Imperial College London

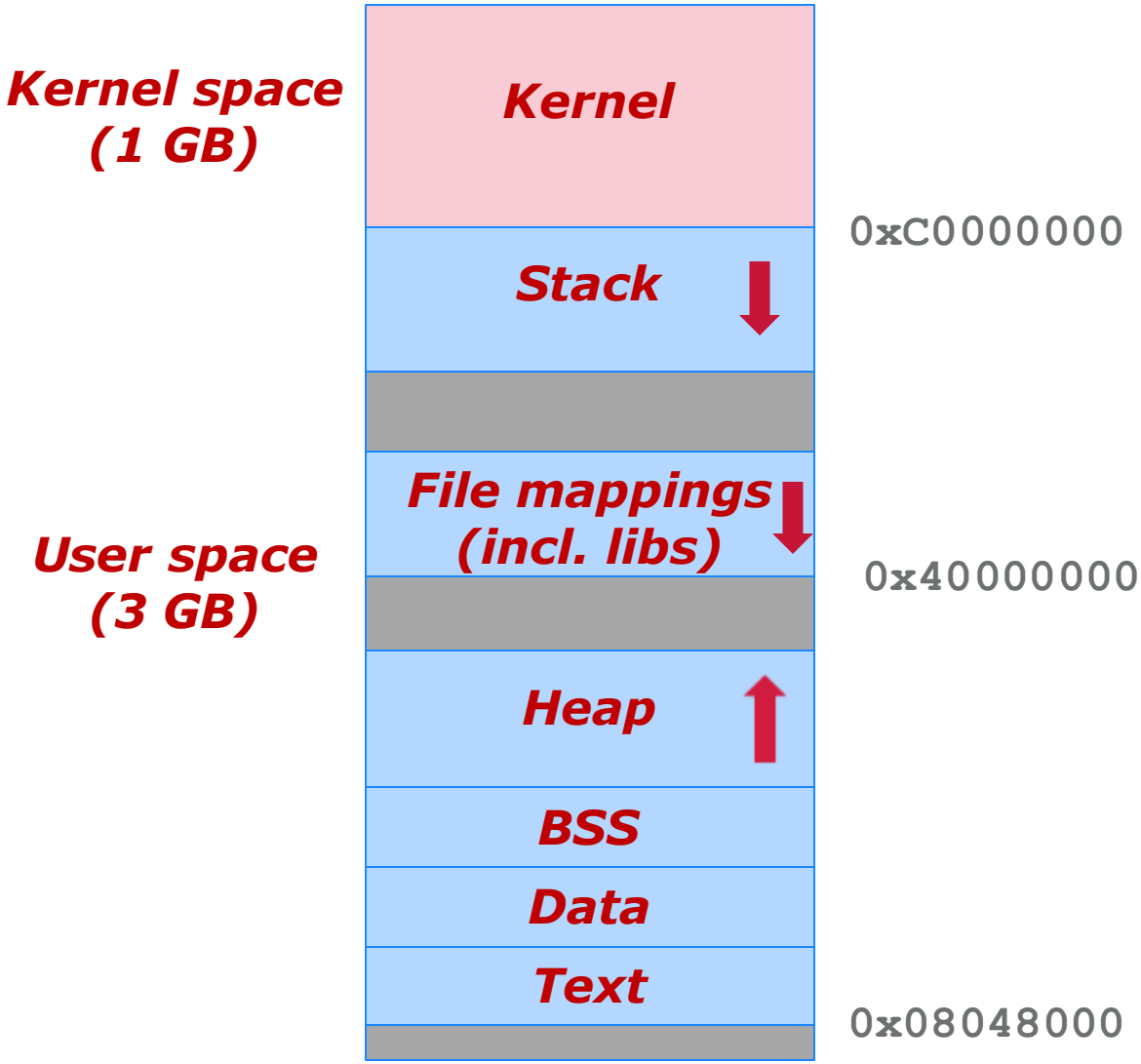
Introduction to Program Analysis for Security

Cristian Cadar
c.cadar@imperial.ac.uk

Software Reliability Course
Autumn 2016

Imperial College London

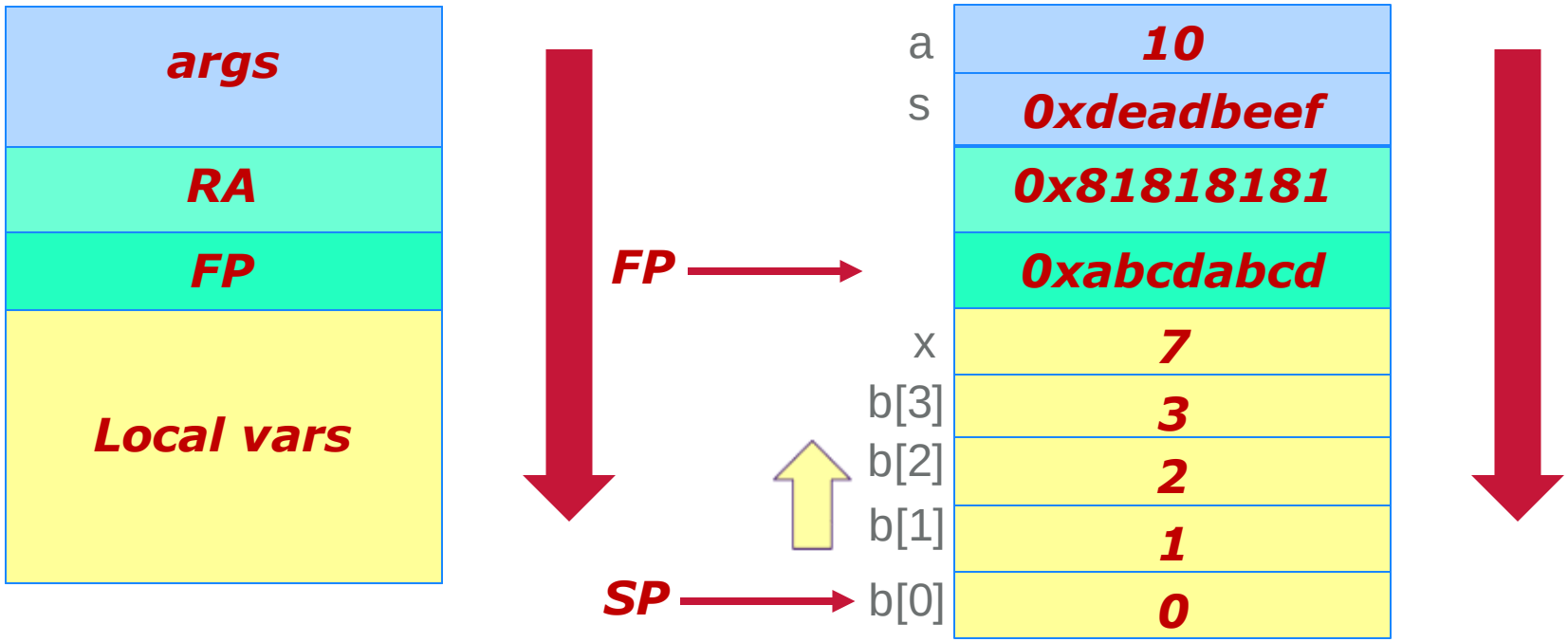
Address Space Layout [Linux x86, no randomization]



Imperial College London

Stack Frame Layout [Linux x86]

```
foo(int a, char* s) {
    int x = 7, b[4] = {0,1,2,3};
    ...
```

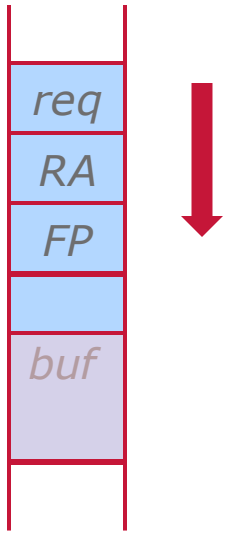


Basic Stack Exploit

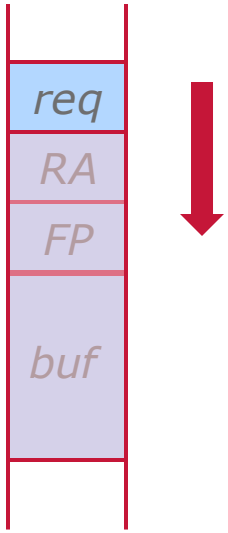
My insecure web server

```
void read_req(char *req) {
    char buf[100];
    strcpy(buf, req);
    do-something(buf);
}
```

