CMPSC 447
Control-Flow Integrity

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

Exploit Vulnerabilities

- How do you exploit a memory error vulnerability?

Memory Error Exploits

- First and most common way to take control of a process – control-flow hijacking
- Write to control memory
  - Call the victim with inputs necessary to overflow buffer or exploit data pointer
  - To overwrite the value of a code pointer (e.g., return address) or data that impacts control (e.g., conditional)
- Direct the process execution to exploit code
  - Inject code (if possible) or reuse existing code
  - Use compromised pointer to jump to the chosen code

Prevent Overflows

PENNSTATE 1855

- How would you prevent adversaries from control-flow hijacking?
  - Use safe string functions correctly (flaw)
  - Apply a comprehensive bounds checking defense (access)
  - Restrict options for control flows (exploit)
- We will examine the latter two today

Check Bounds

- How would you check bounds naively?

Check Bounds

- How would you check bounds naively?
  - Presumably, you need to know the start and end of a buffer
- Then, you need to check bounds – how and when?

Bounds Checks

- SoftBound
  - Records base and bound information for every pointer as disjoint metadata
  - Check and/or update such metadata whenever one dereferences (uses) a pointer
  - Supported by formal proofs of spatial memory safety
- Separating metadata from pointers maintains compatibility with C runtime

SoftBound

- Checking Bounds
  - Whenever a pointer is used to access memory (i.e., dereferenced), SoftBound inserts code (highlighted in grey) for checking the bounds to detect spatial memory violations.

```
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
value = *ptr;     // original load
```

Where `check()` is defined as:

```
void check(ptr, base, bound, size) {
  if ((ptr < base) || (ptr+size > bound)) {
    abort();
  }
}
```

SoftBound

- Need to initialize, maintain, and use bounds information
  - How to create?
  - What ops require changes to bounds info?
  - How to lookup bounds info?

SoftBound

- Creating pointers
  - New pointers in C are created in two ways:
    - (1) explicit memory allocation (i.e. malloc()) and
    - (2) taking the address of a global or stack-allocated variable using the '&' operator.
  - Initialization for malloc

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;
```

SoftBound

- Pointer arithmetic
  - When an expression contains pointer arithmetic (e.g., ptr+index), array indexing (e.g., &(ptr[index])), or pointer assignment (e.g., newptr = ptr;), the resulting pointer inherits the base and bound of the original pointer

```
newptr = ptr + index;     // or &ptr[index]
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

SoftBound

- Pointer metadata retrieval
  - SoftBound uses a table data structure to map an address of a pointer in memory to the metadata for that pointer
  - On load

```
int** ptr;
int* new_ptr;
...
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
newptr = *ptr;       // original load
newptr_base = table_lookup(ptr)->base;
newptr_bound = table_lookup(ptr)->bound;
```

  - On store

```
int** ptr;
int* new_ptr;
...
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
(*ptr) = new_ptr;      // original store
table_lookup(ptr)->base = newptr_base;
table_lookup(ptr)->bound = newptr_bound;
```

SoftBound

- Downsides
  - Has a significant overhead – 67% for 23 benchmark programs
  - Uses extra memory – 64% to 87% depending on implementation
  - Does not support multithreaded programs
- But, achieves full spatial memory safety for C programs
  - We have used in "privilege separation" work (PtrSplit) to be discussed later

Fat Pointers

- Idea
  - Associate base and bounds metadata with every pointer
- Problems
  - Forgery – overwrite base and bounds when overwrite pointer
  - Limited space – have at most 64 bits to express address and metadata
  - Performance – SoftBound demonstrated that these operations could be costly
- Solutions?

Low-Fat Pointers

- Idea
  - Hardware support for fat pointers
- Solutions
  - Forgery – Hardware tags to prevent software from overwriting without detection
  - Limited space – Do not really need entire 64-bit address space – use 46-bit address space and rest for metadata
  - Performance – Hardware instructions to perform desired operations inline
- Result: Memory error protection for 3% overhead

Low-Fat Pointers

- Checking – similar to SoftBound

```
if ((ptr.A >= ptr.base)  && (ptr.A <= ptr.bound))
    perform load or store
else
    jump to error handler
```

- Tagging – common technique from long ago
  - Hardware differentiates data (and code) from references
  - Utilize 8 bits of 64-bit pointer for "type" of pointer
- Encoding
  - Base and bounds within the remaining 10 bits
  - Not many. Optimize use? Align regions

Direct Control of Program

- Once an adversary can specify the value of a code pointer, they can direct the program's execution (control flow)
  - Return address (call stack) – choose next code to run on return instruction
  - Function pointer (stack or heap) – chooses next code to run when invoked
- What exploit options do adversaries have available?

Prevent Code-Reuse Attacks

PENNSTATE 1855

- Most powerful adversary attack is code-reuse attack
- E.g., Using a ROP chain can execute any code in any order
  - As long as it terminates in a return instruction
  - Can also chain calls and jumps
- How would you prevent a program from executing the victim's code in unexpected and arbitrary ways?

