CS5231: Systems Security

Lecture 4: Advanced Memory Error Defenses

Control-flow Hijacking: Code Injection

- Control-oriented a.k.a control-flow hijacking
- Outcome 1: Code Injection
  - ***Definition***: *A memory exploit that hijacks control to jump to attacker's data payload*

- Req 1: Write Attack Payload in memory
- Req 2: Have Attack Payload Be Executable
- Req 3: Divert control-flow to payload

Control-oriented Exploits (II): Code Reuse

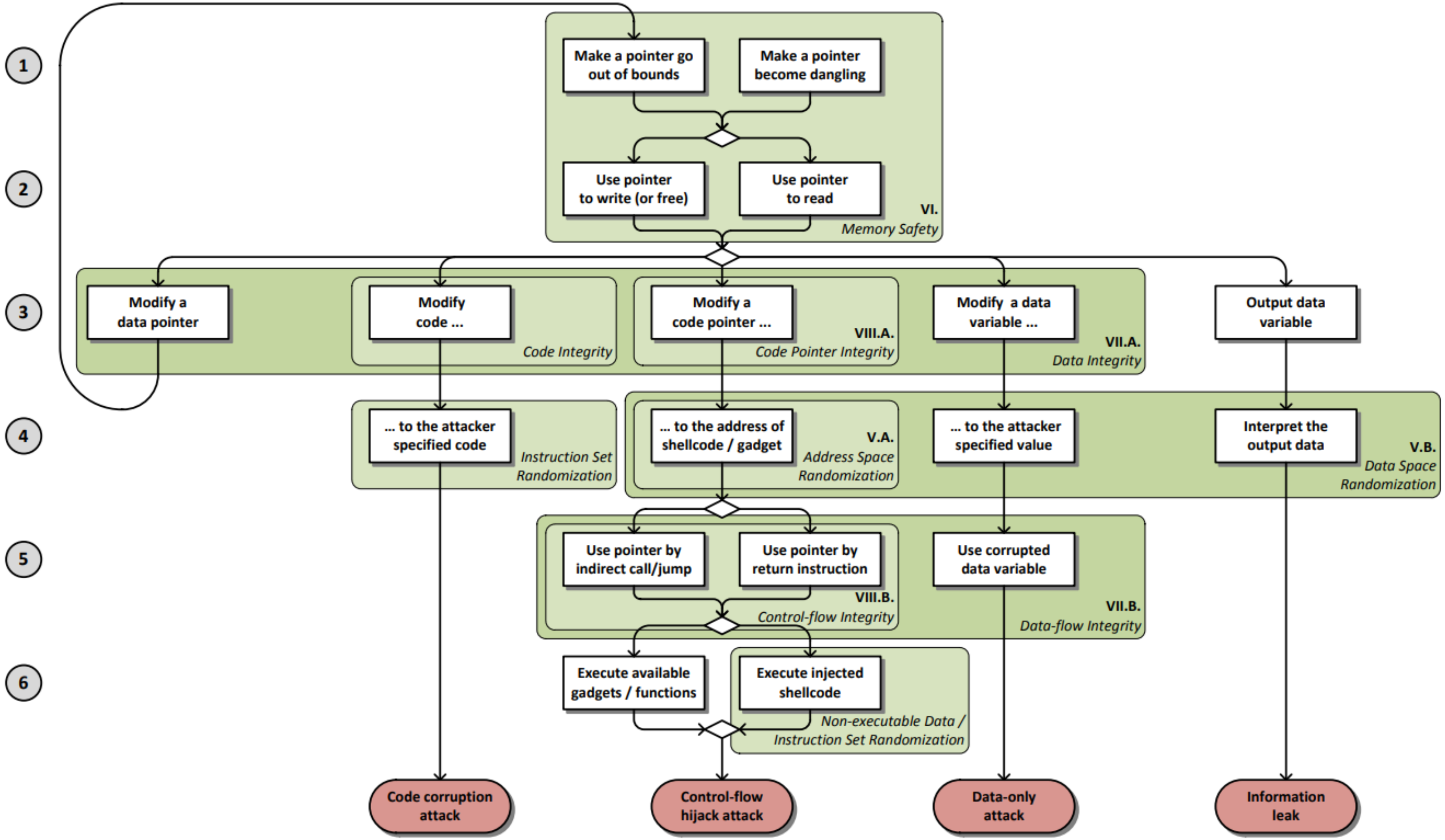
- Outcome 2: Code Reuse
  - ***Definition**: A memory exploit that hijacks control to jump to attacker's controlled code address*
- Requirements for Code Reuse
  - ~~Req 1: Write Attack Payload in memory~~
  - Req 2: Have Attack Payload Be Executable
  - Req 3: Divert control-flow to payload
- Insight: Re-use the existing code as payload

another paper called jump oriented programming

Data-oriented Exploits

- Don’t need any execution of illegitimate code
- Requirements for Code Reuse
  - ~~Req 1: Write Attack Payload in memory~~
  - ~~Req 2: Have Attack Payload Be Executable~~
  - ~~Req 3: Divert control-flow to payload~~
- Insight: changing data to affect the computation done by a program

Taxonomy of Safety Properties



Eternal War on Memory

Memory Error Defense Summary

- Safe coding practice
- Randomization
  - Address-space randomization, data-space randomization, instruction set randomization
- Partial memory safety
  - StackGuard, stack canaries
  - Non-executable data/DEP
- Full memory safety

Full Memory Safety

Definition: Memory Safety

- Goals:
  - Create memory pointers via permitted operations
    - E.g. malloc(), p = &q;
  - Only access memory allocated to the pointer
    - Spatially → within the allocated range
    - Temporally → while the memory is in scope
  - All “objects” are spatially disjoint at all times
- Enforcement:
  - Can be done by compilation or binary rewriting
  - Insert metadata & inline reference monitors

Spatial Safety

1. Distinguish pointers from non-pointers
2. Check object allocation
3. Check each pointer access
   – Recall: Pointers can be incremented, type cast, etc.

- Proposals:
  – Fat pointers

P	[start, end]

  – Shadow-memory data structure [e.g. JK-Tree]
  – Encode the size information in pointer value [BB]
- Overheads: About 30% or more (SPEC)
- Hardware support: Intel MPX

Temporal Safety

1. Track creation and destruction of pointers
2. Ensure: De-allocated pointers are not accessed

- A Proposal:
  - Lock-and-key [CETS]