strlen(req) < 100



strlen(req) = 108

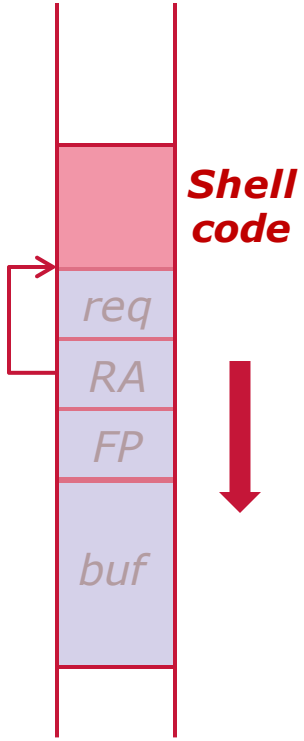


Can crash app (DoS)
Can divert control flow

Basic Stack Exploit

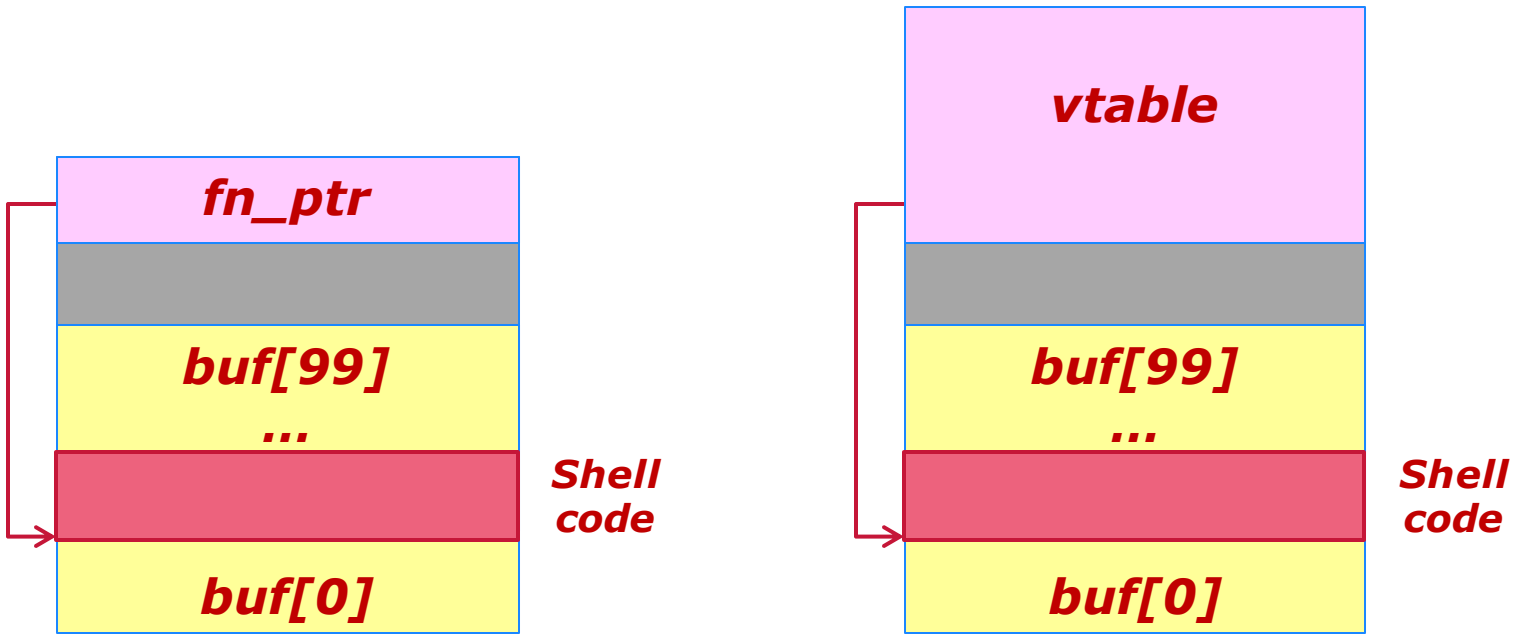
My insecure web server

```
void read_req(char *req) {
    char buf[100];
    strcpy(buf, req);
    do-something(buf);
}
```



Heap-Based Attacks

- E.g., overwrite function pointers in the heap



Use After Free Attacks

- Freeing memory can create dangling pointers
- These can be used to leak memory contents and corrupt memory

```
int *p;

int foo() {
  int a[4] = {1, 2, 3, 4};
  p = a;
  return a[0];
}

int bar() {
  int a[4] = {42, 41, 40, 39};
  return a[0];
}
```

```
int main() {
  foo();
  bar();
  printf("*p = %d\n", *p);
}
```

Use After Free Attacks

- Freeing memory can create dangling pointers
- These can be used to leak memory contents and corrupt memory

```
int main() {
  int *p = malloc(1000);
  *p = 100;
  free(p);

  int *q = malloc(1000);
  *q = 42;

  printf("*p = %d\n", *p);
  return 0;
}
```

Other types of attacks?

- Overwrite longjmp buffers
- Overwrite GOT and/or PLT
- Format string vulnerabilities
- etc.

- Sometimes a one-byte overflow can be enough!

```
void read_req(int input) {
    char auth, buf[128];
    auth = check_credentials();
    buf[input] = 1;
    if (auth)
        enter_privileged_mode();
}
```

Out-of-bounds reads

- Out-of-bounds writes can clearly be exploited by attackers
- What about out-of-bounds reads?

1) OOB reads can leak private data

```
int main(int argc, char** argv) {
    ...
    printf(argv[1]);
    ...
}
```

Out-of-bounds reads

- Out-of-bounds writes can clearly be exploited by attackers
- What about out-of-bounds reads?

1) OOB reads can leak private data
   - Heartbleed bug exploited OOB reads
   - An attacker could trick OpenSSL into allocating a 64KB buffer and leak the contents of the victim's memory, 64KB at a time

Out-of-bounds reads

- Out-of-bounds writes can clearly be exploited by attackers
- What about out-of-bounds reads?

2) OOB reads can be used to divert control flow

```
void foo(int user_input) {
    fun_ptrs valid_targets[100];
    p[user_input]();
    ...
}
```

Types of Attacks

1) Code corruption attacks
   - Try to modify existing code in memory
   - Code segment is marked as read-only, supported by all modern CPUs
2) Control-flow hijacking
   - Corrupt code pointers/code data such as the return address, function pointers, etc.
3) Non-control-data attacks
   - Corrupt any security-critical data other than code data
4) Leak confidential memory
   - Also to find out critical info to conduct attacks 1) - 3)

Defending directly against buffer overflows

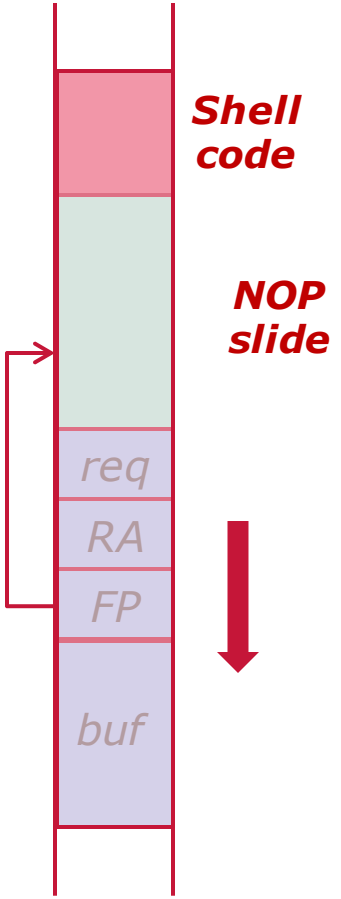
- The root cause of many attacks are buffer overflows!
- Can we prevent them from happening in the first place?

- First line of defence: off-line program analysis tools
  - Static analysis, symbolic execution, model checking, etc.

- Can we prevent them at run-time?
  - One possibility is to compile code with a Safe C compiler
  - But as we've seen, the runtime overhead is often prohibitive

Partial defenses

If we can't eliminate buffer overflows completely, are there any effective partial solutions?

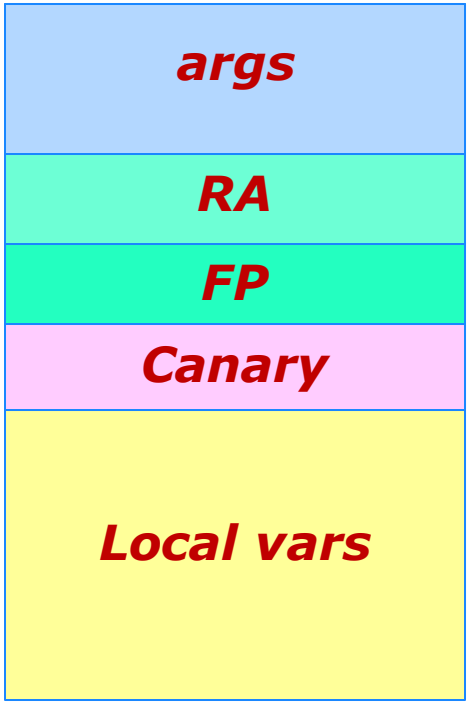


My insecure web server

```
void read_req(char *req) {
    char buf[100];
    strcpy(buf, req);
    do-something(buf);
}
```

Defense: Stack Canaries

- Add *canaries* to stack frames and verify their integrity prior to function return
- Need to recompile code, but no source modifications are needed and binary-compatible with existing libs
- Small performance overhead: push canary value on function prologue, check integrity on epilogue
- Implemented as patches to gcc, Clang, MS compiler: enabled by default



StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks
Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole,Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang, USENIX Security 1998

Defense: Stack Canaries

Three types of canary values:

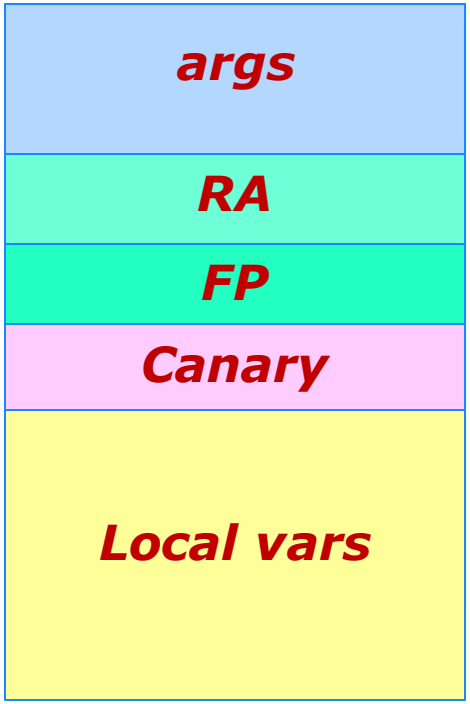
1) Terminator canaries: typically includes

`NULL(0x00)  CR(0x0d),`

`LF(0x0a) EOF(0xff)`

Example attack:

- memcpy!



