CS165 – Computer Security

Advanced Memory Error Defenses

October 29, 2025

Memory Error Defenses

- We have discussed some
  - Canaries
  - Address Space Layout Randomization
  - Data Execution Protection (No Execute)
- Do these defenses work?

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 (e.g., EBP) to decode ASLR
      - Exploit function pointers other than the return address
    - DEP/NX does not prevent execution of code memory

Control Hijacking

- 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

Protect the Return Address

- There is a defense that prevents the return address from being modified without detection
  - More reliable than stack canaries
  - Called shadow stack

Shadow Stack

- Idea: Check whether the return address has been modified directly
  - Not use a separate item like a canary
- On Call: record the value of the return address in a safe memory location (i.e., the "shadow")
- On Return: compare the value of the return address to be assigned to the %eip to the "shadow" recorded
  - Reject unless they match

Why Not Do This Already?

- Idea: Check whether the return address has been modified directly
  - Not use a separate item like a canary
- Seems like an obvious and easy defense
  - But the performance of recording the return address twice
  - And protecting the shadow return address from modification
  - Is significantly higher than the canary defense
- What can we do if a software defense is easy, but expensive?

Intel CET

- Implement the defense in hardware
- Specifically, Intel Control-Flow Enforcement Technology (CET)
  - Implements shadow stack (and more)
  - To prevent return-oriented programming attacks
  - Windows supports Intel CET
  - So do Linux compilers (gcc and clang)
    - With the `-fcf-protection` flag

Control Hijack w/ Function Ptrs

```
int main()
{
    int (*f)() = &function;
    int val = f();
    return val;
}
```

- If an adversary can modify the value of variable “f”, then they can choose which code to run (e.g., gadget)

Defense for ROP Attacks

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

Defense for ROP Attacks

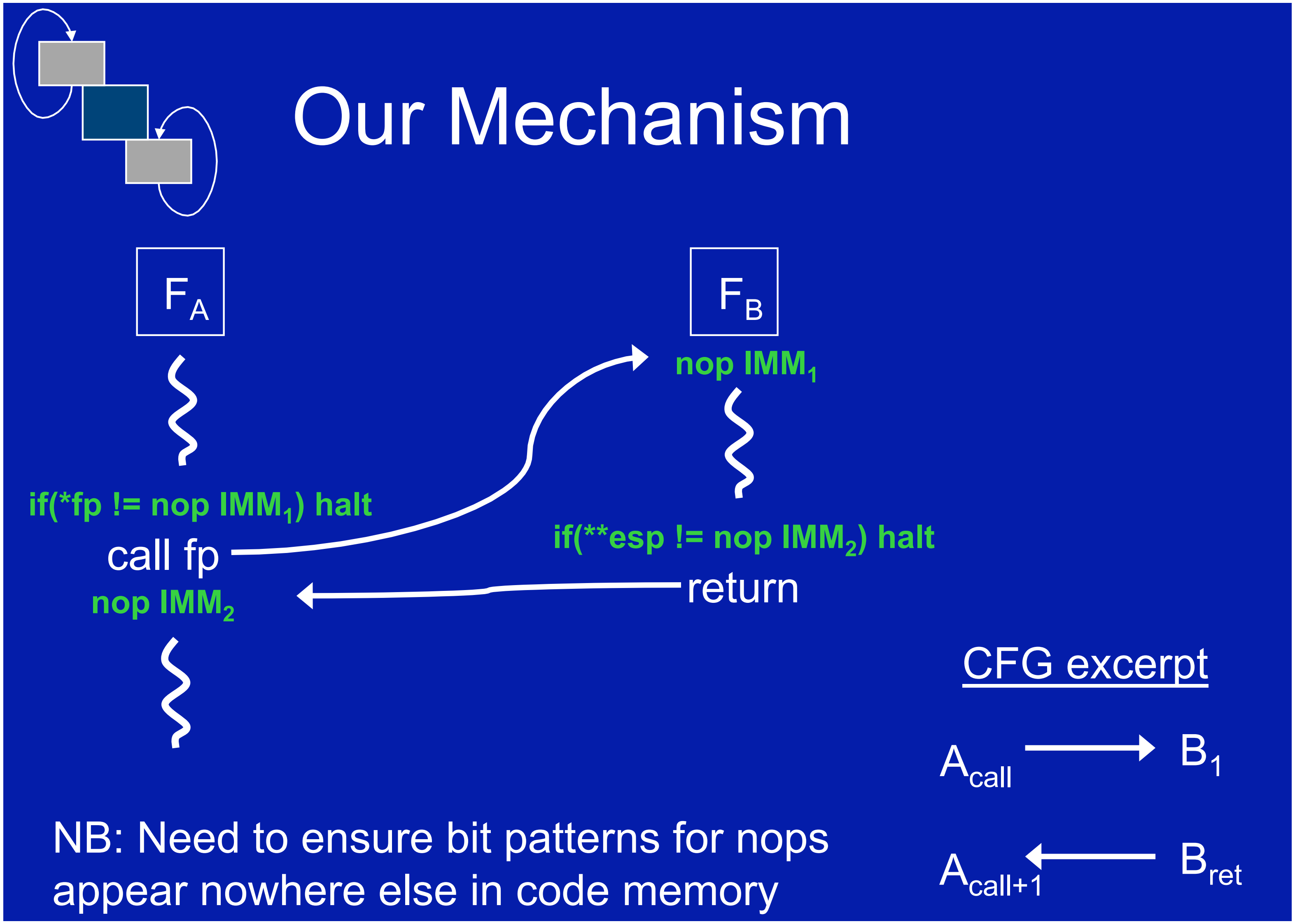
- There is a defense that prevents many ROP attacks
  - Called control-flow integrity
- Control-flow integrity restricts the values of function pointers to only those that are legally possible
  - Given the program code

