Defenses and Secure Coding

With material from Mike Hicks, Dave Levin, and Michelle Mazurek

Recall

What is memory safety?

A memory safe program execution:

1. Only creates pointers through **standard means**
   - `p = malloc(…)`, or `p = &x`, or `p = &buf[5]`, etc.

2. Only uses a pointer to access memory that **“belongs” to that pointer**

Combines two ideas:

temporal safety and **spatial safety**

Recall

Spatial safety

- View pointers as ***capabilities***: triples (**p**,***b***,***e***)
  - **p** is the actual pointer (current address)
  - ***b*** is the base of the memory region it may access
  - ***e*** is the extent (bounds) of that region (count)
- **Access allowed** *iff* ***b*** ≤ **p** ≤ (**e**-`sizeof`(`typeof`(**p**)))

No buffer overflows

- A buffer overflow violates spatial safety

```
void copy(char *src, char *dst, int len)
{
  int i;
  for (i=0;i<len;i++) {
    *dst = *src;
    src++;
    dst++;
  }
}
```

- Overrunning bounds of source and/or destination buffers implies either `src` or `dst` is illegal

No format string attacks

- The call to **`printf`** dereferences illegal pointers

```
char *buf = “%d %d %d\n”;
printf(buf);
```

  - View the stack as a buffer defined by the number and types of the arguments it provides
  - The extra format specifiers construct pointers beyond the end of this buffer and dereference them

- Essentially a kind of buffer overflow

Temporal safety

- Violated when trying to access **undefined memory**
  - Spatial safety assures it was to a legal region
  - Temporal safety assures that region is still in play
- Memory regions either **defined** or **undefined**
  - Defined means allocated (and active)
  - Undefined means unallocated, uninitialized, or deallocated
- Pretend memory is infinitely large, no reuse

No dangling pointers

- Accessing a freed pointer violates temporal safety

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

The memory dereferenced no longer belongs to p.

- Accessing uninitialized pointers is similarly not OK:

```
int *p;
*p = 5; // violation
```

Integer overflows?

```
int f() {
  unsigned short x = 65535;
  x++; // overflows to become 0
  printf(“%d\n”,x); // memory safe
  char *p = malloc(x); // size-0 buffer!
  p[1] = ‘a’; // violation
}
```

- Integer overflows are themselves allowed
  - But can’t become illegal pointers
- Integer overflows often enable buffer overflows

For **more on memory safety**, see
http://www.pl-enthusiast.net/2014/07/21/memory-safety/

How to get memory safety?

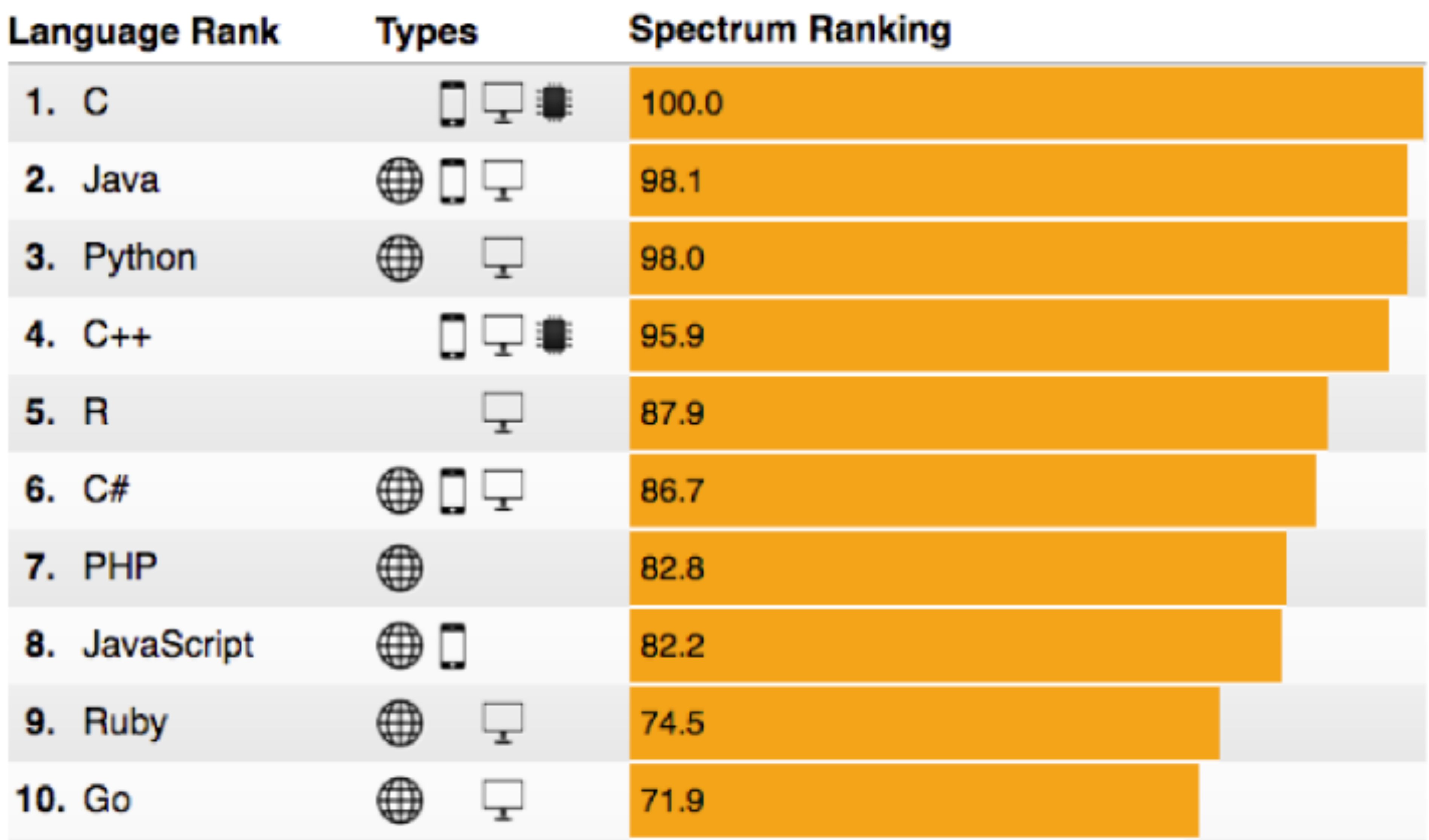
- The easiest way to avoid all of these vulnerabilities is to use a memory-safe language

