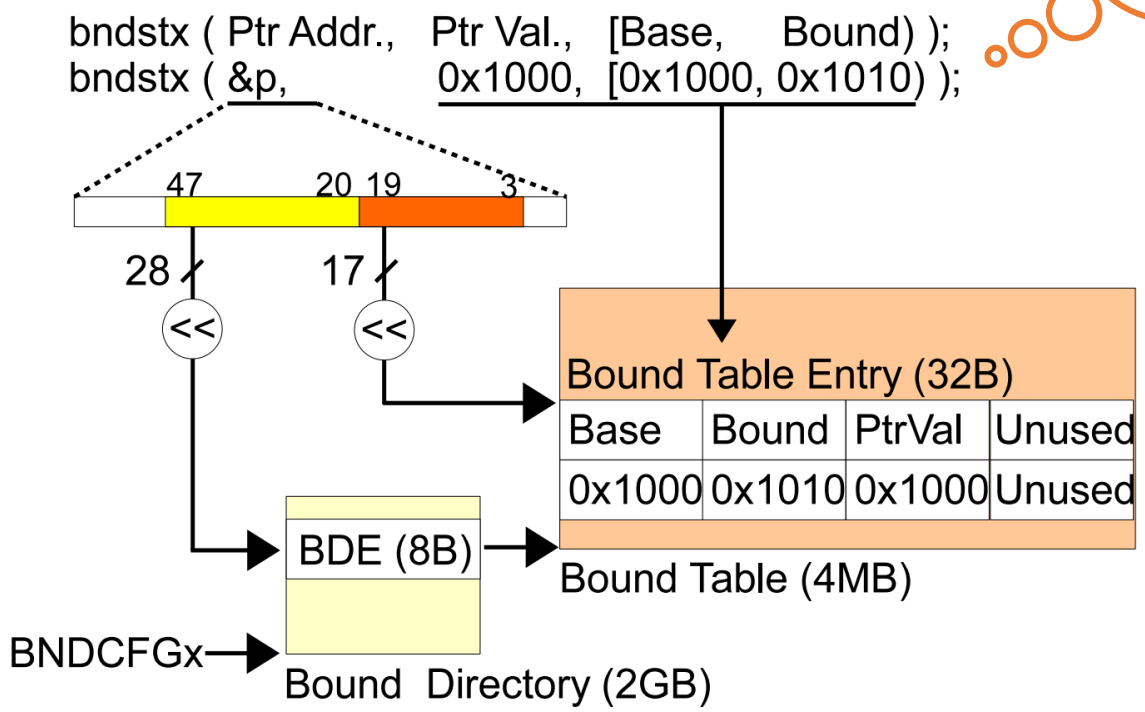
- Bndmk: create base and bound metadata
- Bndldx/bndstx: load/store metadata from/to bound tables
- Bndcl/bndcu: check pointer with lower and upper bounds