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

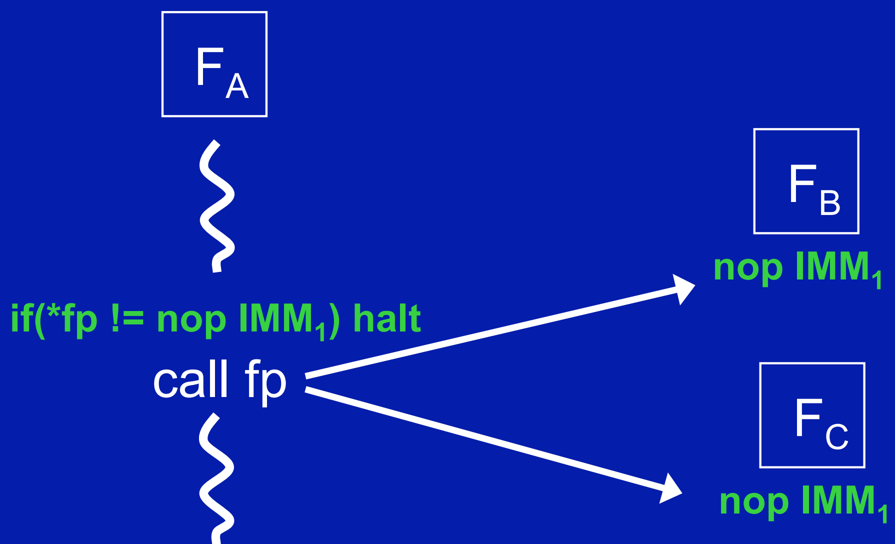
- A function call using a function pointer
  - What happens?

