CS165 – Computer Security

Memory Error Defenses

February 20, 2024

Memory Error Defenses

- We have discussed some
  - Canaries
  - Address Space Layout Randomization
  - Data Execution Protection (No Execute)
- How do these defenses work? Or fail to work?
  - Review

Memory Error Defenses

- We have discussed some
  - Canaries
  - Address Space Layout Randomization
  - Data Execution Protection (No Execute)
- These defenses do not prevent ROP attacks
  - Why not?

Memory Error Defenses

- We have discussed some
  - Canaries
  - Address Space Layout Randomization
  - Data Execution Protection (No Execute)
- These defenses do not prevent ROP attacks
  - Why not?
    - Bypass canaries and ASLR
      - Disclose canary values on stack
      - Disclose stack pointer values (EBP) to determine ASLR base
    - DEP/NX does not prevent execution of code memory

Defense for ROP Attacks

- There is a defense that prevents many ROP attacks
  - Called control-flow integrity

Control Hijack

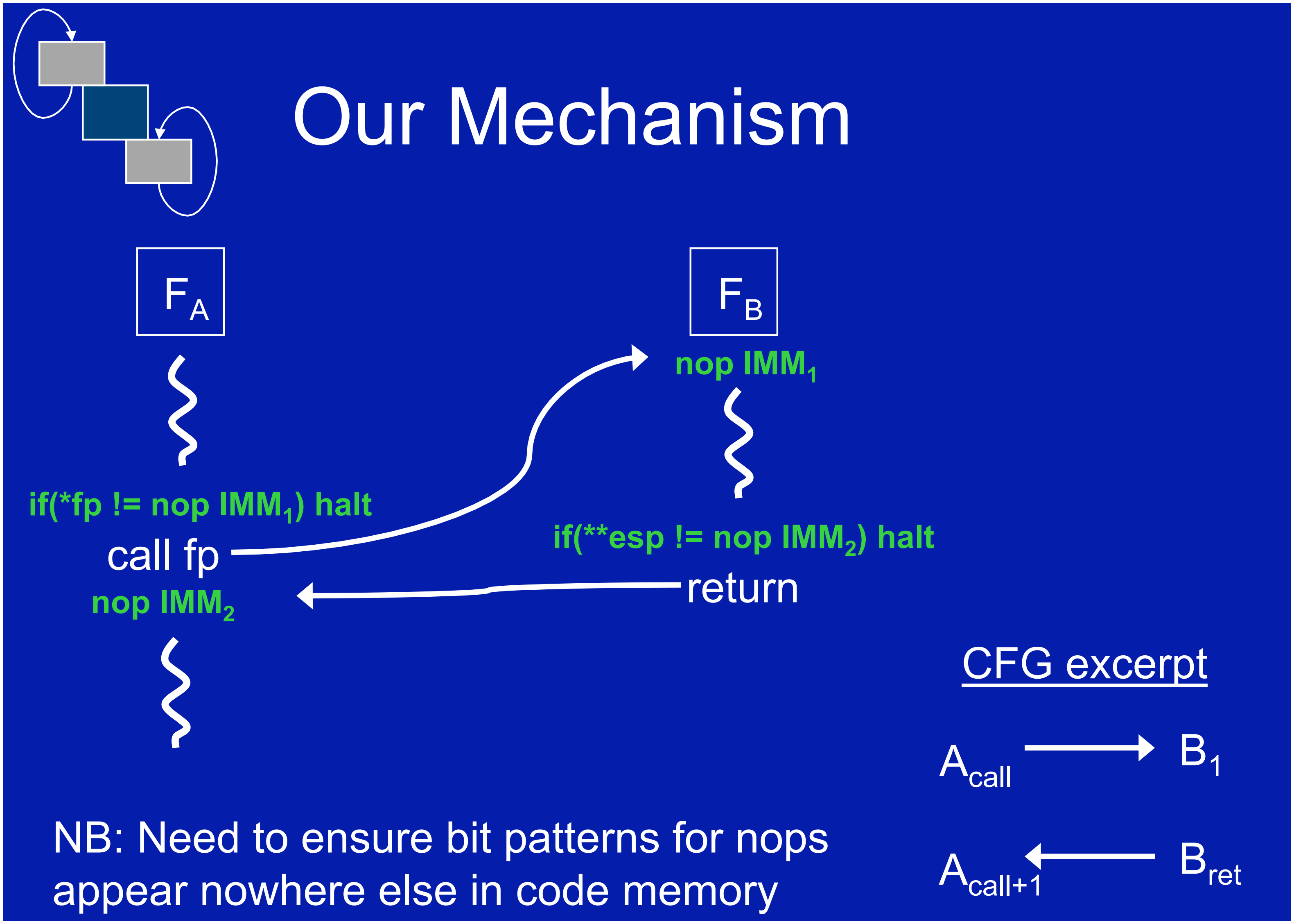
- Two main ways that C/C++ allows code targets to be computed at runtime
  - Return address (stack) – choose instruction to run on “ret” (i.e., function return)
    - *Why is the return address determined dynamically?*
  - Function pointer (stack or heap) – chooses instruction to run when invoked
    - Also called an indirect call
- If adversary can change either they can hijack control
- Difficult to prevent modification of code pointers
  - No broad defense at present (too expensive)

Indirect Call

- A function call using a function pointer
  - What happens?

```
int F_A()
{
  int (*fp)();
  …
  fp = &F_B;
  …
  fp();
  …
}
```