Original Program

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

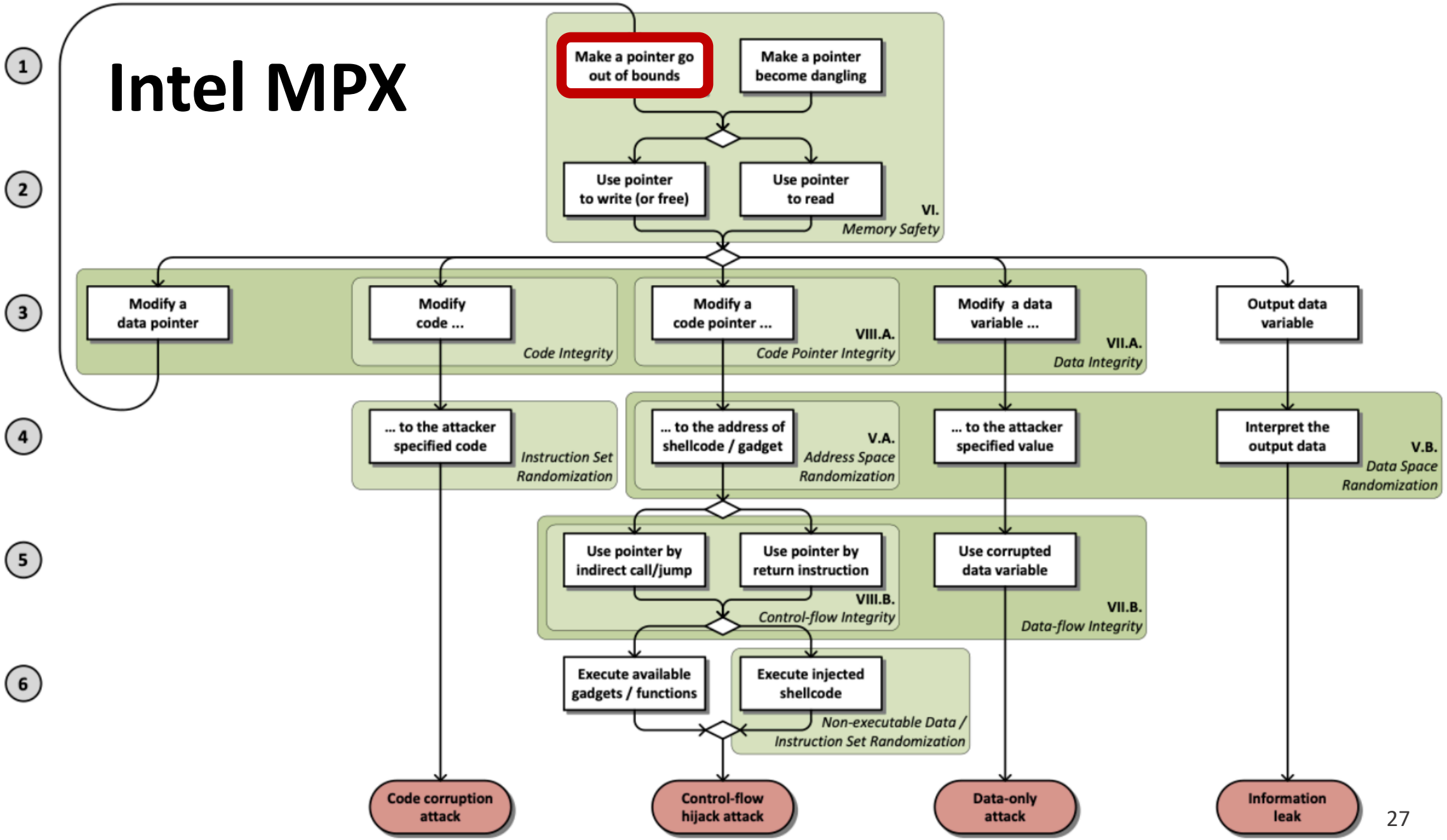
Instrumented Program

```
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';
```



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

Intel MPX


1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI.
Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A.
Code Pointer Integrity
Modify a data variable ...
VII.A.
Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A.
Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B.
Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B.
Control-flow Integrity
Use corrupted data variable
VII.B.
Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