P	K	L

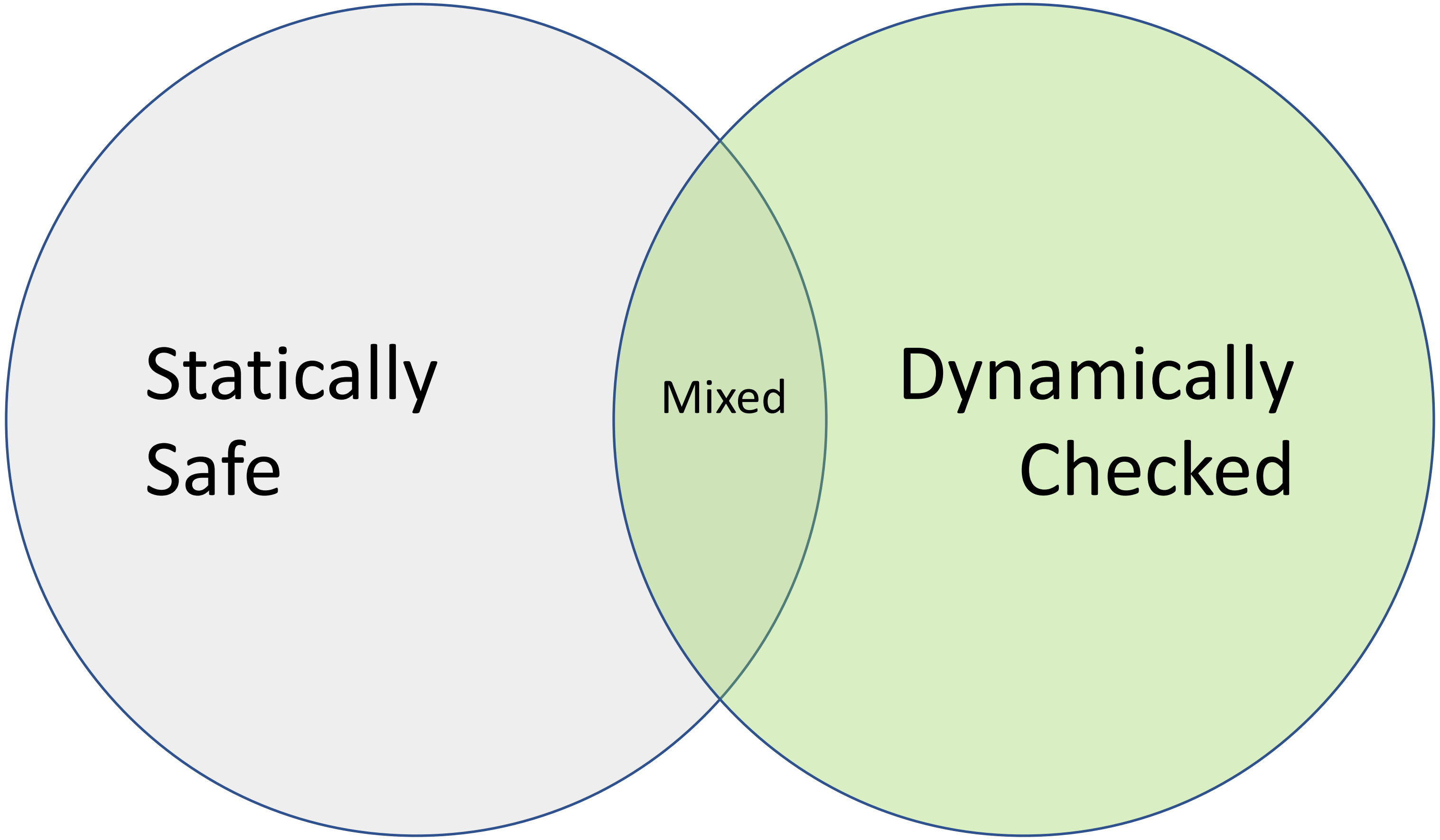
    - The key K and lock L will match only if P is live
    - When de-allocated, change the key K
    - Where to store the lock & key?
      - Fat pointers, shadow memory data structure…
  - Canary-based defenses: Set to NULL on de-allocate [DN]
- Overheads:
  - About 50% or more for lock-n-key
  - Unclear, but could be about 10% for canary-based

Approaches to Enforce Memory Safety: Static vs. Dynamic

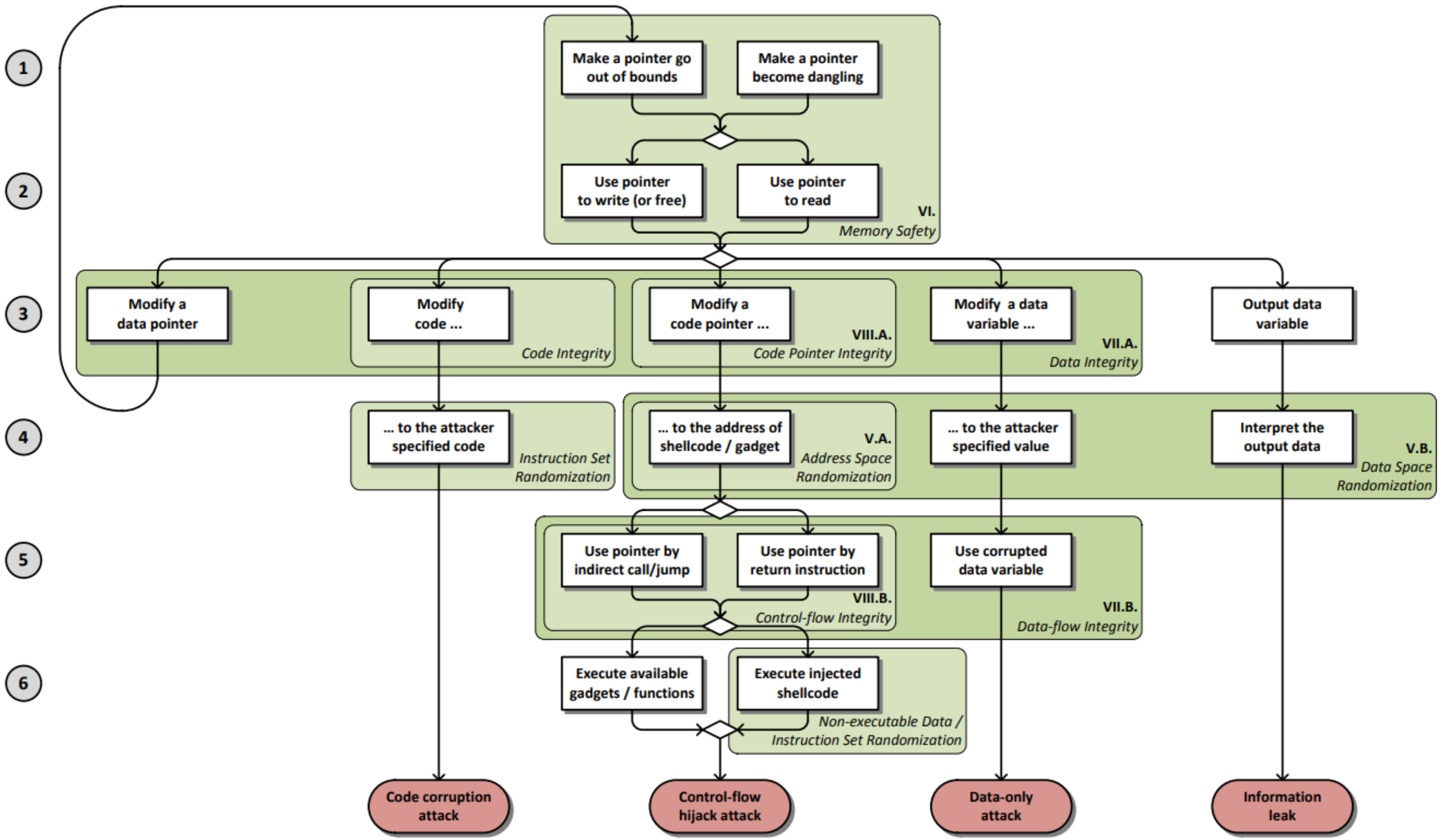
- Statically disallow
  - Type casts
  - Unchecked buffer accesses
  - Pointer Arithmetic
  - Explicit Alloc / Dealloc
- Examples:
  - Memory-safe languages
  - Safe C subsets (e.g. Cyclone)