Control-Flow Integrity

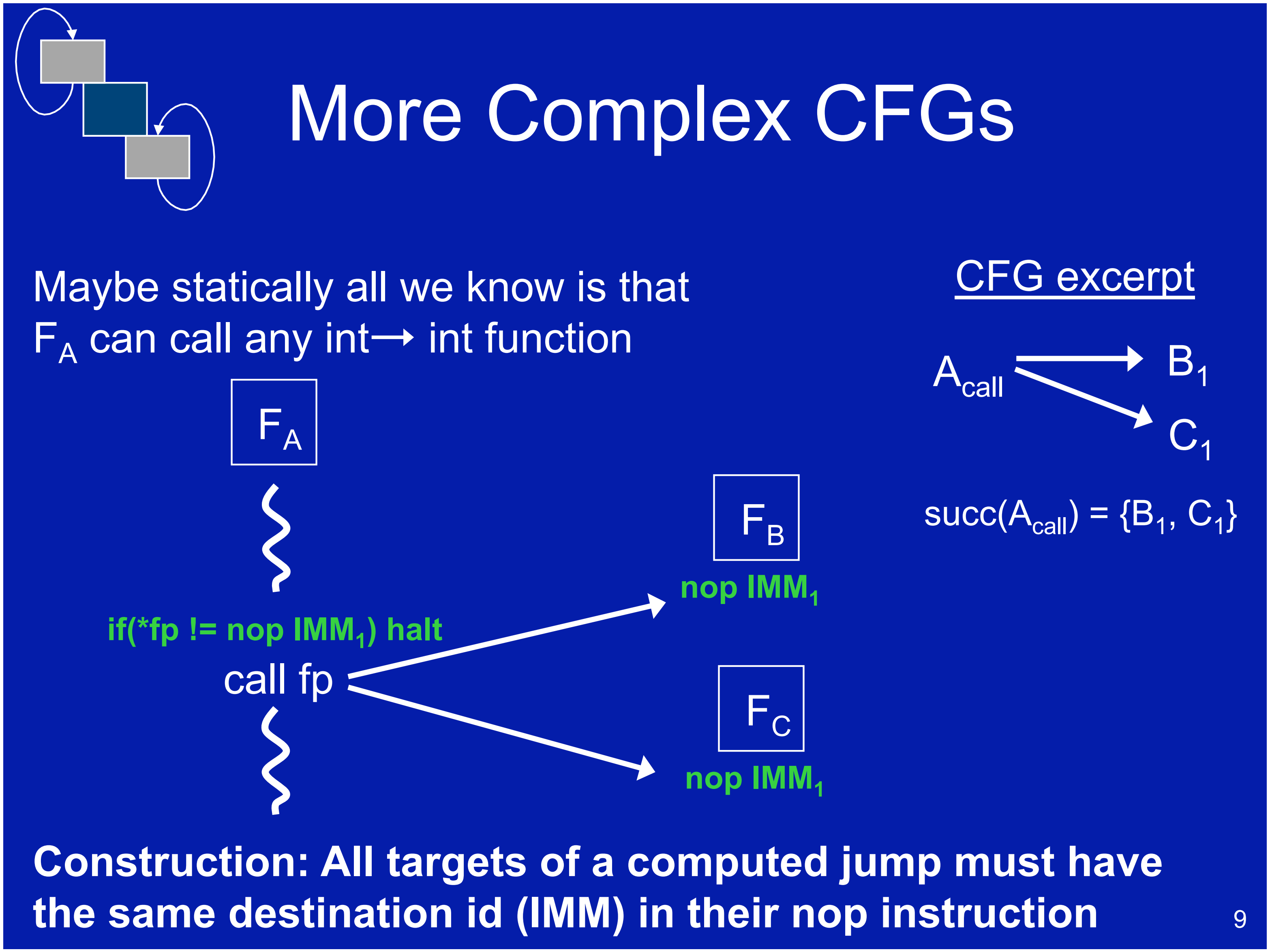


Indirect Call

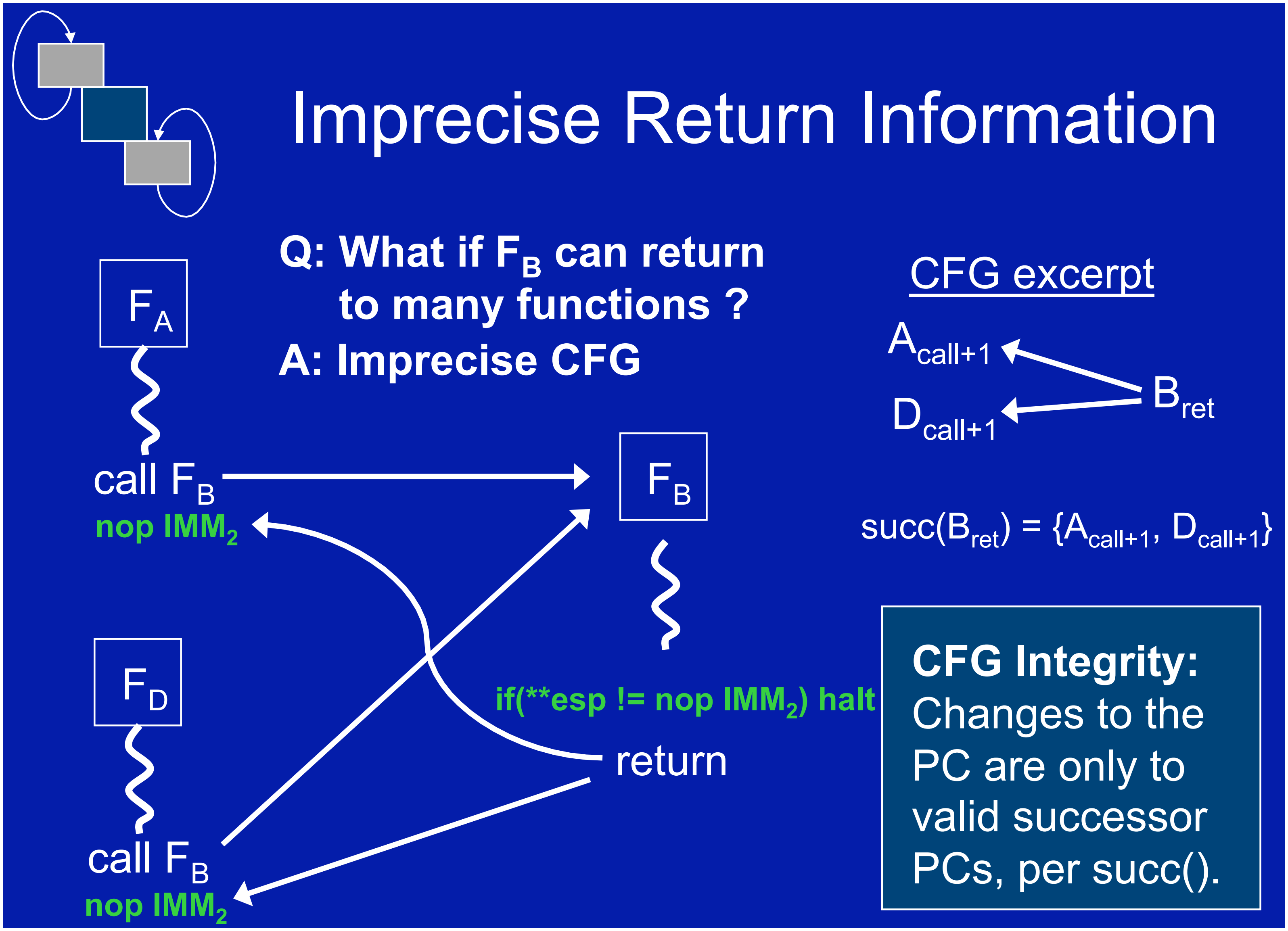
- A function call using a function pointer
  - What happens?

```
int F_A()
{
  int (*fp)();
  …
  if (a > 0) fp = &F_B;
  else fp = &F_C;
  …
  fp();
  …
}
```