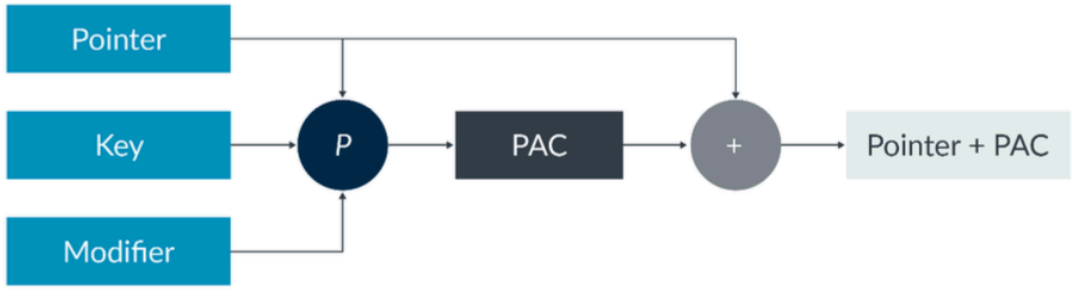
Analysis of Intel MPX

Intel MPX is **impractical** for fine-grained memory safety

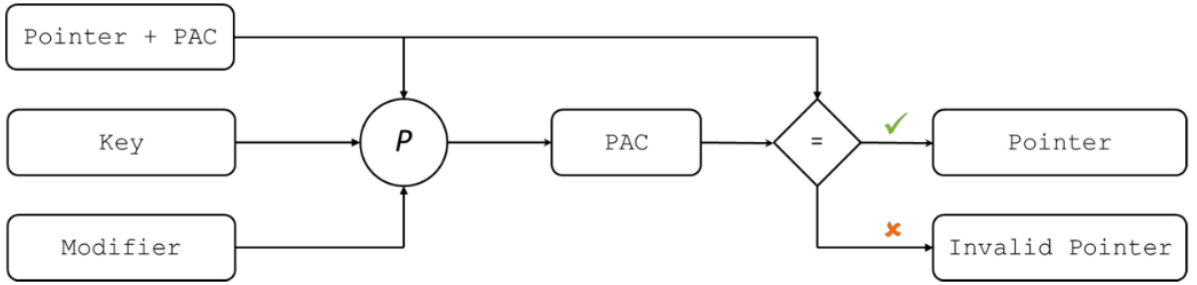
- High overheads
  - Check is sequential
  - loading/storing bounds registers involves two-level address translation
- Does not provide temporal safety
- Does not support multithreading transparently

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

ARM PA (Pointer Authentication)