Prevent Code-Reuse Attacks

- How would you prevent a program from executing gadgets rather than the expected code?
  - Control-flow integrity
    - Force the program to execute according to an expected CFG

Control Flow Graph

- Is a graph G=(V,E)
  - Graph vertices: V – set of program instructions
  - Graph edges: E=(a, b) – meaning *b* can succeed *a* in some execution
- For a function, a CFG relates the instructions and the possible ordering of instruction executions
- Many of these can be predicted from the code

Control Flow Graph

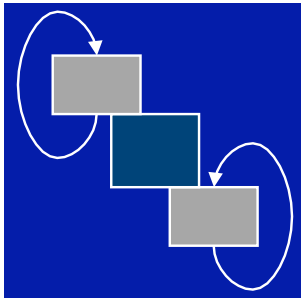
- Each line corresponds to one or more instructions
- Non-trivial edges
  - Line 1 → 11
  - Line 3 → 5
  - Line 7 → 9
- All flow edges known from code

```
0:  /* i, n are ints, and char b[12] */
1:  if (i > 0) {
2:      n = i + 2;
3:      if (n == 7)
4:          b[n+i] = 'a';
5:      else {
6:          n = i + 8;
7:          if (n < 12)
8:              b[n] = 'a';
9:          }
10:  }
```

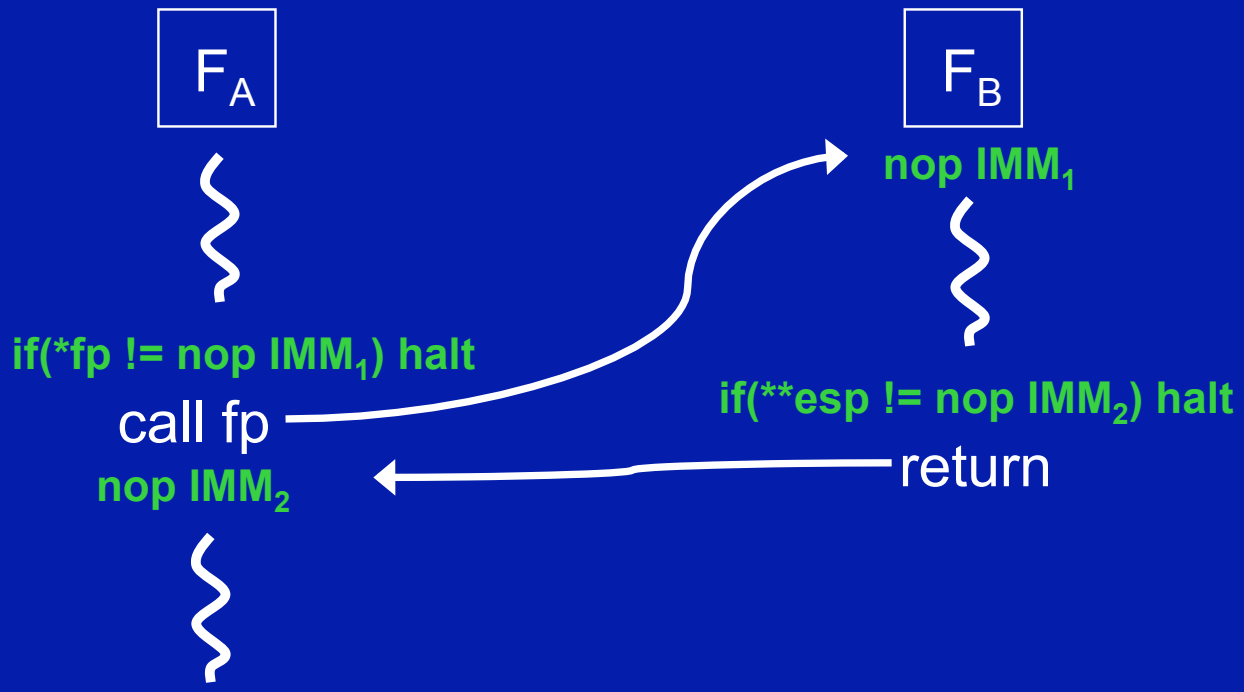
CFG Ambiguity

- There is ambiguity about the target of some instructions
  - Called indirect control flows
- Those instructions are
  - Returns
  - Indirect Calls
  - Indirect Jumps
- Their targets are computed at runtime
  - Can you give an example? How to limit to the CFG?

