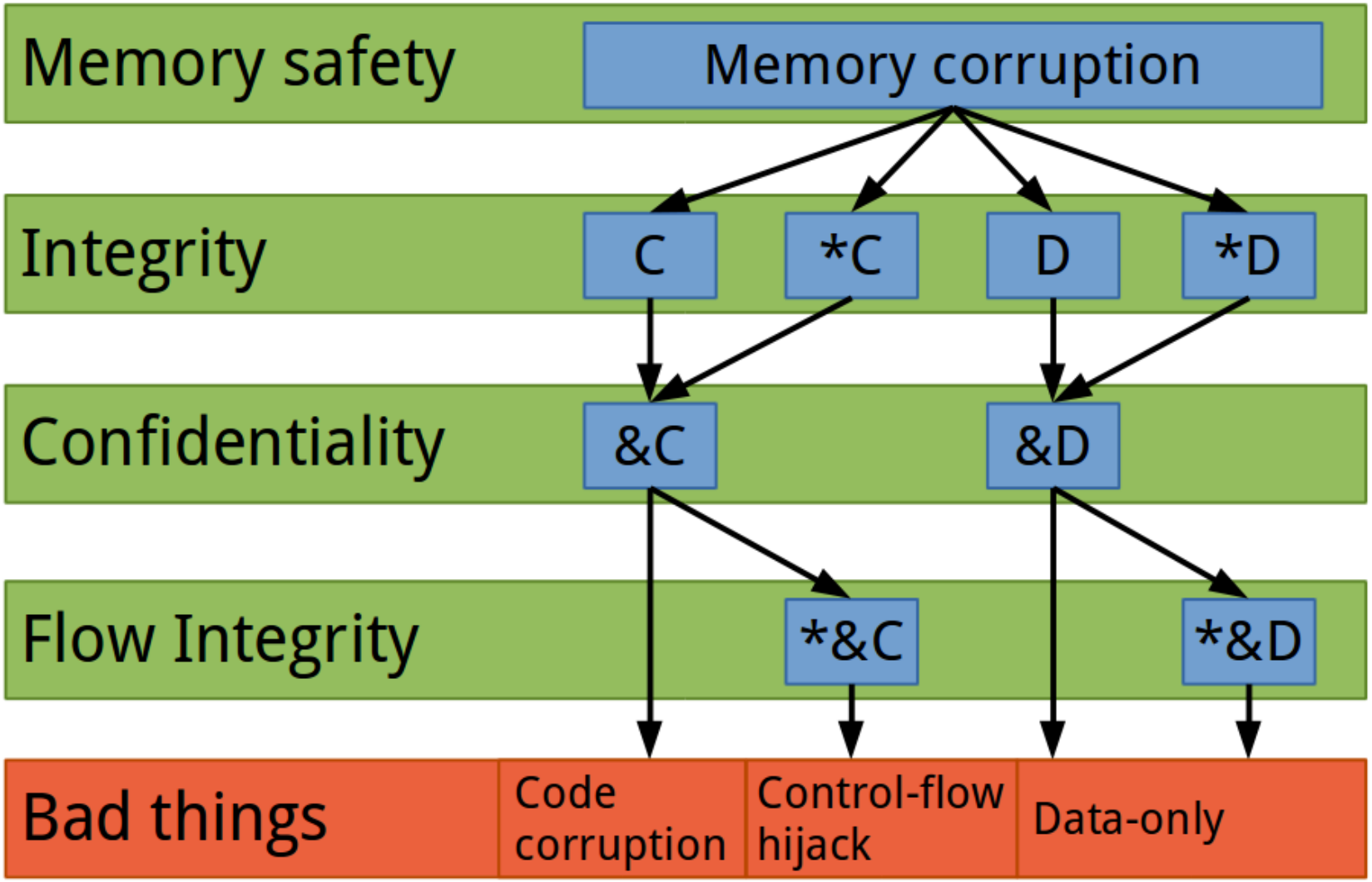
CS412 Software Security

Advanced Mitigations

Mathias Payer

EPFL, Spring 2019

Model for Control-Flow Hijack Attacks



Advanced mitigations

- Stack integrity
- Control-Flow Integrity (CFI)
- Code Pointer Integrity
- Sandboxing