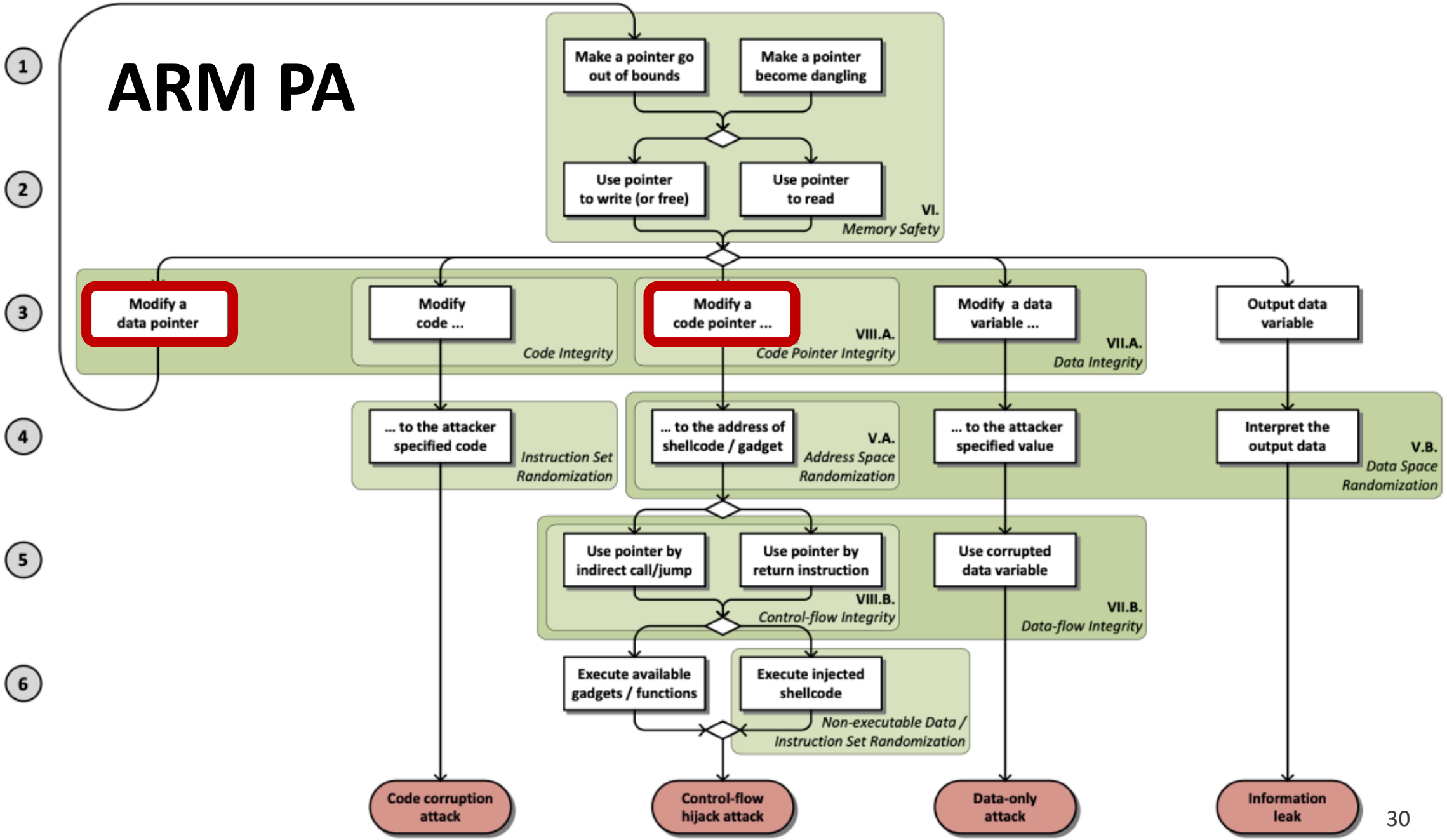
Generate PAC



Check PAC

- Keys: 128 bit, stored in system registers
- Modifier: either another processor register or 0
- PAC: a tweakable message authentication code (MAC)
  - **Where to store PAC?**

ARM PA


1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI. Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A. Code Pointer Integrity
Modify a data variable ...
VII.A. Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A. Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B. Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B. Control-flow Integrity
Use corrupted data variable
VII.B. Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

Discussion Questions

Discussion Questions

- Given that backwards compatibility has been a large issue for many proposed mitigation techniques, is it at all feasible to transition to new binaries all in unison (during a transition, say, to a new architecture/OS)?

