CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

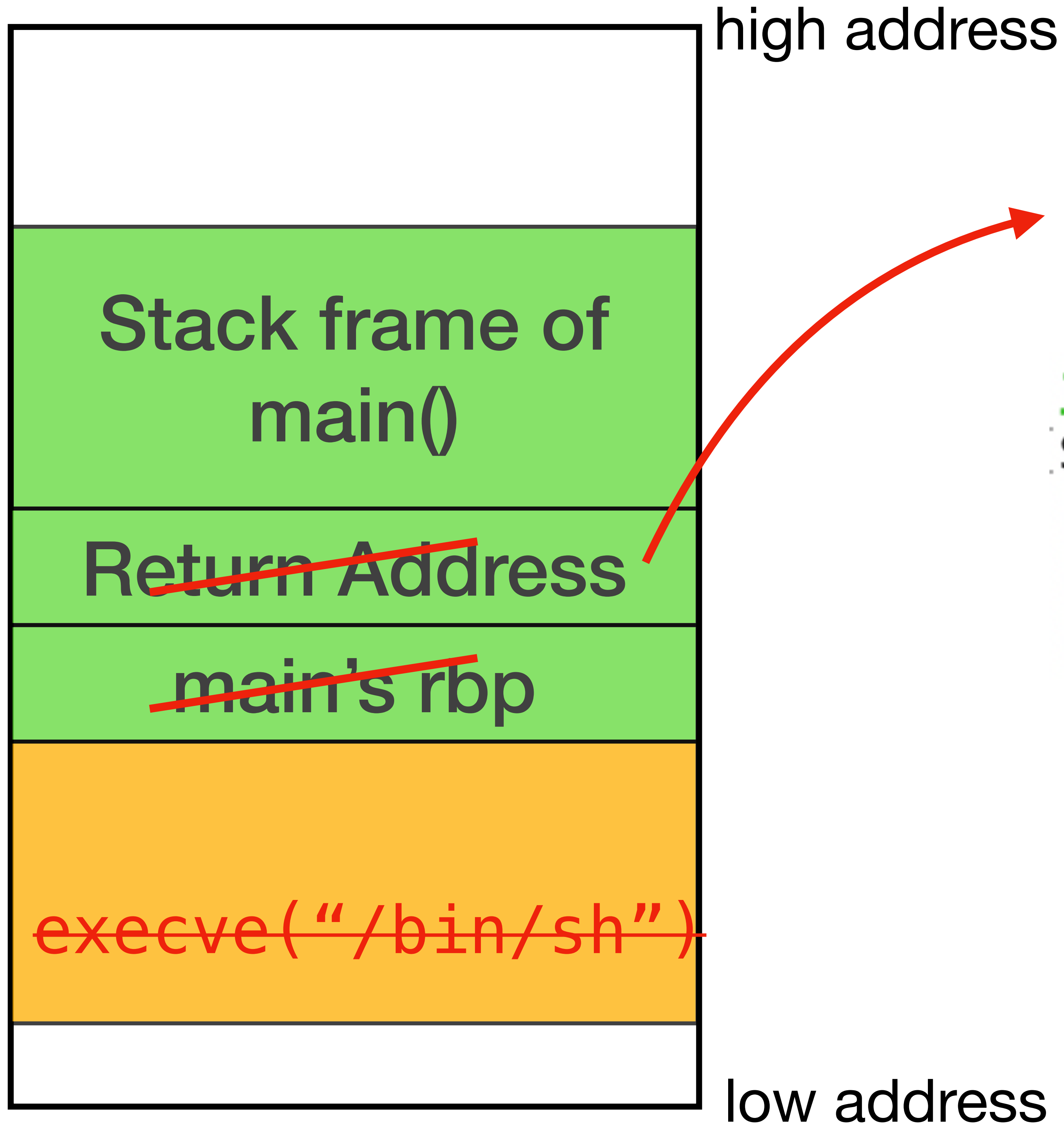
Department of Computer Science

George Washington University

Outline

- Review: ASLR
- Protecting Return Addresses
- Control-flow Integrity

Exploiting Existing and Executable Code



How about "returning" to some library code?

```
jie@gwsyssec: ~/courses/csci6545/lectures
$ ldd demo
        linux-vdso.so.1 (0x00007ffffadfd000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f48a2c00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f48a2efc000)
```

```
(gdb) info proc mappings
process 74581
Mapped address spaces:

          Start Addr           End Addr       Size     Offset  Perms  objfile
      0x555555554000     0x555555555000     0x1000        0x0  r--p   /home/jie/courses/csci6545/lectures/demo
      0x555555555000     0x555555556000     0x1000     0x1000  r-xp   /home/jie/courses/csci6545/lectures/demo
      0x555555556000     0x555555557000     0x1000     0x2000  r--p   /home/jie/courses/csci6545/lectures/demo
      0x555555557000     0x555555558000     0x1000     0x2000  r--p   /home/jie/courses/csci6545/lectures/demo
      0x555555558000     0x555555559000     0x1000     0x3000  rw-p   /home/jie/courses/csci6545/lectures/demo
      0x7ffff7c00000     0x7ffff7c28000    0x28000        0x0  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7c28000     0x7ffff7dbd000   0x195000    0x28000  r-xp   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7dbd000     0x7ffff7e15000    0x58000   0x1bd000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e15000     0x7ffff7e16000     0x1000   0x215000  ---p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e16000     0x7ffff7e1a000     0x4000   0x215000  r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e1a000     0x7ffff7e1c000     0x2000   0x219000  rw-p   /usr/lib/x86_64-linux-gnu/libc.so.6
      0x7ffff7e1c000     0x7ffff7e29000     0xd000        0x0  rw-p
      0x7ffff7fa6000     0x7ffff7fa9000     0x3000        0x0  rw-p
      0x7ffff7fbb000     0x7ffff7fbd000     0x2000        0x0  rw-p
      0x7ffff7fbd000     0x7ffff7fc1000     0x4000        0x0  r--p   [vvar]
      0x7ffff7fc1000     0x7ffff7fc3000     0x2000        0x0  r-xp   [vdso]
      0x7ffff7fc3000     0x7ffff7fc5000     0x2000        0x0  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7fc5000     0x7ffff7fef000    0x2a000     0x2000  r-xp   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7fef000     0x7ffff7ffa000     0xb000    0x2c000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffb000     0x7ffff7ffd000     0x2000    0x37000  r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffff7ffd000     0x7ffff7fff000     0x2000    0x39000  rw-p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0  rw-p   [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0  --xp   [vsyscall]
```

Life of a C Program: Execution

Loading → Execution → Termination

Loading

- Initializing memory layout
- (Optional) Dynamic linking, e.g. `libc`
- Environment initialization, e.g., stack setup
- Setting program counter (PC) to `_start()`

Execution

- `_start()` calls `main()`
- `main() runs the program`

Termination

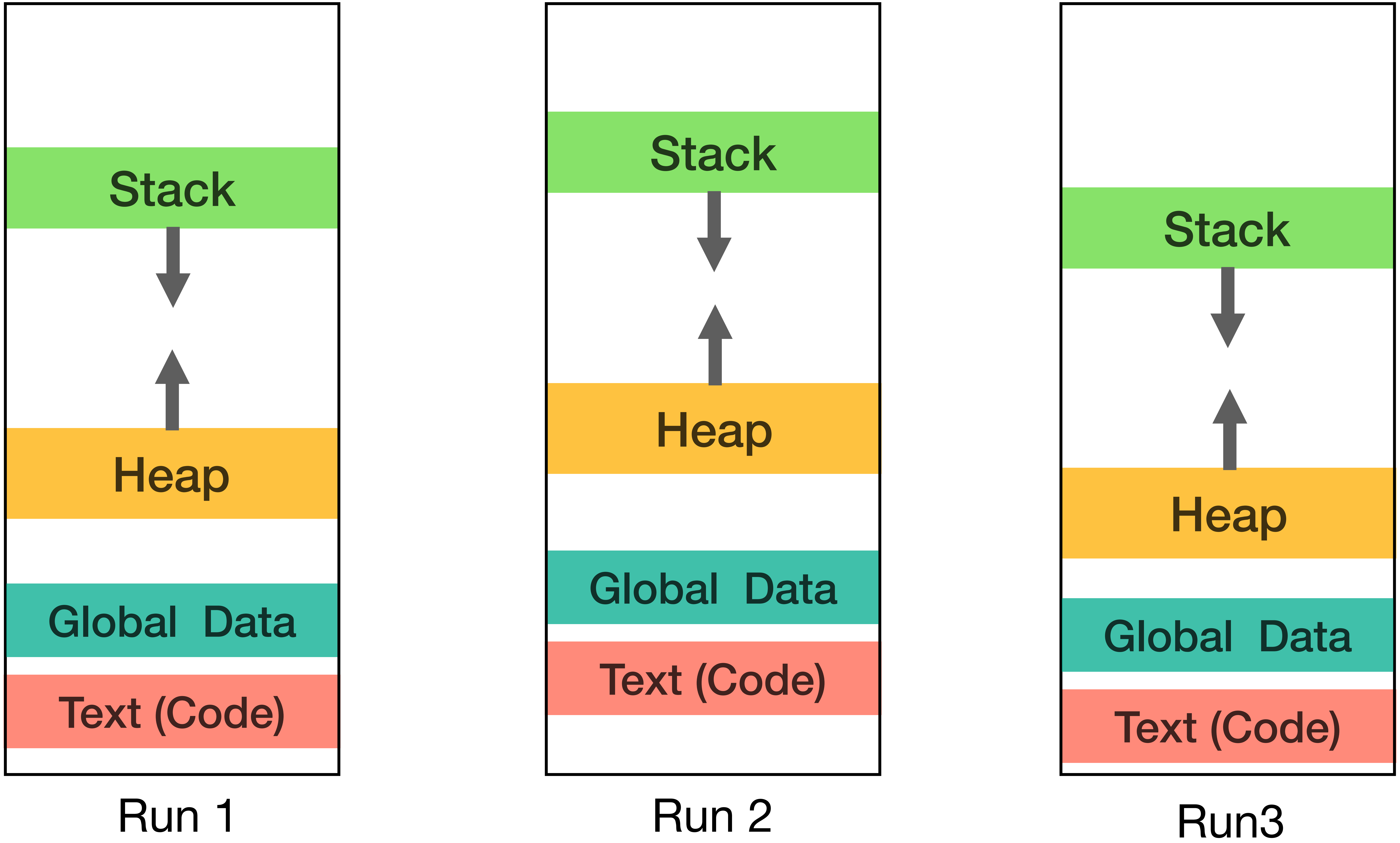
- `main()` returns,
- `_start()` calls `exit()`
- cleanup and shutdown

Address Space Layout Randomization (ASLR)

Introducing randomness into memory regions of a program

- During program initialization, done by the program loader
- Can also happen during static linking time
- Making it hard to figure out attacked target addresses

Address Space Layout Randomization (ASLR)



Address Space Layout Randomization (ASLR)

- **When** to randomize address space?
  - Only at loading time or also at run-time?
  - What should the randomization frequency be?
- **What** to randomize?
  - Which memory regions to randomize?
  - Should we randomize each memory objects?
- **How** to randomize?
  - How many bits to randomize?