- Modern languages are memory safe
  - Java, Python, C#, Ruby
    - Haskell, Scala, Go, Objective Caml, Rust

- In fact, these languages are **type safe,** which is even better (more on this shortly)

Recall

C and C++ still very popular



spectrum.ieee.org/computing/software/the-2016-top-programming-languages

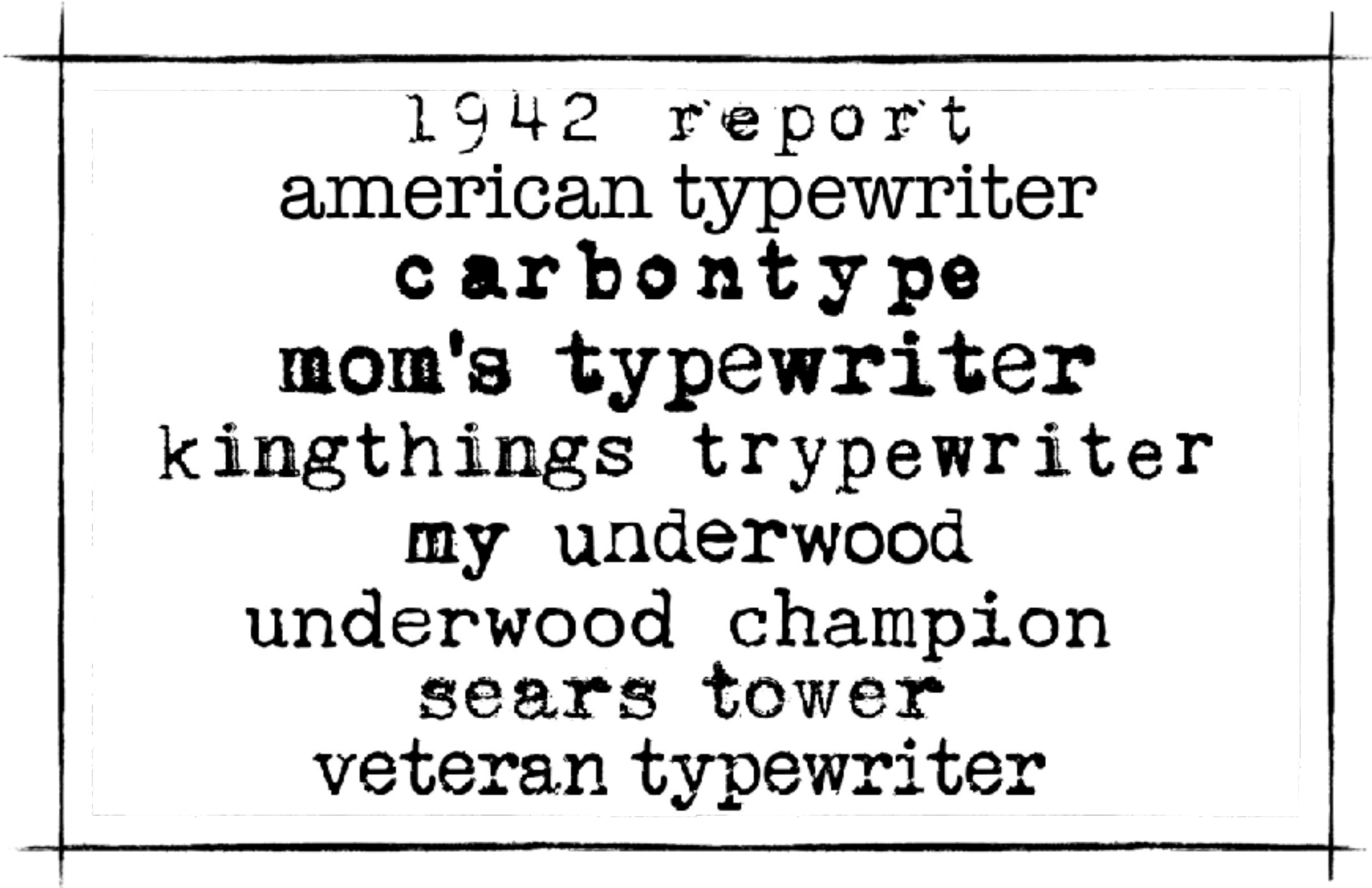
Memory safety for C

- **C/C++ are here to stay**.
  - You **can** write memory safe programs with them
  - But the language provides no guarantee
- Compilers could add code to **check for violations**
  - Out-of-bounds: immediate failure (Java *ArrayBoundsException)*
- This idea has been around for more than 20 years. **Performance has been the limiting factor.**
  - Work by Jones and Kelly in 1997 adds 12x overhead
  - Valgrind memcheck adds 17x overhead

Research progress

- **CCured** (2004), 1.5x slowdown
  - But no checking in libraries
  - Compiler rejects many safe programs
- **Softbound/CETS** (2010): 2.16x slowdown
  - Complete checking, highly flexible
- **Intel MPX** hardware (2015 in Linux)
  - Hardware support to make checking faster

https://software.intel.com/en-us/blogs/2013/07/22/intel-memory-protection-extensions-intel-mpx-support-in-the-gnu-toolchain



Type Safety

https://a.dilcdn.com/bl/wp-content/uploads/sites/8/2014/02/typewriterfont.jpg

Type safety

- Each object is ascribed a **type** (`int`, pointer to `int`, pointer to function), and
- Operations on the object are always *compatible* with the object's type
  - Type safe programs do not "go wrong" at run-time
- **Type safety** is **stronger** than memory safety

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
cmp("hello","bye"); // crash!
```

Memory safe, NOT type safe

Aside: Dynamic Typing

- Dynamically typed languages
  - Don’t require type declaration
  - e.g., Ruby and Python
  - Can be viewed as type safe
- Each object has **one type: Dynamic**
  - Each operation on a Dynamic object is permitted, but *may be unimplemented*
  - In this case, it *throws an exception*
  - Checked at **runtime** not **compile time!**

Types for Security

- Use types to enforce **security property** invariants
  - Invariants about data's privacy and integrity
  - Enforced by the type checker
- **Example**: **Java with Information Flow (JIF)**