- Does the increased space of potential memory safety vulnerabilities in a JIT compilation environment translate to noticeably less secure programs in practice?

- Can code verification be used to completely ensure that a piece of code is void of memory attacks? Or do some memory attacks exist where even static verification cannot detect or avoid?

Discussion Questions

- Practical solutions to memory safety (e.g. Rust) exist, but it is difficult to get software developers to make the switch due to time/resource constraints. Is it practical to focus on memory safety issues that aren't likely to be critical exploits?

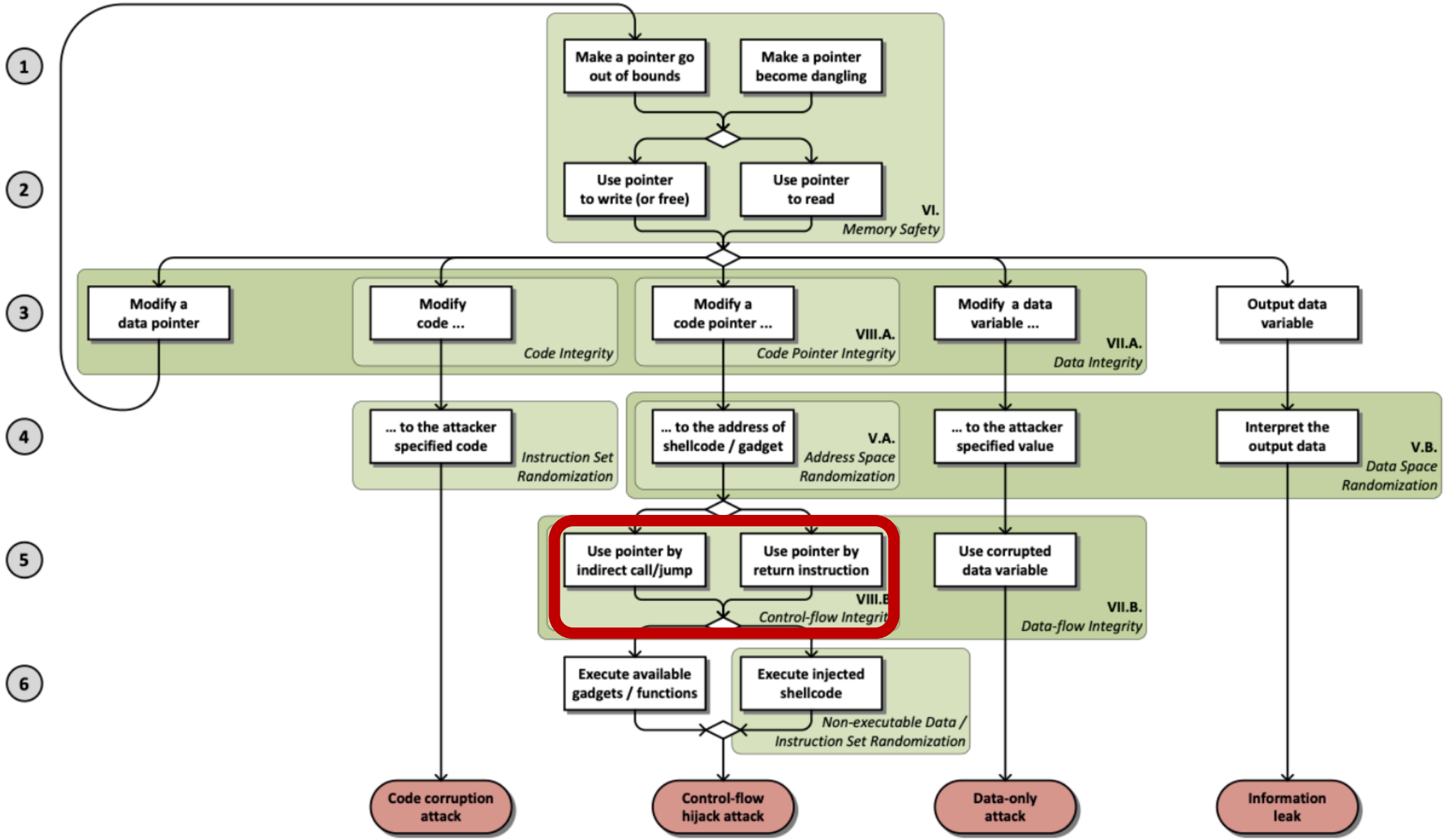
- How much of memory corruption is a hardware related problem? Is it possible to produce corruption free programs using formal verification?