Case Study: PaX's ASLR on x86-32 Systems

- Randomized bits: number of bits ASLR can vary for a memory region
- Attacked bits: number of bits attackers can bypass (e.g., partial info leak)

- Rs: number of randomized bits for the stack
- Rm: number of randomized bits for the mmap() area
- Rx: number of randomized bits for the main executable
- As: number of bits of stack randomness attacked in one attempt
- Am: number of bits of `mmap()` randomness attacked in one attempt
- Ax: number of bits of main executable randomness attacked in on attempt
- Probability of success within x number of attempts:
  - Brute-force attacks: $Pb(x) = x / 2^n$
  - Random guss attacks: $Pb(x) = 1 - (1 - 2^{-n})^x$

where $n = Rs-As + Rm-Am + Rx-Ax$, i.e., the number of randomized bits to find.

Attacking PaX ASLR

Attack the Apache http server with `ret2libc`

- Server takes requests for connections from remote users
  - ‣ Creating a new child process to handle the request
- A stack buffer overflow bug was exploited when handling user input
  - ‣ The vulnerable buffer's address is stored as a local variable on stack
- ASLR setting:
  - ‣ Starting address of each memory region is randomized
  - ‣ Randomized bits: 16 bits for `mmap()` and code, and 24 bits for stack
    - Kernel maintains a `delta_mmap` variable as the offset to the start address of the `mmap()` region, which is `0x40000000`.
- Attacking goal: Invoke `system()` with argument to launch a shell

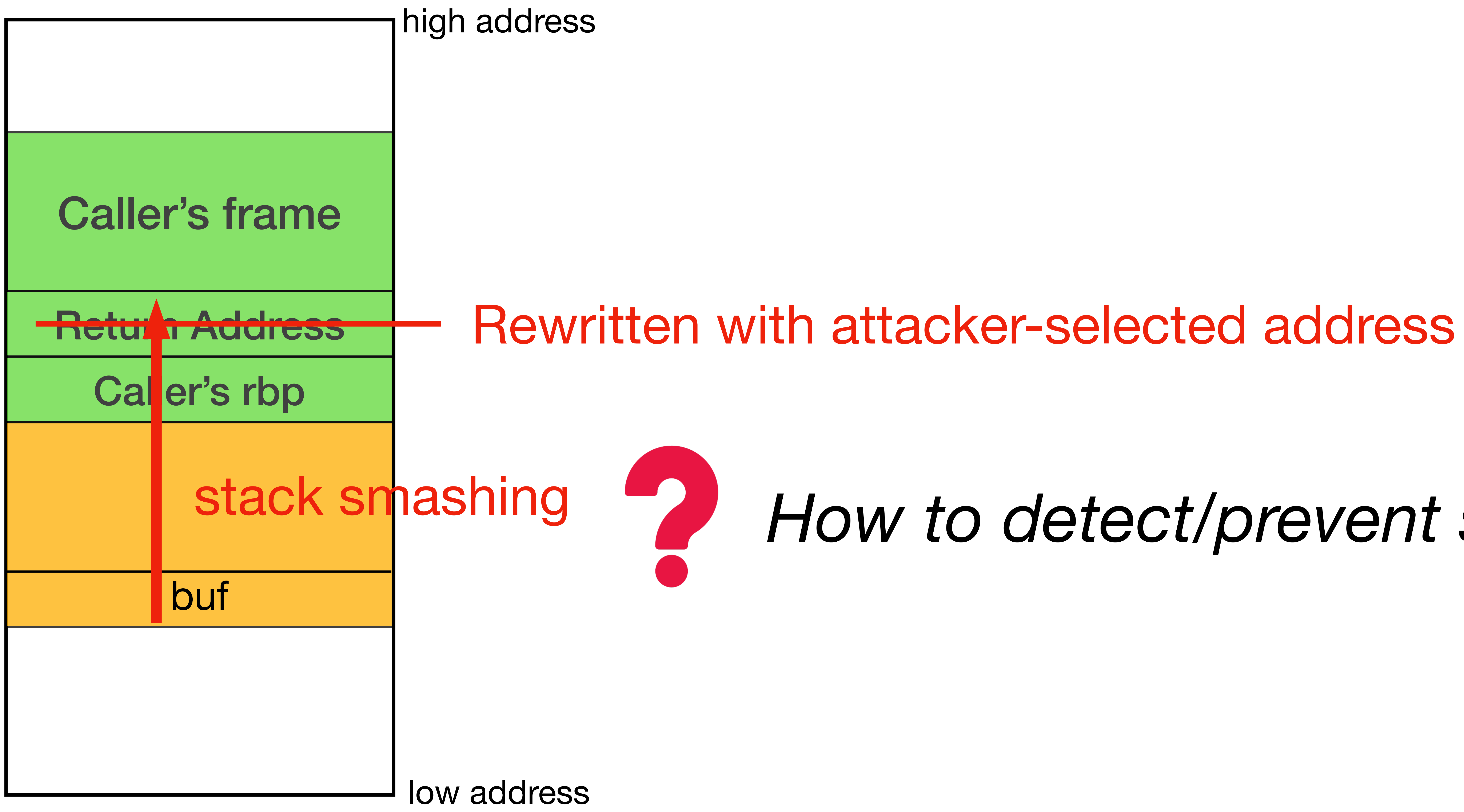
Shacham, Hovav, et al. "On the effectiveness of address-space randomization." ACM conference on Computer and Communications Security. 2004.

How Hard/Easy is the Attack?

- 16 bits of randomization for delta_mmap
  - Only need to try at most 2^16 = 65,536 times
- Experimental setup
  - Exploit executed on a 2.4 GHz Pentium 4 Linux machine
  - Against a PaX ASLR protected Linux running on Athlon 1.8 GHz machine
  - Running 10 trials
- Experimental results

Average	Max	Min
216 s	810 s	29 s

Return Address Corruption



How to detect/prevent stack smashing?

Stack Canaries

canary in the coal mine (also **canary in a coal mine**)

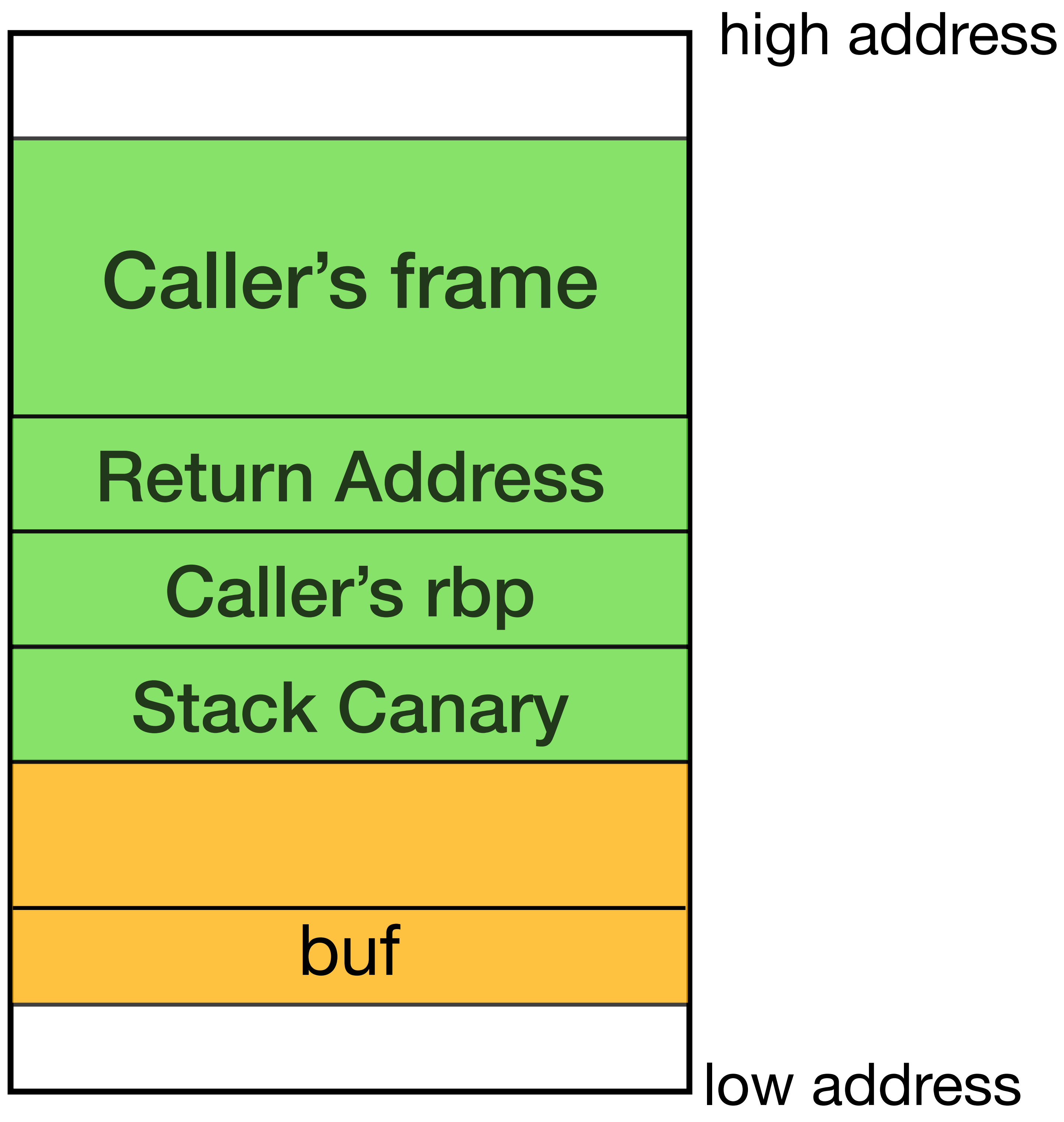
an early indicator of potential danger or failure: *native brook trout are very much the canary in the coal mine for the health of a stream.*

[with reference to the former practice of taking live canaries into coal mines to test for the presence of toxic gases; death of the canary would serve as an indication that such gases were present]