Control-Flow Integrity



Control-Flow Integrity



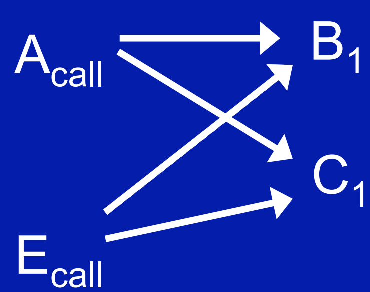
Control-Flow Integrity

No “Zig-Zag” Imprecision

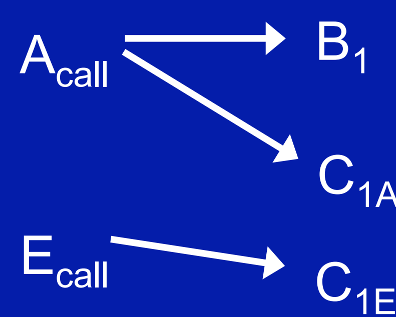
Solution I: Allow the imprecision

Solution II: Duplicate code to remove zig-zags

CFG excerpt

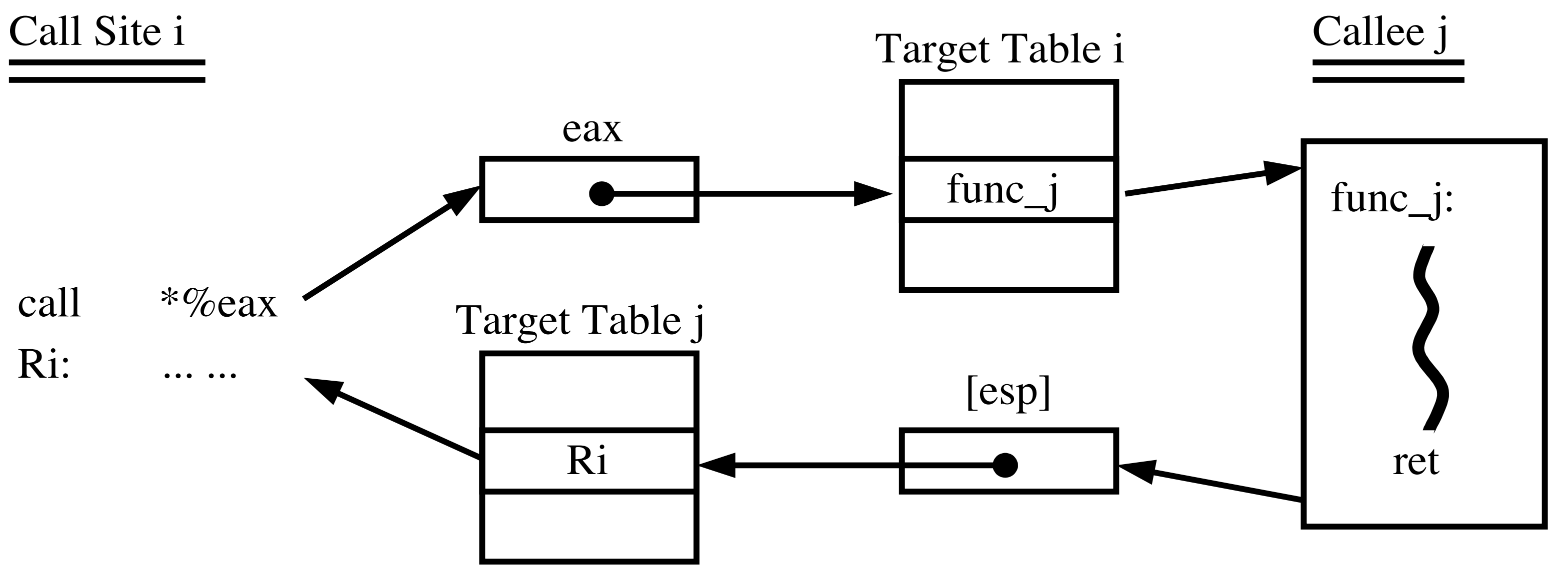


CFG excerpt



Restricted Pointer Indexing

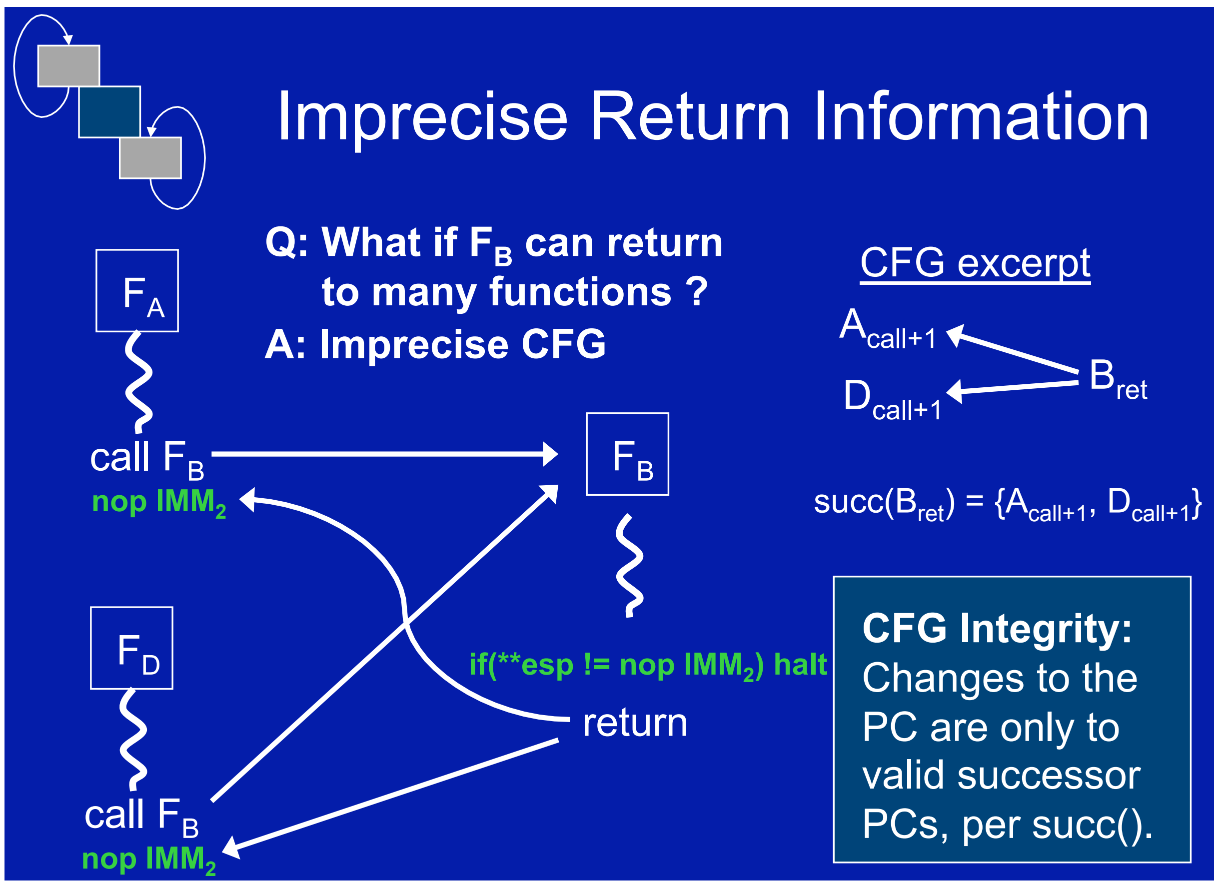
- One table for call and return for each call/return site



- Limit an indirect call to a predefined set of functions
  - Possible assignments to the function pointer for call site I
- Limit a return to a predefined set of callers
  - Only the callers of Callee j