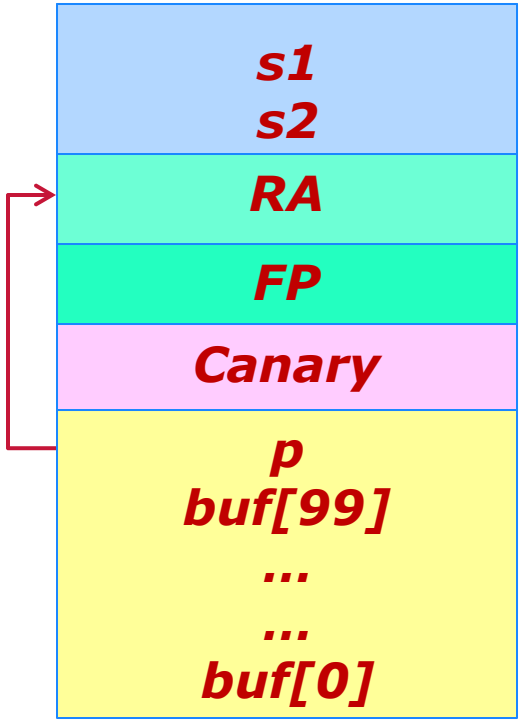
Defense: Stack Canaries

Three types of canary values:

2) Random canaries: value chosen at load time

Example attack:

```
void read_req(char *s1, *s2) {
    char *p;
    char buf[100];
    strcpy(buf, s1);
    strncpy(p, s2, 8);
}
```



Overwrite p to point to address of RA!

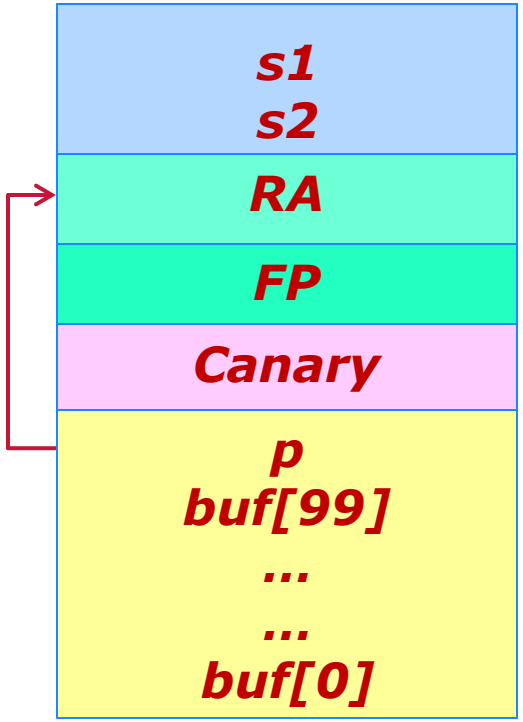
Defense: Stack Canaries

Three types of canary values:

3) Random XOR canaries:

Canary = Rand-val XOR RA

```
void read_req(char *s1, *s2) {
    char *p;
    char buf[100];
    strcpy(buf, s1);
    strncpy(p, s2, 8);
}
```



If RA was hacked, the check will fail

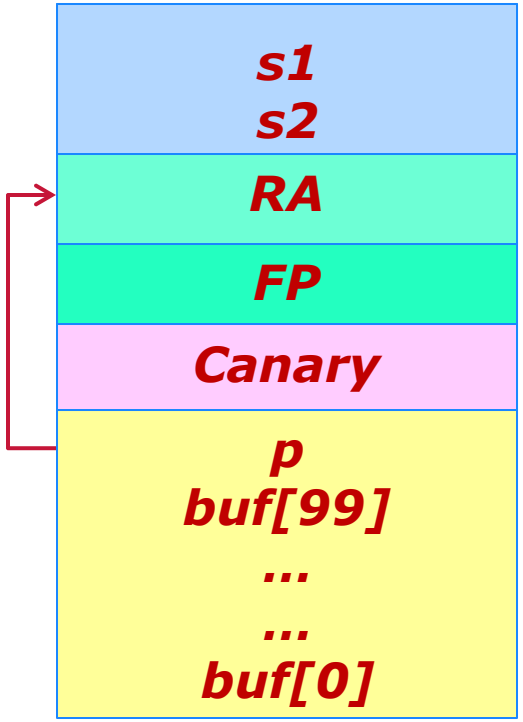
Defense: Stack Canaries

Attacks against random (xor) canaries:

- Overwrite global variable holding random value?

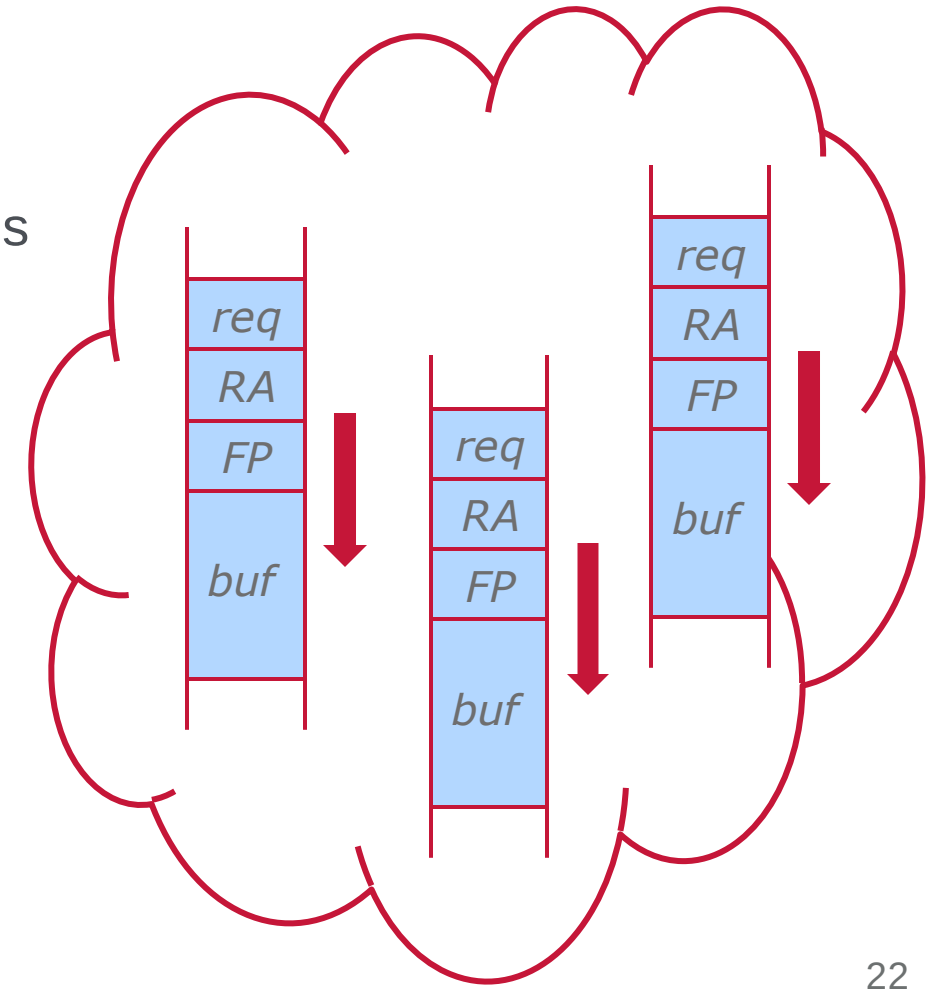
  → allocate canary table in separate R/O page

- Canary value may still leak



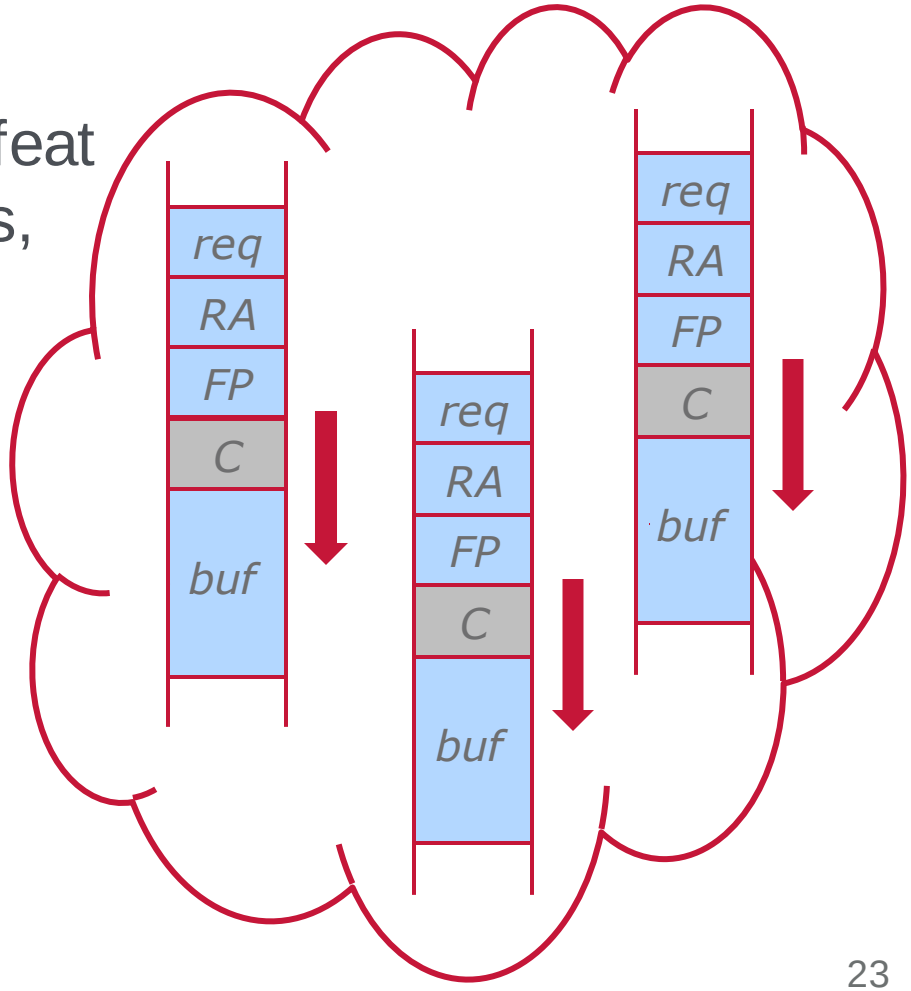
Leaking Canary Value: Nginx attack (2013)