- Dynamically check
  - Spatial Errors
  - Temporal Errors
- Check Type casts?
  - Need to track type-info at runtime
- Examples:
  - See previous slides

Approaches to Enforce Memory safety: Static + Dynamic



Summary of Memory Defenses



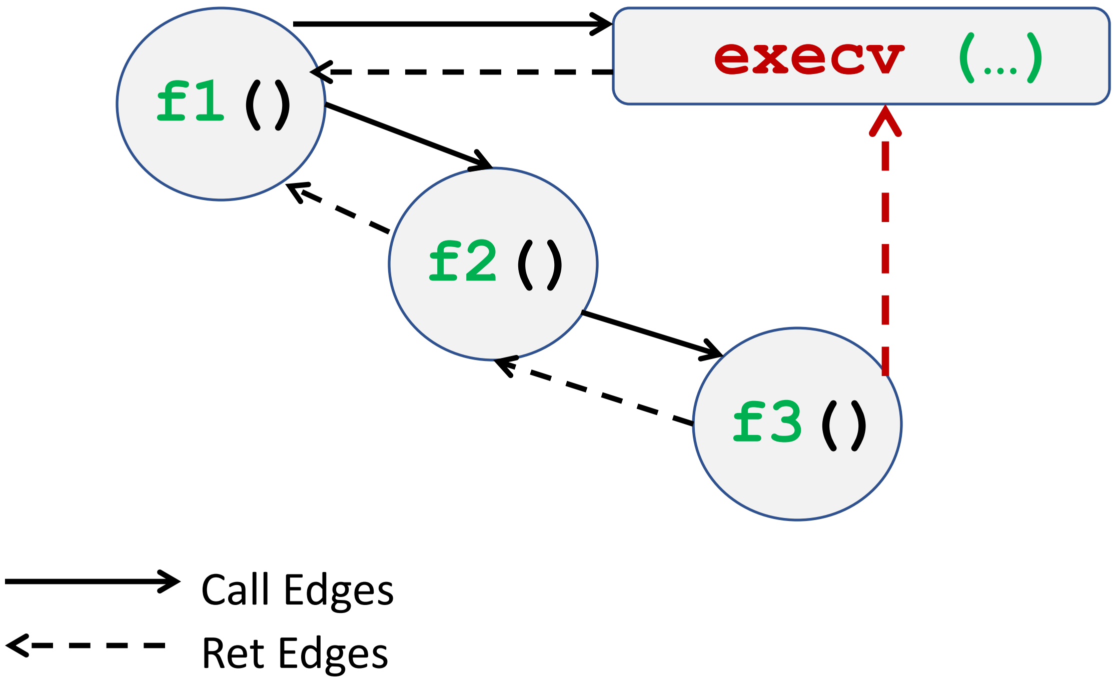
Eternal War on Memory

Control Flow Integrity

Control-flow Integrity

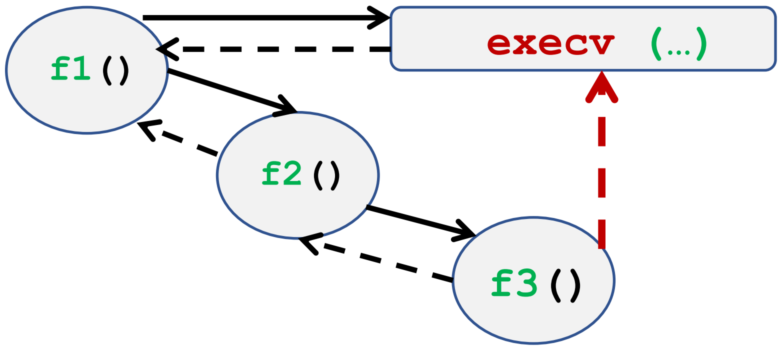
- Goal of CFI enforcement:
  - Control Flow Integrity

    "Follow the statically determine CFG at runtime"



Control Flow Integrity

- ***Definition** of Control Flow Integrity*
  - Each control transfer jumps to a statically-known set of locations
    - E.g.: Returns -> Return points, Call Instructions -> calls
- CFI blocks all control-flow hijacking exploits



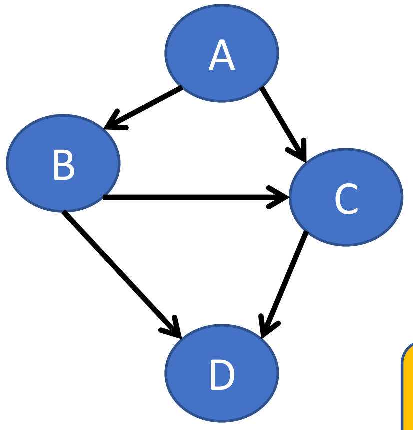
Control Flow Graph

CFI: Theoretical power

- Goal of CFI enforcement:
  - Control Flow Integrity

    “Follow the statically determined CFG at runtime”
- Can block all control-flow hijacking attacks!

Legitimate Call Sequences:
A -> B -> D
A -> C -> D
A -> B -> C



Illegitimate Call Sequences:
A -> C -> B -> D
A->D

CFI can protect against all such bad call sequences

Inline Reference Monitors (II): CFI – Implementation 1

- Goal of CFI enforcement:
  - Control Flow Integrity

    "Follow the statically determined CFG at runtime"

```
jmp ecx ; computed jump
```

```
cmp ecx, 0x80480aa      ;
jne error_label         ;
……                      ;
jmp ecx                 ; jump to dst
```

Control-flow Integrity: Return Edges?

- A function can have several callers….
- If a small set of return points
  - Instrument code to enforce return target
- If a large possible set of return points
  - Use a shadow stack!
  - Shadow stack can be protected by SFI

CFI Implementations

- Can we do faster than CFI-1?