```
int{Alice, Bob} x;
int{Alice, Bob, Chuck} y;
x = y; //OK: policy on x is stronger
y = x; //BAD: policy on y is weaker
```

Types have ***security labels*** that govern ***information flow***

http://www.cs.cornell.edu/jif

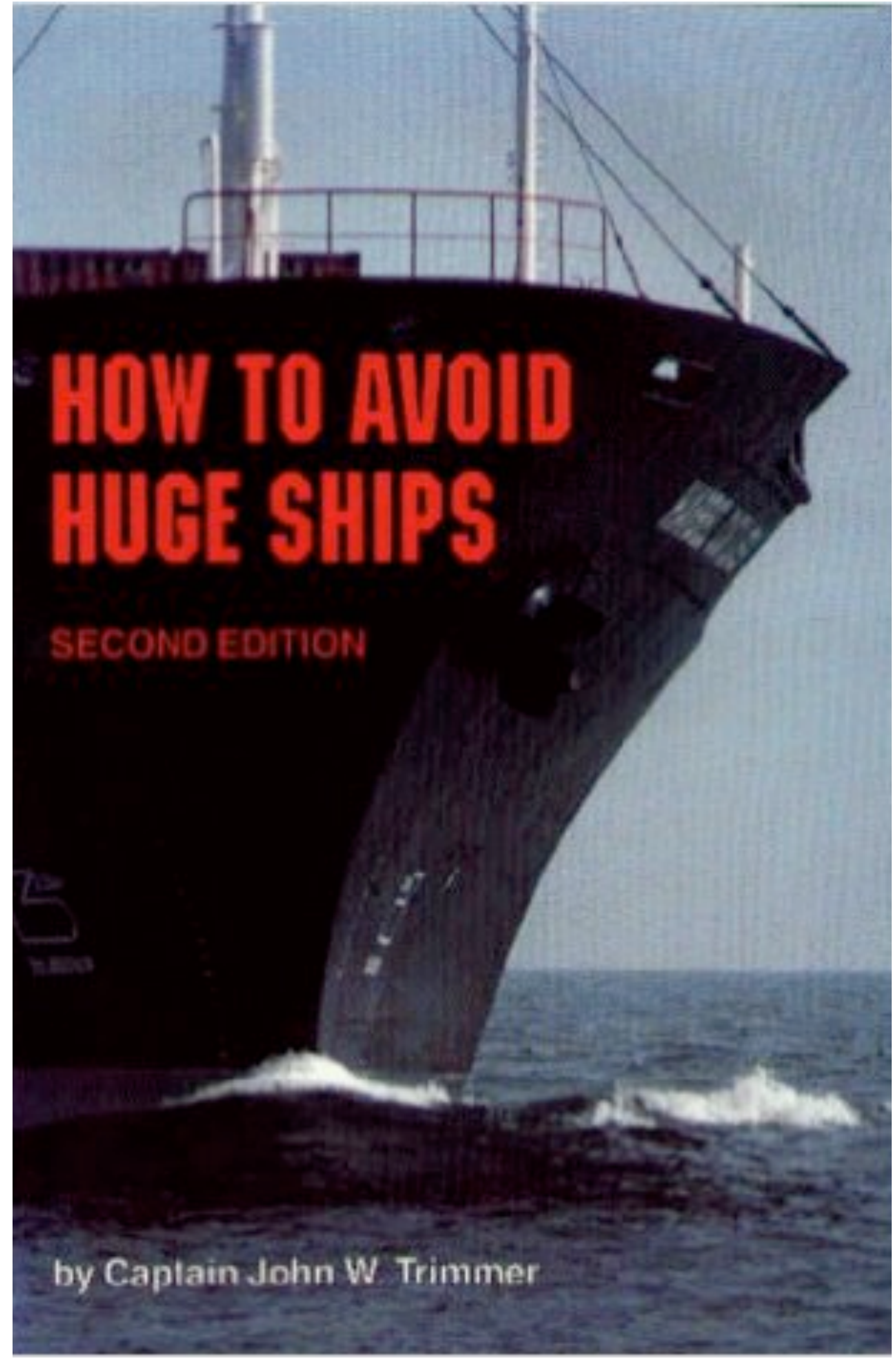
Why not type safety?

- C/C++ often chosen **for performance** reasons
  - Manual memory management
  - Tight control over object layouts
  - Interaction with low-level hardware
- **Enforcement** of type safety is typically **expensive**
  - **Garbage collection** avoids temporal violations
    - Can be as fast as malloc/free, often uses much more memory
  - **Bounds** and **null-pointer checks** avoid spatial violations
  - **Hiding representation** may **inhibit optimization**
    - Many C-style casts, pointer arithmetic, & operator, not allowed

A new hope?

- Many applications do not need C/C++
  - Or the risks that come with it
- New languages aim to provide **similar features** to C/C++ while **remaining type safe**
  - Google's **Go**, Mozilla's **Rust**, Apple's **Swift**

Avoiding exploitation



https://www.amazon.com/Avoid-Huge-Ships-John-Trimmer/dp/0870334336

Until we have a widespread type-safe replacement for C, what can we do?

- Make bugs **harder to exploit**
  - Crash but not code execution
- **Avoid bugs** with better programming
  - Secure coding practices, code review, testing

Better together: Try to avoid bugs, *but also* add protection if some slip through

Avoiding exploitation

Recall the steps of a stack smashing attack:

- Putting attacker code into memory