- Nginx: popular HTTP server and reverse proxy
- Thread pool architecture
  - When a thread dies, a new one is spawned



Leaking Canary Value: Nginx attack (2013)

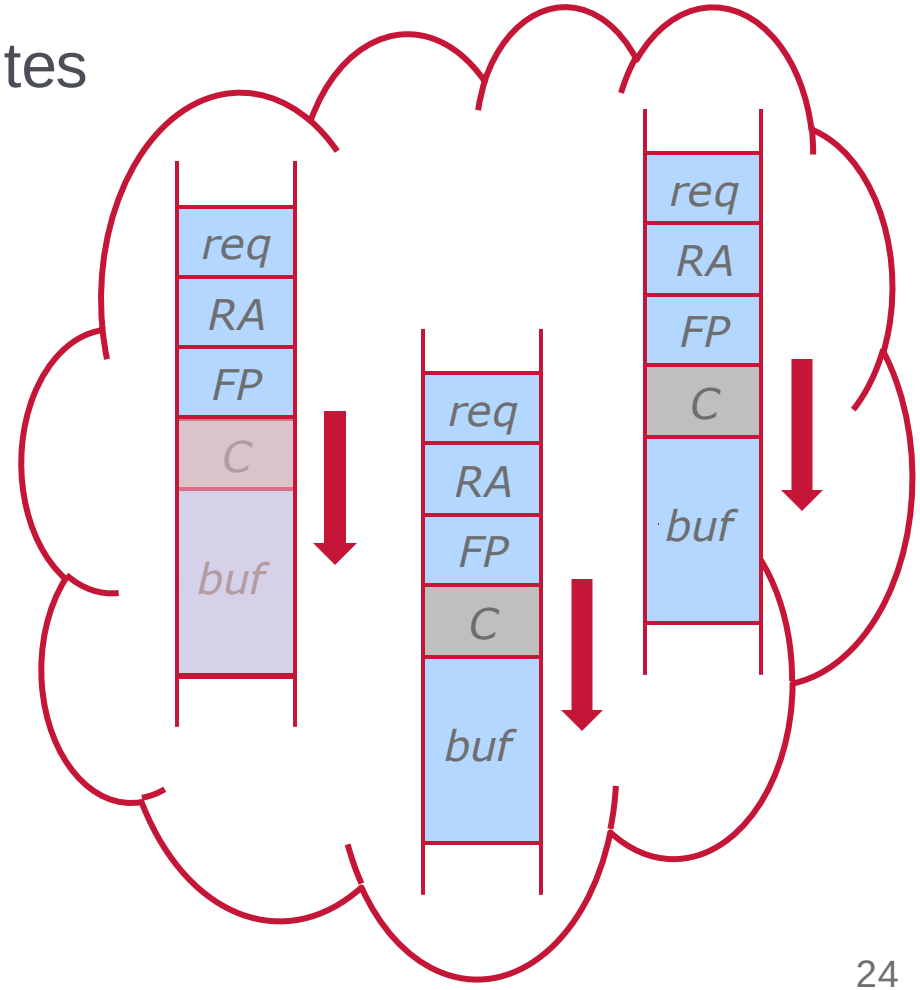
- Nginx: popular HTTP server and reverse proxy
- Sophisticated attack had to defeat various protection mechanisms, including stack canaries



Leaking Canary Value: Nginx attack (2013)

Attack against canaries:

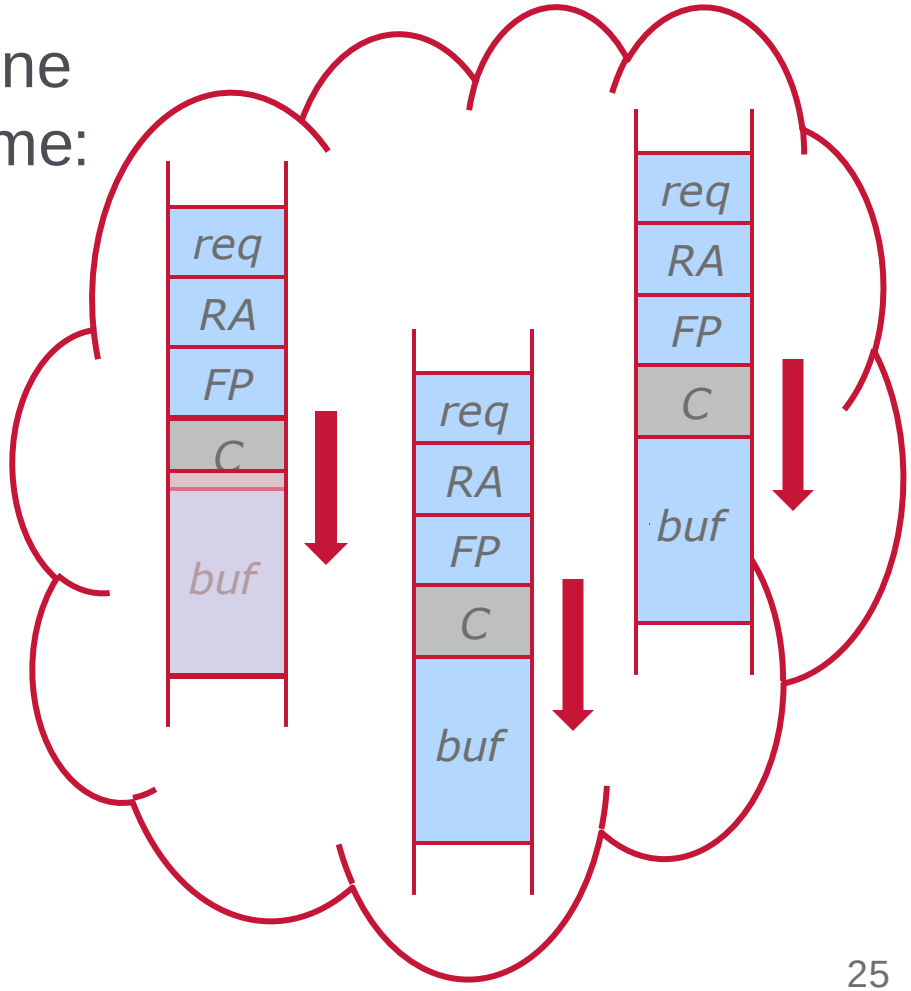
- Send request that only overwrites the stack canary:
  - Response: canary value correct!
- How many tries?



Leaking Canary Value: Nginx attack (2013)

Attack against canaries:

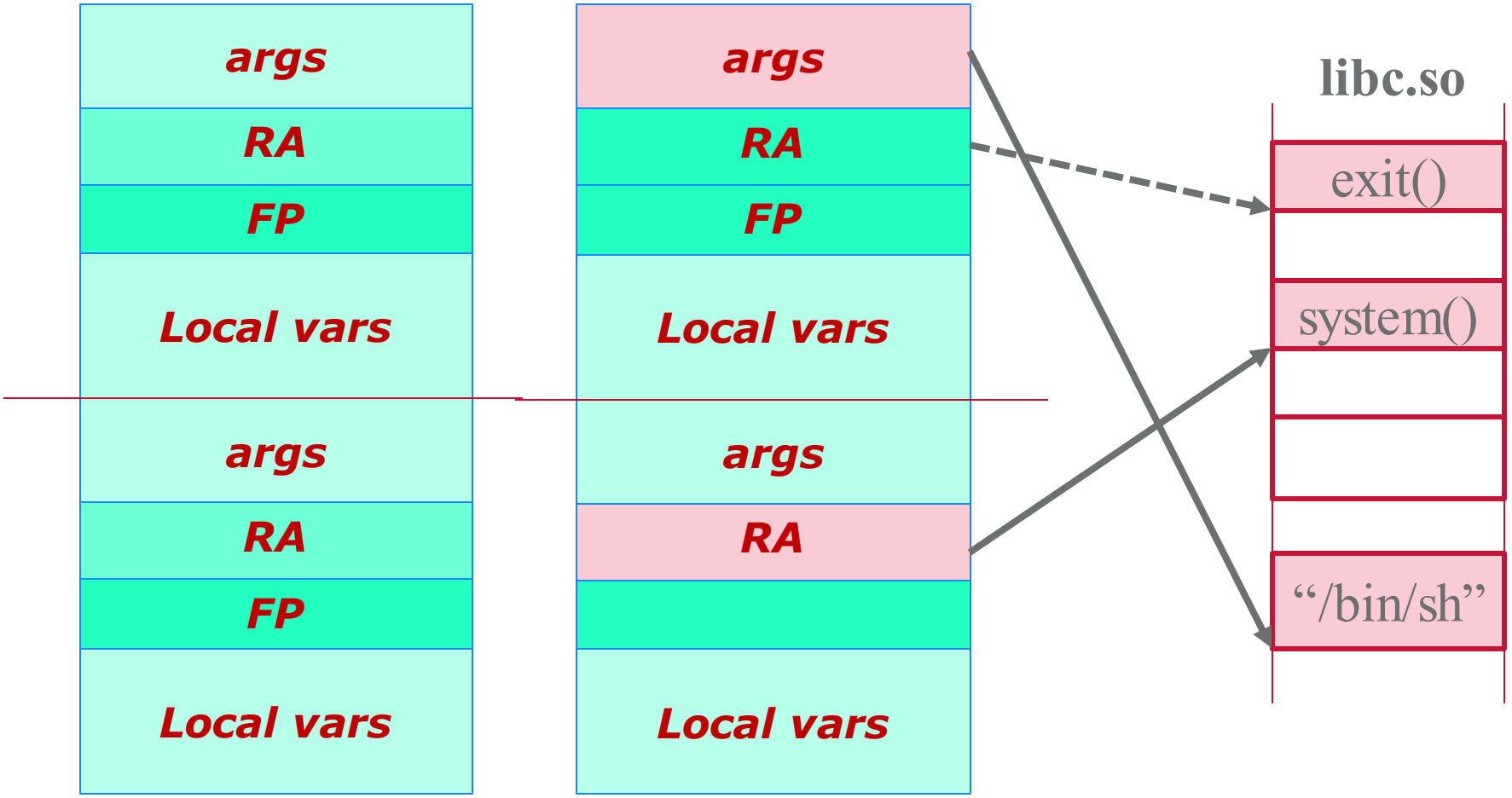
- Send request that overwrites one byte of the stack canary at a time:
  - Reponse: byte correct!
- How many tries?