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

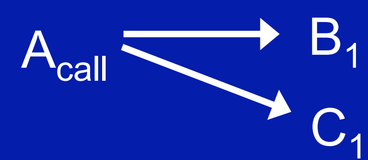
Control-Flow Integrity

More Complex CFGs

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



CFG excerpt



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

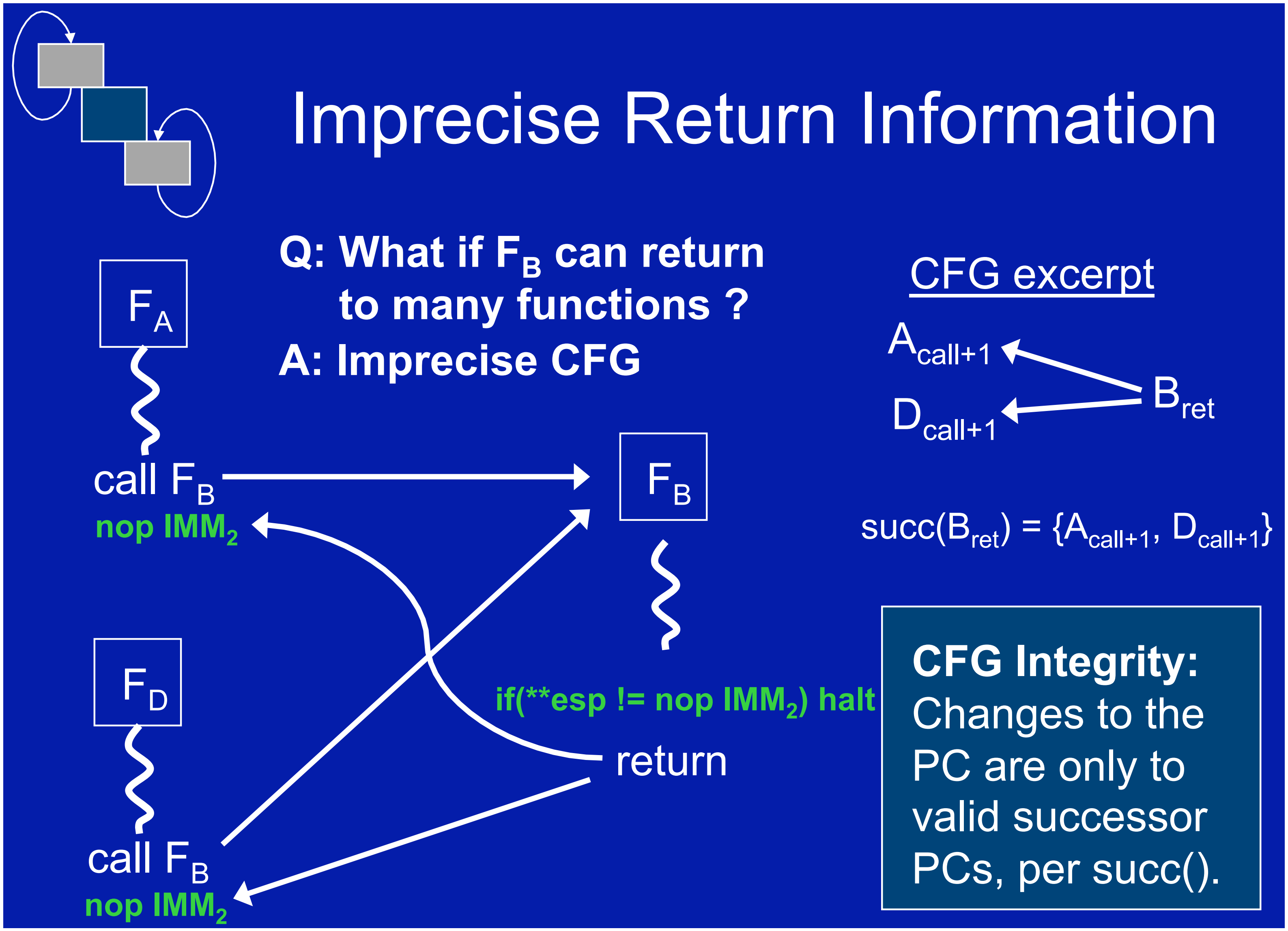
Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction

Indirect Call

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

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

Control-Flow Integrity



Indirect Call

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

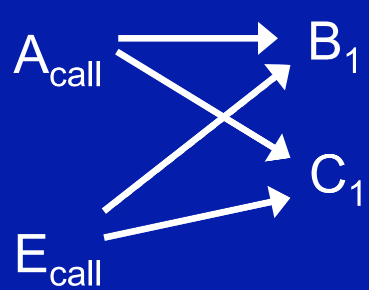
```
int F_D()
{
   int (*fp)();
   …
   fp = &F_B;
   …
   fp();
   …
}
```