Control-Flow Integrity



Our Mechanism



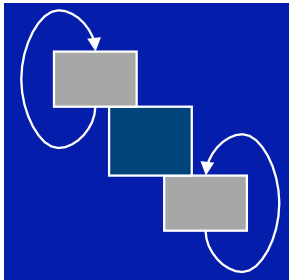
CFG excerpt

$A_{call} \rightarrow B_1$

$A_{call+1} \leftarrow B_{ret}$

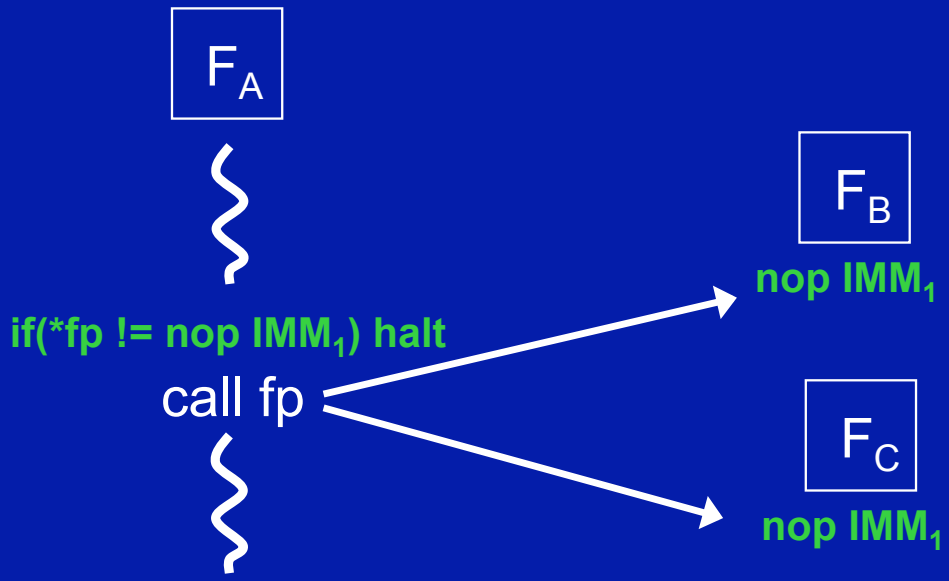
NB: Need to ensure bit patterns for nops appear nowhere else in code memory

Control-Flow Integrity

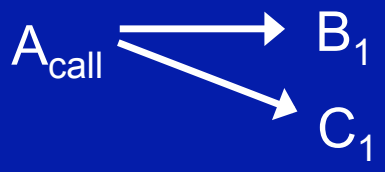


More Complex CFGs

Maybe statically all we know is that
F_A can call any int⟶ int function



CFG excerpt

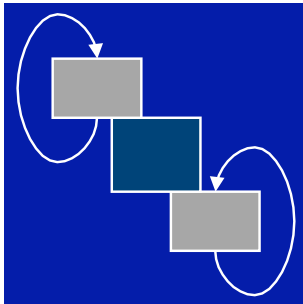


succ(A_{call}) = {B_1, C_1}

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction

9

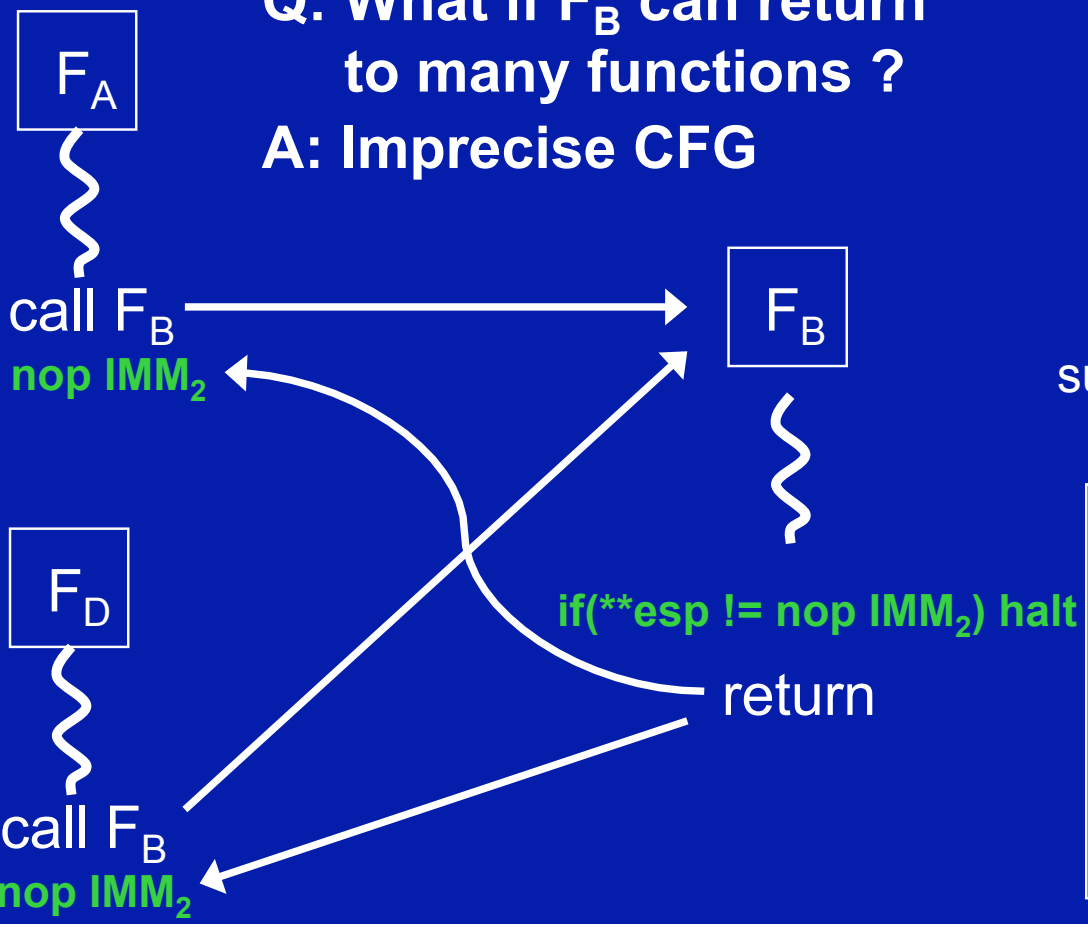
Control-Flow Integrity



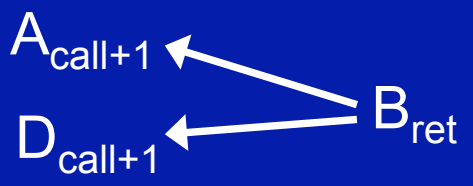
Imprecise Return Information

Q: What if F_B can return to many functions ?