Stack Canaries

A random value put on the stack to detect stack buffer overflows

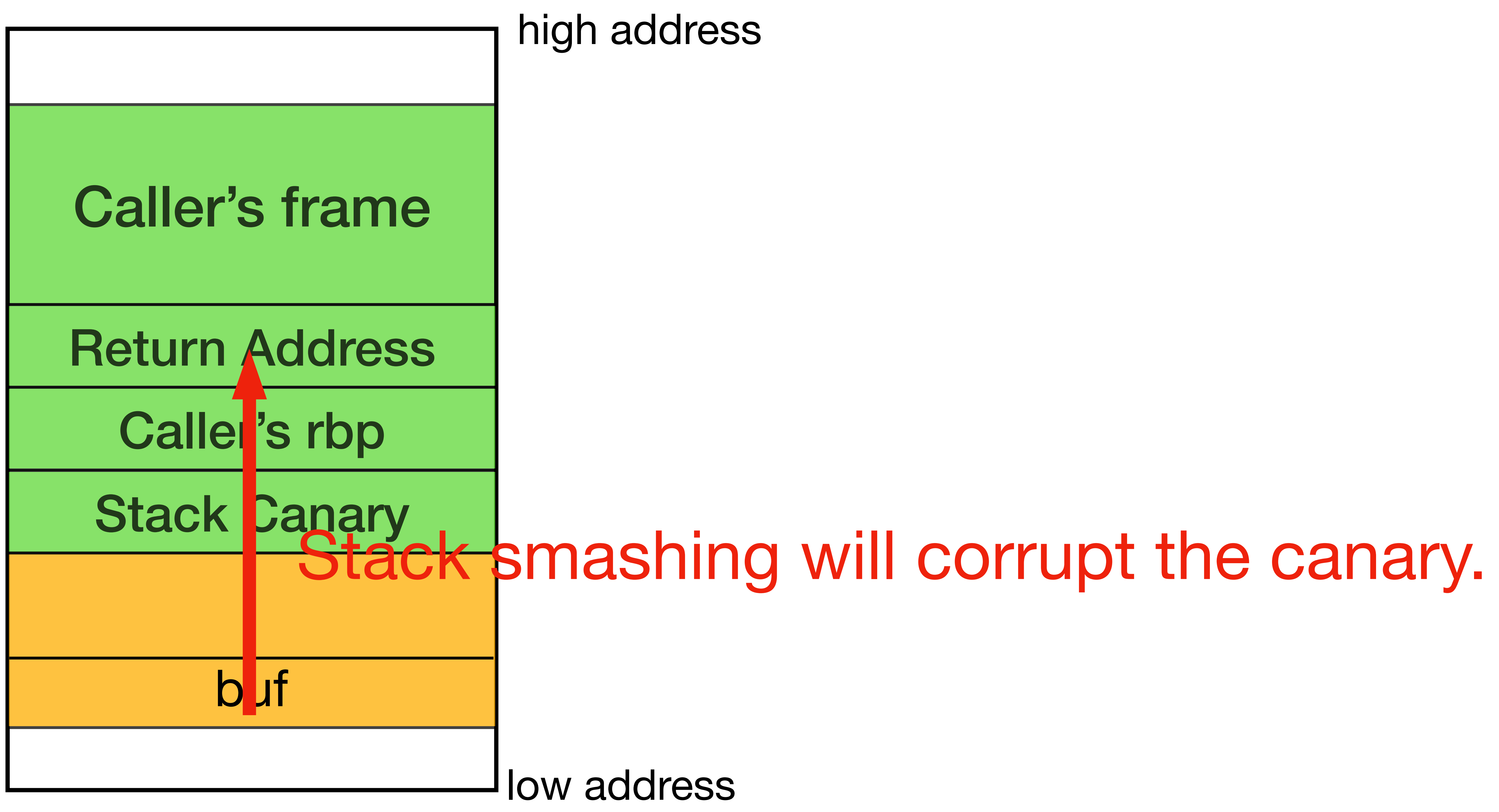
- Located close to the return address
- Function checking if the canary changed before using the return address



Stack Canaries

A random value put on the stack to detect stack buffer overflows

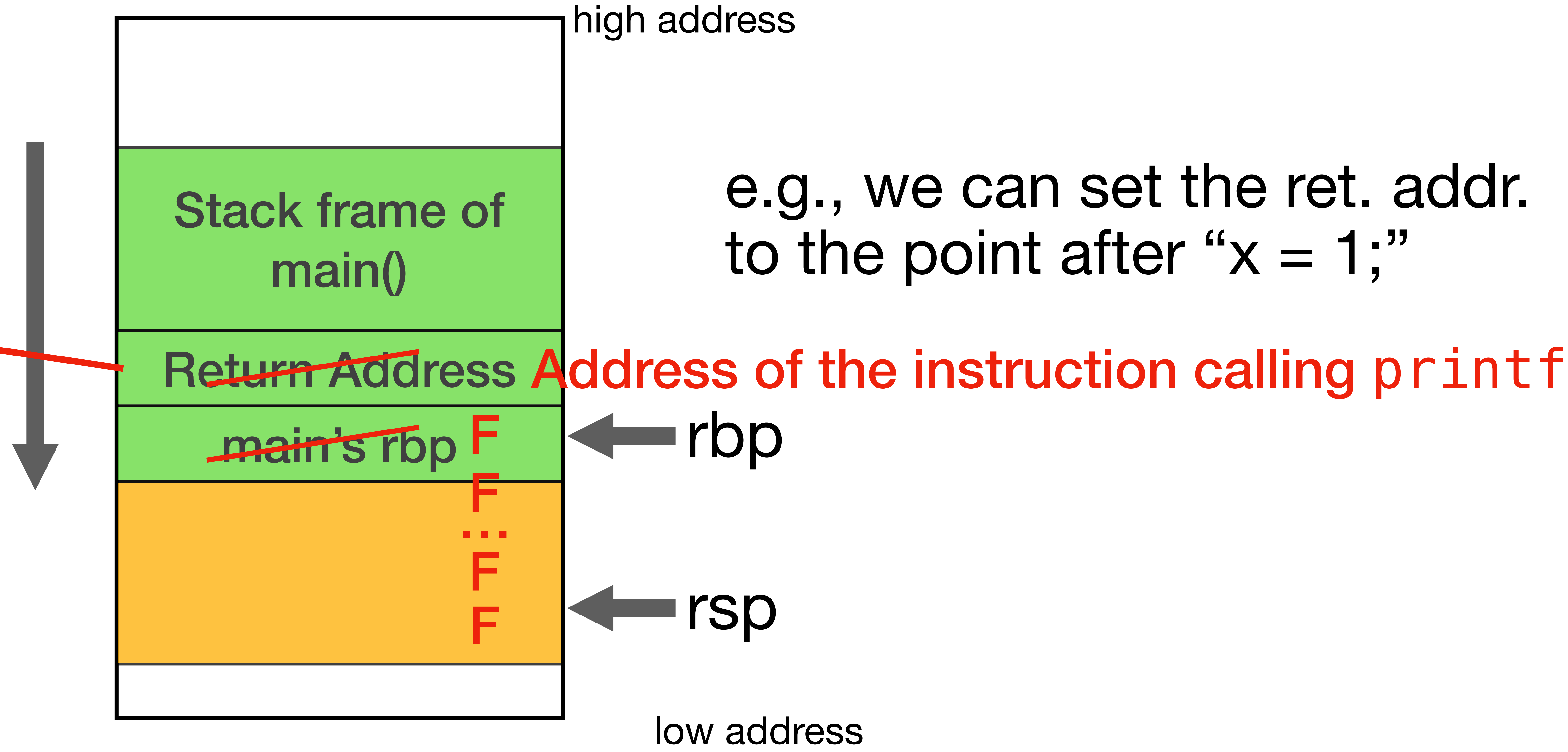
- Located close to the return address
- Function checking if the canary changed before using the return address



Smashing the Stack: Figure out a Nasty Input

```c
void foo(int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}

int main() {
      int x;
      x = 0;
      foo(1,2);
      x = 1;
      printf("%d\n",x);
      return 0;
}
```

./demo ?   What to input?



e.g., we can set the ret. addr. to the point after “x = 1;”

Smashing the Stack

```
void foo(int a, int b) {
     char buffer[12];
     gets(buffer);
     return;
}

int main() {
     int x;
     x = 0;
     foo(1,2);
     x = 1;
     printf("%d\n",x);
     return 0;
}
```

```
clang demo.c -o demo
```

```
Dump of assembler code for function foo:
   0x0000000000001150 <+0>:     push   %rbp
   0x0000000000001151 <+1>:     mov    %rsp,%rbp
   0x0000000000001154 <+4>:     sub    $0x20,%rsp
   0x0000000000001158 <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000000115b <+11>:    mov    %esi,-0x8(%rbp)
   0x000000000000115e <+14>:    lea    -0x14(%rbp),%rdi
   0x0000000000001162 <+18>:    mov    $0x0,%al
   0x0000000000001164 <+20>:    call   0x1040 <gets@plt>
   0x0000000000001169 <+25>:    add    $0x20,%rsp
   0x000000000000116d <+29>:    pop    %rbp
   0x000000000000116e <+30>:    ret
```

Function Frame with Stack Canaries

```
0x00001160 <+0>:  push   %rbp
0x00001161 <+1>:  mov    %rsp,%rbp
0x00001164 <+4>:  sub    $0x20,%rsp
0x00001168 <+8>:  mov    %fs:0x28,%rax
0x00001171 <+17>: mov    %rax,-0x8(%rbp)
0x00001175 <+21>: mov    %edi,-0x18(%rbp)
0x00001178 <+24>: mov    %esi,-0x1c(%rbp)
0x0000117b <+27>: xor    %eax,%eax
0x0000117d <+29>: lea    -0x14(%rbp),%rdi
0x00001181 <+33>: call   0x1050 <gets@plt>
0x00001186 <+38>: mov    %fs:0x28,%rax
0x0000118f <+47>: mov    -0x8(%rbp),%rcx
0x00001193 <+51>: cmp    %rcx,%rax
0x00001196 <+54>: jne    0x11a2 <foo+66>
0x0000119c <+60>: add    $0x20,%rsp
0x000011a0 <+64>: pop    %rbp
0x00000000000011a1 <+65>:ret
0x00000000000011a2 <+66>:call   0x1030 <__stack_chk_fail@plt>
```

AMD64/x86-64 ISA

- General-purpose registers
  - rax–rdx, rsi, rdi, r8–r15
  - rbp, rsp
- Program counter
  - rip
- Segment registers
  - cs, ss, ds, ss, es, fs, gs
- Control registers
  - cr0, cr2, cr3, cr4

Segment Registers on AMD64

- A legacy feature from x86-32 for segmentation-based addressing
    - CS (code segment)
    - DS (data segment)
    - SS (stack segment)
    - ES (extra segment)
    - FS, GS (general-purpose segment)
        - Usually used for Thread-local Storage (TLS)

Function Frame with A Stack Canary

```
0x00001160 <+0>:   push   %rbp
0x00001161 <+1>:   mov    %rsp,%rbp
0x00001164 <+4>:   sub    $0x20,%rsp
0x00001168 <+8>:   mov    %fs:0x28,%rax
0x00001171 <+17>:  mov    %rax,-0x8(%rbp)
0x00001175 <+21>:  mov    %edi,-0x18(%rbp)
0x00001178 <+24>:  mov    %esi,-0x1c(%rbp)
0x0000117b <+27>:  xor    %eax,%eax
0x0000117d <+29>:  lea    -0x14(%rbp),%rdi
0x00001181 <+33>:  call   0x1050 <gets@plt>
0x00001186 <+38>:  mov    %fs:0x28,%rax
0x0000118f <+47>:  mov    -0x8(%rbp),%rcx
0x00001193 <+51>:  cmp    %rcx,%rax
0x00001196 <+54>:  jne    0x11a2 <foo+66>
0x0000119c <+60>:  add    $0x20,%rsp
0x000011a0 <+64>:  pop    %rbp
0x00000000000011a1 <+65>:ret
0x00000000000011a2 <+66>:call   0x1030 <__stack_chk_fail@plt>
```