Defense: Non-executable Stack and Heap (W^X)

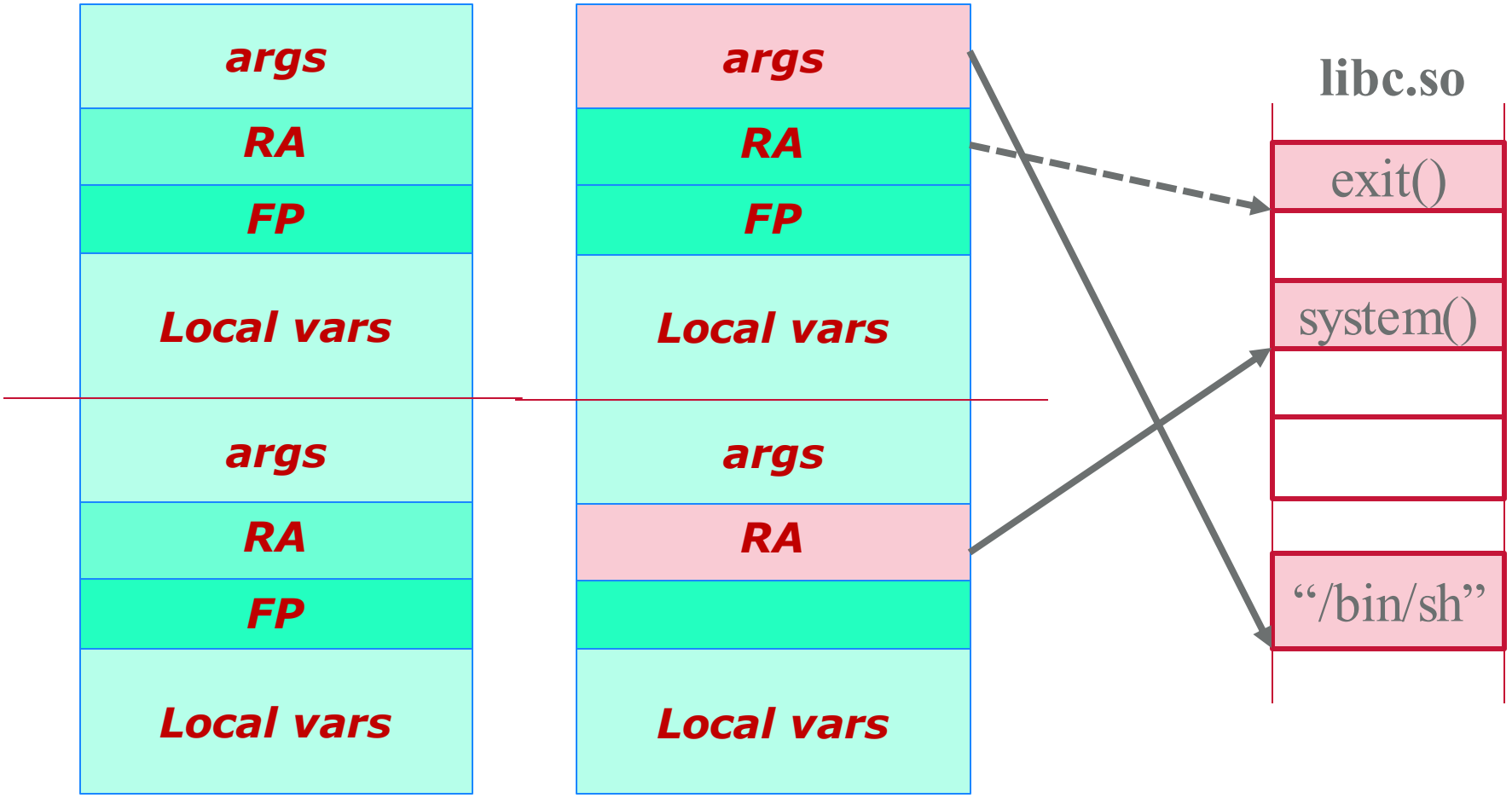
- Basic stack exploit requires code to run on the stack!
- Prevent overflow code execution by marking stack and heap segments as non-executable
- More generally, a page cannot be both W and X
  - NX-bit on AMD, XD-bit on Intel
  - NX bit in every Page Table Entry (PTE)
  - Deployment examples (both 2004):
    - Linux (via PaX project)
    - Windows since XP SP2 (Data Execution Prevention: (DEP))
- Limitations:
  - Some apps need executable heap (e.g. JITs)
  - Does not defend against **return-to-libc** attacks

Return-to-libc Attacks: use existing X pages!



Return-Oriented Programming

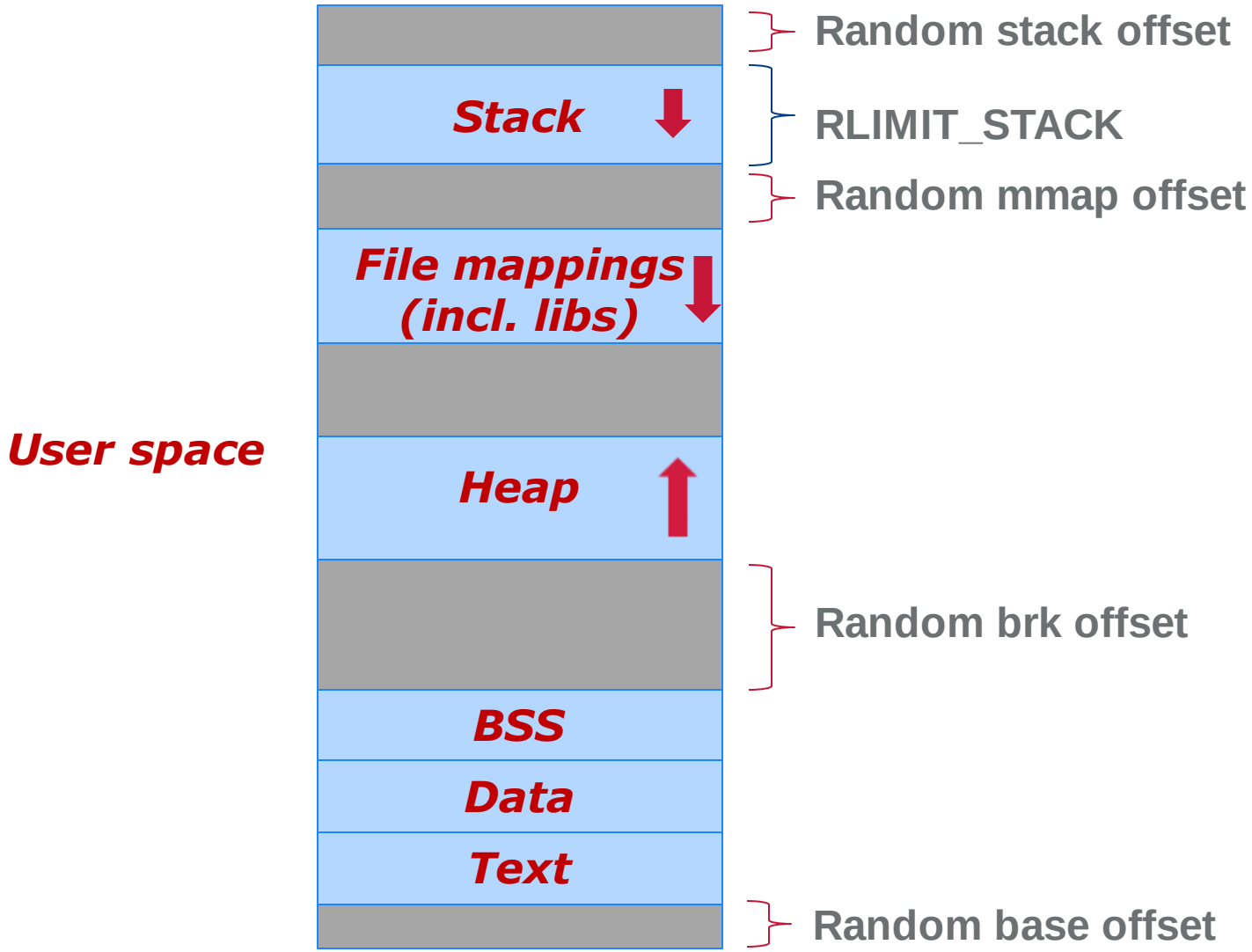
Generalize to arbitrary code fragments ending in RET!