- Can these bounds checking systems actually do anything about temporal errors?

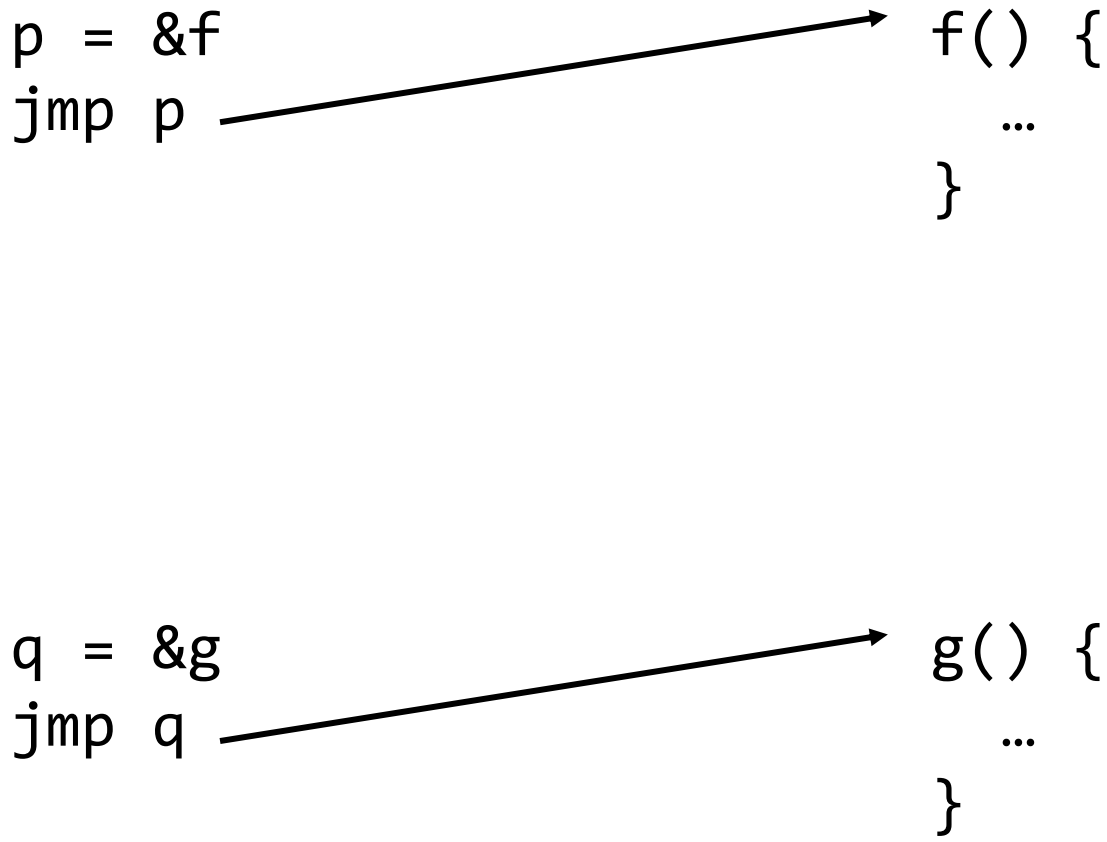
- Are there memory corruption-based availability attacks?