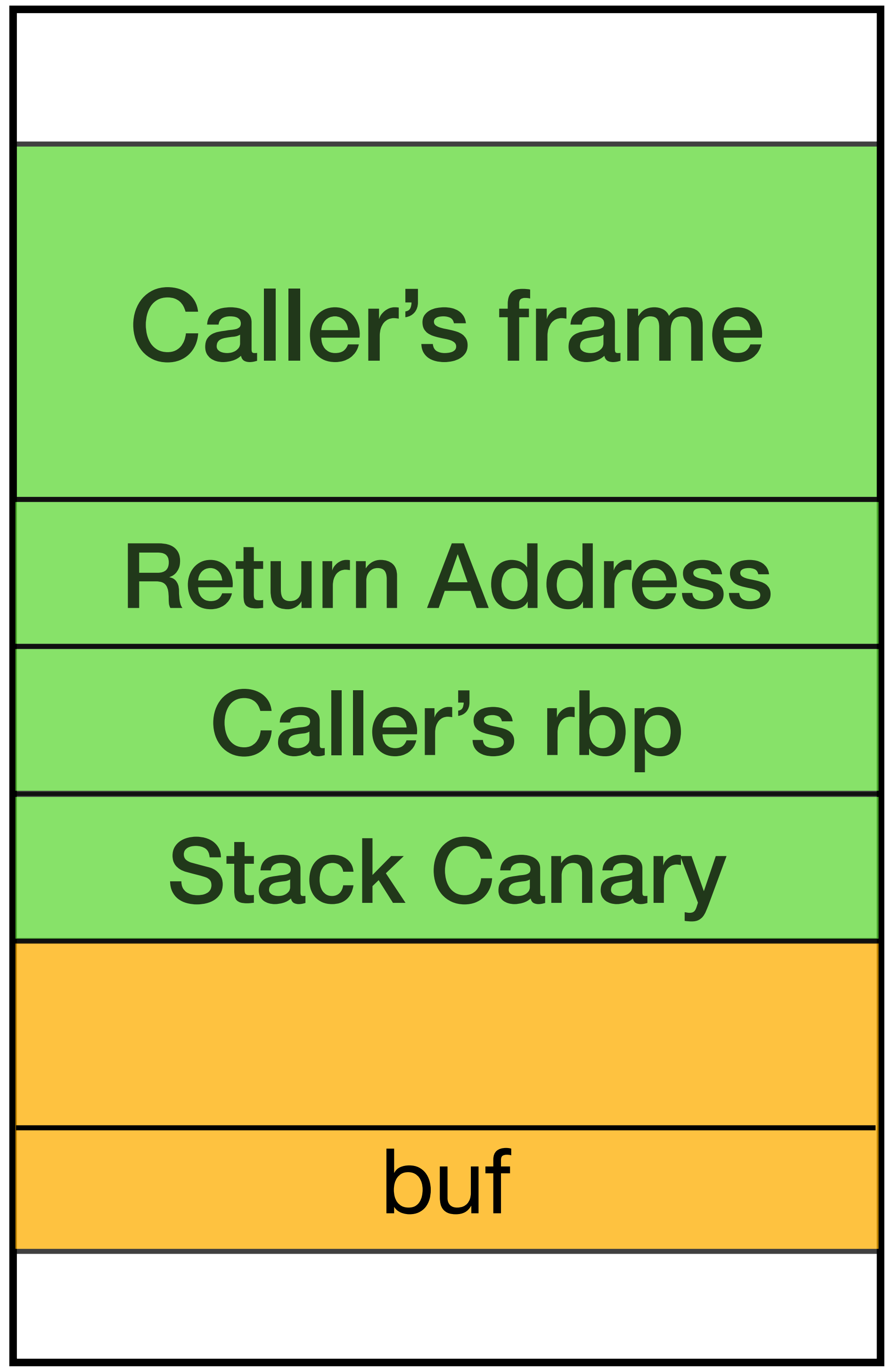
- Load canary from `%fs:0x28` onto the stack
  - `fs`: segment register
  - Canary was initialized during program initialization

- Load original canary to a register
- Load canary saved by this function to a register
- Compare the two register
  - If equal, keep normal execution
  - If unequal, jump to `__stack_chk_fail()`

Use Compilers to Add Stack Canaries

- clang
  - `-fstack-protector`
    - Add stack canaries for functions with a `char` array or calls to `alloca()`
  - `-fstack-protector-strong`
    - Add stack canaries for all functions with arrays, alloca, or taking addr of local vars
  - `-fstack-all`
    - Add stack canaries for all functions
- gcc has `-fstack-protector` on by default

Weaknesses of Stack Canaries



- Disclosure attacks
  - ‣ Buffer overread may leak the value of stack canaries.
    - Infamous buffer overread example: Heartbleed attack
    - Leak from the segment register
    - Leak from the stack
- Most effective in detecting *consecutive* stack overflows
  - ‣ Cannot detect arbitrary out-of-bound memory corruption
- Only protecting return addresses
  - ‣ Other security important data may still be corrupted
    - e.g., function pointers defined as local variables

Root Causes for the Weaknesses of Stack Canaries

- An extra layer of abstraction

"All problems in computer science can be solved by another level of indirection."

- David Wheeler

Software and Hardware Abstractions

Abstraction is the act of representing essential features without including the background details or explanations.

- Allow encapsulation of ideas without having to go into implementation details.
- Require an explicit definition on how to interoperate between layers

- In software, an API abstracts the underlying implementation by defining how a library can be used.
- In operating systems, the system call interface abstracts low level implementations.
- In hardware, an ISA abstracts the underlying implementation of the instructions into logic and state.

Root Causes for the Weaknesses of Stack Canaries

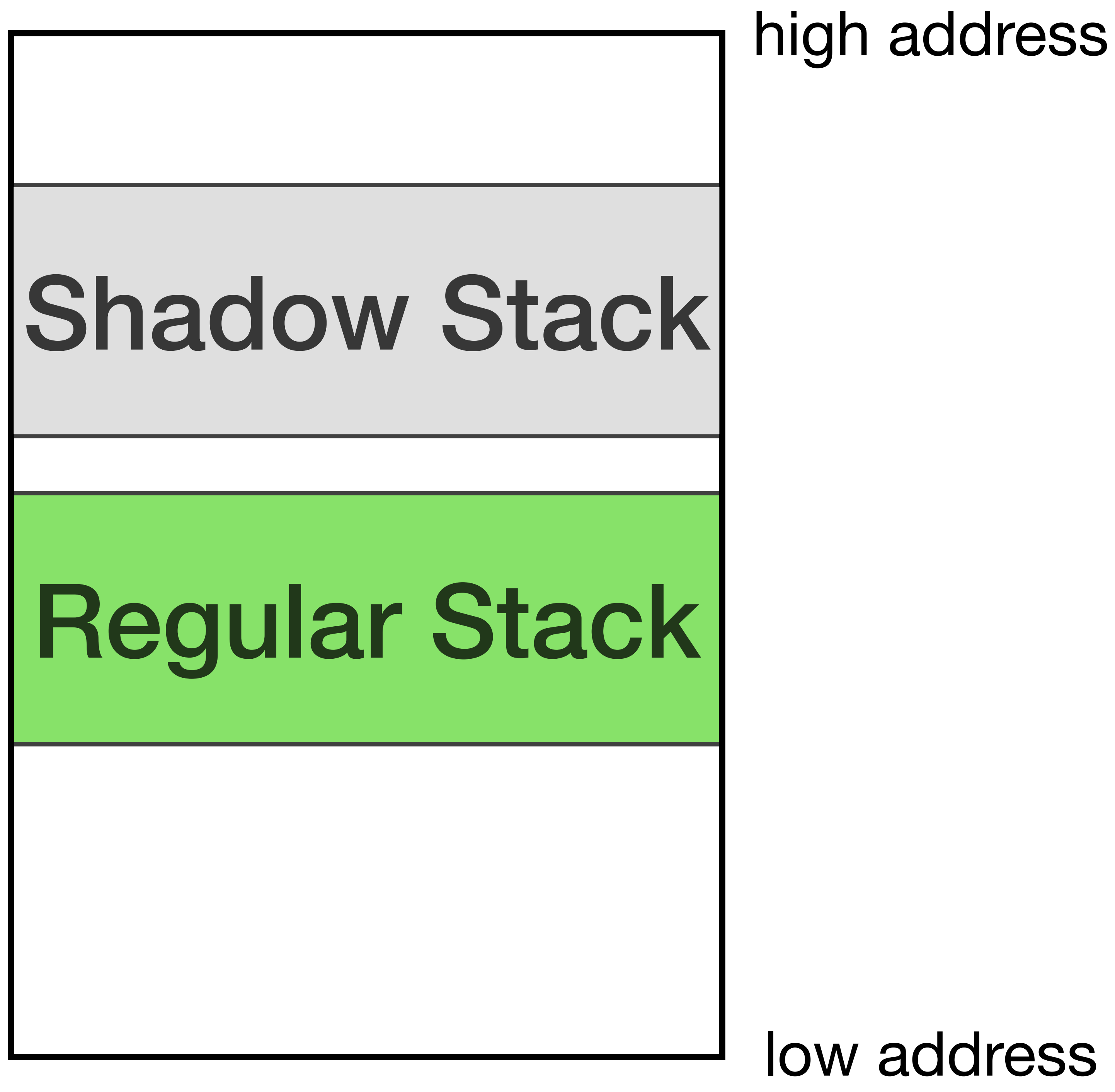
- An extra layer of abstraction
  - Which adds complexity to the system
- Key elements for security are located in the “danger zone”
  - Canaries on the stack are close to vulnerable buffers