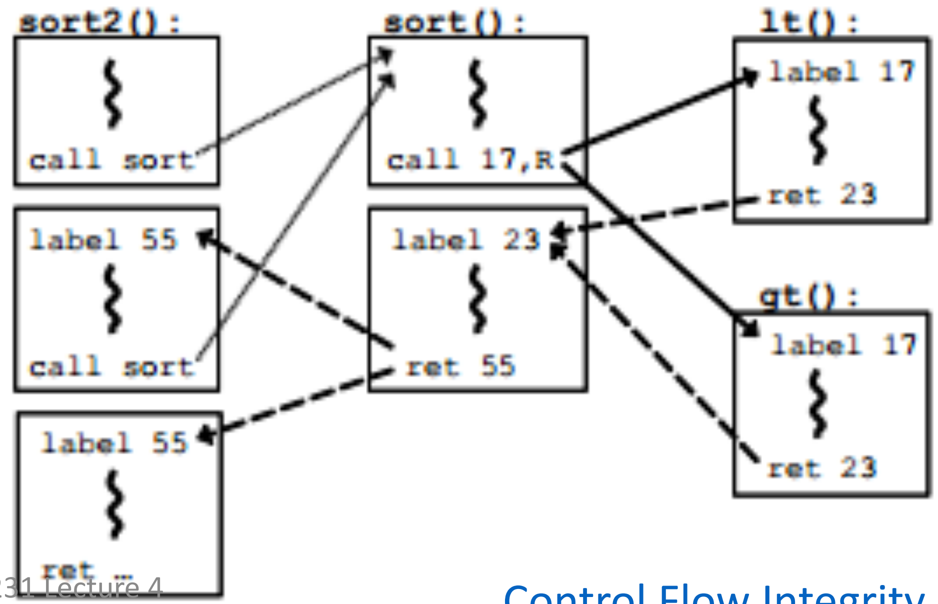
```
jmp ecx ; computed jump
```

```
cmp ecx, 0x80480aa    ;
jne error_label       ;
......                ;
jmp ecx               ; jump to dst
```

CFI Implementation With Randomized Tags

- Each code block must start with a tag
  - The tag should be a random, secret value
  - If f can jump to block g,h,…
    - Then blocks g,h,… should have the same tag

```
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 );
}
```



CFI Implementation With Randomized Tags

- Each code block must start with a tag
  - The tag should be a random, secret value
  - If f can jump to block g,h,…
    - Then blocks g,h,… should have the same tag

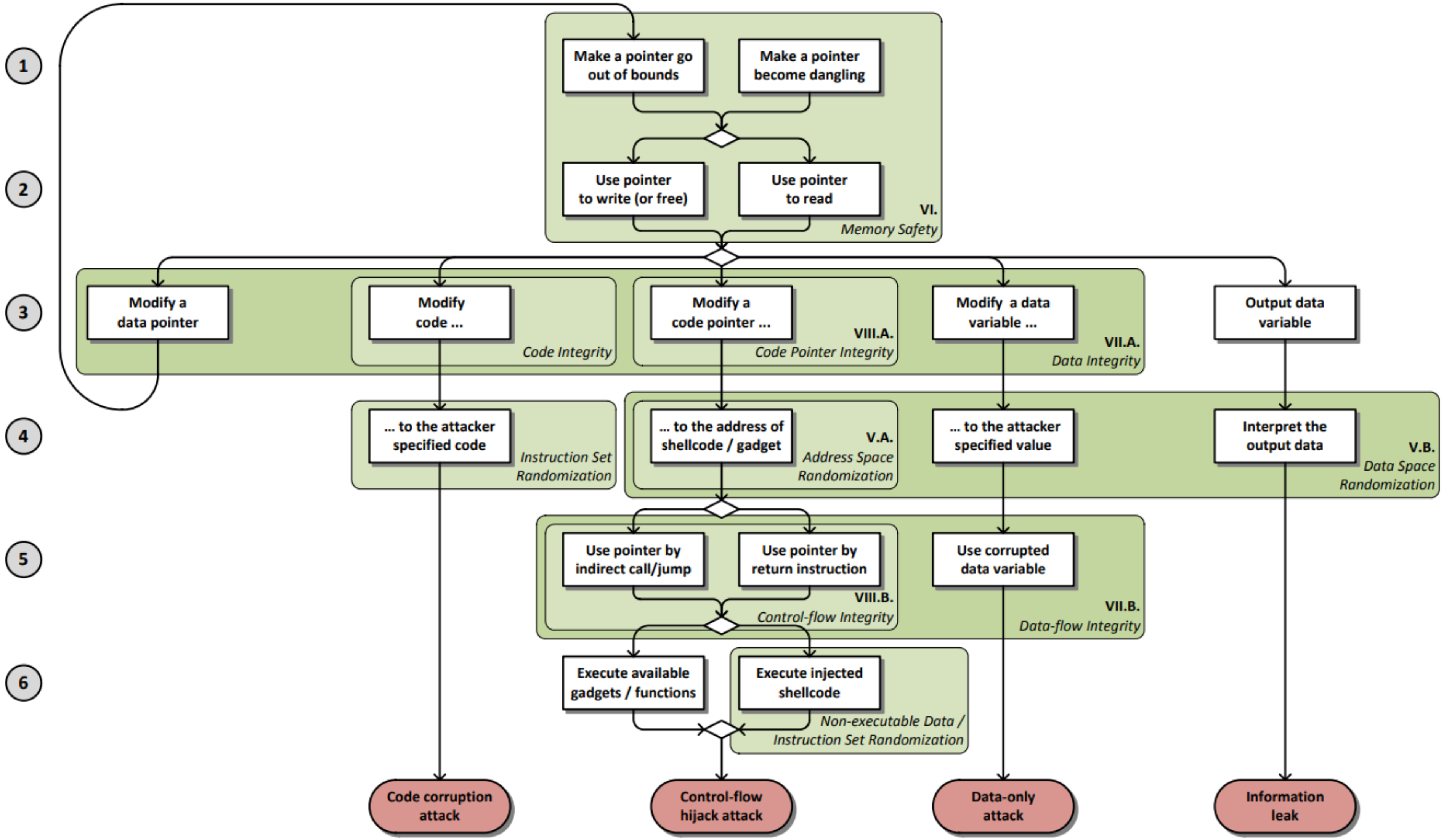
```
jmp ecx ; computed jump
```

```
cmp [ecx], 12345678h   ; comp ID & dst
jne error_label        ; if != fail
lea ecx, [ecx+4]       ; skip ID at dst
jmp ecx                ; jump to dst
```

CFI, In Practice

- Powerful in theory, but…
  - It is challenging to recover the precise control flow graph at compile time
- Why?
  - Do we know what will a function pointer point to?
  - Pointer analysis:
    - Theoretically is undecidable
    - Practically is a difficult problem for real-world programs
- Implemented in LLVM – Clang
- Implemented in Microsoft V. Studio compiler

Taxonomy of Safety Properties



[Eternal War on Memory]