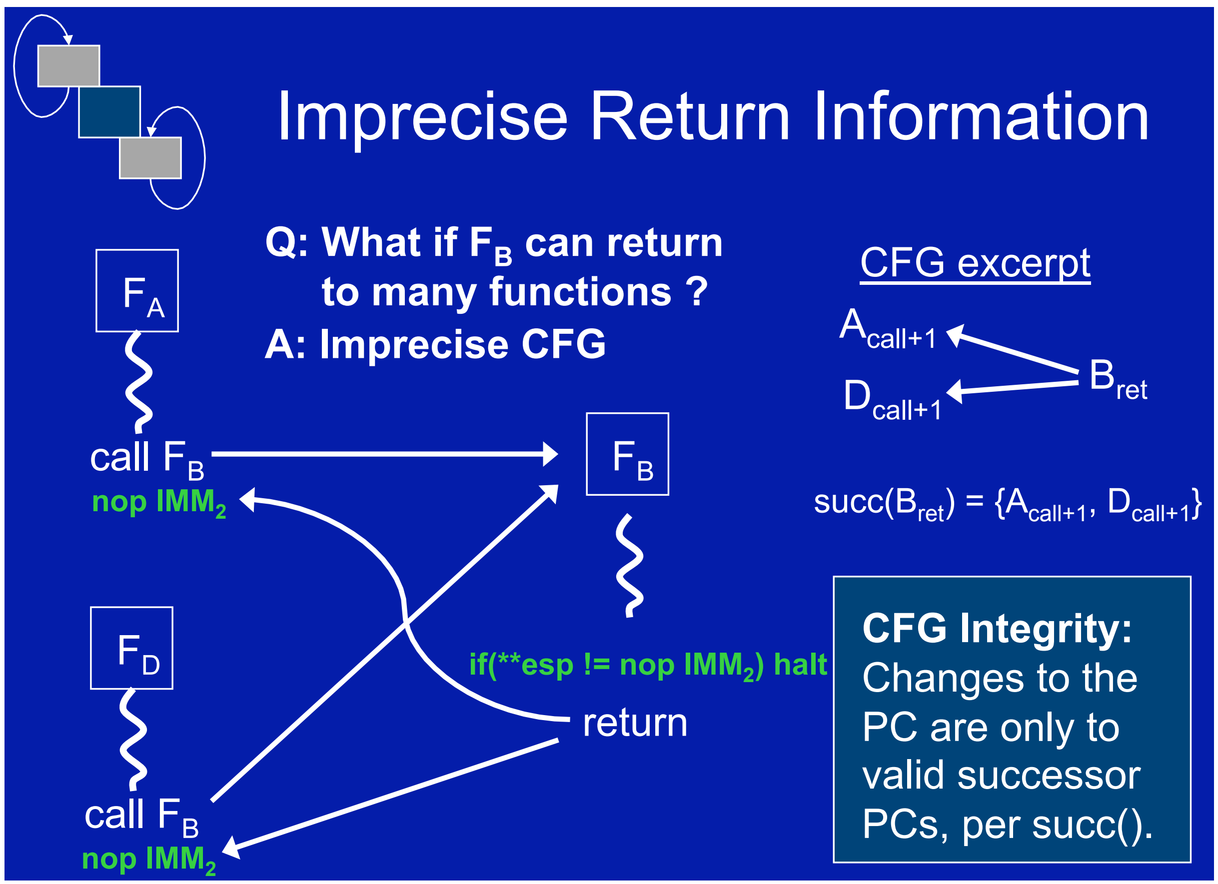
Limiting Returns

- Can't we do better for limiting returns
  - Don't we know where a return should go?



Shadow Stack

- Store the return address in a secure (shadow) location
  - Then, check that the return address matches the shadow



CFI Policies

- CFI limits the indirect call and return targets
  - But there are multiple CFI policies that may be enforced

CFI Policies

- CFI limits the indirect call and return targets
  - But there are multiple CFI policies that may be enforced
- Coarse CFI
  - What code locations could you execute from on a call?
  - Or return?

CFI Policies

- CFI limits the indirect call and return targets
  - But there are multiple CFI policies that may be enforced
- Coarse CFI
  - Any function start (for indirect calls)
    - That is, a function pointer can be used to call any function
  - Follow any call site (for returns)
    - A return address can return to any call site
- Reduces the fraction of instructions significantly
  - But, does not prevent attacks in practice
  - Why?

CFI Policies

- CFI limits the indirect call and return targets
  - But there are multiple CFI policies that may be enforced
- Fine CFI
  - Want to reduce the set of indirect call and return targets to those that are really possible
  - What can we do for calls/returns?

CFI Policies

- Fine CFI
  - For calls: match function pointers with functions of the same function signature
    - Signature: return type, number of arguments, argument types

CFI Policies

- Fine CFI
  - For calls: match function pointers with functions of the same function signature
    - Signature: return type, number of arguments, argument types
  - Suppose you have the function pointer declaration
    - `void (*fun_ptr)(int);`
  - Which function could be a legal target?
    - `void *function(int x)`
    - `void function1(int *x)`
    - `void function2(int y1, int y2)`
    - `void function3(int z)`

CFI Policies

- Fine CFI
  - For calls: match function pointers with functions of the same function signature
    - Signature: return type, number of arguments, argument types
  - Suppose you have the function pointer declaration
    - `void (*fun_ptr)(int);`
  - Which function could be a legal target?
    - `void *function(int x)`
    - `void function1(int *x)`
    - `void function2(int y1, int y2)`
    - `void function3(int z)`

CFI Policies

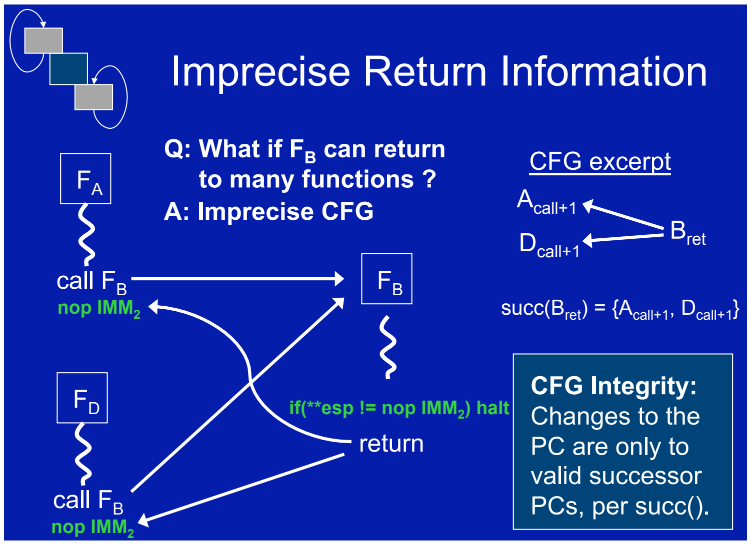
- Fine CFI
  - For returns: Always return to the call site that invoked the function
    - How do we ensure that?

CFI Policies

- Fine CFI
  - For returns: Always return to the call site invoked
    - Shadow stack
      - Record return address in a safe location
      - Check return address against shadow value on return
      - Now implemented in Intel CET hardware

CFI Policies

- Fine CFI
  - For returns: Always return to the call site invoked
    - Shadow stack
      - Record return address in a safe location
      - Check return address against shadow value on return
      - Now implemented in Intel CET hardware



Prevent All ROP attacks?

- Does CFI prevent all ROP attacks?

Prevent All ROP attacks?

- Does CFI prevent all ROP attacks?
  - No. CFI cannot detect attacks that use legal control flows

Prevent All ROP attacks?

- Does CFI prevent all ROP attacks?
  - No. CFI cannot detect attacks that use legal control flows
- E.g., change a data pointer value used in a system call
  - Consider `open(char *file)`
  - If we can change the "file" pointer to reference an adversary-controlled string, we can achieve our goal
    - Without changing the program's control flow

Shouldn’t we just fix memory errors?