Stronger Return Address Protection: Shadow Stacks

Shadow Stack for Return Address Integrity

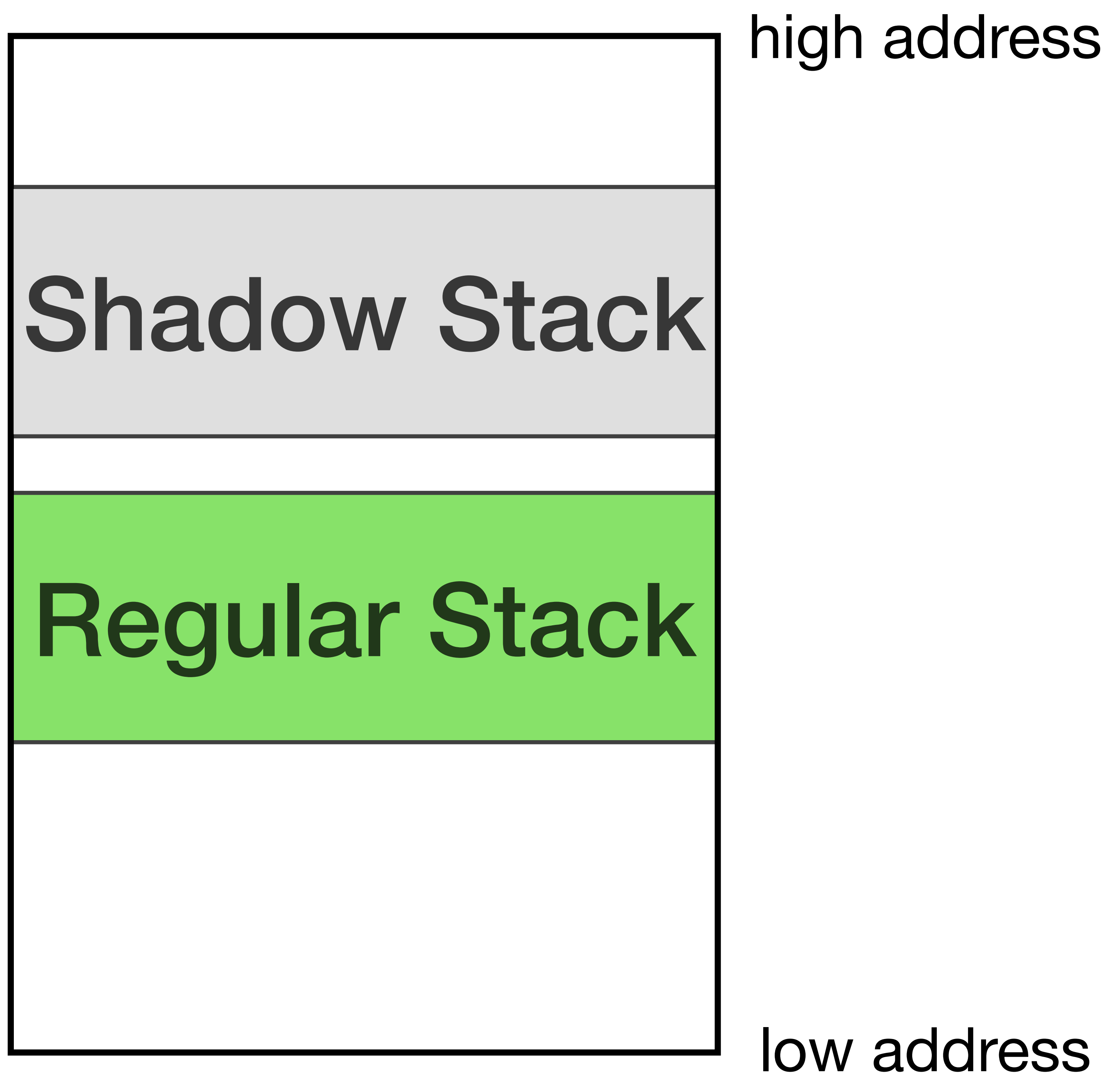
A separate stack dedicated to storing a copy of each return address

- A program can use the return address on the shadow stack
  - Checking the validity of the original return address
  - Directly using the copy on the shadow stack to return



Shadow Stack for Return Address Integrity

- Where to put the shadow stack?
- How to organize the shadow stack?
  - i.e., where exactly to store copies of return addresses?
- How to index the shadow stack?
  - i.e., how to find the return addresses on the shadow stack?



Two Types of Shadow Stack

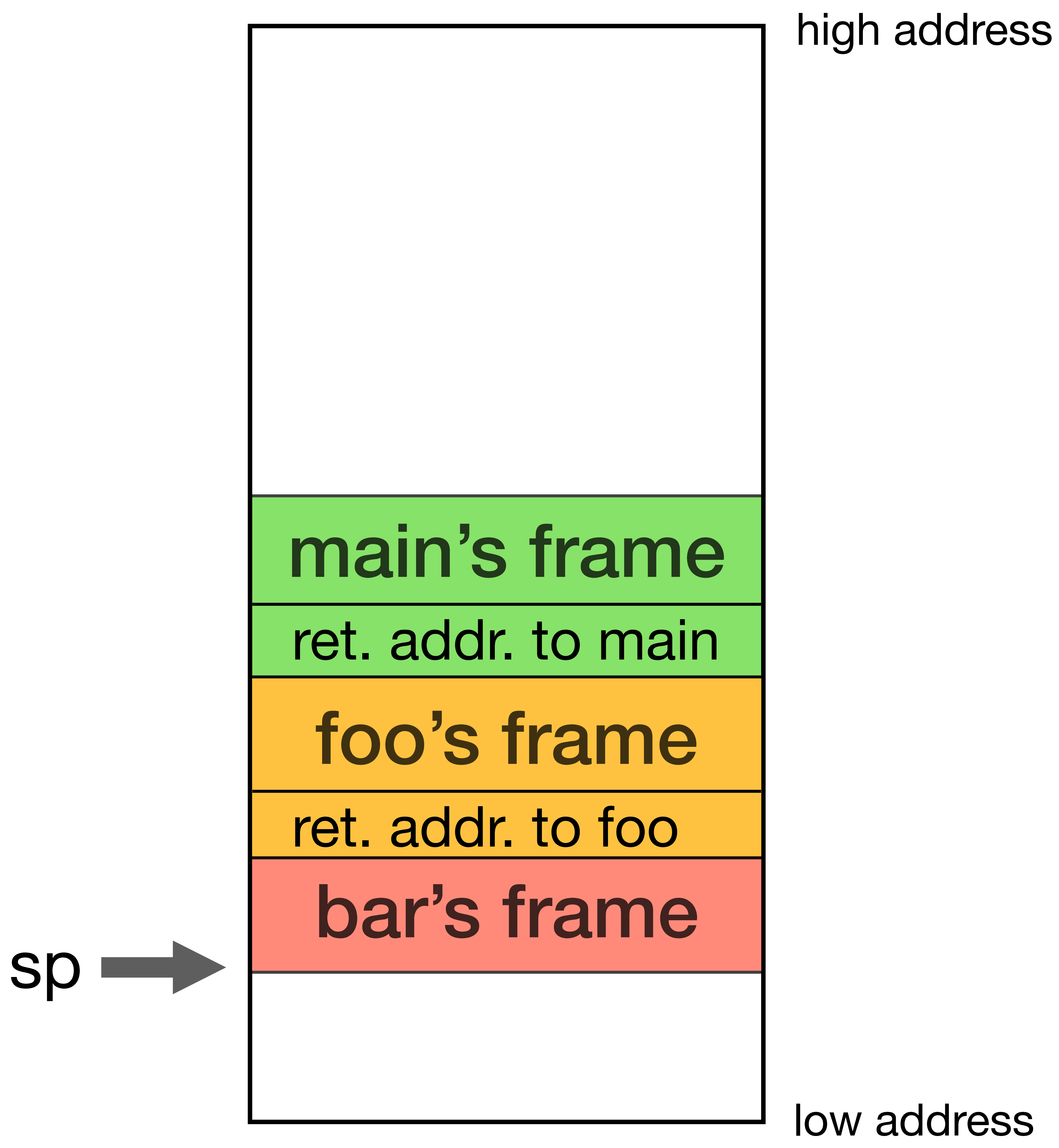
- Compact shadow stack
- Parallel shadow stacks

Compact Shadow Stack

- Where to put the shadow stack?
  - Usually also in the stack region, but could be in other memory regions.
- How to organize the shadow stack?
  - i.e., where exactly to store copies of return addresses a the shadow stack?
    - All return addresses are put together
- How to index the shadow stack?
  - Maintain a special shadow stack pointer (ssp) to the top of the shadow stack