A: Imprecise CFG



CFG excerpt



$succ(B_{ret}) = \{A_{call+1}, D_{call+1}\}$

CFG Integrity: Changes to the PC are only to valid successor PCs, per succ().

Destination Equivalence

- Eliminate impossible return targets
  - Two *destinations* are said to be *equivalent* if they connect to a common source in the CFG.

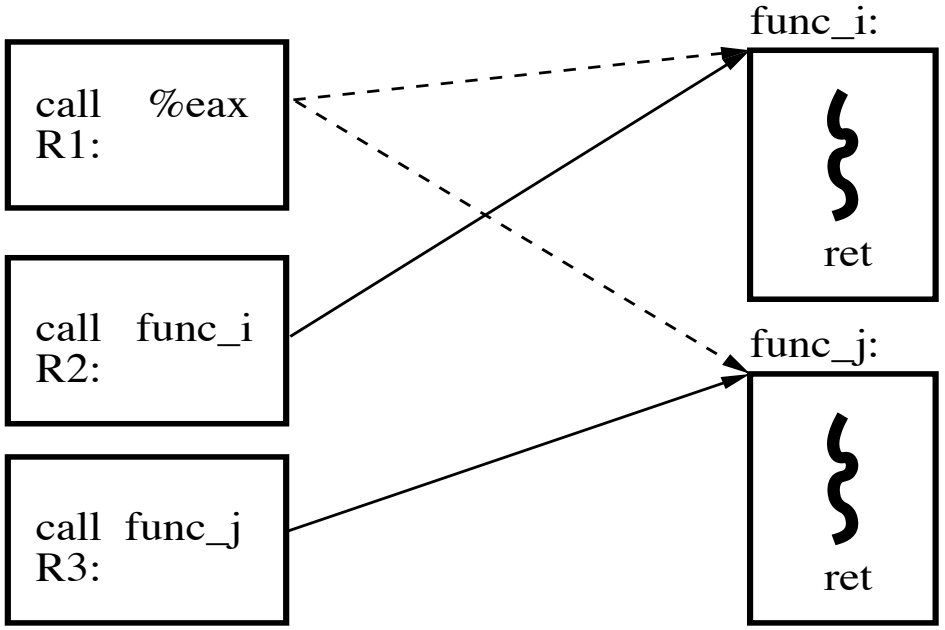


Figure 4. Destination equivalence effect on ret instructions (a dashed line represents an indirect *call* while a solid line stands for a direct *call*)

Destination Equivalence

- Eliminate impossible return targets
  - Can *R2* be a return target of *func_j*?

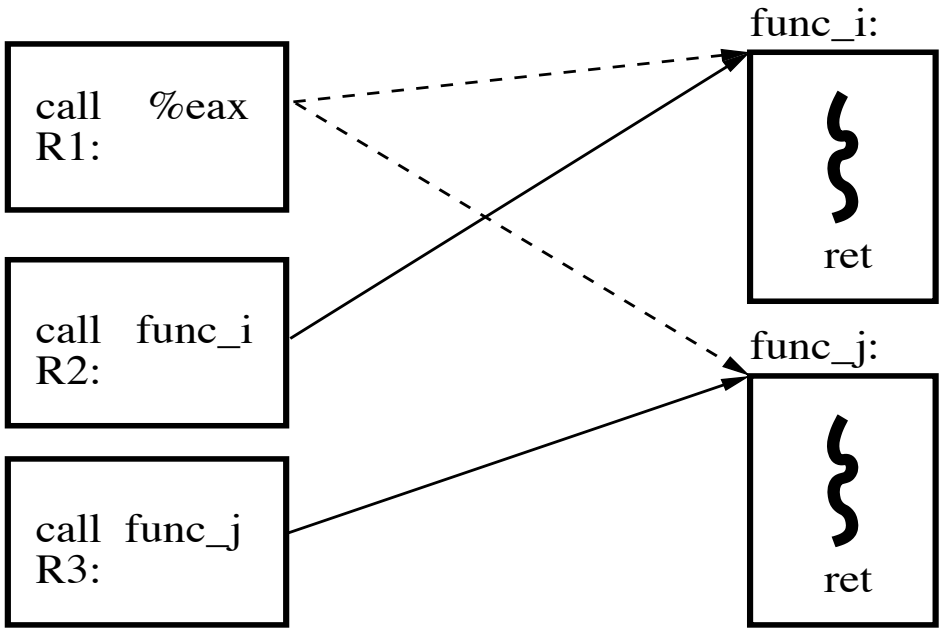
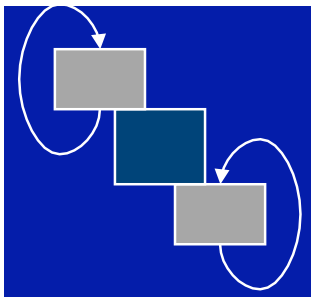


Figure 4. Destination equivalence effect on ret instructions (a dashed line represents an indirect *call* while a solid line stands for a direct *call*)