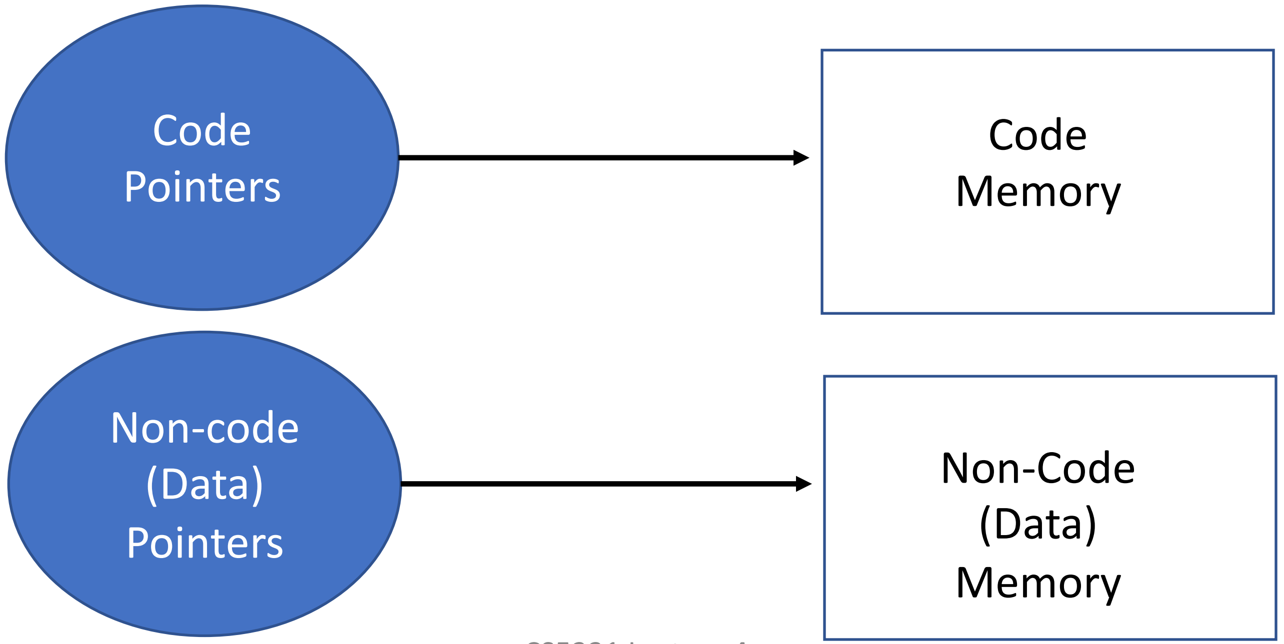
Pointer Integrity

Pointer Integrity

- Runtime Property:
  - Pointers should point to valid addresses only
- Code Pointer (Definition):

  Rule 1: A pointer that can be legally point to code

  Rule 2: Pointers that legally can point to pointers of Type 1, by transitively dereferencing and legal arithmetic operations



Code Pointer Integrity

Code Pointer Integrity (CPI) Defeats CI

- ***Definition** of Code Pointer Integrity :*
  1. *Enforce that code pointers point to code-segment only!*
  2. *Enforce that control transfers use code pointers.*
- Recall, the requirements for code injection:
  - Req 1: Write Attack Payload in memory
  - Req 2: Have Attack Payload Be Executable
  - Req 3: Divert control-flow to payload
- Rule 1 of CPI defeats requirement 3
  - Code segment is not writable
  - Enforcement Details: [CPI Paper (OSDI'14)]

Protecting Code Pointers

- Examples of Code pointers:
  - Return Address Storage
  - Jump Tables / Global Offset Tables
  - Function Pointers
  - Virtual Method Tables (e.g. C++ classes)

```
mov ecx, 0x4[esp]
call [ecx]
```

```
mov edx, 0x14[esp]
jmp [edx]
```

```
ret
```

- *Code Pointer Corruption:*
  - *Forging the runtime value of a code pointer to an "invalid" one!*
  - Valid value: A value that is possible under a memory safe execution of the program

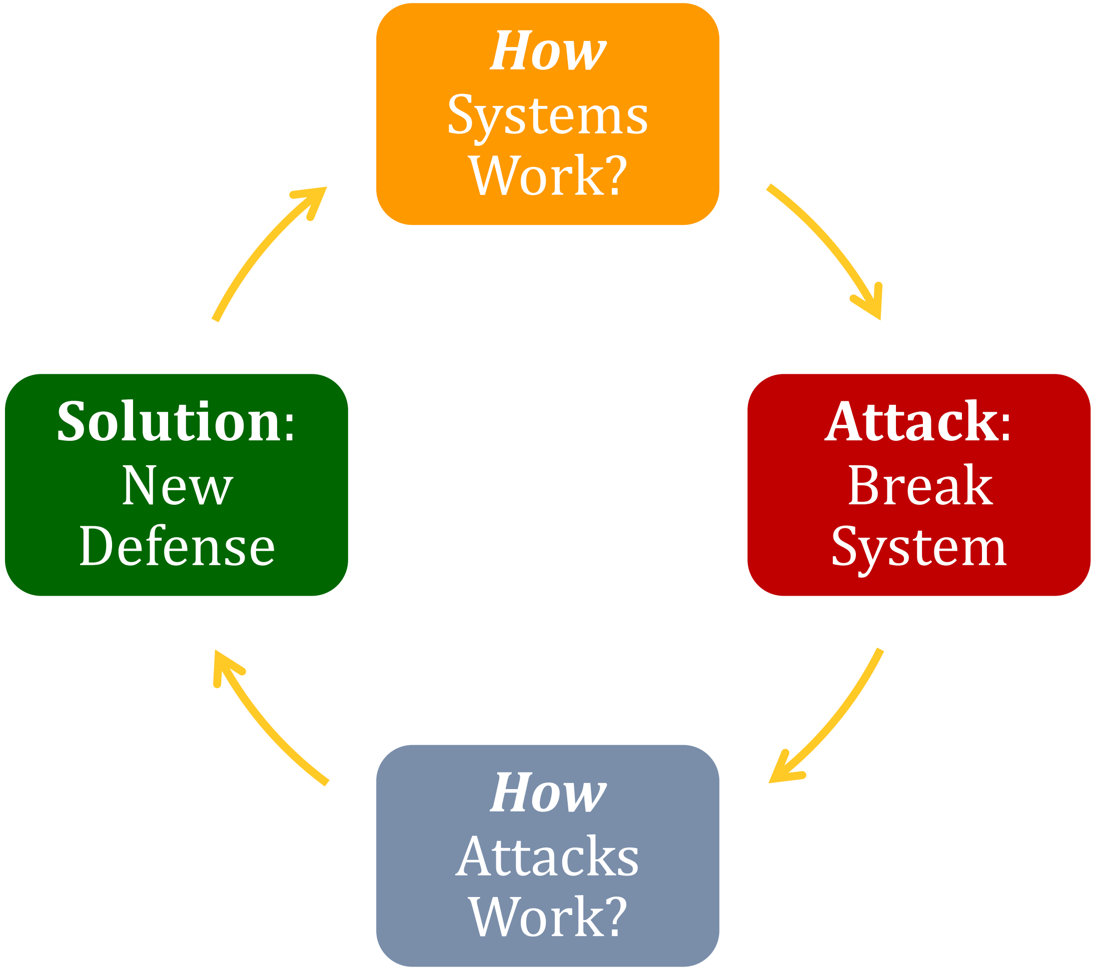
Data & Code Pointer Integrity

Is a research topic…

- One approach: **Pointer authentication**
  - Available in ARM processors as hardware primitive
- The basic idea:
  - Cryptographically bind a pointer address to its legitimate value when it is created
  - When legitimate instructions use this value, they can check whether the value has been tampered

https://www.usenix.org/system/files/sec19fall_liljestrand_prepub.pdf

Arms Race between Attackers and Defenders



A New Round in Arms Race: Data-Oriented Programming

Non-Control Data Attacks

- Corrupt/leak several bytes of security-critical data

```
//set root privilege  *
seteuid(0);
......
//set normal user privilege
seteuid(pw->pw_uid);
//execute user's command
```

```
//offset depends on IE version +
safemode = *(DWORD *)
            (jsobj + offset);
if(safemode & 0xB == 0) {
    Turn_on_God_Mode();
}
```