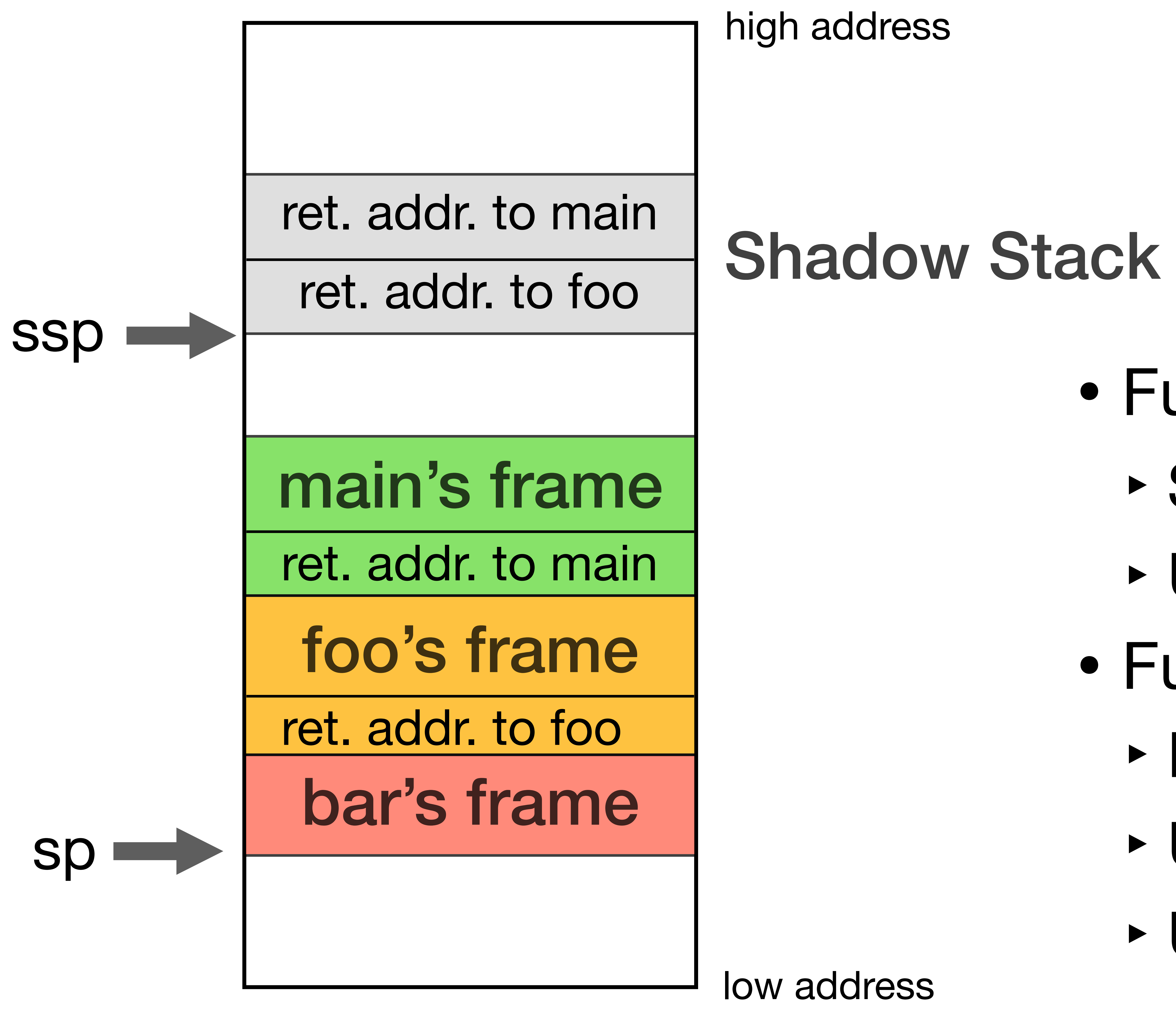
Compact Shadow Stack

e.g., `main()` calls `foo()` and `foo()` calls `bar()`



Compact Shadow Stack

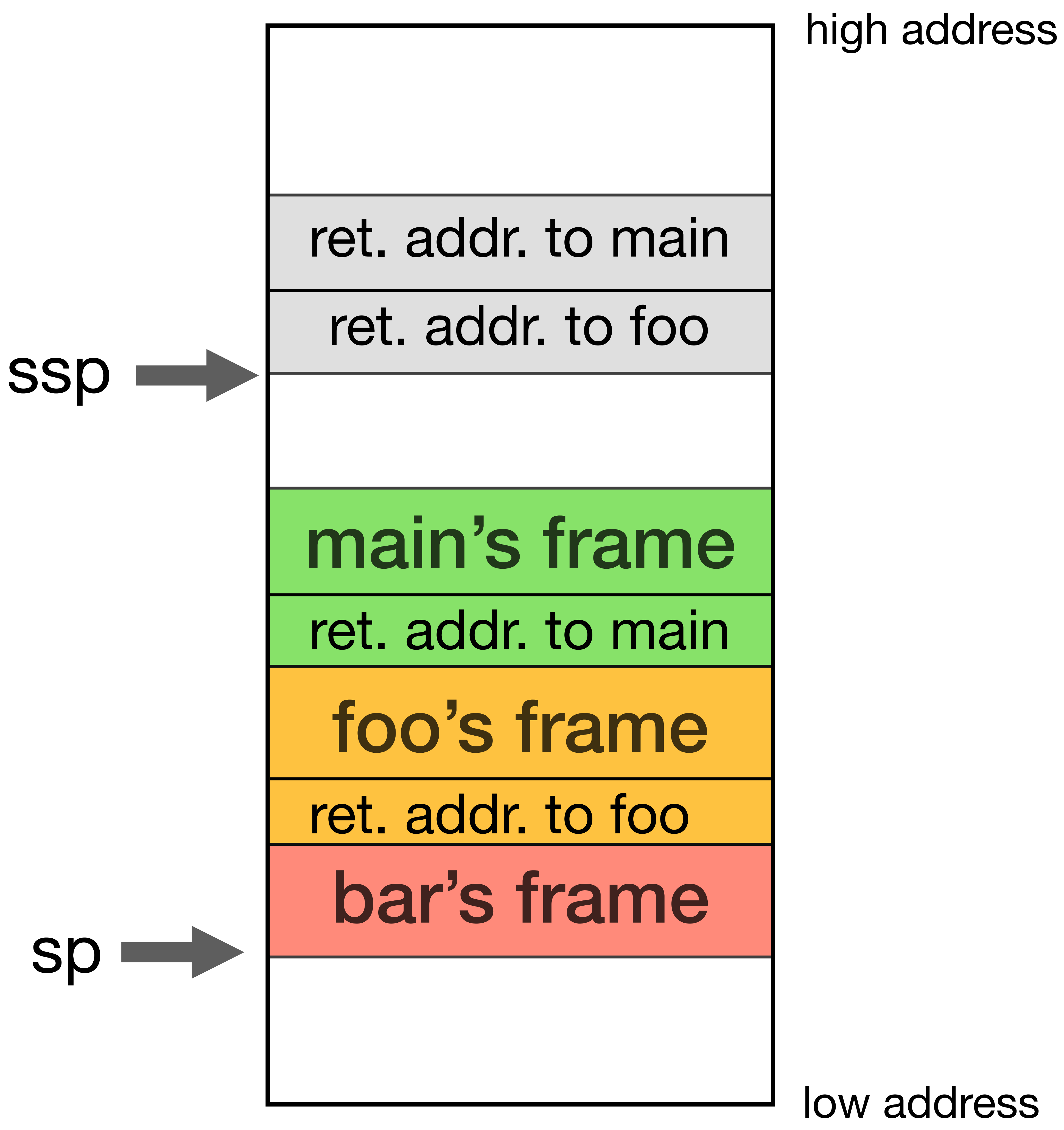
e.g., `main()` calls `foo()` and `foo()` calls `bar()`



- Function prologue
  - Save ret addr to shadow stack via ssp
  - Update ssp
- Function epilogue
  - Load ret addr from shadow stack via ssp
  - Use the saved ret addr
  - Update ssp

How to Maintain the Shadow Stack Pointer

e.g., `main()` calls `foo()` and `foo()` calls `bar()`



- In a global variable
  - Slow (takes two load instructions)
  - Multi-threaded issues
- Via segment register
  - `fs/gs` stores the base addr to index ssp
    - Medium speed (takes one load instruction)
- In a dedicated register
  - Good performance in general

LLVM's Implementation of Shadow Stack for AMD64

- Introduced in LLVM-7 for AMD64 and AArch64

```c
int foo() {
  return bar() + 1;
}
```

Compile with –02

```
push   %rax
callq  bar
add    $0x1,%eax
pop    %rcx
retq
```

LLVM's Implementation of Shadow Stack for AMD64

```c
int foo() {
  return bar() + 1;
}
```

Compile with

`-fsanitize=shadow-call-stack`

```
mov     (%rsp),%r10
xor     %r11,%r11
addq    $0x8,%gs:(%r11)
mov     %gs:(%r11),%r11
mov     %r10,%gs:(%r11)
push    %rax
callq   bar
add     $0x1,%eax
pop     %rcx
xor     %r11,%r11
mov     %gs:(%r11),%r10
mov     %gs:(%r10),%r10
subq    $0x8,%gs:(%r11)
cmp     %r10,(%rsp)
jne     trap
retq

trap:
ud2
```

LLVM's Implementation of Shadow Stack for AMD64

```
int foo() {
  return bar() + 1;
}
```

Compile with

`-fsanitize=shadow-call-stack`

```
mov     (%rsp),%r10          # Save ret addr to r10
xor     %r11,%r11            # Set r11 (gs's offset) to 0
addq    $0x8,%gs:(%r11)      # Increment ssp's offset by 8 bytes
mov     %gs:(%r11),%r11      # Load ssp's offset into r11
mov     %r10,%gs:(%r11)      # Save ret addr to shadow stack
push    %rax
callq   bar
add     $0x1,%eax
pop     %rcx
xor     %r11,%r11            # Clear r11
mov     %gs:(%r11),%r10      # Load ssp's offset into r10
mov     %gs:(%r10),%r10      # Load ret addr from shadow stack
subq    $0x8,%gs:(%r11)      # Decrement ssp's offset by 8 bytes
cmp     %r10,(%rsp)          # Compare two ret addr
jne     trap                 # If not equal, jump to trap()
retq

trap:
ud2                          # Invalid instruction
```

LLVM’s Implementation of Shadow Stack for AMD64

- Introduced in LLVM-7 for AMD64 and AArch64
- Support for AMD64 was removed since LLVM-9
  - High performance overhead
  - Security weakness: Subtle Time-Of-Check-Time-Of-Use (TOCTOU)

LLVM's Implementation of Shadow Stack for AMD64

```
mov     (%rsp),%r10         # Load ret addr into r10
xor     %r11,%r11           # Set r11 (gs's offset) to 0
addq    $0x8,%gs:(%r11)     # Increment ssp by 8 bytes
mov     %gs:(%r11),%r11     # Load ssp into r11
mov     %r10,%gs:(%r11)     # Save ret addr to shadow stack
push    %rax
callq   bar
add     $0x1,%eax
pop     %rcx
xor     %r11,%r11           # Clear r11
mov     %gs:(%r11),%r10     # Load ssp into r10
mov     %gs:(%r10),%r10     # Load ret addr from shadow stack
subq    $0x8,%gs:(%r11)     # Decrement ssp by 8 bytes
cmp     %r10,(%rsp)         # Compare two ret addr
jne     trap                # If not equal, jump to trap()
retq

trap:
ud2                         # Invalid instruction
```

What if the return address was corrupted by another thread before it was loaded and saved onto the shadow stack?

What if the return address was corrupted after it passed the check but before it was popped from the regular stack?

LLVM's Implementation of Shadow Stack for AArch64

- Directly use the return address on the shadow stack instead of checking validity
- AArch64 uses a link register (`lr/x30`) for return address
  - Always storing return address into `lr` before storing it onto the stack
  - Always loading return address from stack to `lr` before using the return address
  - Preventing TOCTOU

LLVM's Implementation of Shadow Stack for AArch64

```
int foo() {
  return bar() + 1;
}
```

```
stp   x29, x30, [sp, #-16]!  # Store frame pointer (x29) and lr onto the stack
mov   x29, sp                # Set the frame pointer for the current frame
bl    bar                    # Call bar()
add   w0, w0, #1             # Add 1 to the return value of bar()
ldp   x29, x30, [sp], #16    # Restore frame pointer and load ret addr into  lr
ret
```

With `-fsanitize=shadow-call-stack`

```
str   x30, [x18], #8      # Store lr to memory pointed by ssp (x18)
                          # and increment ssp
stp   x29, x30, [sp, #-16]!
mov   x29, sp
bl    bar
add   w0, w0, #1
ldp   x29, x30, [sp], #16
ldr   x30, [x18, #-8]!    # Decrement ssp and load lr from memory
                          # pointed by ssp
ret
```