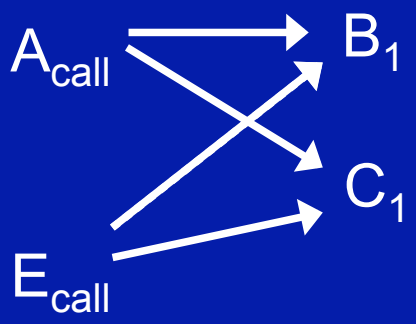
Control-Flow Integrity



No "Zig-Zag" Imprecision

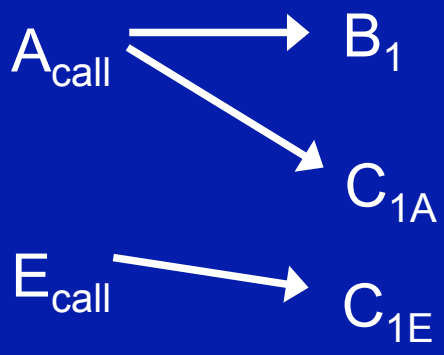
Solution I: Allow the imprecision

CFG excerpt



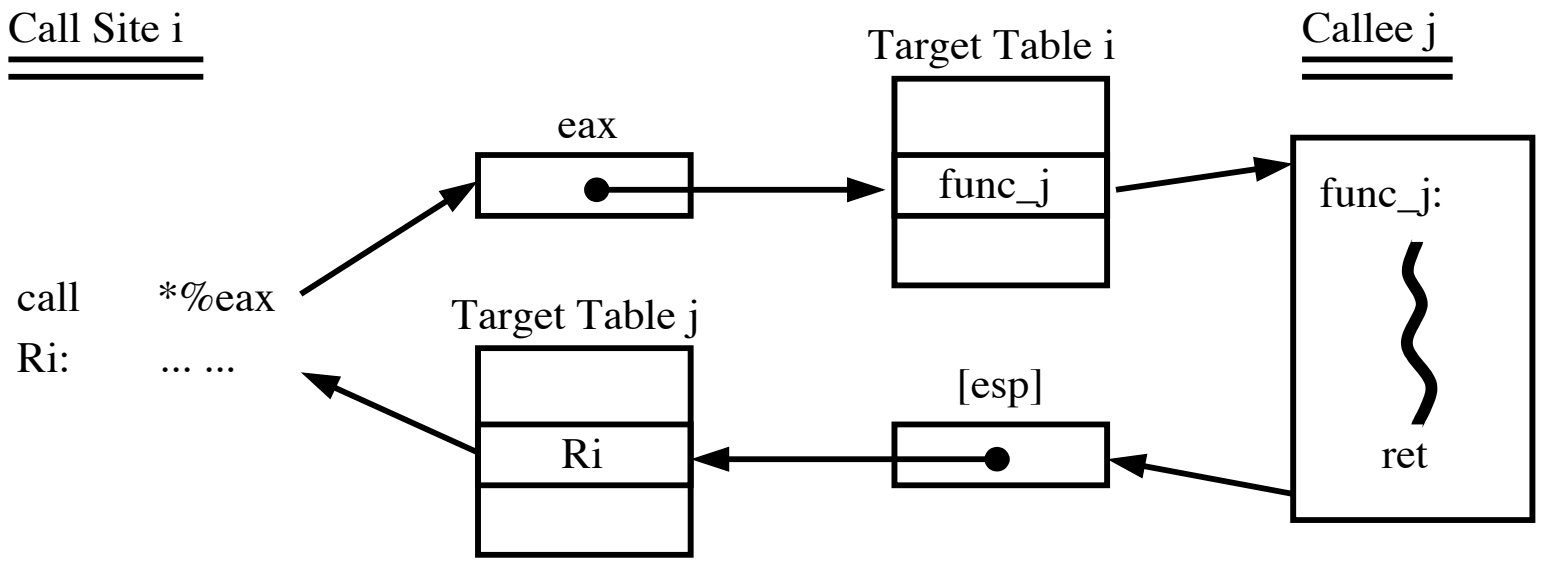
Solution II: Duplicate code to remove zig-zags

CFG excerpt



Restricted Pointer Indexing

- One table for call and return for each function



- Why can't *func_j* return to *R2* with this approach?

Other Problems with CFI

- CFI enforcement has overhead - Can we reduce?
- Idea: only check CFI for the last N branches
  - kBouncer inspects the last 16 indirect branches taken each time the program invokes a system call
    - Why 16? Uses Intel's Last Branch Record (LBR), which can store 16 records
  - ROPecker also checks forward for future gadget sequences (short sequences ending in indirection)
- These hacks can be circumvented by extending the ROP chains
  - Bottom line – no shortcuts

Control-Flow Graph

- Computing an accurate estimate of a CFG is intractable in general
  - Indirect calls (forward edges)
  - Returns (backward edges)
- Depends on predicting the value of a pointer
  - I.e., solving the points-to problem (undecidable)
- OK, maybe this is hard for function pointers (indirect calls), but this should be easy for returns, right?
  - You return to one of the possible callers
    - Generally, yes, but there are exceptions

Forward Edges

- How do we compute the possible targets for function pointers?