Control-Flow Integrity

No “Zig-Zag” Imprecision

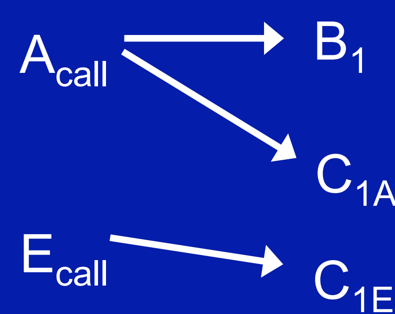
Solution I: Allow the imprecision

CFG excerpt



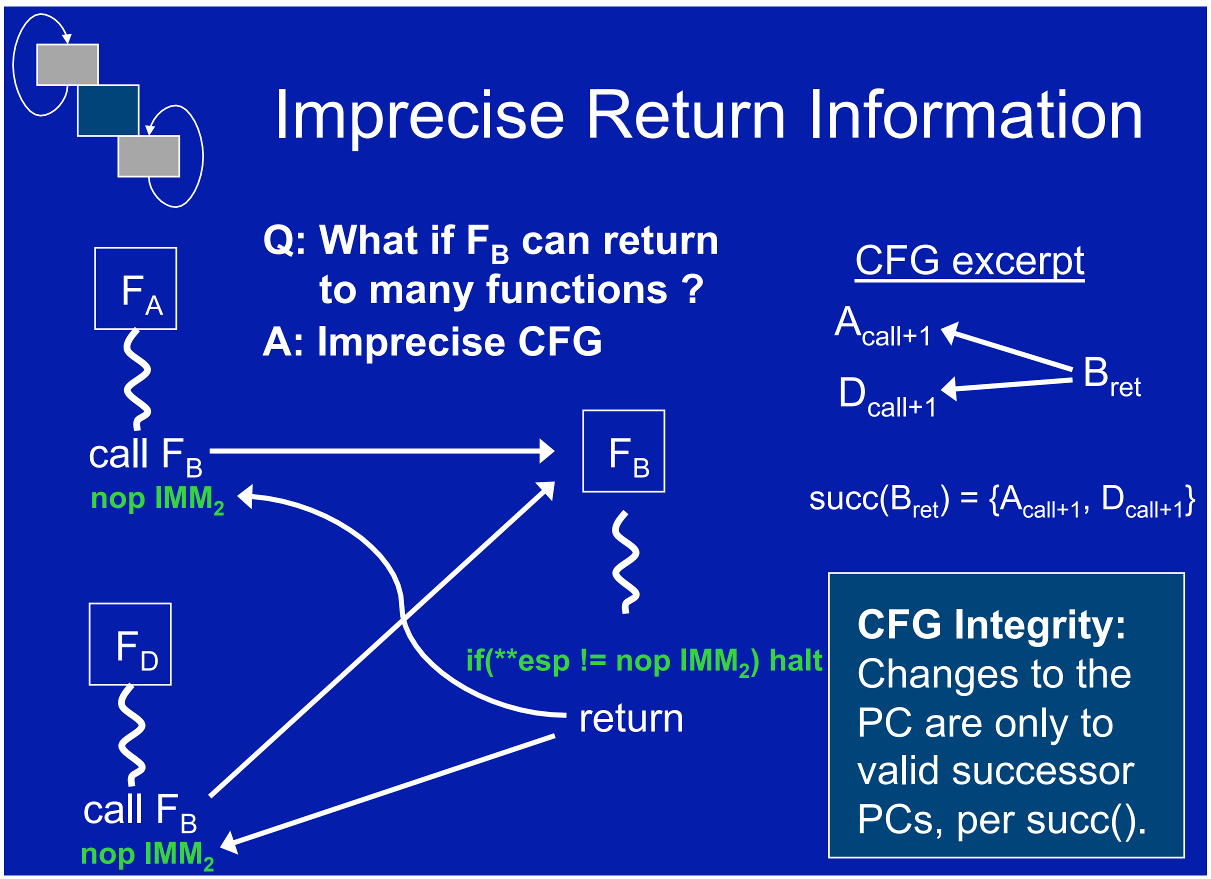
Solution II: Duplicate code to remove zig-zags