Compact Shadow Stack

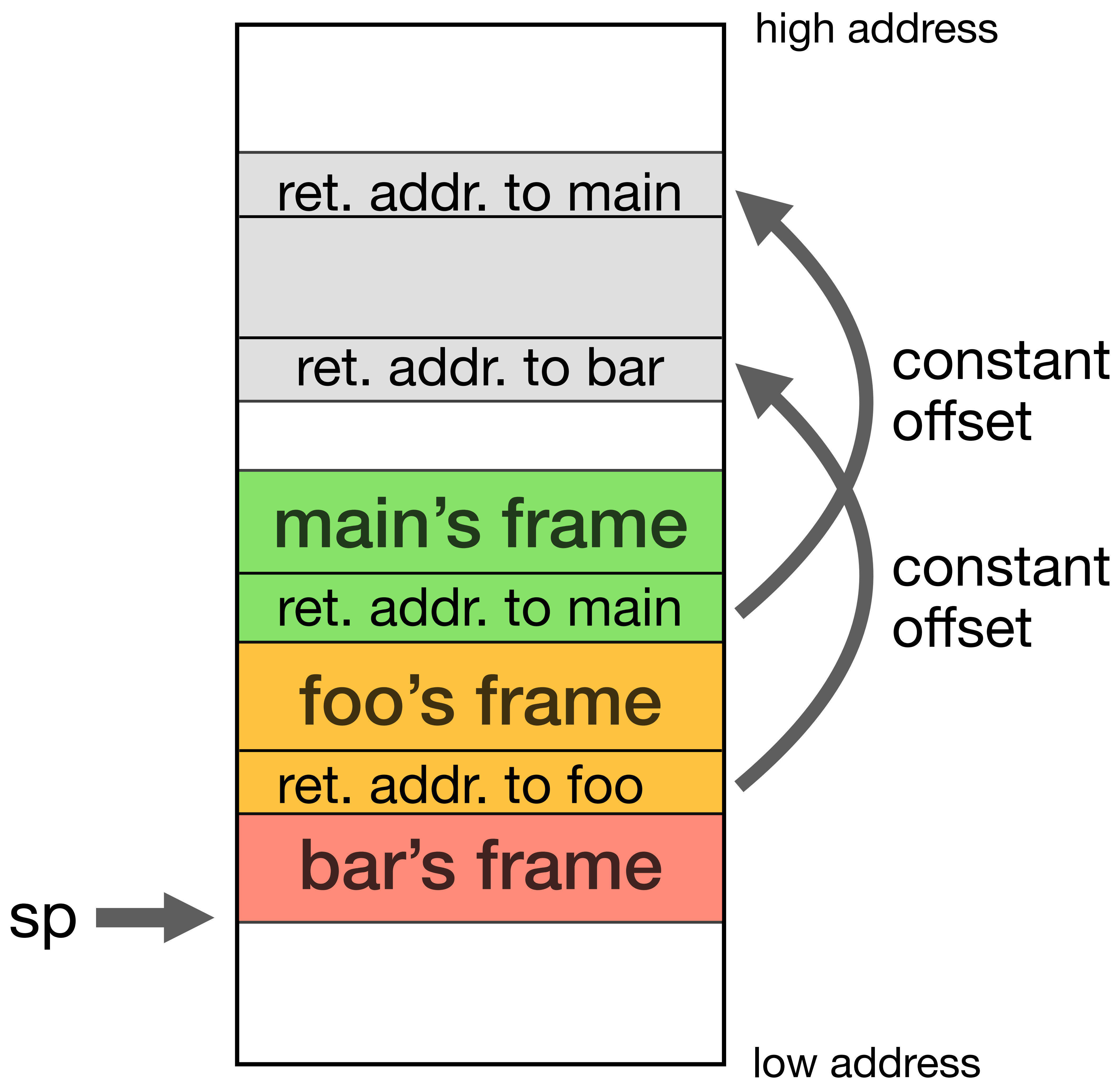
- Where to put the shadow stack?
  - Usually also in the stack region, but could be in other memory regions.
- How to organize the shadow stack?
  - i.e., where exactly to store copies of return addresses a the shadow stack?
    - All return addresses are put together
- How to index the shadow stack?
  - Maintain a special shadow stack pointer (ssp) to the top of the shadow stack

Parallel Shadow Stack

- Where to put the shadow stack?
  - Usually also in the stack region, but could be in other memory regions.
- How to organize the shadow stack?
  - i.e., where exactly to store copies of return addresses a the shadow stack?
    - Return addresses on the shadow stack are scattered to match the layout of the regular stack.
- How to index the shadow stack?
  - Use a constant offset from the regular stack pointer

Parallel Shadow Stack

e.g., `main()` calls `foo()` and `foo()` calls `bar()`



- Shadow stack is of the same size as the regular stack.
- Constant offset between each ret addr on the regular and shadow stack
- Use sp + offset to index the ret addr on the shadow stack
  - Prologue: Save ret addr to sp + offset
  - Epilogue: Load ret addr from sp + offset

Parallel Shadow Stack

- Example of shadow stack updating during function prologue for ARM

```
mov.w ip , #0xe00000   # Move the offset to a reigster
str.w lr , [sp, ip ]   # Store lr to (sp + offset)
```

Strengths and Weaknesses of Parallel Shadow Stack

- Strengths
  - Fast
- Weaknesses
  - Memory consumption overhead is high.
  - Hard-coded offset is a security hazard (easily accessible to adversaries).
  - Compatibility issues for multi-threaded programs
    - Constrained address space layout

Reduce the Weaknesses of Parallel Shadow Stack

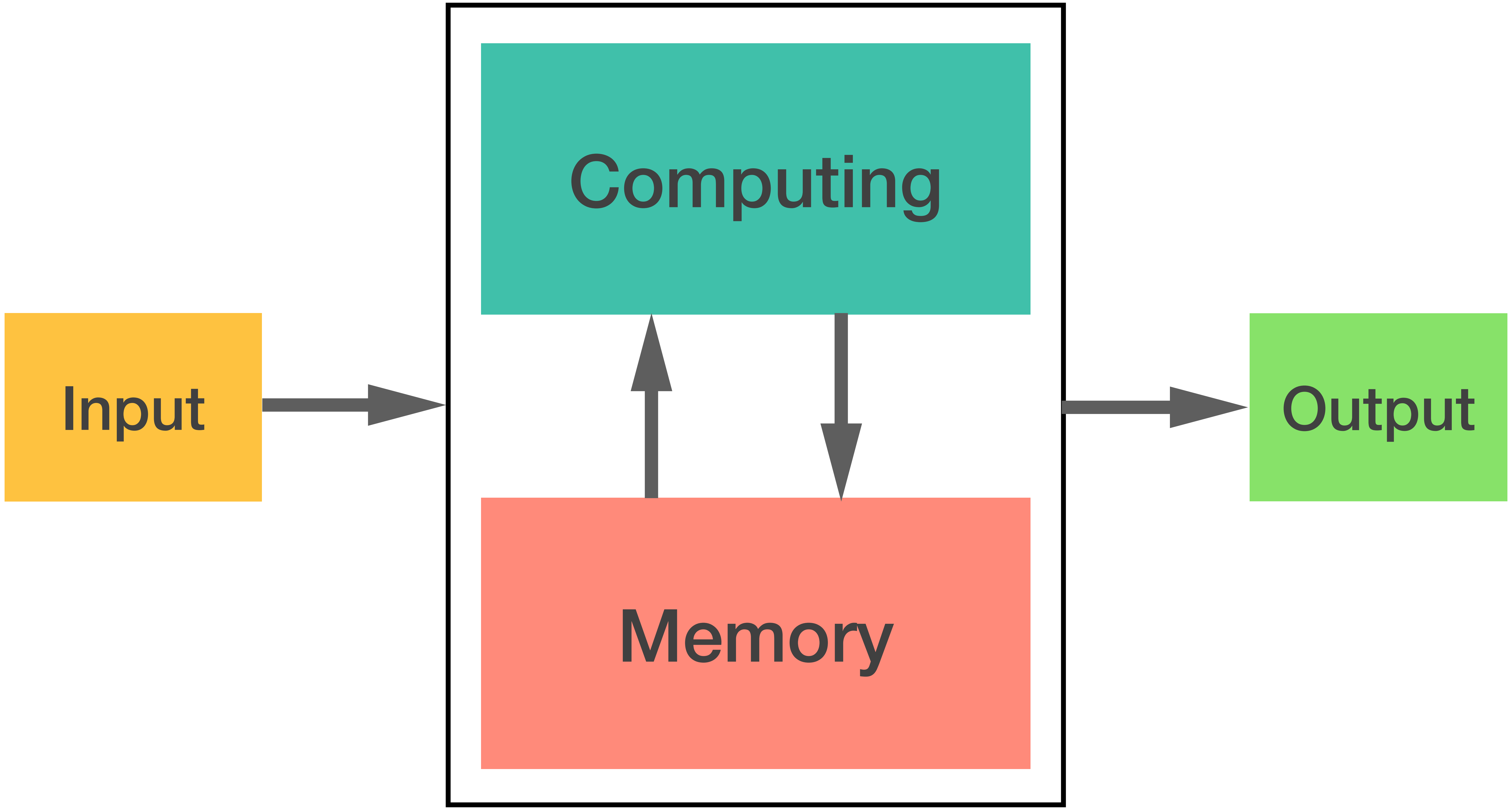
- Memory consumption overhead is high.
  - Use smaller offset
    - But less secure
- Hard-coded offset is a security hazard (easily accessible to adversaries).
  - Encode the offset in a dedicated register
    - Can have different offsets for each thread
    - Higher performance penalty
- Compatibility issues for multi-threaded programs.
  - Smaller offset also mitigates this issue.

Weaknesses of Shadow Stack for Return Addresses

- Increased complexity due to additional abstraction
  - Increasing the complexity of the protected software
  - Performance and memory overhead
  - New security risks
    - e.g. race conditions due to x86's use of stack for return addresses
- Limited scope of protection
- Integrity of shadow stack itself
  - Shadow stack protects return addresses, who protects shadow stack?

What are essential to a programming language?

Architecture of Modern Computers



Essential Components of a Programming Language

- Data types
  - int, char, boolean, etc.
- Operators
  - arithmetic, move, comparison, etc.
- Control flow
  - absolute/conditional transfer, loop

```
#include <stdio.h>

void foo() {
    printf("Hello from foo\n”);
}

void bar() {
    printf("Hello from bar\n");
}

int main(int argc, char *argv[]) {
    if (argc > 2) {
        foo();
    } else {
        bar();
    }

    return 0;
}
```

Essential Components of a Programming Language

- Data types
  - int, char, boolean, etc.
- Operatiors
  - arithmetic, move, comparison, etc.
- Control flow
  - absolute/conditional transfer, loop

Control Flow

- Data types
  - int, char, boolean, etc.
- Operators
  - arithmetic, move, comparison, etc.
- Control flow
  - absolute/conditional transfer, loop

Attacks often start with misusing data types, and then misdirecting the control flow to launch malicious computations.

Control-flow Integrity is Critical to Software Security.

Control Flow

- Data types
  - int, char, boolean, etc.
- Operators
  - arithmetic, move, comparison, etc.
- Control flow
  - absolute/conditional transfer, loop
    - Function returns, calls, indirect jumps, goto statements (dangerous and rare today)

Control Flow

- Function returns: Backward control flow
- Function calls: Forward control flow
  - Direct calls: Call by function name
    - Hardcoded function address; usually cannot change
  - Indirect calls: Call by function pointers
    - Dynamically-computed function address

Forward Control Flow

```
#include <stdio.h>

void foo() {
    printf("Hello from foo\n”);
}

void bar() {
    printf("Hello from bar\n");
}

int main(int argc, char *argv[]) {
    void (*fn_p)() = foo;
    if (argc > 2) {
        fn_p = bar;
    }

    fn_p();

    return 0;
}
```

Which function to call is computed at run-time.

Overflowing Heap Critical User Data

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];               /* vulnerable input buffer */
    void (*process)(char *);    /* pointer to function */
} chunk_t;

void showlen(char *buf) {
    int len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk_t *next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Overflow the buffer on the heap to set the function pointer to an arbitrary address.