  - (No zeroes or other stoppers)
- Getting `%eip` to point to attacker code
- Finding the return address

How can we make these attack steps more difficult?

- Side note: How to implement fixes?
- Goal: change libraries, compiler, or OS
  - Fix *architectural design*, not code
  - Avoid changing (lots of) application code
  - One update fixes all programs at once

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

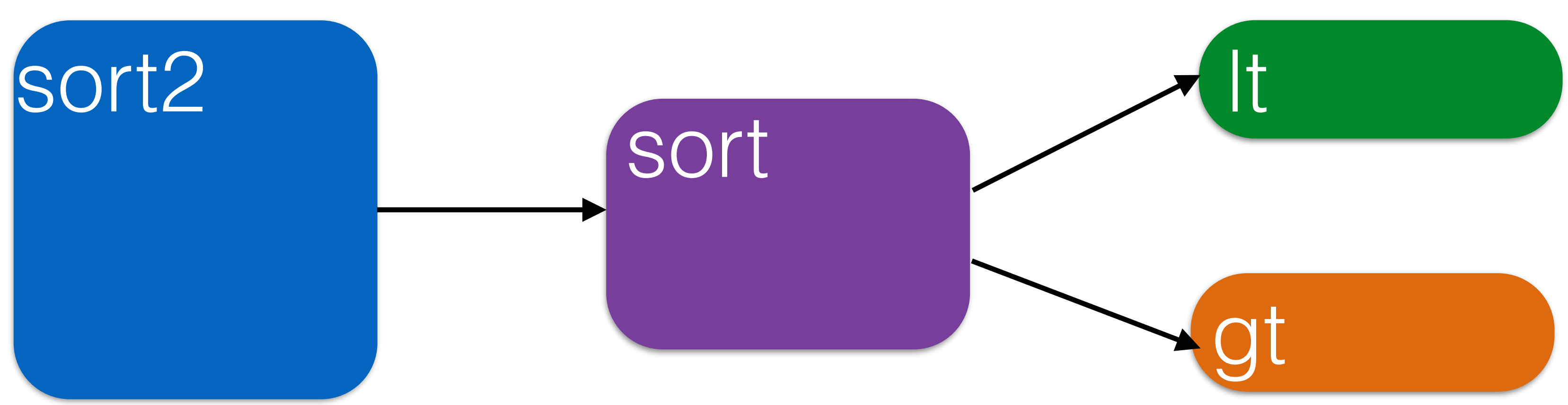
- *Detect deviations from expectation efficiently*

- *Avoid compromise of the detector*

Call Graph

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

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

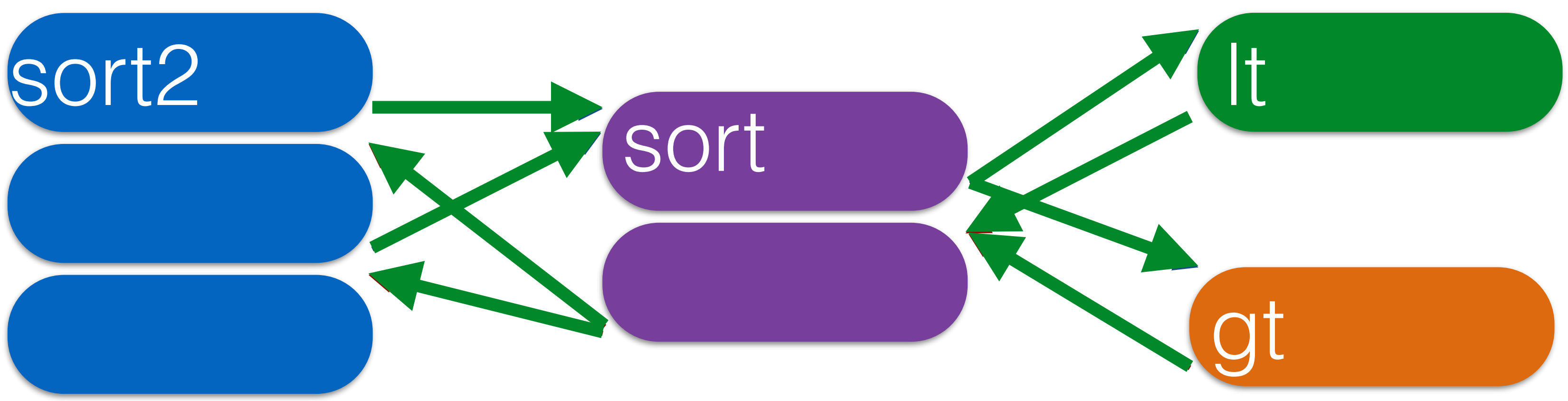


Which functions call other functions

Control Flow Graph

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

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



Break into ***basic blocks***

Distinguish ***calls*** *from* ***returns***

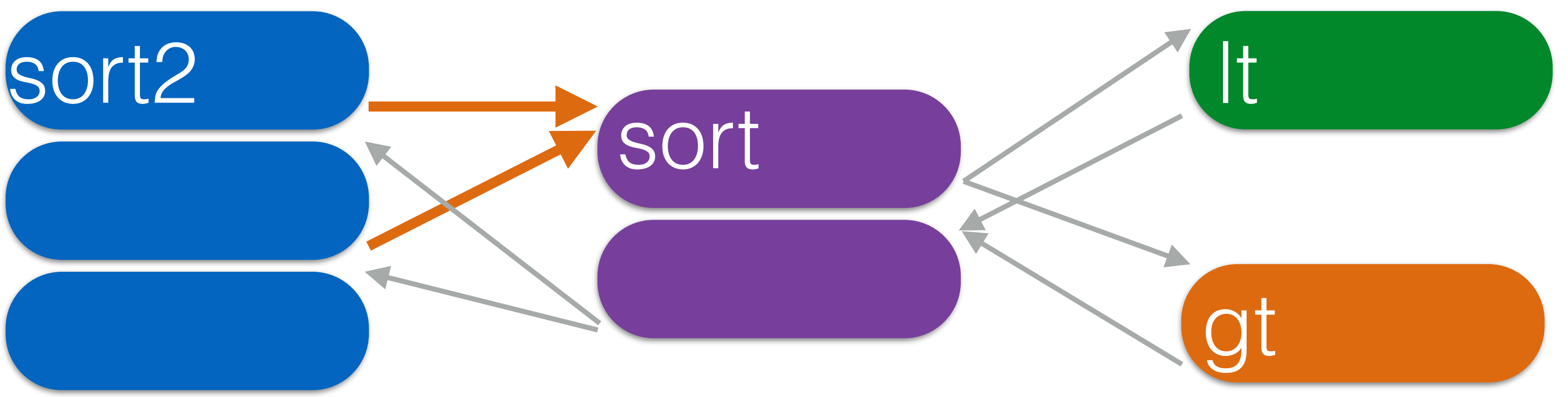
CFI: Compliance with CFG

- **Compute the call/return CFG** in advance
  - During compilation, or from the binary
- **Monitor the control flow** of the program and ensure that it only follows paths allowed by the CFG
- Observation: **Direct calls** need not be monitored
  - Assuming the code is immutable, the target address cannot be changed
- Therefore: **monitor only indirect calls**
  - `jmp`, `call`, `ret` with non-constant targets

Control Flow Graph

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