Backup

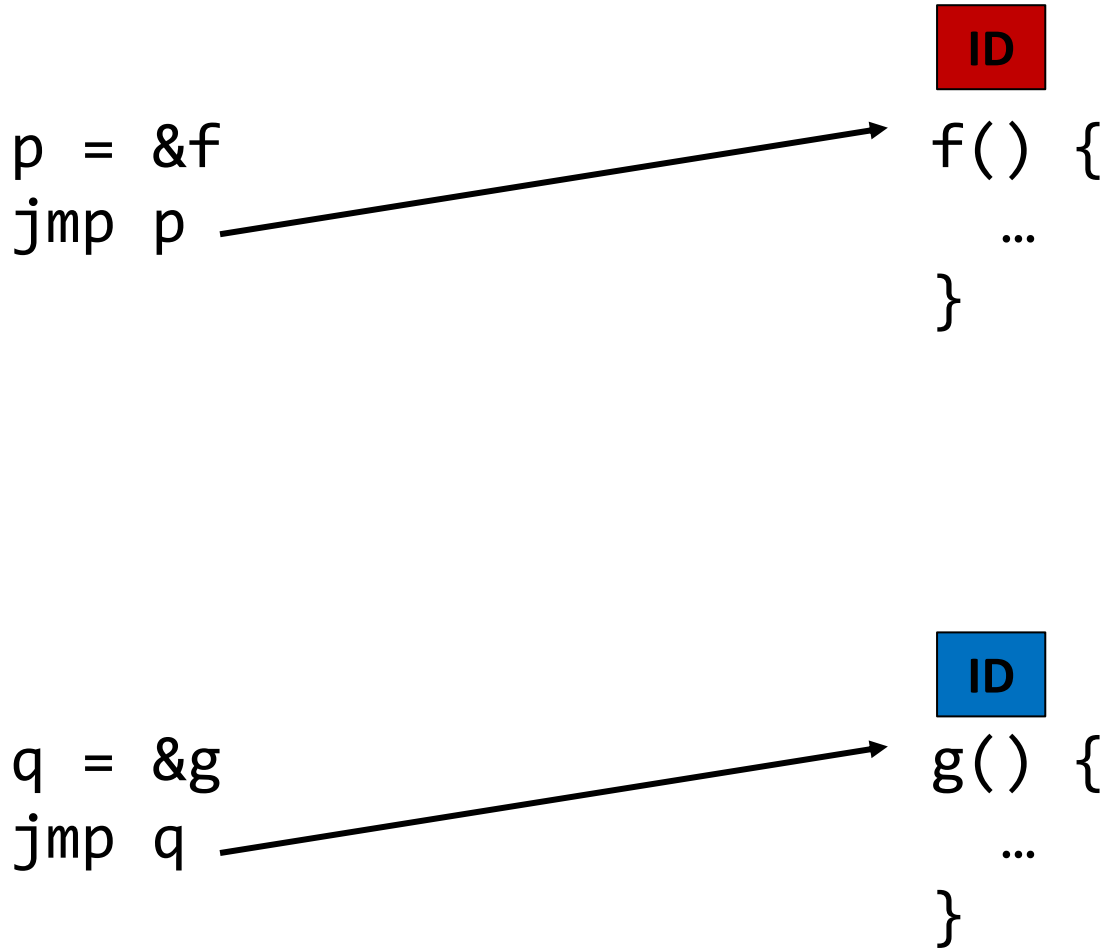
CFI: Control Flow Integrity



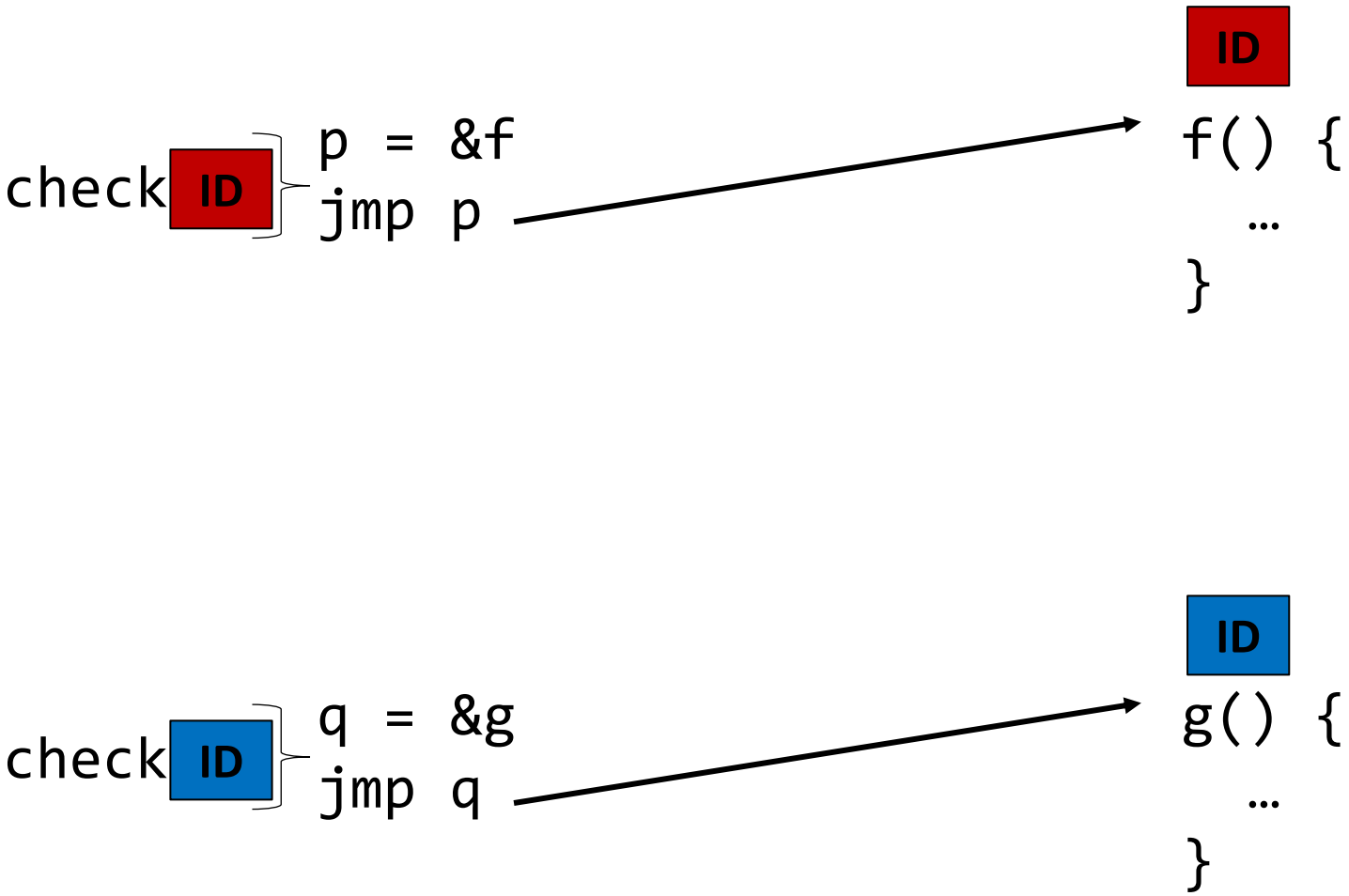