Control-flow Graph (CFG)

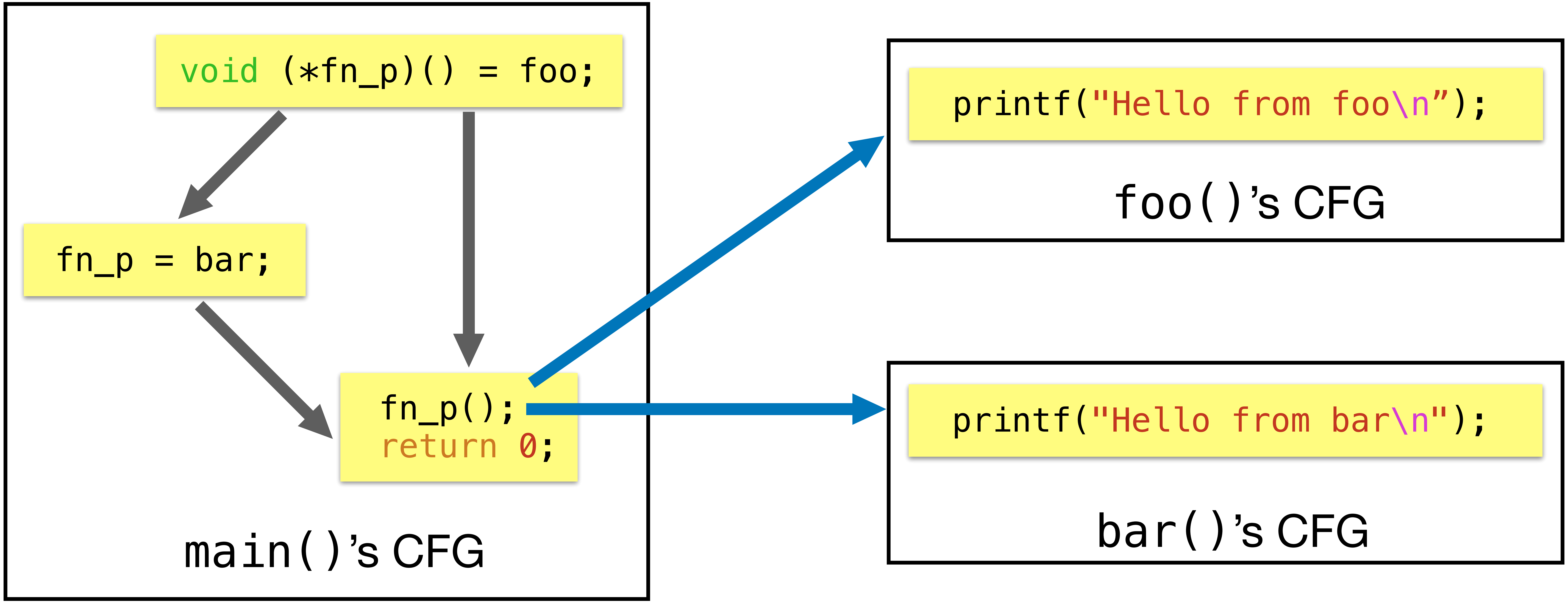
A program representation using graph notations.

- Node: A sequence of instructions without control flow transfers
- Edge: Control flow transfer between nodes
- Usually represent one function, but can also represent a whole program
- Critical to analyzing programs

Forward Control Flow

```
#include <stdio.h>

void foo() {
    printf("Hello from foo\n”);
}

void bar() {
    printf("Hello from bar\n");
}

int main(int argc, char *argv[]) {
    void (*fn_p)() = foo;
    if (argc > 2) {
        fn_p = bar;
    }

    fn_p();

    return 0;
}
```

Example of Control-flow Graph (CFG)



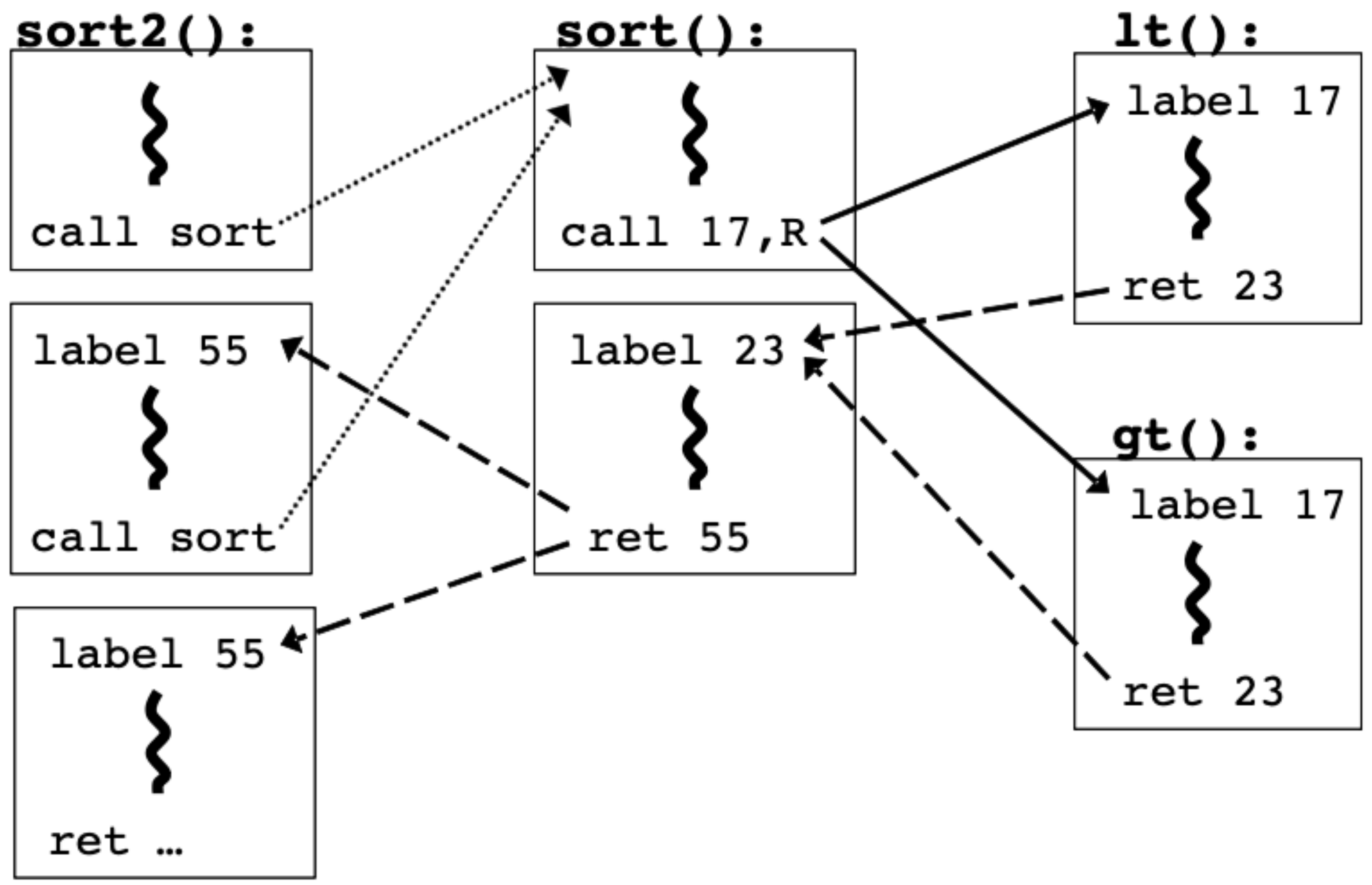
How to Enforce Control-flow Integrity

- Compute a CFG
- For indirect control flow transfers, compute their target destinations
  - Mostly via compiler or binary rewriting, but possible at run-time
- Before an indirect transfer, check the validity of the destination
- Two CFI policies:
  - Label-based and type-based

Label-based CFI

- Assign and insert a label (ID) before each indirect transfer destination
- Before executing an indirect transfer, check the destination's label
  - Similar to using stack canaries / shadow stacks

```
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 );
}
```



Direct forward transfer

Indirect forward transfer

Backward transfer

Example of Label-based CFI

Bytes (opcodes)	x86 assembly code	Comment
FF E1	jmp ecx	; a computed jump instruction

CFI instrumentation

```
81 39 78 56 34 12    cmp   [ecx], 12345678h   ; compare data at destination
75 13                jne   error_label        ; if not ID value, then fail
8D 49 04             lea   ecx, [ecx+4]       ; skip ID data at destination
FF E1                jmp   ecx                ; jump to destination code
```

Bytes (opcodes)	x86 assembly code	Comment
8B 44 24 04	mov eax, [esp+4]	; first instruction
		; of destination code
...		

CFI instrumentation

```
78 56 34 12          DD 12345678h         ; label ID, as data
8B 44 24 04          mov   eax, [esp+4]   ; destination instruction
...
```

A subtle issue:

- Provide a potential ROP gadget incrementing `ecx` by 4

Issues/Weaknesses of Label-based CFI

- Collision of labels with existing data/code
  - Generally harder on x64 than RISC arch such as ARM
    - e.g., in ARM, "`mov r0, r0`" (`0x4600`) can serve as a unique label
- Very challenging to compute a precise CFG
  - A practical implementation:
    - Use the same label for all function entrance
    - Use the same label for all instructions following a call

Type-based CFI

- Use function signature (types of return value and args) as the target identifier
- Each signature is assigned a unique type ID
  - e.g., “`void foo()`” and “`int bar()`” will have different type IDs.
- Before an indirect call, check destination’s type ID against a predetermined ID

Type-based CFI Implementation: Option 1

- Maintain a mapping table (e.g. hash table)
  - Key: function address
  - Value: function’s type ID
  - The table is stored in a read-only memory region.
- Compiler generates a type ID using the function pointer’s signature/type.
- For an indirect call
  - Query the mapping table to get the current function pointer’s type ID
  - Compare this ID with the pregenerated type ID

Type-based CFI Implementation: Option 2

- Compiler computes the type ID of each function
- Insert the ID as a label at the entrance of function
- Do a label-based CFI check

Weaknesses of Type-based CFI

- Needs source code or compiler IR; cannot do binary rewriting
- Only works for indirection function calls but not returns
- Allows indirect calls to a group of functions
  - What if we have “`int myFunc(const char *str)`”, which has the same type ID as “`int system(const char *command)`”?

Effectiveness of CFI

- Average target set size (also known as size of average equivalent classes)
    - Compute the number of possible call targets
- Average Indirect-target Reduction (AIR)
    - How much a CFI mechanism reduces the number of valid targets
- Open problem: *how effective are these metrics?*
    - Not all functions are equal!

Use Compilers to Enforce CFI

- clang
  - Mostly type-based
  - Multiple options (7 in Clang-22) to control protection granularity
  - Not on by default
- gcc
  - Currently requires special Intel hardware; no pure software support

Fundamental Weaknesses of CFI

- Performance and code size overhead
- Cannot be 100% accurate
- Backward CFI (i.e., protecting return addresses) is critical.
- An extra lay of complexity
  - May need support from OS
- Limited protection scope
  - Does not prevent data-only attacks
  - Needs W^X/DEP