Stack integrity

- Stack integrity ensures that (i) the return instruction pointer and (ii) the stack pointer cannot be modified
  - Return instruction pointers are code pointer; stack integrity guarantees return instruction pointer integrity, i.e., only valid return instruction pointers are dereferenced.
  - Pointers to other stack frames are stored on the stack, stack integrity ensures the integrity of this metadata as well. Note that modifying the base pointer indirectly modifies the return instruction pointer.
- Stack canaries are a weak form of stack integrity
- Shadow stacks are a strong form of stack integrity

Shadow stack

- A shadow stack is a second stack for each thread that keeps track of control data (e.g., return instruction pointer, base pointer, or code pointers)
- Note that not all implementations protect all types of data
- Data on the shadow stack is integrity protected
  - Implicitly: as the shadow stack contains only control data, buffer overflows are not possible
  - Explicitly: some shadow stacks are write protected
- How would you implement a shadow stack? Discuss!

Shadow stack

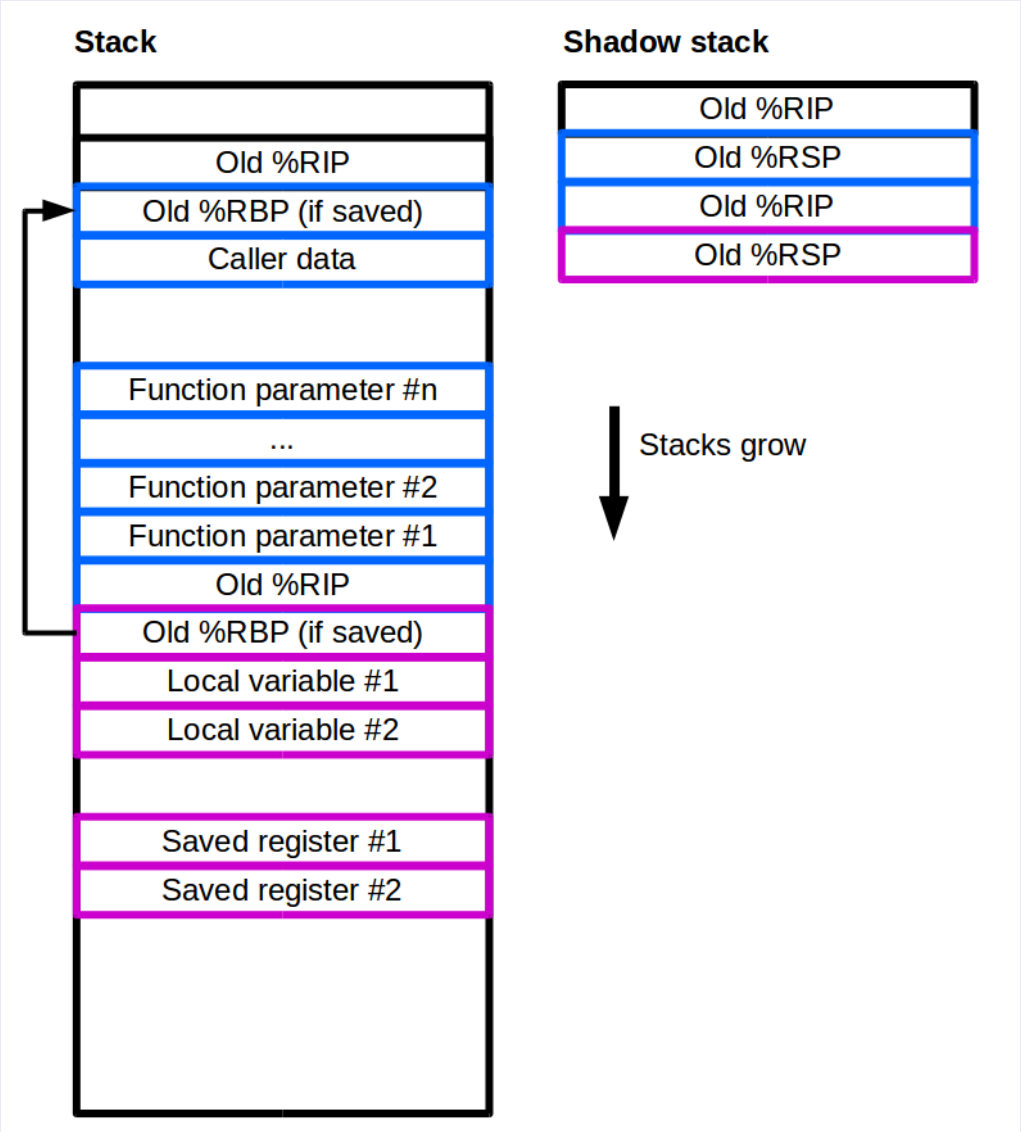


Figure 2:

Safe stack

- A shadow stack always keeps two allocated stack frames for each function invocation (maybe of different size)
- Core idea: for each variable in a stack frame decide if its safe
- Variables are safe if they are only used in a safe context, i.e., they don't escape the current function and are only used with bounded pointer arithmetic
- Push any unsafe variables to the unsafe stack
- Performance benefit: an unsafe stack frame is only allocated for if there are unsafe variables

Safe stack

```
int foo() {
 char buf[16];
 int r;
 r = scanf("%s", buf);
 return r;
}
```

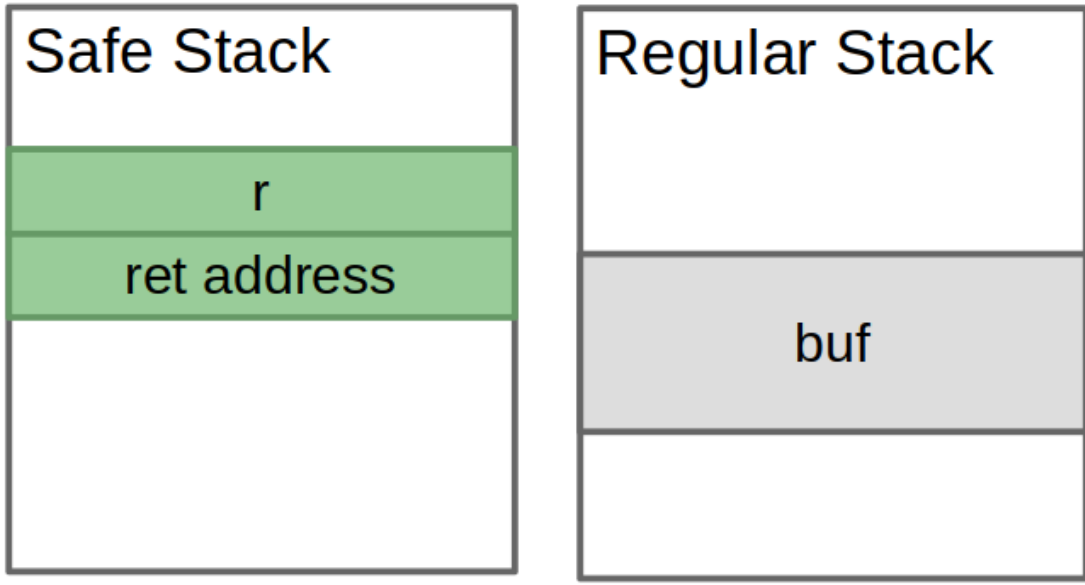


Figure 3:

Control-Flow Integrity

CFI is a defense mechanism that protects applications against control-flow hijack attacks. A successful CFI mechanism ensures that the control-flow of the application never leaves the predetermined, valid control-flow that is defined at the source code/application level. This means that an attacker cannot redirect control-flow to alternate or new locations.

```
CHECK(fn);
(*fn)(x);
```

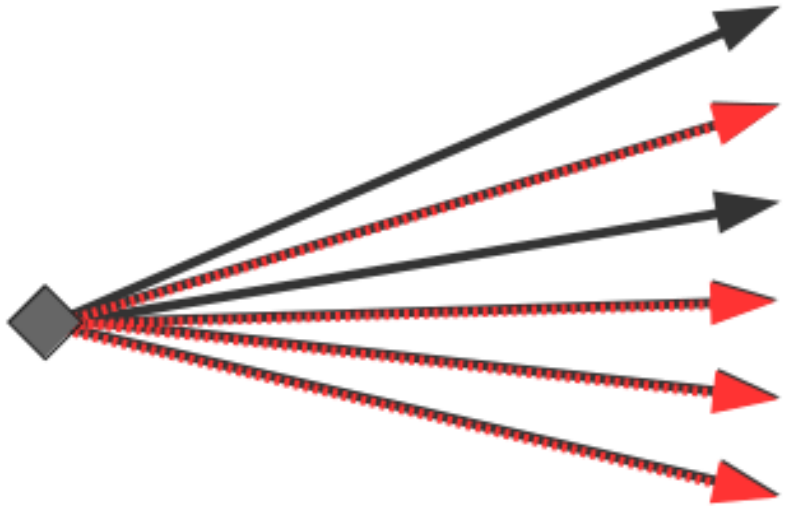


Figure 4:

Basics of a CFI mechanism

Core idea: restrict the dynamic control flow of the application to the control-flow graph of the application.

- Target set construction
- Dynamic enforcement mechanism to execute runtime checks

CFI: target set construction

How do we infer the control-flow graph (for C/C++ programs)? A static analysis (on source code or binary) can recover an approximation of the control-flow graph. Precision of the analysis is crucial!

- Valid functions
- Arity
- Function prototypes
- Class hierarchy analysis

CFI: target set construction

Trade-off between precision and compatibility.
One set of *valid functions* is highly compatible with other software but may result in imprecision given the large amount of functions.
Class hierarchy analysis results in small sets but may be incompatible with other source code and some programmer patterns (e.g., casting to void or not passing all parameters).

CFI: target set construction trade-offs

Microsoft chose compatibility over security. LLVM chose security over compatibility.
Discuss trade-offs.

CFI: runtime checks

The analysis produces target sets for each location of an indirect control-flow transfer. The runtime check leverages the runtime value and the target set to execute a set check. The most efficient implementation uses a set of bit masks.

```
void (*fn)(int) = &func;
...
if (!contains(targetset, fn)) {
  abort("Error: illegal target");
}
fn(12);
```

Note that the check and dispatch are atomic as otherwise, this would result in a TOCTTOU vulnerability.

CFI: limitations

- CFI allows the underlying bug to fire and the memory corruption can be controlled by the attacker. The defense only detects the deviation after the fact, i.e., when a corrupted pointer is used in the program.
- Over-approximation in the static analysis reduces security guarantees
- What kind of attacks are possible?
  - An attacker is free to modify the outcome of any JCC
  - An attacker can choose any allowed target at each ICF location
  - For return instructions: one set of return targets is too broad and even localized return sets are too broad for most cases.
  - For indirect calls and jumps, attacks like COOP (Counterfeit Object Oriented Programming) have shown that full functions can be used as gadgets.

Code-Pointer Integrity

- Memory corruption is abundant.
- Strong memory-safety-based defenses have not been adopted.
- Weaker defenses like strong memory allocators also ignored.
- Only defenses that have *negligible* overhead are adapted.
- What if we can have memory safety but only where it matters?
- Assume we want to protect applications against control-flow hijacking attacks. What data must be protected?
- Code Pointer Integrity (CPI) ensures that all code pointers are protected at all times

CPI attacker model

- Attacker can read data, code (includes stack, bss, data, text, heap)
- Attacker can write data.
- Attacker cannot modify code
- Attacker cannot influence the loading process
- (This is true for most mitigations)

Existing memory safety solutions

- SoftBound+CETS 116% overhead, only partial support for SPEC CPU2006
- CCured: 56% overhead
- AddressSanitizer: 73% overhead, only partial memory safety (probabilistic spatial)
- CPI targets protection of *subset* of data

Checks enforce memory safety!

```
char *buf = malloc(10);
// instr: track bounds
buf_lo = p; buf_up = p+10;
...
char *q = buf + input;
// instr: track bounds
q_lo = buf_lo; q_up = buf_up;
// instr: check bounds
if (q < q_lo || q >= q_up)
  abort();
*q = input2;
...
(*func_ptr)();
```

Checks focus on all data, how can we protect integrity of only code pointers?

Paradigm shift: protect *select* data

Instead of protecting everything a little protect a little completely. Strong protection for a select subset of data. Attacker may modify any unprotected data.

By only protecting code pointers, CPI reduces the overhead of memory safety from 116% to 8.4% while still deterministically protecting applications against control-flow hijack attacks.

What data must be protected?