- Special cases relying on particular data/functions
  - user id, safemode, private key, etc
  - interpreter – printf(), etc

specific

trivial-to-prevent

- *What is the expressiveness of general non-control data attacks?*

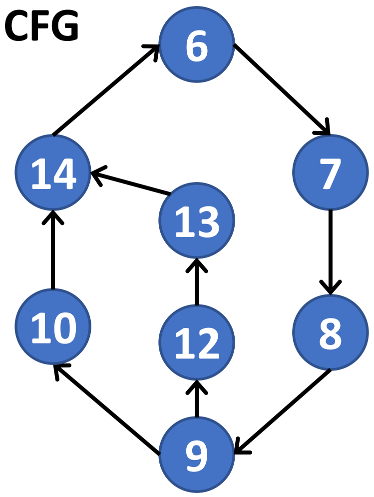
* Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In USENIX 2005.
\+ Yang Yu. Write Once, Pwn Anywhere. In Black Hat USA 2014

Motivating Example

Vulnerable Program

```c
struct server{int *cur_max, total, typ;} *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12]
...
while (quota--) {
  readData(sockfd, buf);          // stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  } //...(following code skipped)...
}
```

CFG



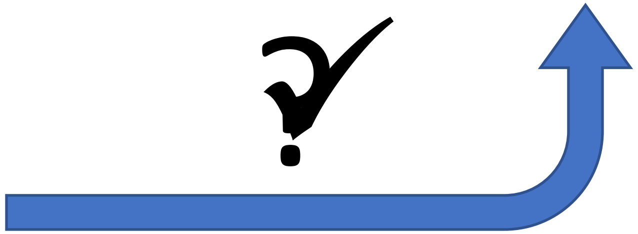
Expected Computation

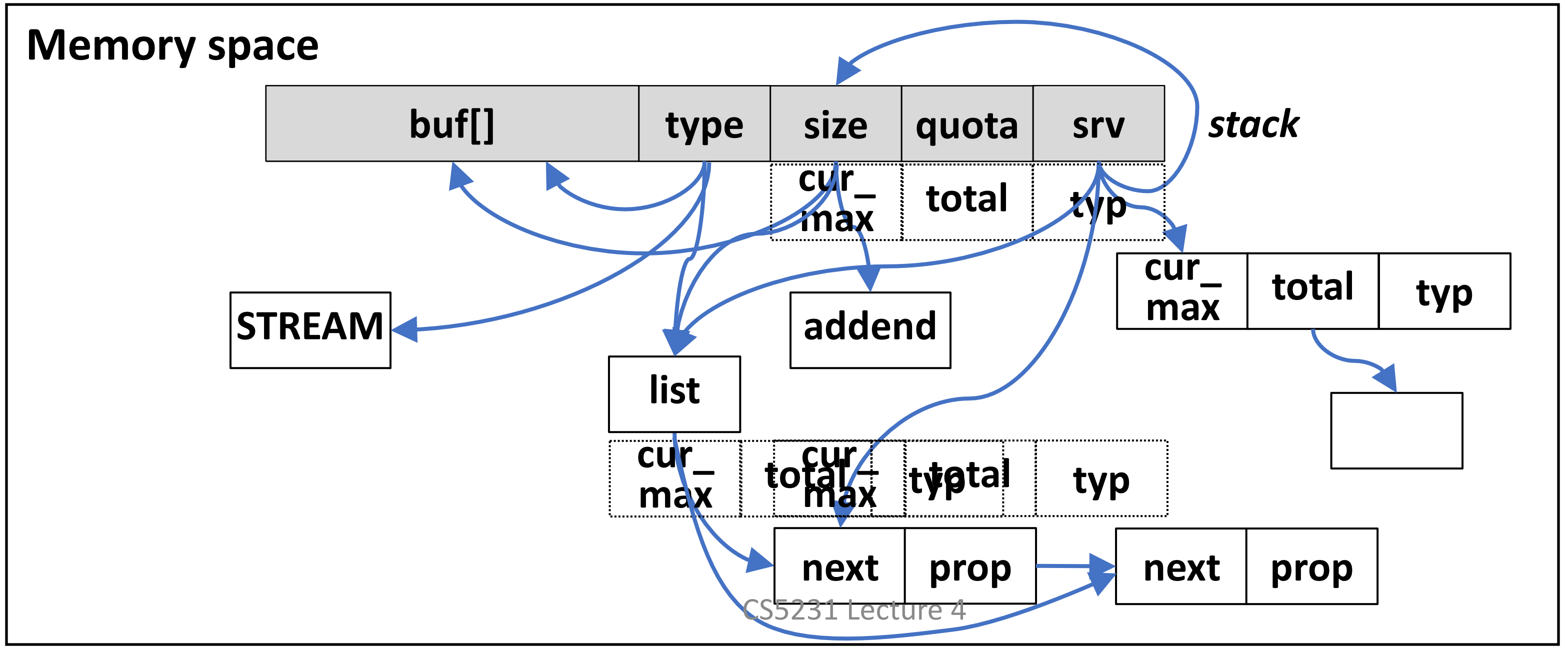
```c
struct Obj {struct Obj *next; int prop;}

void updateList(struct Obj *list, int addend){
 for(; list != NULL; list = list->next)
    list->prop += addend;
}
```

Motivating Example (cont.)

```c
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

```c
for(; list != NULL; list = list->next)
   list->prop += addend;
```





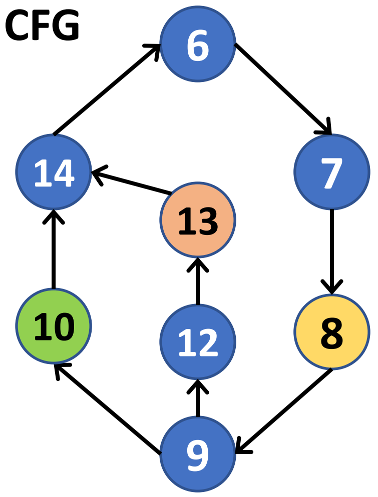
Data-Oriented Programming (DOP)

- General construction
  - w/o dependency on security-critical data / functions
- Expressive attacks
  - towards Turing-complete computation
- Rely on data-oriented gadgets & dispatchers