CFG excerpt



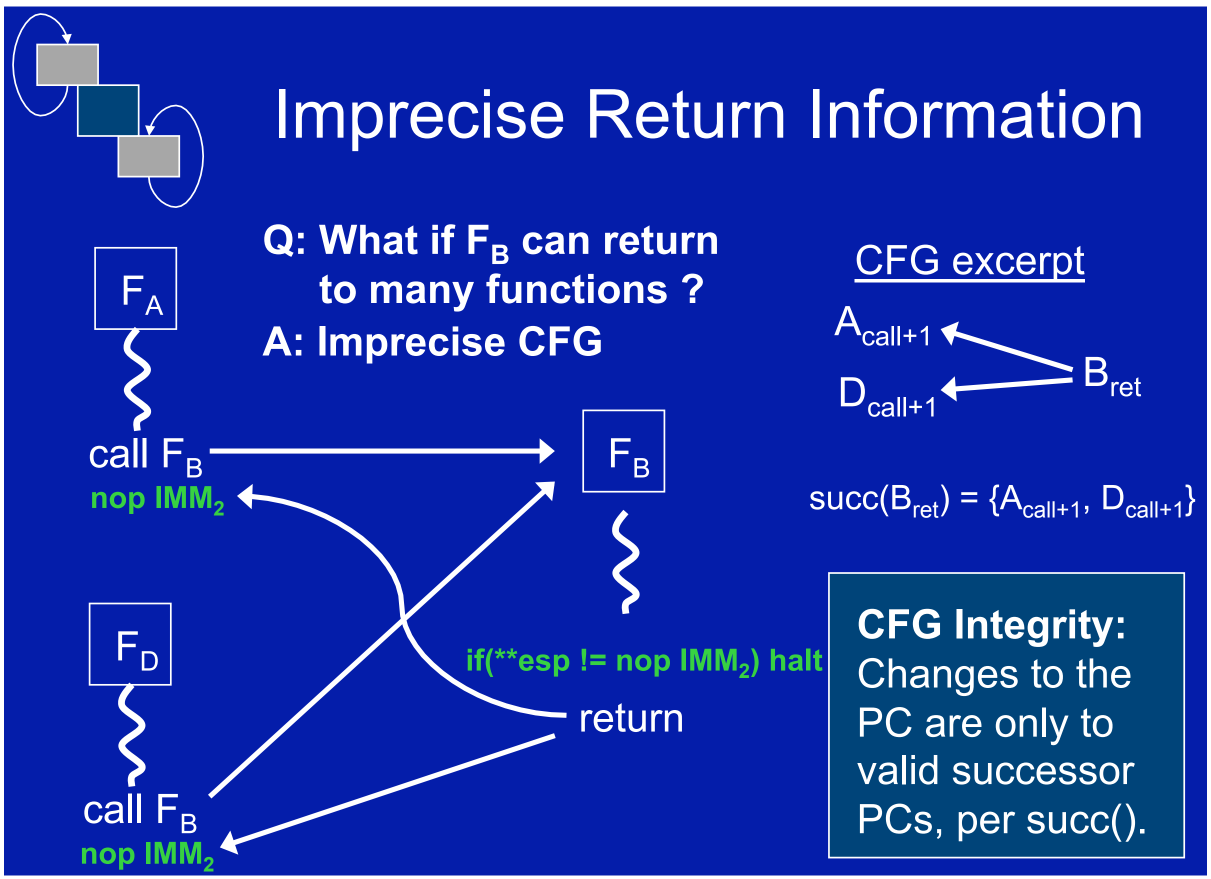
Limiting Returns

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



CFI Enforces 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 ever legitimately be the target of a call instruction?
  - Or a 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

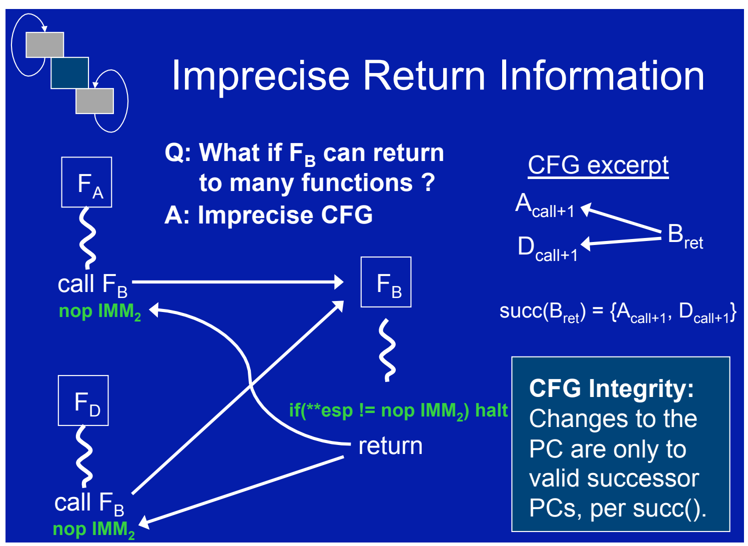
- Fine CFI
  - For returns: Always return to the call site that invoked the function
    - How do we ensure that?

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



Intel CET and CFI

- Intel Control-Flow Enforcement Technology (CET)
  - Implements shadow stack
    - On returns
  - And coarse CFI
    - On indirect calls
  - Linux compiler support (gcc and clang)
    - With the `-fcf-protection` flag

Conclusions

- Can improve resilience to attack on memory errors
  - Prevent return-oriented attacks
- Shadow stack
  - Ensure that return address cannot be modified
    - Ensure function returns to its caller
- Control-flow integrity
  - Limit program control flows to those in program
    - Limit to legal function pointer values
- Doesn't prevent all exploits, but reduces many attack vectors – and is now available

Questions