Defense: Address Space Layout Randomization

- Map code, stack and heap segments, as well as shared libraries to random locations in process memory
  - Attacker does not know where e.g., system() lives!

Address Space Layout [Linux x86, w/ ASLR]



Defeating ASLR with W^X

- Approach 1: guessing location of ROP gadgets
  - ASLR provides probabilistic protection
- Approach 2: leaking information about memory layout and location of ROP gadgets

Preserving control and data integrity

- Ensure that control flow and memory accesses obey the intention of the code
- High-level idea:
  - Compute a static over-approximation of allowed behaviours
  - Flag any violations at runtime

Control-Flow Integrity

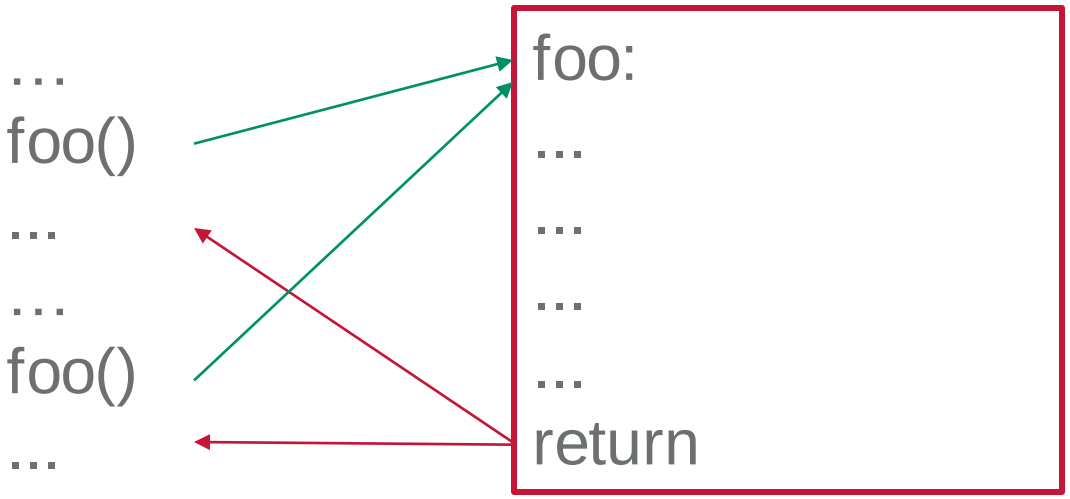
- Attackers try to divert control flow by overwriting return addresses, overwriting function pointers, etc.
- Key idea: compute a static overapproximation of allowable transfers
- Check for violations at runtime

Control-Flow Integrity Principles: Implementations, and Applications.
Martiın Abadi, Mihai Budiu, Ulfar Erlingsson, Jay Ligatti.
In Computer and Communications Security (CCS 2005)

Control-Flow Integrity

- Attackers try to divert control flow by overwriting return addresses, overwriting function pointers, etc.
- Key idea: compute a static overapproximation of allowable transfers
- Check for violations at runtime

Example 1 (pseudo-code)

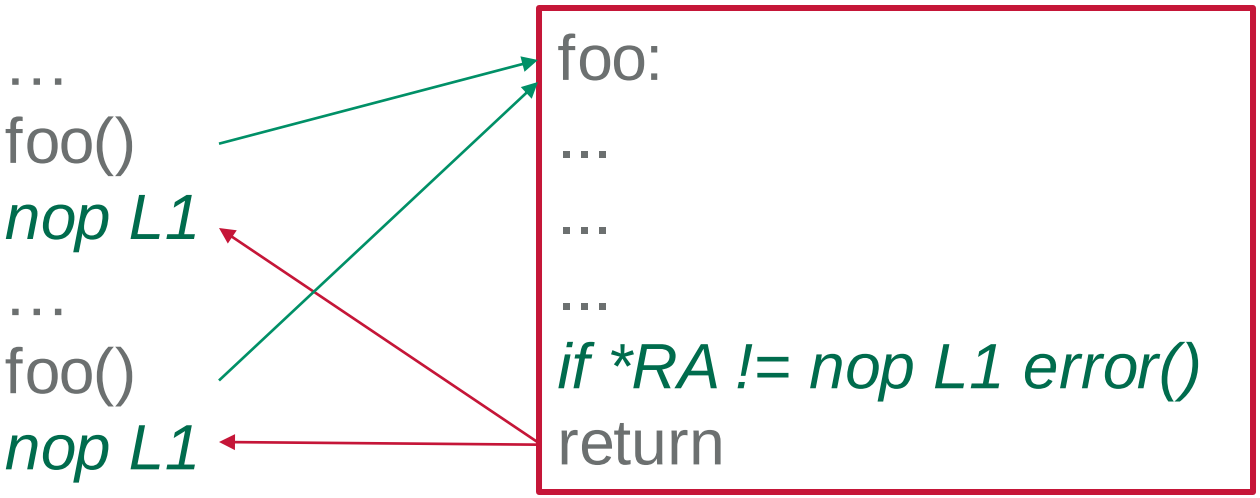


Assume foo is only called from these two places

Control-Flow Integrity

- Attackers try to divert control flow by overwriting return addresses, overwriting function pointers, etc.
- Key idea: compute a static overapproximation of allowable transfers
- Check for violations at runtime

Example 1 (pseudo-code)

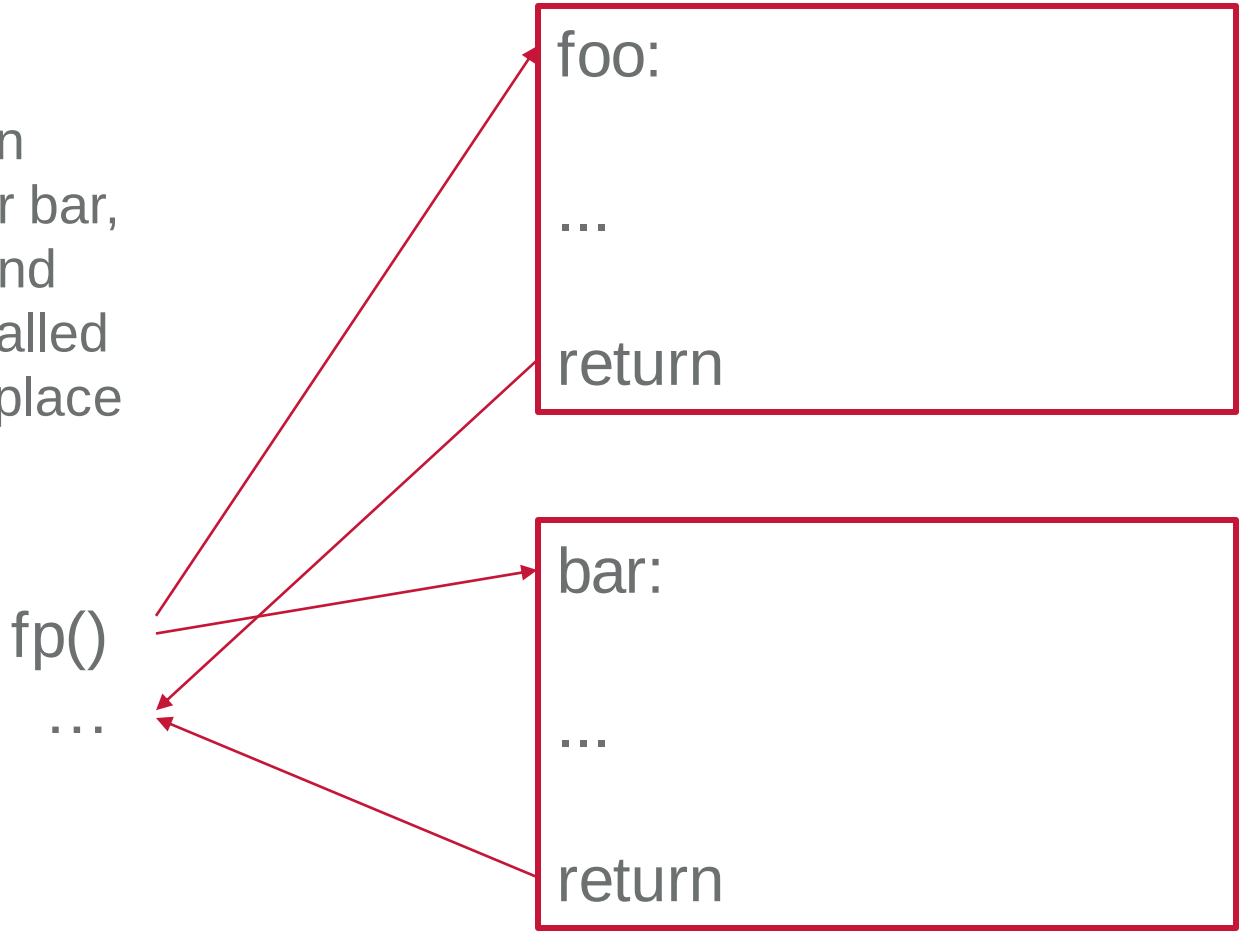


Can implement nop L using x86 prefetching

Control-Flow Integrity

Example 2 (pseudo-code)