CFI



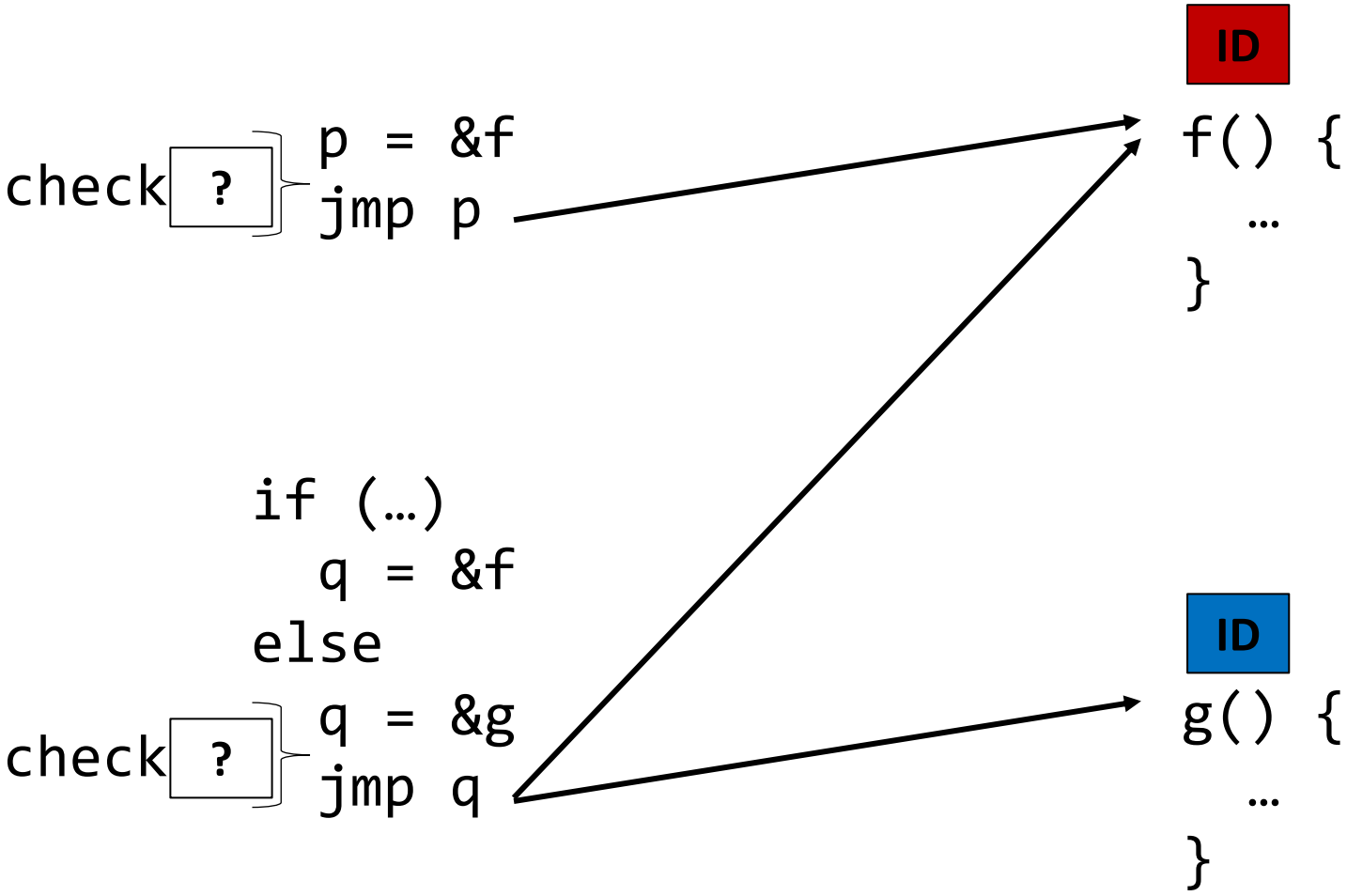
CFI



CFI



CFI



Over-approximation Problem