- Sensitive pointers are code pointers and pointers used to access sensitive pointers
- We can over-approximate and identify sensitive pointer through their types: all types of sensitive pointers are sensitive
- Over approximation only affects performance

Memory layout

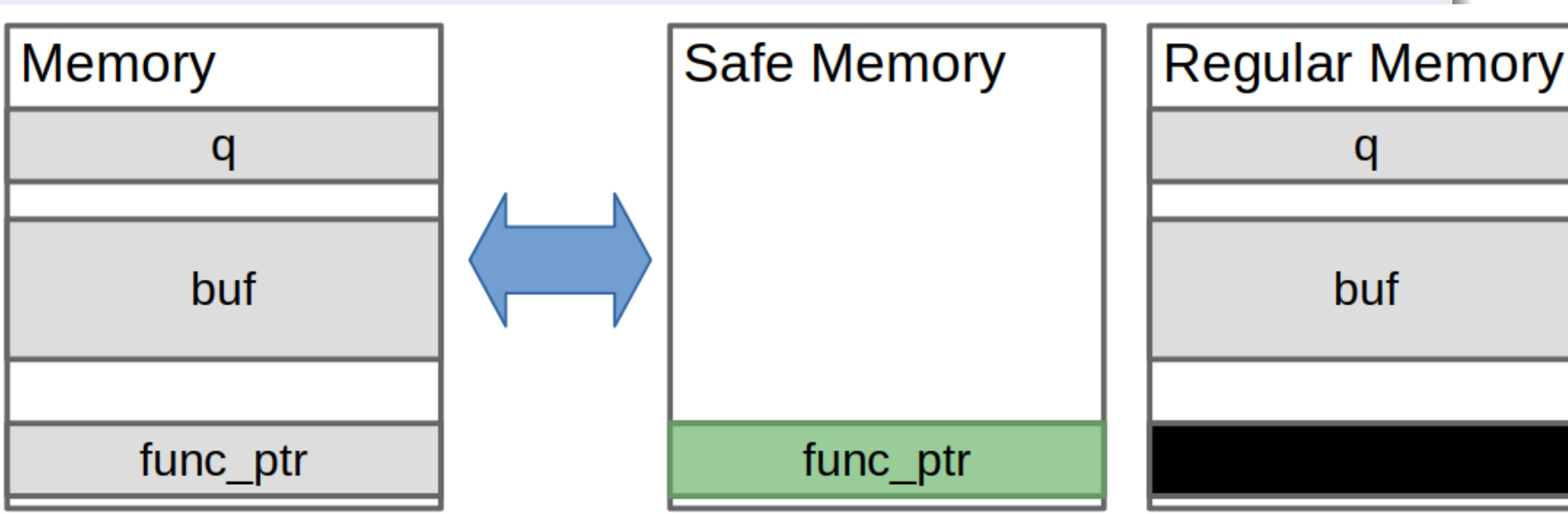


Figure 5:

Memory layout

- Memory view is split into two views: control and data plane
  - The control plane is a view that only contains code pointers (and transitively all related pointers)
  - The data plane contains only data, code pointers are left empty (void/unused data)
- The two planes must be separated and data in the control plane must be protected from pointer dereferences in the data plane

Sandboxing

Different levels of granularity for sandboxing:

- Kernel isolates process memory
- `chroot` / containers isolate processes from each other
- `seccomp` restricts processes from interacting with the kernel
- Software-based Fault Isolation isolated components in a process

Software-based Fault Isolation (SFI)

- Application and untrusted code run in the same address space
- The untrusted code may only read/write the untrusted data segment. How do you implement such a restriction?
  - Segmentation (on x86)
  - Mask memory area: `and $0x00ffffff, %rax; mov $0xc0fe0000, (%rax)`
  - Challenge for CISC ISAs: jumping to unalaigned instructions: `mov $0x80cd01b0, (%rax)` contains `mov $1, %al; int $0x80`
  - Google's NaCL solves the challenge by aligning instructions

Summary and conclusion

- Adopted defenses do not stop all attacks
- Control-flow hijacking is the most versatile attack vector
- Stack integrity protects code pointers on the stack
- CFI restricts targets on the forward edge
- CPI prohibits control-flow hijacking, key insight: enforce memory safety *only* for code pointers
- Sandboxing separates different privilege domains