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

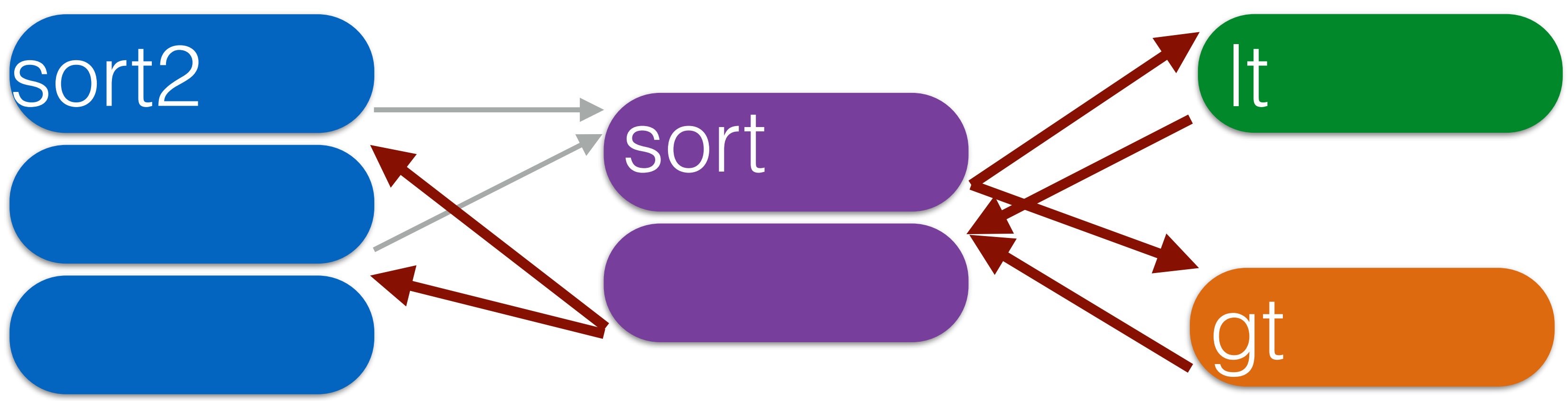


Direct calls *(always the same target)*

Control Flow Graph

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

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



Indirect transfer *(*`call` *via register, or* `ret`*)*

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

- *Detect deviations from expectation efficiently*

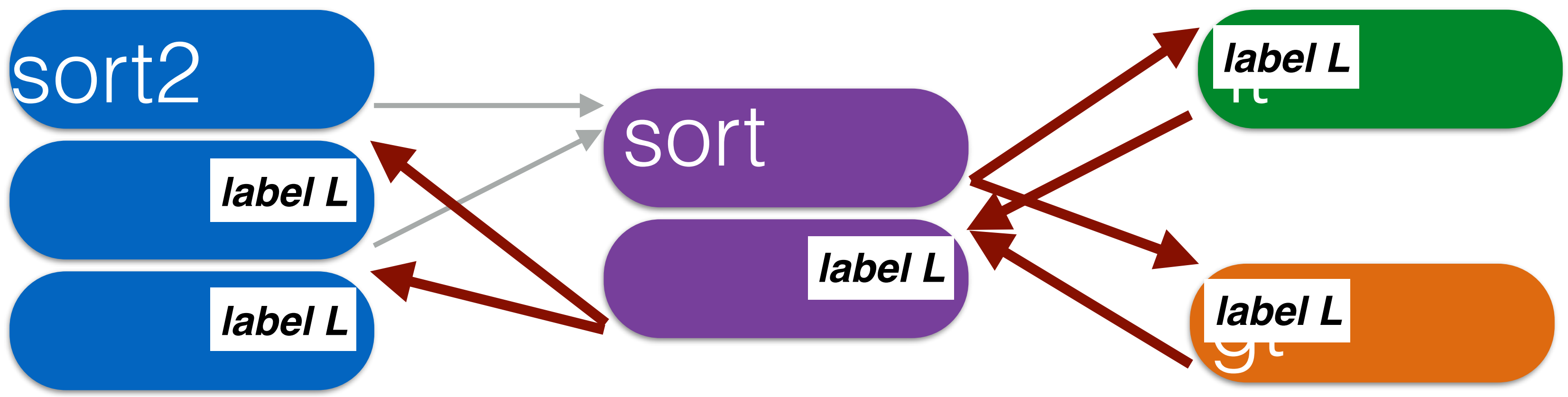
  In-line reference monitor (IRM)

- *Avoid compromise of the detector*

In-line Monitor

- Implement the monitor in-line, as a **program transformation**
- Insert a **label just before the target address** of an indirect transfer
- Insert **code to check the label of the target** at each indirect transfer
  - Abort if the label does not match
- The **labels are determined by the CFG**

Simplest labeling

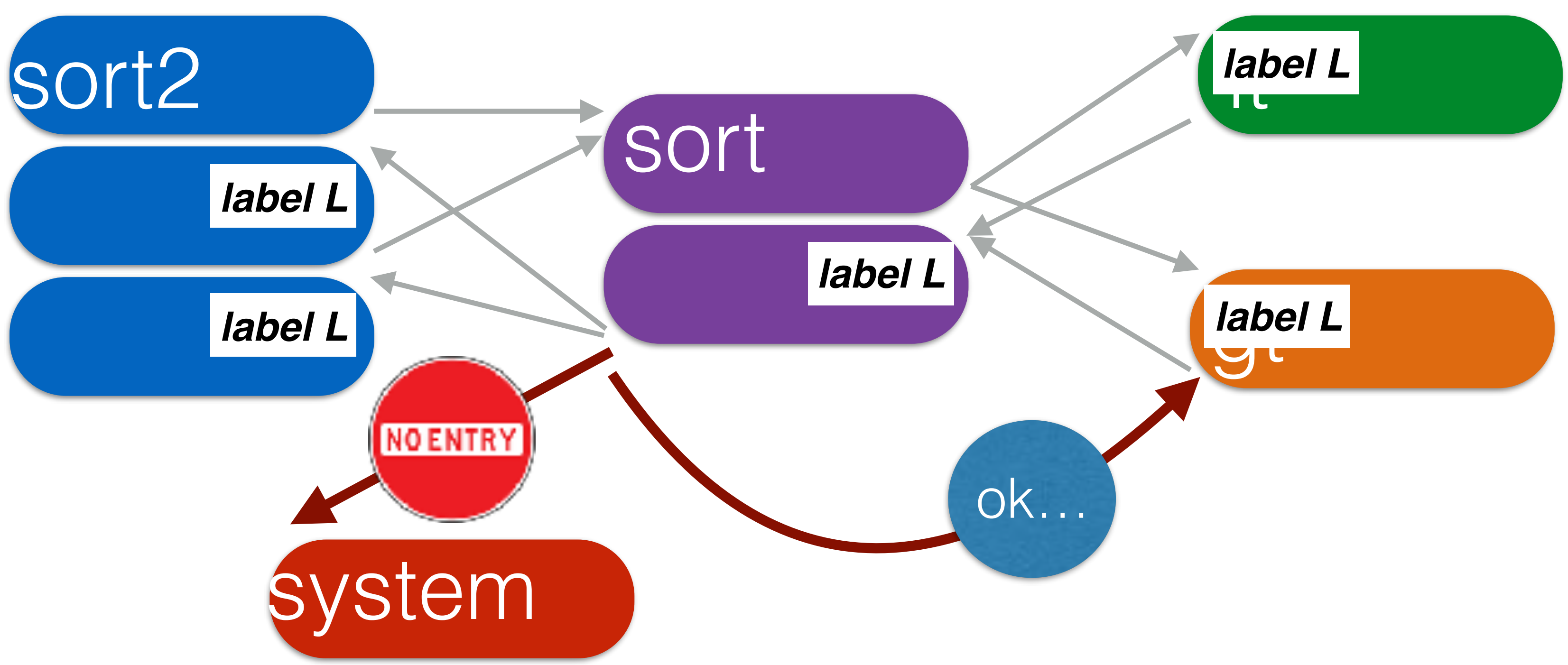


Use the same label at all targets:

label just means it's OK to jump here.

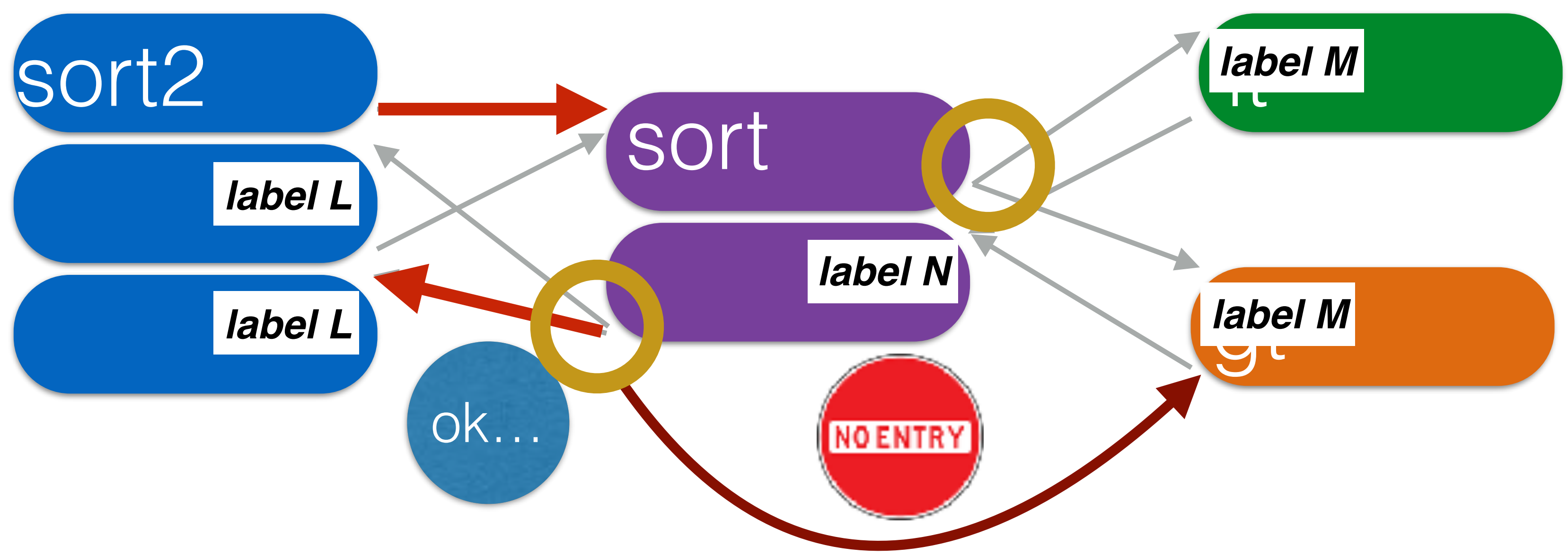
What could go wrong?

Simplest labeling



- Can’t return to functions that aren’t in the graph
- **Can** return to the right function in the wrong order

Detailed labeling



- All potential destinations of **same source** must match
  - Return sites from calls to `sort` must share a label (*L*)
  - Call targets `gt` and `lt` must share a label (*M*)
  - Remaining label unconstrained (*N*)

Prevents more abuse than simple labels,
but still permits call from site A to return to site B

Classic CFI instrumentation

Before CFI