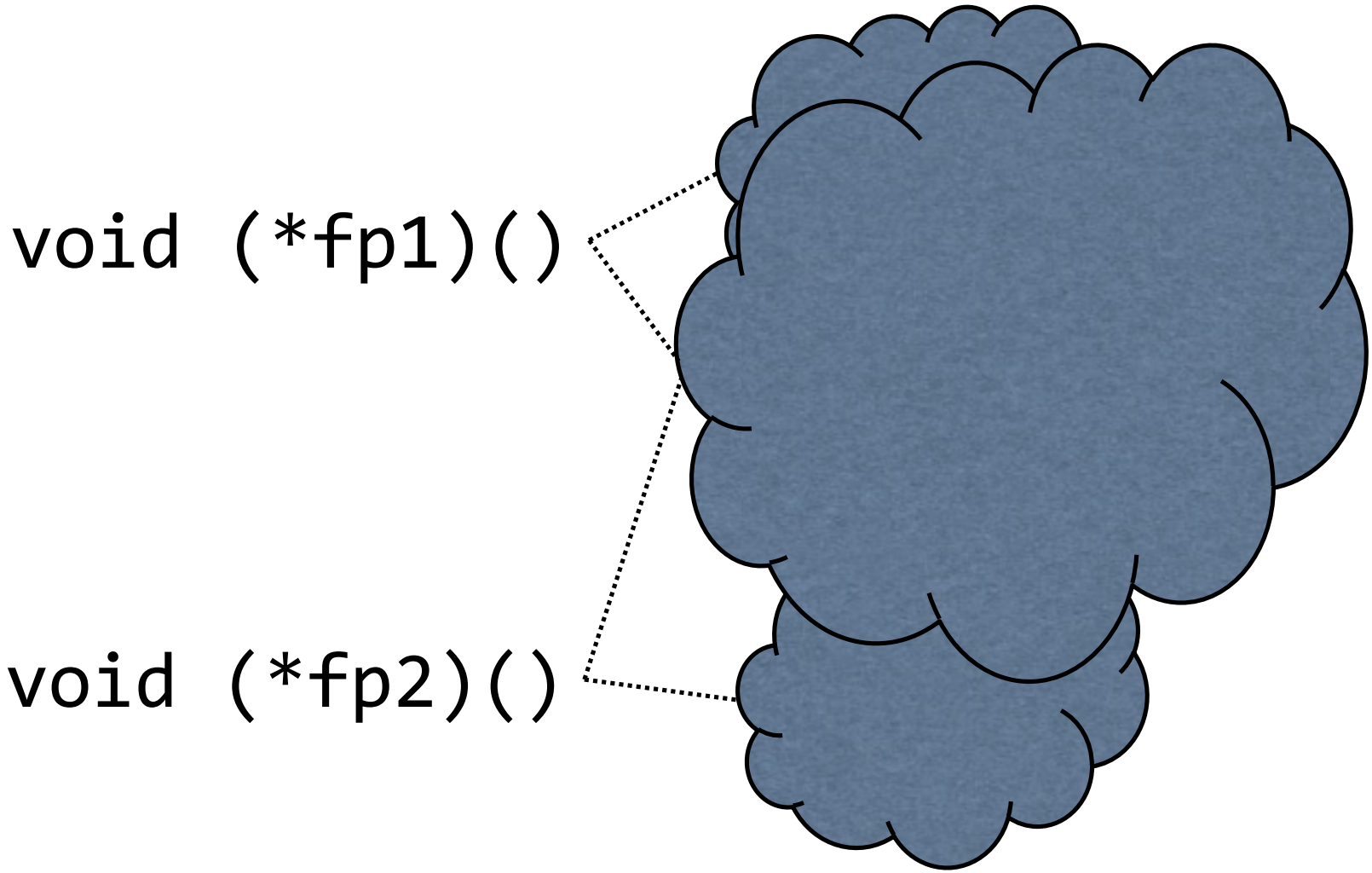
Forward Edges

- How do we compute the possible targets for function pointers?
- What are the possible legal targets of function pointers (i.e., indirect call sites)?

Forward Edges

- How do we compute the possible targets for function pointers?
- What are the possible legal targets of function pointers (i.e., indirect call sites)?
  - (1) Any function start
  - Called coarse-grained CFI
  - As this is the maximal set of legal function pointer targets, it is coarse

Coarse-grained CFI



Forward Edges

- How do we compute the possible targets for function pointers?
- What are the possible legal targets of function pointers (i.e., indirect call sites)?
  - (1) Any function
  - Called coarse-grained CFI
  - As this is the maximal set of legal function pointer targets, it is coarse
- This approach was applied by researchers – and then broken (easily) by other researchers
  - What are some options that would be more accurate?

Signature-based CFI

- How do we compute the possible targets for function pointers?
- What are the expected targets of an indirect call?
  - (2) Functions with the same type signature as the function pointer
  - Suppose you have a function pointer "int (*fn)(char *b, int n)"
    - Which functions should be assigned to that function pointer?

Signature-based CFI

- How do we compute the possible targets for function pointers?
- What are the expected targets of an indirect call?
  - (2) Functions with the same type signature as the function pointer
  - Suppose you have a function pointer "int (*fn)(char *b, int n)"
    - Which functions should be assigned to that function pointer?
- Compute the set of functions that share that signature assuming any of these can be a target
  - Fewer than all functions
  - Intuitively seems like an overapproximation
  - Can a function "void foo(void)" be assigned to "fn" above?