```
while (quota--) {
  readData(sockfd, buf);     //stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  } //...(following code skipped)...
}
```

Data-Oriented Gadgets

- x86 instruction sequence
  - Shown in normal execution
  - Simulating registers with memory
  - ***Load*** *micro-op --> **Semantics** micro-op --> **Store** micro-op*

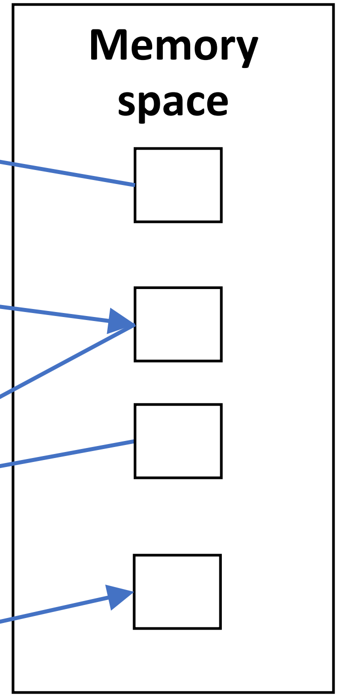


Addition: `srv->total += *size;`

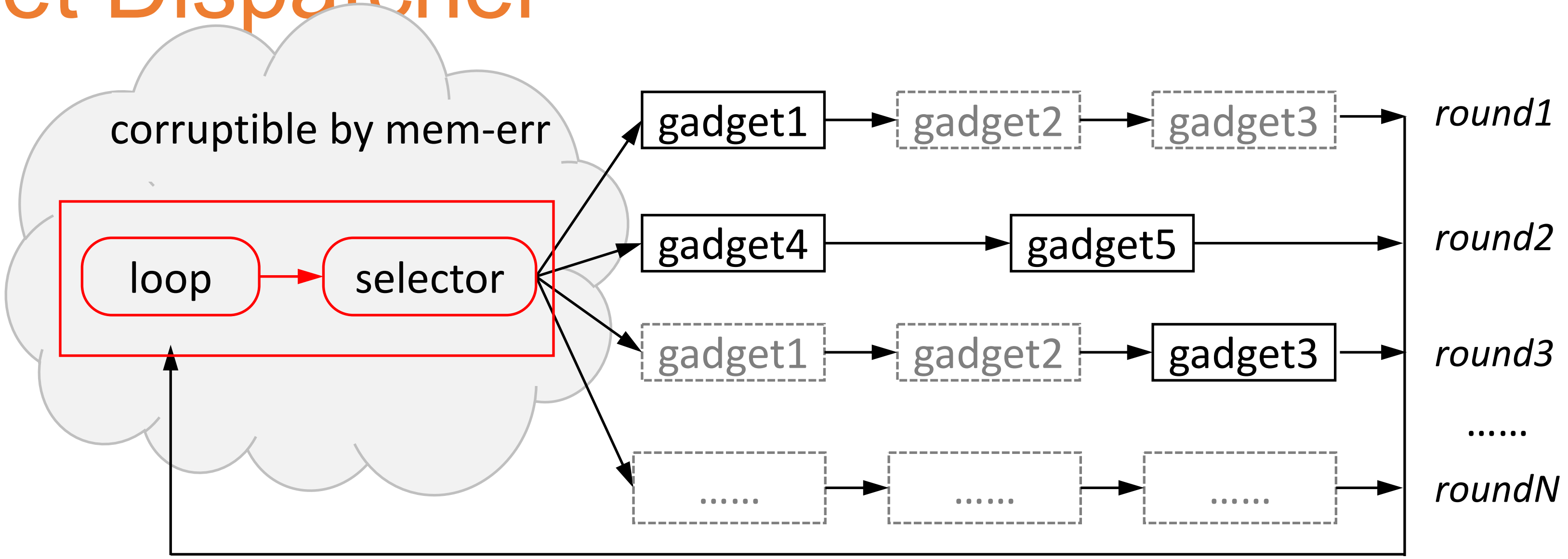
```
mov (%esi), %ebx     //load micro-op
mov (%edi), %eax     //load micro-op
add %ebx, %eax       //addition
mov %eax, (%edi)     //store micro-op
```

Load: `*size = *(srv ->cur_max);`

```
mov (%esi), %ebx     //load micro-op
mov (%edi), %eax     //load micro-op
mov 0xb(%ebx), %eax  //load
mov %eax, (%edx)     //store micro-op
```



Gadget Dispatcher



- Chaining data-oriented gadgets
  - **Loop** ---> repeatedly invoke gadgets
  - **Selector** ---> selectively active gadgets

```
while (quota--) {                          // loop
  readData(sockfd, buf);                   // selector
  if(*type == NONE ) break;
  if(*type == STREAM) *size = *(srv->cur_max);
   else{ srv->typ = *type;  srv->total += *size; }
}
```

Turing Completeness

- DOP emulates a minimal language *MINDOP*
  - *MINDOP* is Turing-complete

Semantics	Statements In C	Data-Oriented Gadgets in DOP
arithmetic / logical	a op b	*p op *q
assignment	a = b	*p = *q
load	a = *b	*p = **q
store	*a = b	**p = *q
jump	goto L	vpc = &input
conditional jump	if (a) goto L	vpc = &input if *p
p – &a; q – &b; op – any arithmetic / logical operation		

Attack Construction