```
FF 53 08                    call   [ebx+8]             ; call a function pointer
```

is instrumented using `prefetchnta` destination IDs, to become:

After CFI

```
8B 43 08                    mov    eax, [ebx+8]        ; load pointer into register
3E 81 78 04 78 56 34 12     cmp    [eax+4], 12345678h  ; compare opcodes at destination
75 13                       jne    error_label         ; if not ID value, then fail
FF D0                       call   eax                 ; call function pointer
3E 0F 18 05 DD CC BB AA     prefetchnta [AABBCCDDh]    ; label ID, used upon the return
```

Fig. 4. Our CFI implementation of a call through a function pointer.

Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return, and pop 16 extra bytes

is instrumented using `prefetchnta` destination IDs, to become:

```
8B 0C 24                    mov    ecx, [esp]          ; load address into register
83 C4 14                    add    esp, 14h            ; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA     cmp    [ecx+4], AABBCCDDh  ; compare opcodes at destination
75 13                       jne    error_label         ; if not ID value, then fail
FF E1                       jmp    ecx                 ; jump to return address
```

Classic CFI instrumentation

```
FF 53 08                  call   [ebx+8]           ; call a function pointer
```

is instrumented using prefetchnta destination IDs, to become:

```
8B 43 08                  mov   eax, [ebx+8]       ; load pointer into register
3E 81 78 04 78 56 34 12   cmp   [eax+4], 12345678h ; compare opcodes at destination
75 13                     jne   error_label        ; if not ID value, then fail
FF D0                     call  eax                ; call function pointer
3E 0F 18 05 DD CC BB AA   prefetchnta [AABBCCDDh]  ; label ID, used upon the return
```

Fig. 4. Our CFI implementation of a call through a function pointer.

Bytes (opcodes)	x86 assembly code	Comment

```
C2 10 00                  ret   10h                ; return, and pop 16 extra bytes
```

is instrumented using prefetchnta destination IDs, to become:

```
8B 0C 24                  mov   ecx, [esp]         ; load address into register
83 C4 14                  add   esp, 14h           ; pop 20 bytes off the stack
3E 81 79 04 DD CC BB AA   cmp   [ecx+4], AABBCCDDh ; compare opcodes at destination
75 13                     jne   error_label        ; if not ID value, then fail
FF E1                     jmp   ecx                ; jump to return address
```

Efficient?

- **Classic CFI** (2005) imposes **16% overhead** on average, **45%** in the **worst case**
  - Works on arbitrary executables
  - Not modular (no dynamically linked libraries)
- **Modular CFI** (2014) imposes **5% overhead** on average, **12%** in the **worst case**
  - C only (part of LLVM)
  - Modular, with separate compilation
  - http://www.cse.lehigh.edu/~gtan/projects/upro/

Control-flow Integrity (CFI)

- *Define "expected behavior":*

  Control flow graph (CFG)

- *Detect deviations from expectation efficiently*

  In-line reference monitor (IRM)

- *Avoid compromise of the detector*

  Sufficient randomness, immutability

Can we defeat CFI?

- **Inject code** that has a **legal label**
  - *Won't work* because we assume **non-executable data**
- **Modify code labels** to allow the desired control flow
  - *Won't work* because the **code is immutable**
- **Modify stack during a check**, to make it seem to succeed
  - *Won't work* because **adversary cannot change registers** into which we load relevant data
    - No time-of-check, time-of-use bug (TOCTOU)

CFI Assurances

- CFI defeats **control flow-modifying** attacks
  - Remote code injection, ROP/return-to-libc, etc.
- But **not manipulation of control-flow** that is **allowed by the labels**/graph
  - Called **mimicry attacks**
  - The simple, single-label CFG is susceptible to these
- **Nor data leaks or corruptions**
  - Heartbleed would not be prevented
  - Nor the `authenticated` overflow
    - Which is allowed by the graph

```
void func(char *arg1)
{
  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, str);
  if(authenticated) { …
}
```

Secure?

- MCFI can **eliminate 95.75% of ROP gadgets** on x86-64 versions of SPEC2006 benchmark suite
    - By ruling their use non-compliant with the CFG
- Average Indirect-target Reduction (AIR) **> 99%**
    - Essentially, the percentage of **possible targets of indirect jumps** that CFI rules out

Secure Coding

Secure coding in C

- Since the language provides few guarantees, **developers** must use **discipline**
- Good **reference guide**: **CERT C Coding Standard**
  - https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard
    - Similar guides for other languages (e.g., Java)
  - See also: *David Wheeler*: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/internals.html

*Combine with **advanced code review and testing***

Design vs. Implementation

- In general, we strive to follow principles and rules
  - A **principle** is a design goal with many possible manifestations.
  - A **rule** is a specific practice consistent with sound principles.
    - The difference between these can sometimes be fuzzy
- Here we look at **rules** for **good C coding**
  - In particular, to **avoid implementation errors** that could result in violations of memory safety
- Later: **Consider principles and rules more broadly**

General **Principle**: *Robust* coding

- Like **defensive driving**
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash (or worse)
  - It's about ***minimizing trust***
- Each module **pessimistically checks its assumed preconditions** (on outside callers)
  - Even if you "know" clients will not send a NULL pointer
  - … Better to throw an exception (or even exit) than run malicious code

Rule: Enforce input compliance

Recall

Read integer

Read message

Echo back (partial) message

```
int main() {
 char buf[100], *p;

 while (1) {
  p = fgets(buf,sizeof(buf),stdin);
  len = atoi(p);
  p = fgets(buf,sizeof(buf),stdin);
  len = MIN(len,strlen(buf));
  for (i=0; i<len; i++) {
   if (!iscntrl(buf[i]))
    putchar(buf[i]);
   else putchar('.');
  }
  printf("\n");
 }
 ...
```

Sanitizes input to be compliant

~~*len may exceed actual message length!*~~

Rule: Enforce input compliance

```
char digit_to_char(int i) {
  char convert[] = "0123456789";
  if(i < 0 || i > 9)
    return '?';

}
```

possible overflow

- **Unfounded trust** in received input is a recurring source of vulnerabilities
    - We will see many more examples in the course

Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length

```
char str[4];
char buf[10] = “good”;
strcpy(str,”hello”); // overflows str
strcat(buf,”day to you”); // overflows buf
```

- Safe versions check the destination length

```
char str[4];
char buf[10] = “good”;
strlcpy(str,”hello”,sizeof(str)); //fails
strlcat(buf,”day to you”,sizeof(buf));//fails
```

Detour: strncpy vs. strlcpy

Recall

```
void vulnerable(char *name_in)
{
    char buf[10];
    strncpy(buf, name_in, sizeof(buf));
    printf("Hello, %s\n", buf);
}
```

name_in = "0123456789ABC"

does not append NULL

prints until NULL

- strncpy is "safe" because it won't overwrite
  - But string not properly terminated
  - Always add `buf[sizeof(buf) -1] = 0;`
- strlcpy is better — copies (n-1) bytes max and appends the null for you!

Replacements

- … for string-oriented functions
  - `strcat` ⟹ `strlcat`
  - `strcpy` ⟹ `strlcpy`
  - `strncat` ⟹ `strlcat`
  - `strncpy` ⟹ `strlcpy`
  - `sprintf` ⟹ `snprintf`
  - `vsprintf` ⟹ `vsnprintf`
  - `gets` ⟹ `fgets`
- Microsoft versions different
  - `strcpy_s`, `strcat_s`, `...`

Rule: Don’t forget NUL terminator

- Strings require one additional character to store the NUL. Forgetting that could lead to overflows.

```
char str[3];
strcpy(str,”bye”); // write overflow
int x = strlen(str); // read overflow
```

- Using safe string library calls will catch this mistake