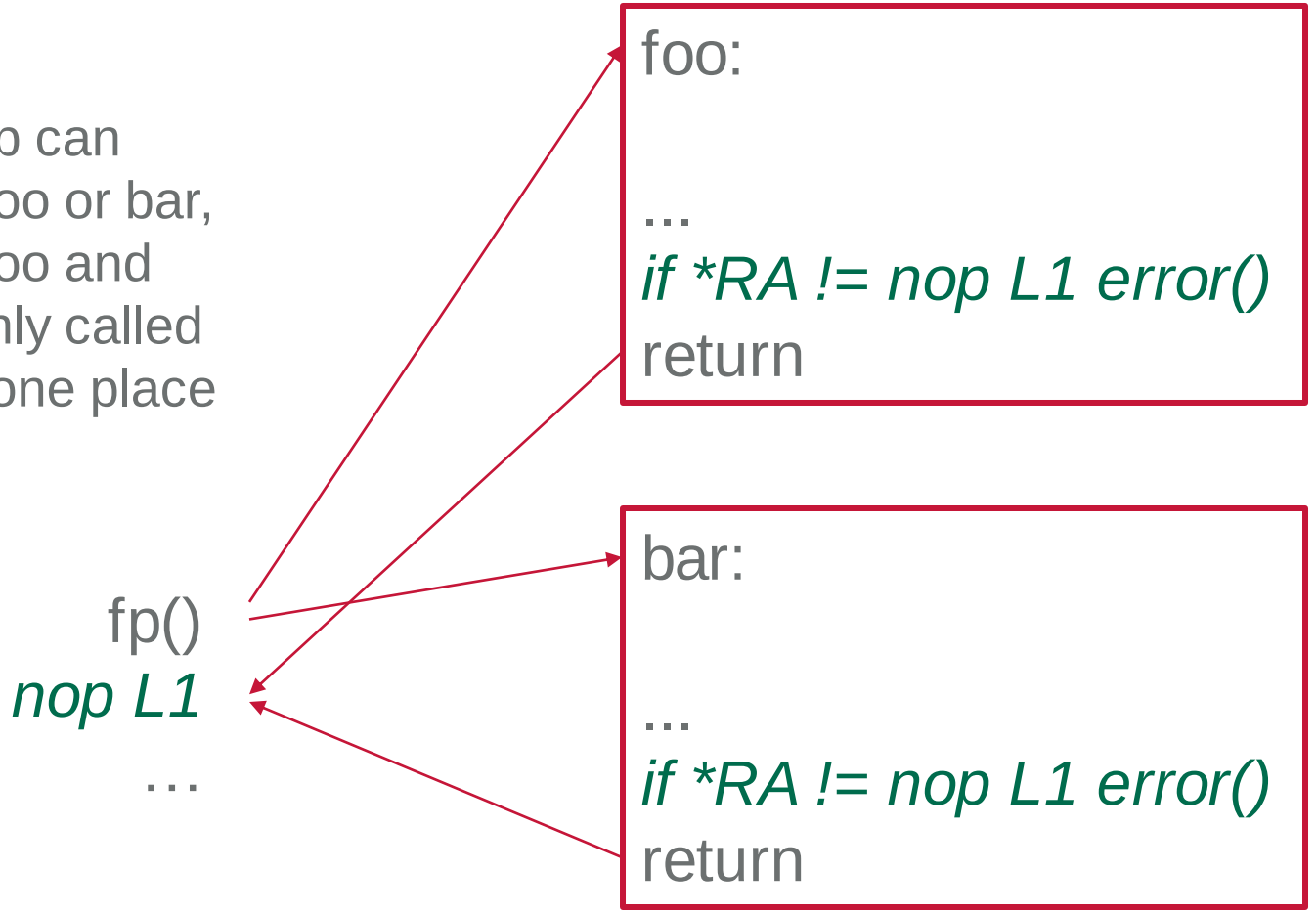
Assume fp can only call foo or bar, Assume foo and bar are only called from this one place



Control-Flow Integrity

Example 2 (pseudo-code)

Assume fp can only call foo or bar, Assume foo and bar are only called from this one place



Control-Flow Integrity

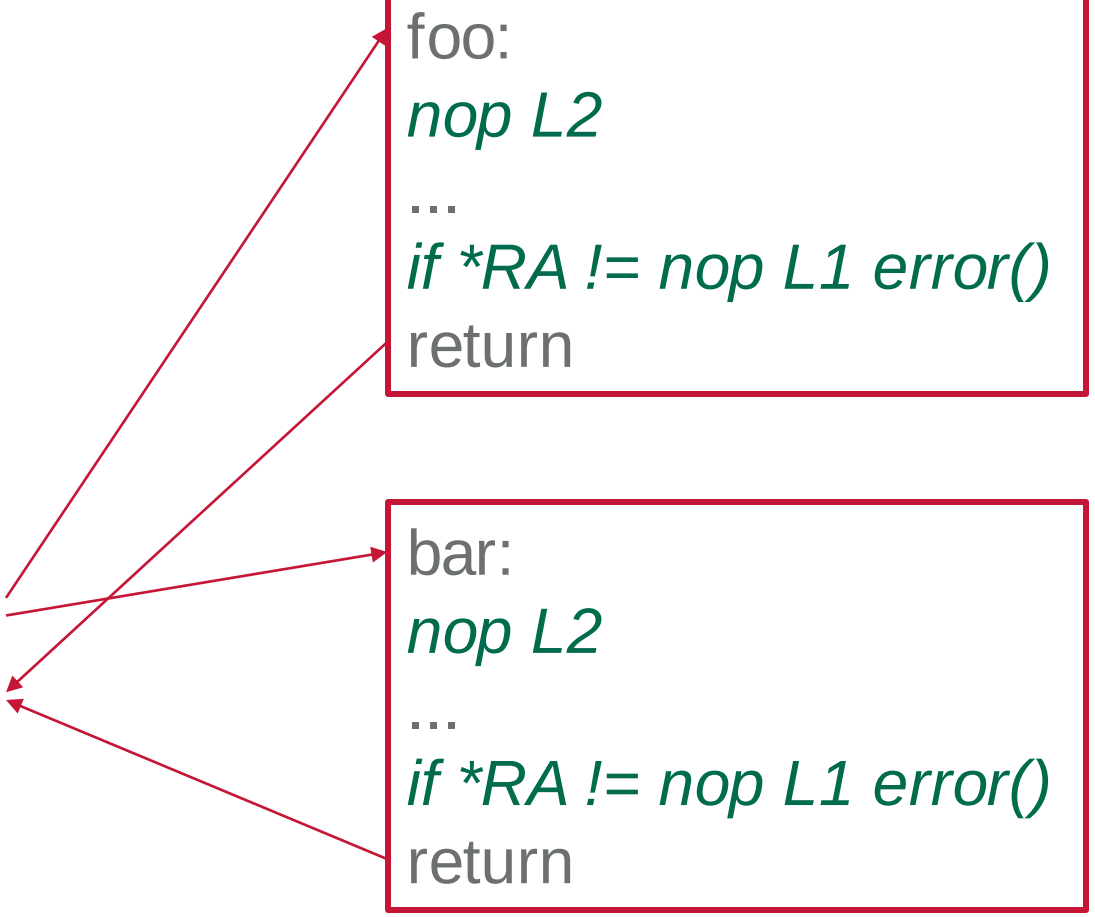
Example 2 (pseudo-code)

Assume fp can only call foo or bar, Assume foo and bar are only called from this one place

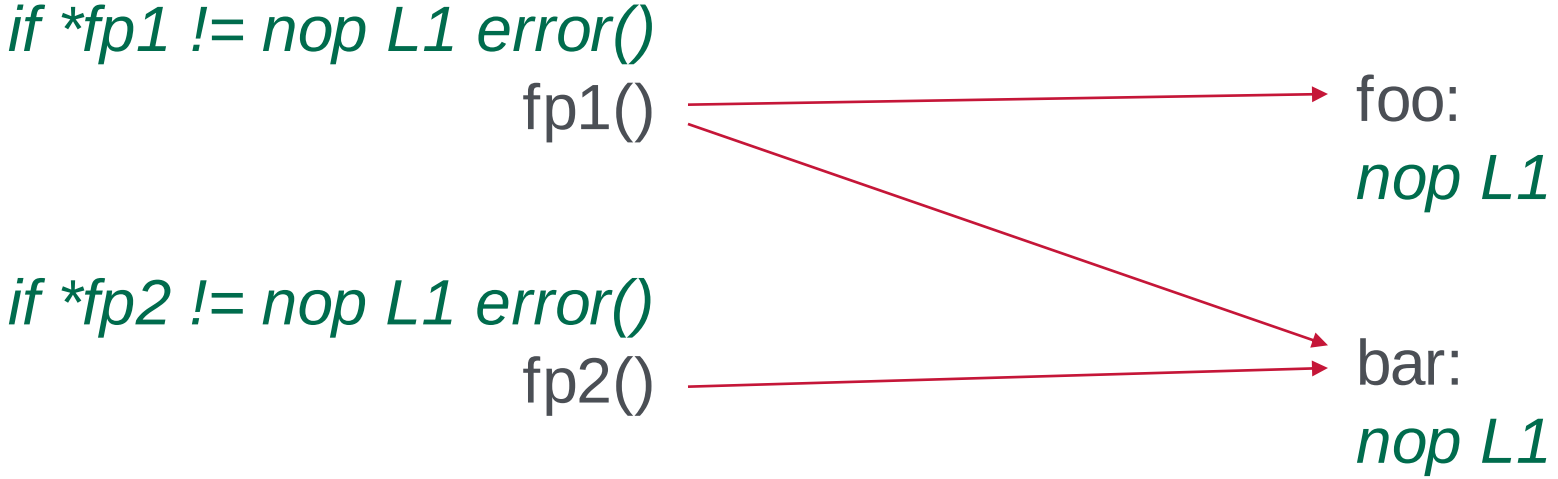
```
if *fp != nop L2 error()
fp()
nop L1
…
```