Taint-based CFI

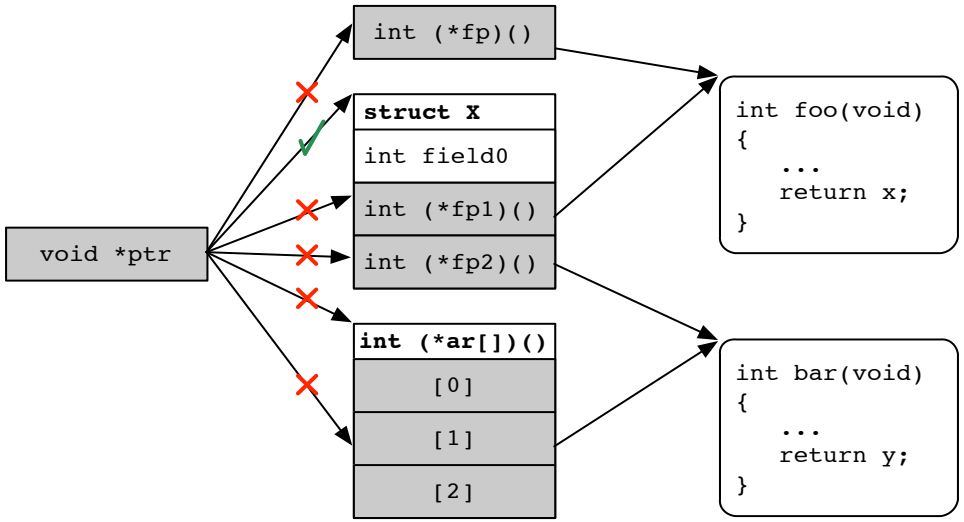
- How do we compute the possible targets for function pointers?
- What are the expected targets of an indirect call?
  - (3) Function targets that may reach indirect call sites
  - `fn = function_a;` // find definitions for function pointers
  - `…; fn(x);` // uses of function pointers (indirect calls)
  - And determine which assignments can reach which uses
- Problem
  - Taint analysis with points-to analysis may greatly overapproximate
  - Taint analysis without points-to analysis is not guaranteed to catch all

Assumptions

1. No arithmetic operations on function pointers

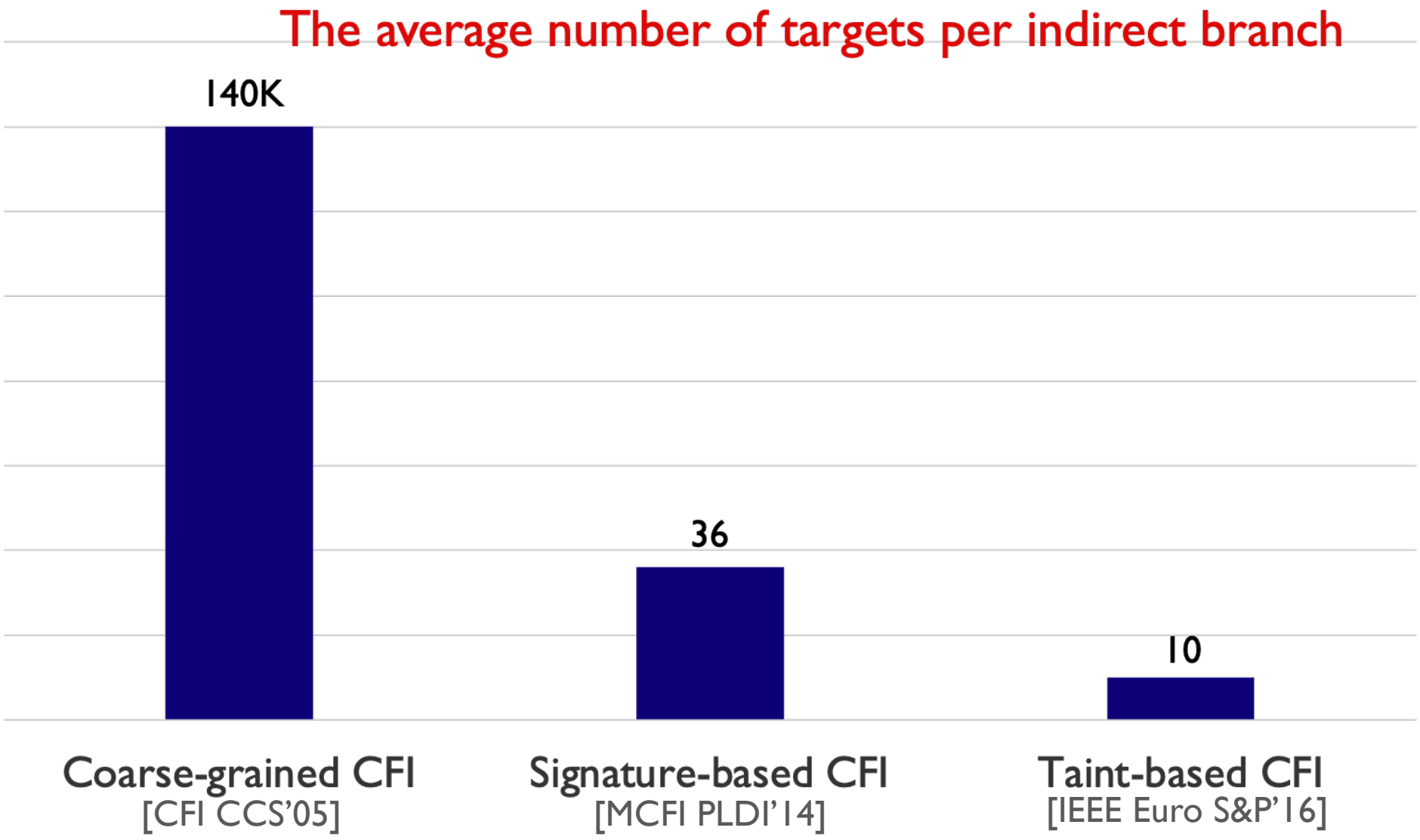
```
void (*fptr)(int) = &foo;
fptr += 10;
```