```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
   } //...(code skipped)...
}
```

- Gadget identification
  - statically identify load-semantics-store chain from LLVM IR
- Dispatcher identification
  - static identify loops with gadgets from LLVM IR
- Gadget stitching
  - select gadgets and dispatchers (manual)
  - check stitchability (manual)

Evaluation – Feasibility

Nine x86 programs with nine vulnerabilities

- x86 Gadgets
  - 7518 in total
  - 1273 reachable via selected CVEs
  - 8 programs can simulate all MINDOP operations
    - manually confirmed 2 can build Turing-complete attacks
- x86 Dispatchers
  - 5052 in total, 1443 contains gadgets

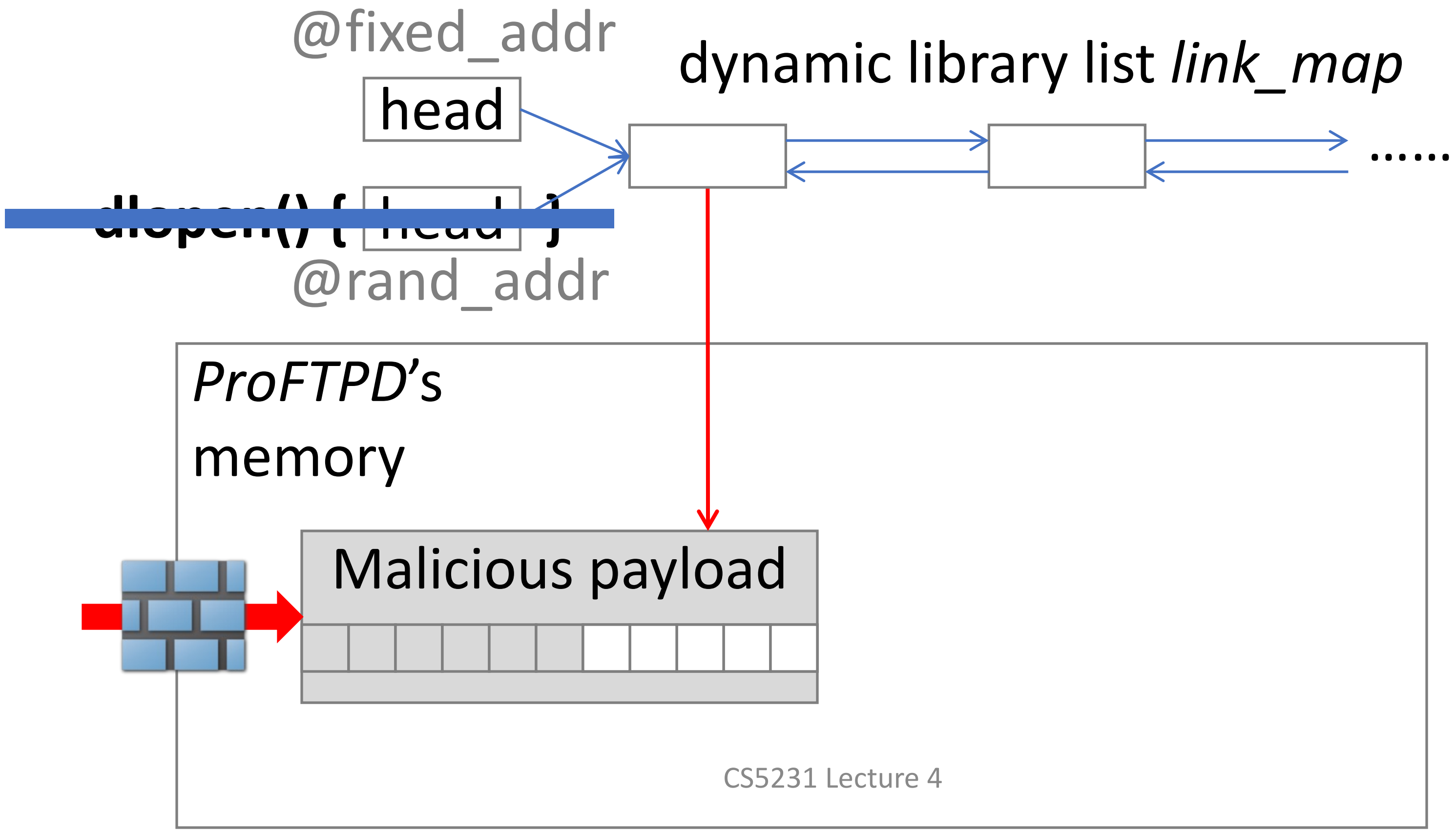
----> DOP elements are abundant

Case Study: Exploiting dlopen

- *dlopen* allows arbitrary computation
  - send malicious payload
  - corrupt link list & call dlopen

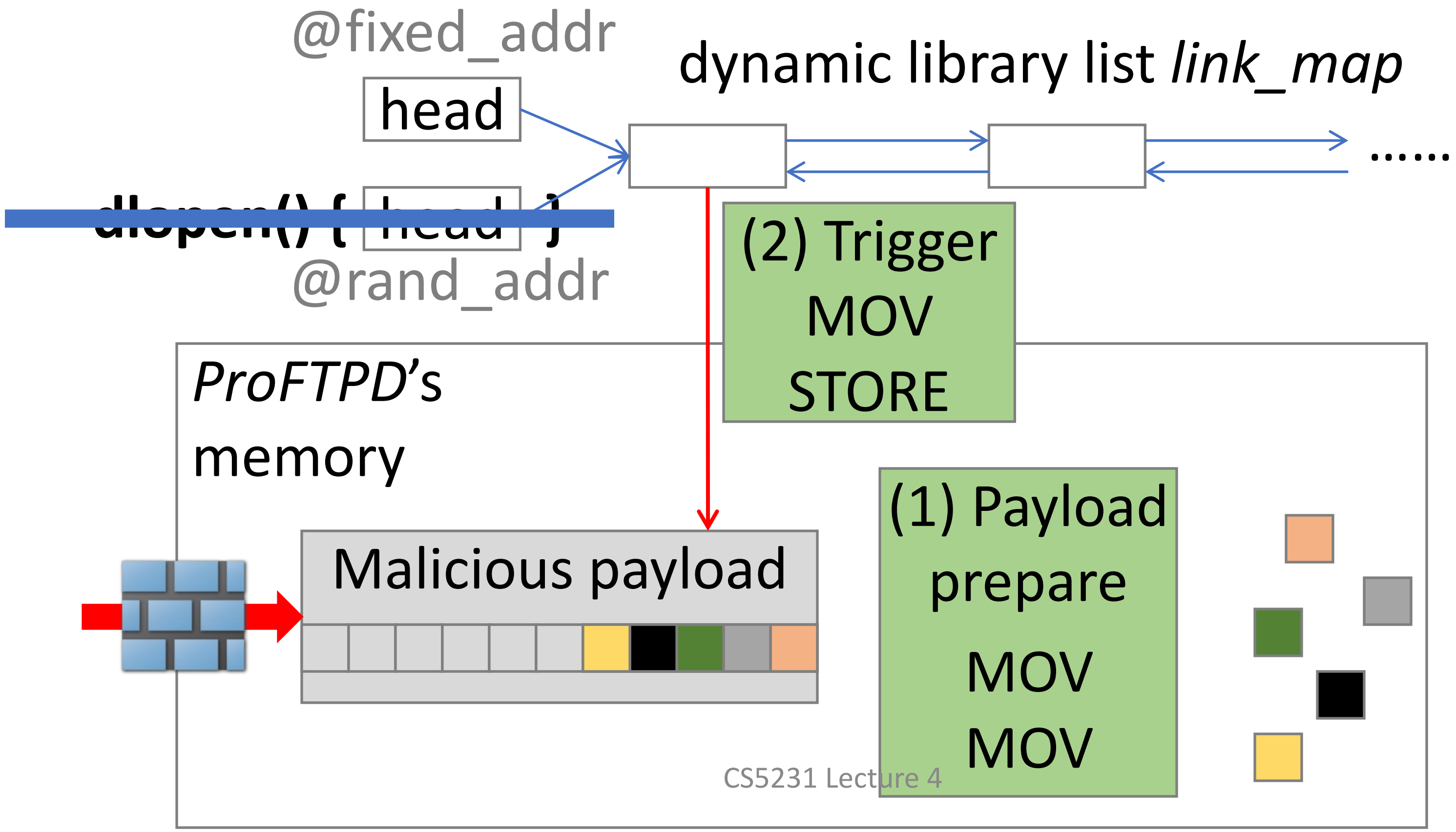
invalid input

no call to dlopen



Case Study: Exploiting dlopen

- DOP attack addresses the problems
  - construct payload in memory — invalid input
  - force call to dlopen — no call to dlopen



> 700 requests