- Can you find the flaw(s)?

```
int
im_vips2dz( IMAGE *in, const char *filename ){
  char *p, *q;
  char name[FILENAME_MAX];
  char mode[FILENAME_MAX];
  char buf[FILENAME_MAX];
  ...

  im_strncpy( name, filename, FILENAME_MAX );
  if( (p = strchr( name, ':' )) ){
    *p = '\0';
    im_strncpy( mode, p + 1, FILENAME_MAX );
  }

  strcpy( buf, mode );
  p = &buf[0];
  ...
}
```

Dynamic Analysis Options

- Regression Testing
  - Run program on many normal inputs and look for bad behavior in the responses
    - Typically looking for behavior that differs from expected – e.g., a previous version of the program
- Fuzz Testing
  - Run program on many abnormal inputs and look for bad behavior in the responses
    - Looking for behaviors that may cause the program to stop executing at all – crash or hang

Dynamic Analysis Options

- Why might fuzz testing be more appropriate for finding vulnerabilities?

Dynamic Analysis Options

- Why might fuzz testing be more appropriate for finding vulnerabilities?
  - Memory errors often lead to crashes
  - Other errors may cause the program to hang

Fuzz Testing

- Fuzz Testing
  - Idea proposed by Bart Miller at Wisconsin in 1988
- Problem: People assumed that utility programs could correctly process any input values
  - Accessible to all
- Found that they could crash 25-33% of UNIX utility programs

Fuzz Testing

- Basic Approach
  - Generate random inputs
  - Run programs using lots of random inputs
  - Detect program crashes
  - Correlate with the random inputs that caused the crashes
- Detect inputs that cause crashes

Example Found

- Fuzz Testing
  - Produce random inputs for processing

```
format.c (line 276):
...
while (lastc != '\n') {
rdc();
}
...

input.c (line 27):
rdc()
{ do { readchar(); }        // assigns 'lastc' to 0 on EOF
while (lastc == ' ' || lastc == '\t'); return (lastc);
}
```

  - Eventually produce line with EOF in the middle

Fuzz Testing

- Idea: Search for flaws in a program by running the program under a variety of inputs
- Challenge: Selecting input values for the program
  - What should be the goals in choosing input values for fuzz testing?

Challenges

- Idea: Search for flaws in a program by running the program under a variety of inputs
- Challenge: Selecting input values for the program
  - What should be the goals in choosing input values for fuzz testing?
  - *Find as many exploitable flaws as possible*
  - *With the fewest possible input values*
- How should these goals impact input choices?

Black Box Fuzzing

- Like Miller – Feed the program random inputs and see if it crashes
- Pros: Easy to configure
- Cons: May not search efficiently
  - May re-run the same path over again (low coverage)
  - May be very hard to generate inputs for certain paths (checksums, hashes, restrictive conditions)
  - May cause the program to terminate for logical reasons (fail format checks and stop)

Black Box Fuzzing

- May be difficult to pass "authenticate_user" and "check format" with random inputs to get to "update"

```
function( char *name, char *passwd, char *buf )
{
if ( authenticate_user( name, passwd )) {
    if ( check_format( buf )) {
        update( buf );
    }
  }
}
```

Grey Box Fuzzing

- Rather than treating the program as a black box, instrument the program to track the paths run
- Save inputs that lead to new paths
  - Mutate off those inputs to generate inputs
  - To bias toward running new paths
- Example
  - American Fuzzy Lop (AFL)
- "State of the practice" at this time

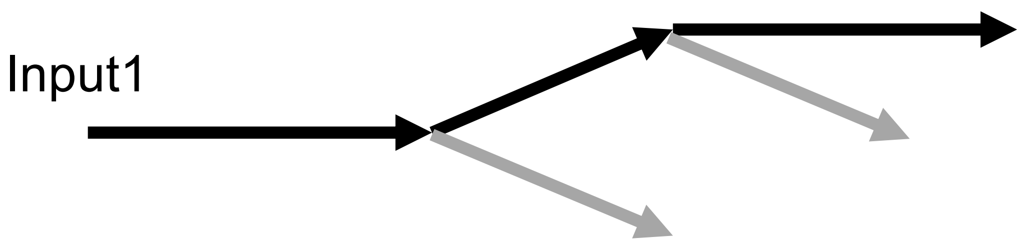
AFL

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats

- See
  - http://lcamtuf.coredump.cx/afl/

AFL Instrumentation

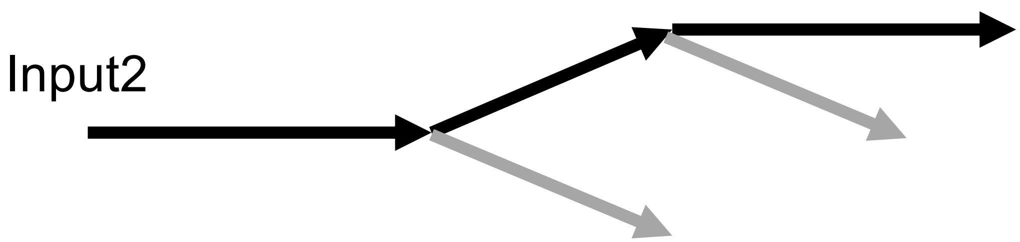
- Instrument conditional statements to track the paths executed – and detect new paths



- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt

AFL Instrumentation

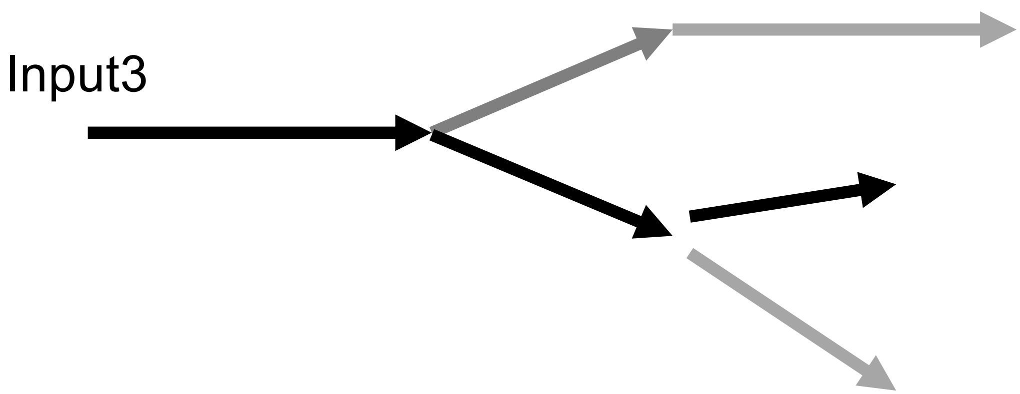
- Instrument conditional statements to track the paths executed – and detect new paths



- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt

AFL Instrumentation

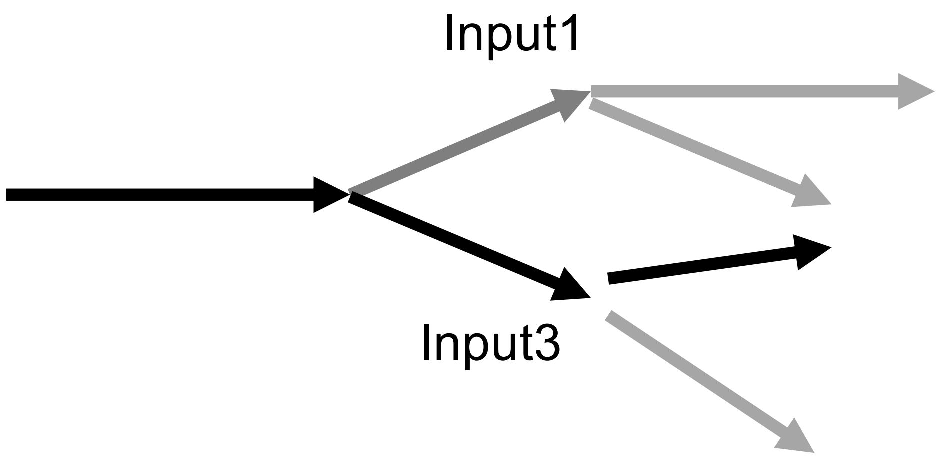
- Instrument conditional statements to track the paths executed – and detect new paths



- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt

AFL Instrumentation

- Instrument conditional statements to track the paths executed – and detect new paths



- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt

AFL Use

- Run the fuzzer using afl-fuzz

```
path-to/afl-fuzz -i <input-dir> -o <output-dir> <path-to-bin> [args]
```

- For example

```
path-to/afl-fuzz -i input/ -o output/ ./cs165-p3 @@ outfile
```

- Where
  - `input/` directory with the input file
  - `output/` is the directory where the AFL results will be placed
  - `@@` shows that the arg (input file) to be fuzzed
- Output stats about coverage and inputs for hangs/crashes

AFL Results

- Shows the results of the fuzzer
  - E.g., provides inputs that will cause the crash

```
                 american fuzzy lop 2.51b (cmpsc497-p1)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 2 hrs, 16 min, 32 sec      │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 13 min, 31 sec      │  total paths : 41     │
│ last uniq crash : 0 days, 0 hrs, 43 min, 58 sec      │ uniq crashes : 11     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 3 (7.32%)         │    map density : 0.11% / 0.40%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.62 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 8/8             │ favored paths : 6 (14.63%)             │
│ stage execs : 12.3k/41.9k (29.31%)  │  new edges on : 7 (17.07%)             │
│ total execs : 243k                  │ total crashes : 2479 (11 unique)       │
│  exec speed : 30.98/sec (slow!)     │  total tmouts : 10 (5 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 7/15.4k, 32/15.4k, 0/15.4k            │    levels : 3          │
│  byte flips : 0/1929, 0/1926, 0/1920                │   pending : 39         │
│ arithmetics : 8/71.7k, 4/5434, 0/0                  │  pend fav : 5          │
│  known ints : 0/6938, 0/35.5k, 0/56.3k              │ own finds : 40         │
│  dictionary : 0/0, 0/0, 0/1270                      │  imported : n/a        │
│       havoc : 0/178, 0/0                            │ stability : 17.69%     │
│        trim : 0.00%/930, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 19%]
```

Why Not Use Safe Languages?

- A "type safe" language cannot have memory errors
  - E.g., Java (older) and Rust (recent)
- Also, "memory safe" versions of C have been proposed

Why Not Use Safe Languages?

- A safe language is "safe" with respect to what requirements?

Why Not Use Safe Languages?

- A safe language is “safe” with respect to what requirements?
  - Spatial, temporal, and type
- Java programs must satisfy all three classes of safety
  - Via runtime checks (spatial and type) and garbage collection (temporal)
- Rust “safe” programs must satisfy all three classes too
  - Via runtime checks (spatial and type) and a specialized mechanism to track the live pointers to an object (temporal)
- May have ”unsafe” Rust code also – no guarantees