```
char str[3];
strlcpy(str,”bye”,3); // blocked
int x = strlen(str); // returns 2
```

Rule: Understand pointer arithmetic

- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

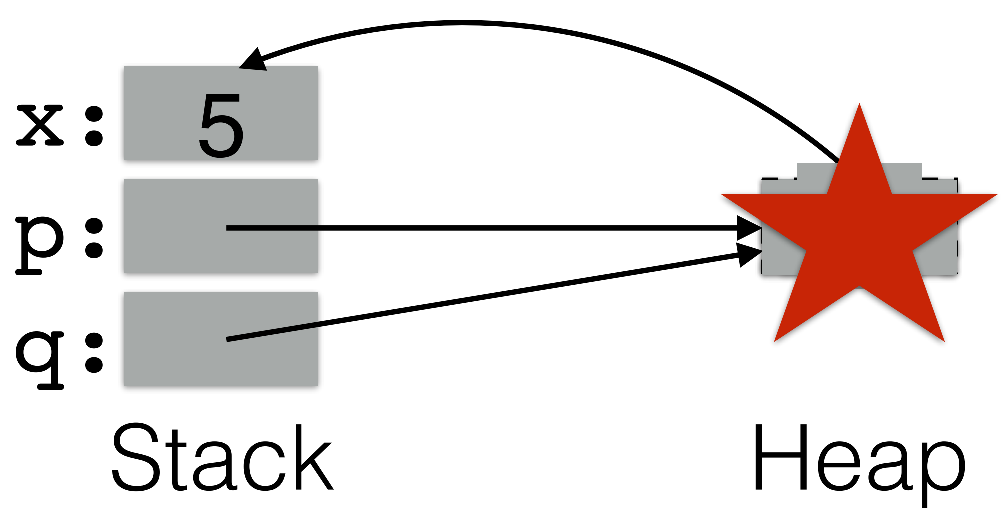
```
int buf[SIZE] = { … };
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
  *buf_ptr++ = getnext(); // will overflow
}
```

- So, **use the right units**

```
while (!done() && buf_ptr < (buf + SIZE)) {
  *buf_ptr++ = getnext(); // stays in bounds
}
```

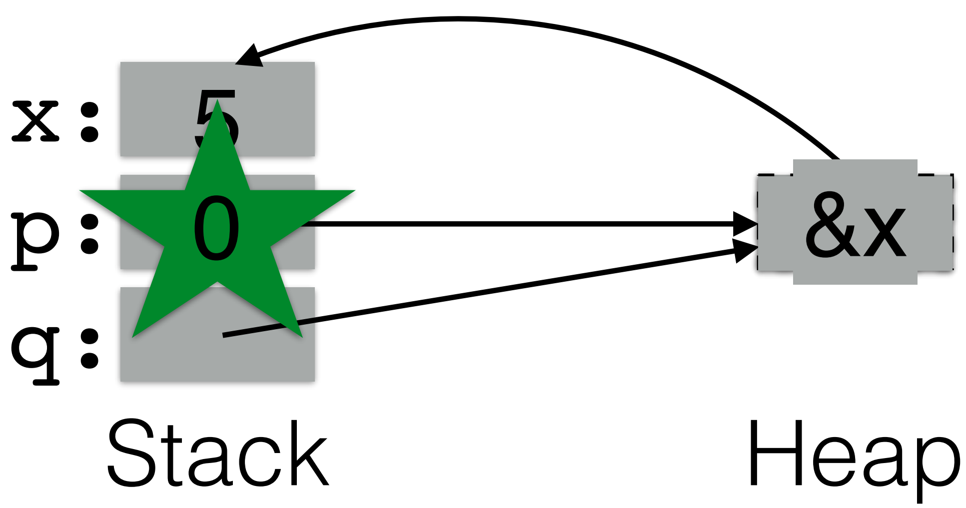
Principle: Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //reuses p's space
*q = &x;
*p = 5; //(good) crash
**q = 3;
```



Principle: Manage memory properly

```
int foo(int arg1, int arg2) {
  struct foo *pf1, *pf2;
  int retc = -1;

  pf1 = malloc(sizeof(struct foo));
  if (!isok(arg1)) goto DONE;
    …
  pf2 = malloc(sizeof(struct foo));
  if (!isok(arg2)) goto FAIL_ARG2;
  …
  retc = 0;

 FAIL_ARG2:
  free(pf2); //fallthru
 DONE:
  free(pf1);
  return retc;
}
```

- **Rule: Use *goto chains* to avoid duplicated or missed code**
  - Mimics try/finally in languages like Java
- Confirm your logic!
  - *Gotofail* bug

Anatomy of a `goto` fail

```
static OSStatus
SSLVerifySignedServerKeyExchange(...)
{
   OSStatus         err;
   ...

   if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      goto fail;
   if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      goto fail;
      goto fail;  // triggers if if fails: err == 0
   if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;
   ...  // SSL verify called somewhere in here

fail:
   SSLFreeBuffer(&signedHashes);
   SSLFreeBuffer(&hashCtx);
   return err; // returns err = 0 (SUCCESS), without SSL verify function
}
```

Rule: Favor safe libraries

- Designed to ensure safe use of strings, pointers, etc.
  - Encapsulate **well-thought-out design**. *Take advantage!*
- **Smart pointers**
  - Pointers with only safe operations
  - Lifetimes managed appropriately
  - First in the Boost library, now a C++11 standard
- **Networking**: Google protocol buffers, Apache Thrift
  - For dealing with network-transmitted data
  - Ensures input validation, parsing, etc.
  - Efficient

Rule: Use a safe allocator

- ASLR challenges libc exploits by making the library base unpredictable
- **Challenge heap-based overflows** by making the **addresses** returned by `malloc` **unpredictable**
  - Can have some negative performance impact
- Example implementations:
  - **Windows Fault-Tolerant Heap**
    - http://msdn.microsoft.com/en-us/library/windows/desktop/dd744764(v=vs.85).aspx
  - **DieHard** (on which fault-tolerant heap is based)
    - http://plasma.cs.umass.edu/emery/diehard.html

Gashlycode Tinies

by **Andrew Myers** @Cornell
inspired by the *Gashlycrumb Tinies* by *Edward Gorey*

Gashlycode Tinies

A is for Amy whose **malloc was one byte short**
B is for Basil who **used a quadratic sort**

C is for Chuck who **checked floats** for **equality**
D is for Desmond who **double-freed memory**

E is for Ed whose **exceptions weren't handled**
F is for Franny whose stack **pointers dangled**

G is for Glenda whose **reads** and **writes raced**
H is for Hans who **forgot** the **base case**

I is for Ivan who did **not initialize**
J is for Jenny who **did not know *Least Surprise***

K is for Kate whose **inheritance depth** might shock
L is for Larry who **never released** a **lock**

M is for Meg who used **negatives** as **unsigned**
N is for Ned with **behavior** left **undefined**

O is for Olive whose index was **off by one**
P is for Pat who ignored **buffer overrun**

Q is for Quentin whose **numbers** had **overflows**
R is for Rhoda whose code left the **rep exposed**

S is for Sam who **skipped retesting** after **wait()**
T is for Tom who **lacked TCP_NODELAY**

U is for Una whose **functions were most verbose**
V is for Vic who **subtracted when floats were close**

W is for Winnie who **aliased arguments**
X is for Xerxes who thought **type casts made good sense**

Y is for Yorick whose **interface** was **too wide**
Z is for Zack whose **code nulls were** often **spied**