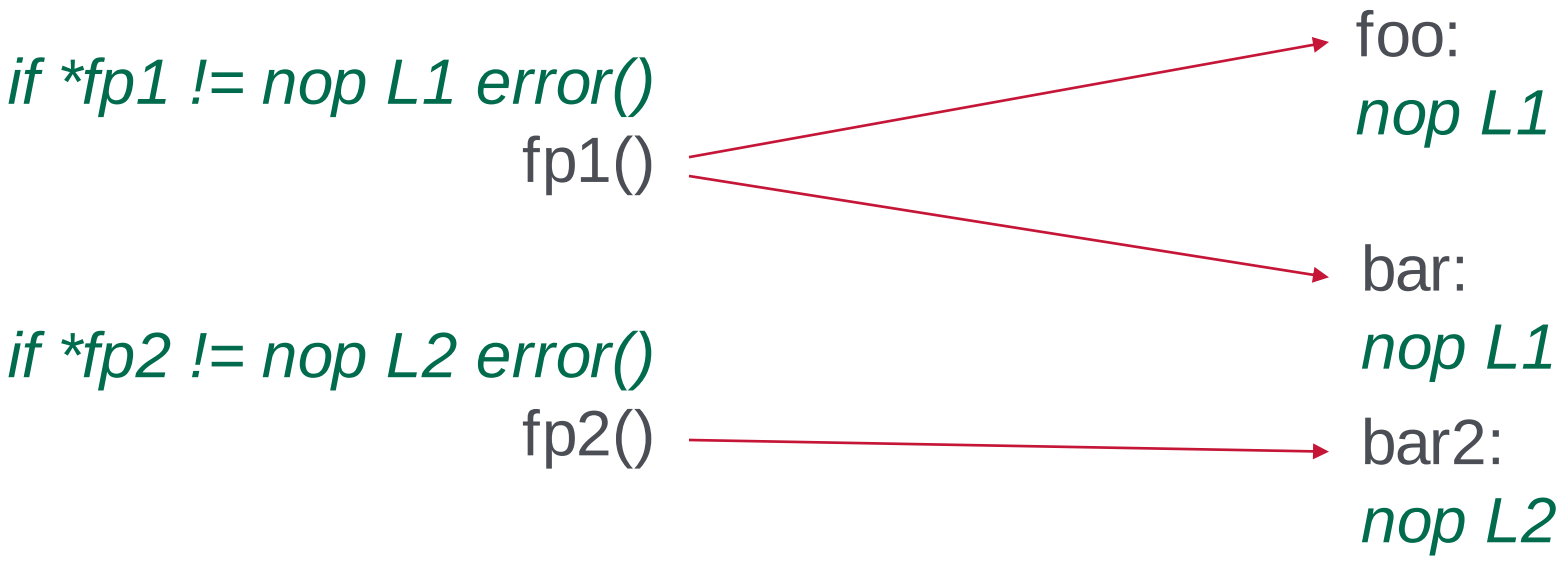
```
foo:
nop L2
...
if *RA != nop L1 error()
return
```

```
bar:
nop L2
...
if *RA != nop L1 error()
return
```



CFI: Refinements





Write Integrity Testing (WIT)

- CFI cannot prevent against non-control-data attacks

```
void read_req(int input) {
    char auth, buf[128];
    auth = check_credentials();
    buf[input] = 1;
    if (auth)
        enter_privileged_mode();
}
```

- Key idea: for each write, compute a static over-approximation of possible write targets
- Check for violations at runtime

Preventing memory error exploits with WIT.
Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, Miguel Castro
In IEEE Symposium on Security and Privacy (S&P 2008)

Write Integrity Testing (WIT)

- WIT first performs a static points-to analysis that for each unsafe write (via a pointer) computes the set of objects which it may access
- All objects in the same set are given a unique color (ID)
- The same color is assigned to the write instruction
- Color sets are merged if there are objects in common until a fixed point is reached
- The color of each memory byte is recorded in a table
  - We use shadow memory

WIT: Example

```
void read_req(int input) {
    char auth, buf[128];
    auth = check_credentials();
    if (color(buf+input) != green) error();
    buf[input] = 1;
    if (auth)
        enter_privileged_mode();
}
```

Data-Flow Integrity (DFI)

- Key idea: for each read, compute a static over-approximation of possible instructions that wrote it
- Check for violations at runtime

- Can also catch out-of-bounds reads

- This static over-approximation is given by a standard reaching definitions data-flow analysis

Securing software by enforcing data-flow integrity
Miguel Castro, Manuel Costa, Tim Harris
In Symposium on OS Design and Implementation (OSDI 2006)

DFI: Example

```
1: void read_req(int input) {
2:     char auth, buf[128];

3:     auth = check_credentials();

4:     buf[input] = 1;

5:     if (auth)
6:         enter_privileged_mode();
7:     }
```

DFI: Example

```
1: void read_req(int input) {
2:     char auth, buf[128];
       store[&auth] = 3
3:     auth = check_credentials();
       store[&buf[input]] = 4
4:     buf[input] = 1;
       if (store[&auth] != 3) error();
5:     if (auth)
6:         enter_privileged_mode();
7:     }
```

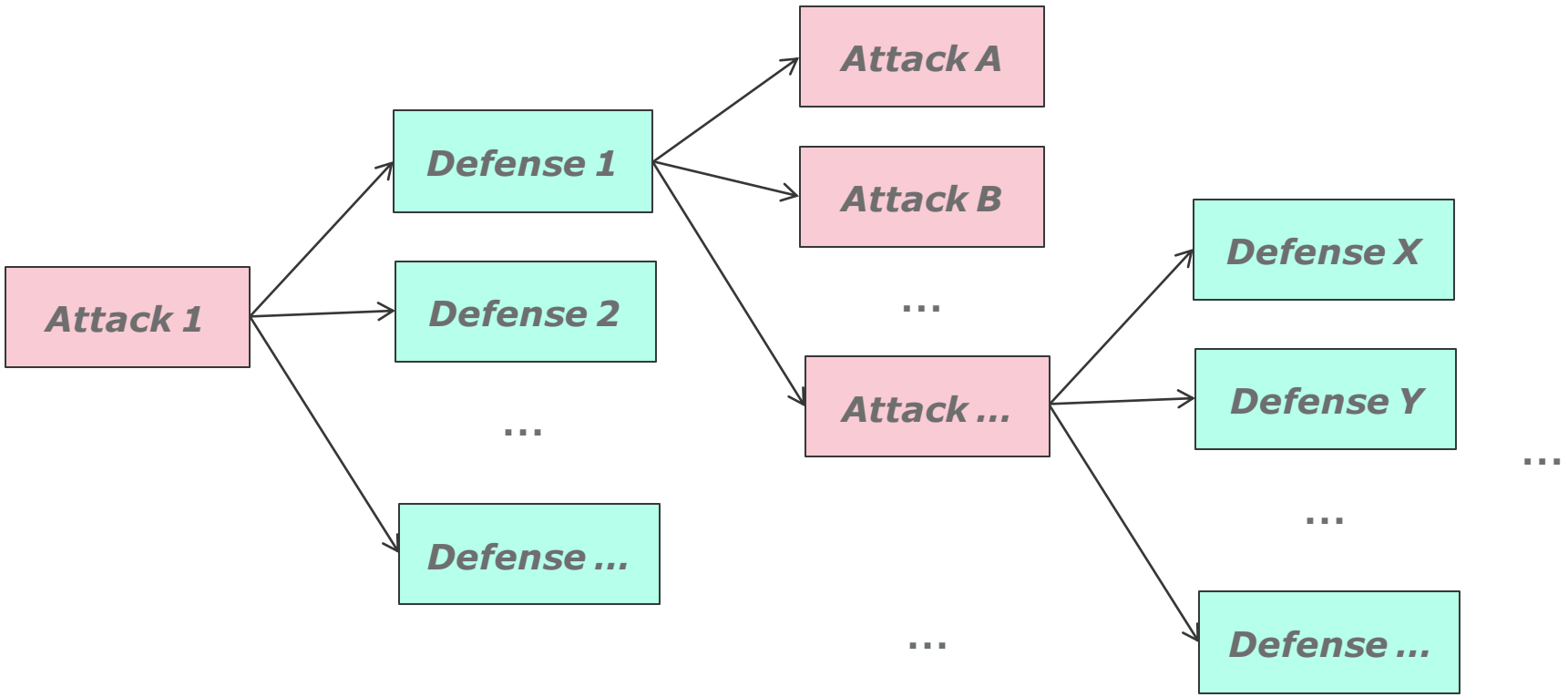
Unsafe writes need to be instrumented too, to protect the DFI shadow memory

CFI, WIT, DFI: Performance

Approximate numbers, as reported by the papers

	CFI	WIT	DFI
Average	16	10	104
Max	45	25	155

Basic Security Game



Software Security: Attacks and Defenses

- Designing effective security solutions is hard
- Asymmetrical fight:
  - Attacker: can exploit **any** code, in any way
  - Defender: needs to prevent **all** possible attacks
- In practice, there are rarely universal defenses
  - But each extra defense reduces attack opportunities, and makes some attacks harder