2. No data pointers to function pointers

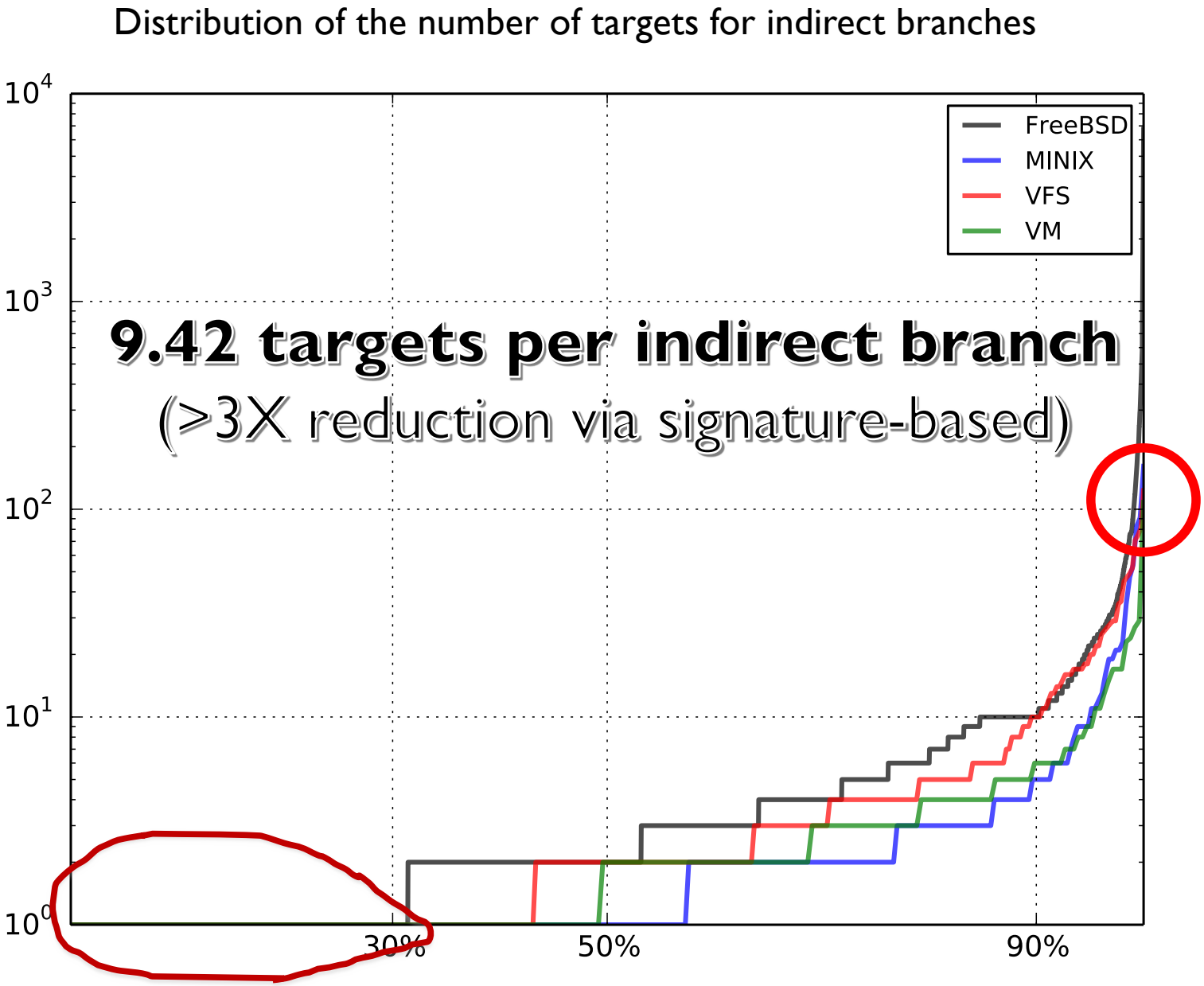


3. No type casts from data pointer types (int *) to function pointer types

Example: FreeBSD



Distribution of Taint Targets



Take Away

- Memory errors are the classic vulnerabilities in C programs (buffer overflow)
- Need two steps to exploit memory errors
  - Illegal memory write – often, but not always, initiated by overflow
  - Direct control flow – to adversary-chosen code
- Defenses have been proposed to prevent both steps
  - Bounds checks via bounds metadata and/or fat pointers
  - Control-flow integrity has been suggested as the way to block ROP attacks