Issues to Overcome

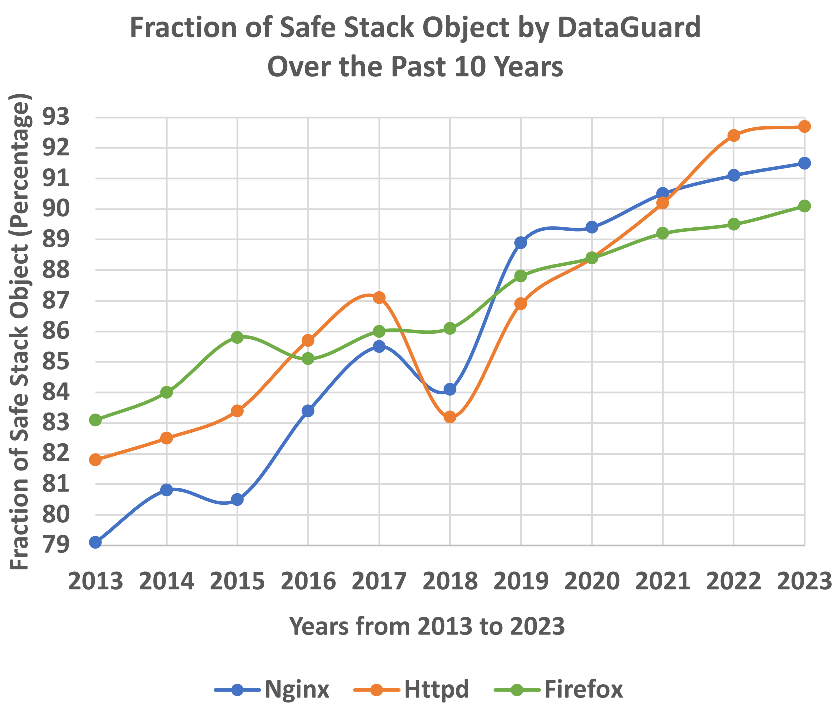
- Usability
  - Early "memory safe" C languages were not popular
  - C# is still less popular than C/C++
- Performance
  - Type-safe languages incur overhead from checks to ensure safety
  - Java has a significant overhead compared to C
  - Story: JavaOS project
- Functionality
  - May use unsafe C libraries
  - JVM is written in C

Rust

- Usability
  - Has concepts to manage temporal safety (ownership)
  - Requires type-safe usage (more effort to program)
- Performance
  - Has runtime checks to enforce spatial safety
  - But, appears to require fewer checks than for C
- Functionality
  - Allows the definition of “unsafe” Rust modules
  - Uses C libraries
- Efforts to replace some C code in Linux with Rust

C Is Getting Safer

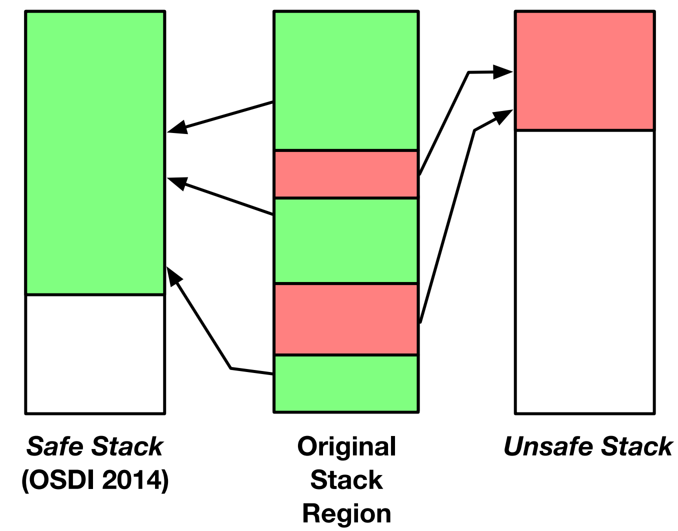
- Likely due to fuzz testing, the fraction of C objects whose accesses are all memory safe is increasing



- Over 90+% of stack objects
  - Over 75+% of heap objects likewise

Isolate Memory-Safe Objects

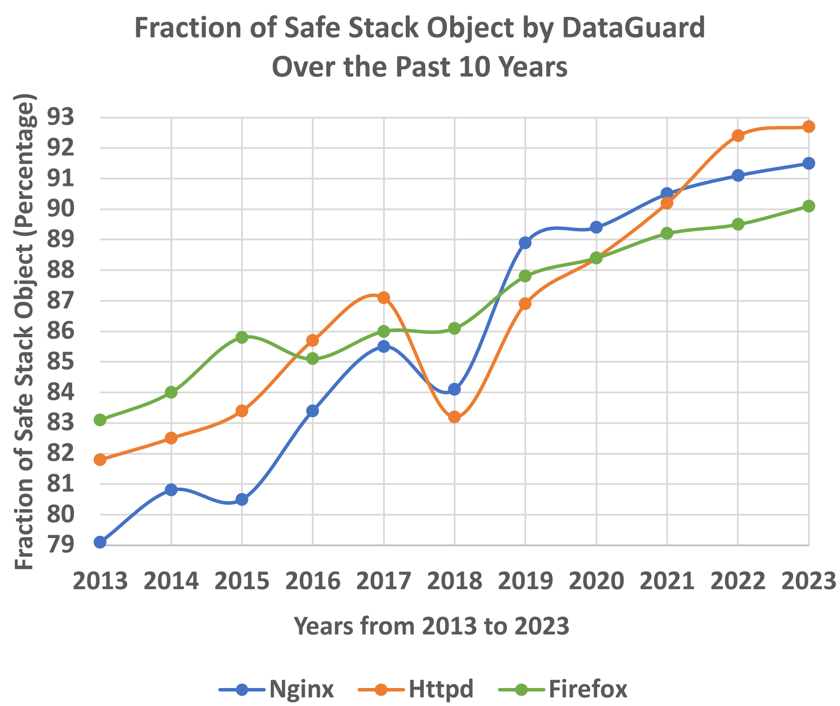
- Memory safe objects can be protected by isolation
  - All accesses in "safe" region must be safe
  - No references to safe region from "unsafe" objects/region
  - → Safe region is safe from memory errors



- Protected by ASLR or new hardware cheaply
- Issue: May have to protect unsafe cases too

C Is Getting Safer, But…

- Likely due to fuzz testing, the fraction of C objects whose accesses are all memory safe is increasing



- But, manual code and AI-generated C code is currently much less safe

Conclusions

- Can improve resilience to attack on memory errors
- Control-flow integrity
  - Limit control flows restrict ROP attacks
    - But, can still launch attacks that follow legal control flows
- Fuzz testing
  - Systematic approach to test programs for crash/hang
    - But, cannot achieve complete coverage
- Safe languages
  - Memory errors are not possible in these languages
    - But, impact on usability, performance, functionality

Questions