Architectural Support for Security

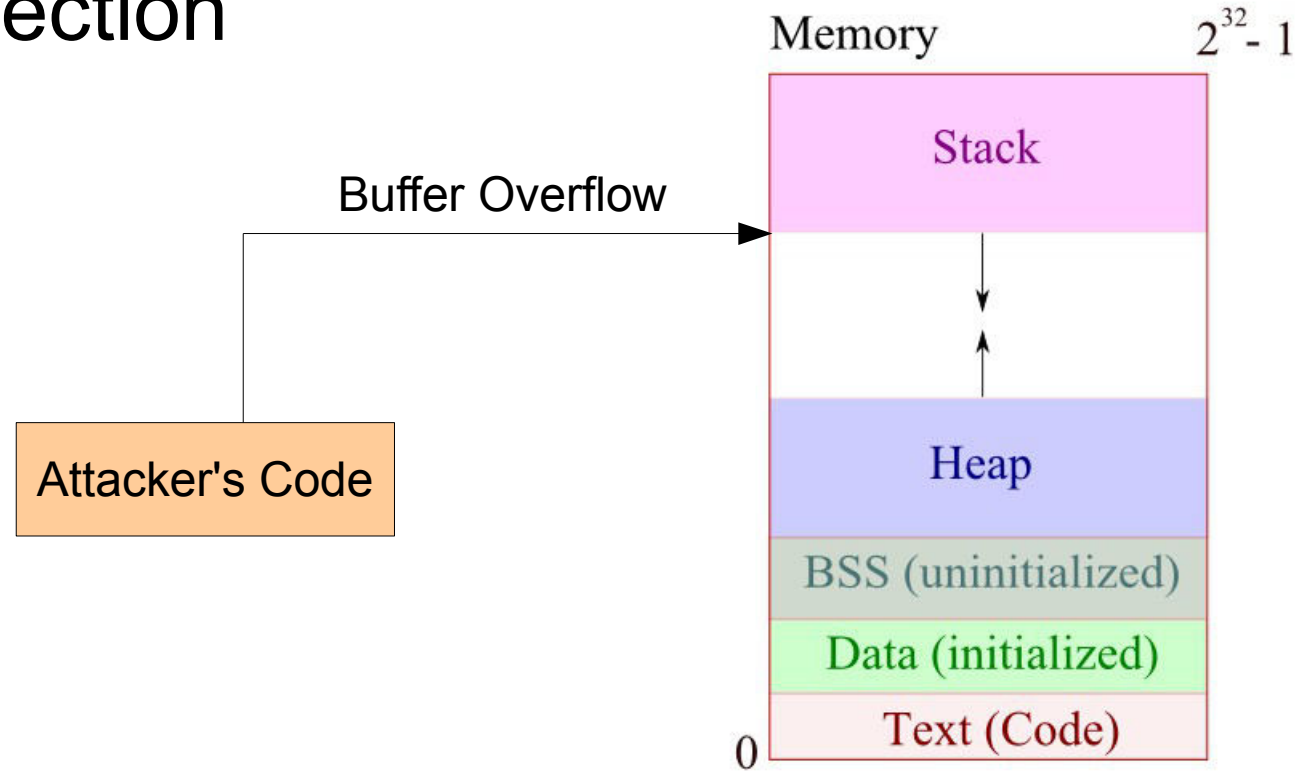
Tim Loffredo

Branch Regulation: Low-Overhead Protection from Code Reuse

- Prevents "Code Reuse Attacks"
  - CRAs are a BIG PROBLEM!
- New Architectural Component: Secure Call Stack
- Good Performance (2% Overhead)
- But First...

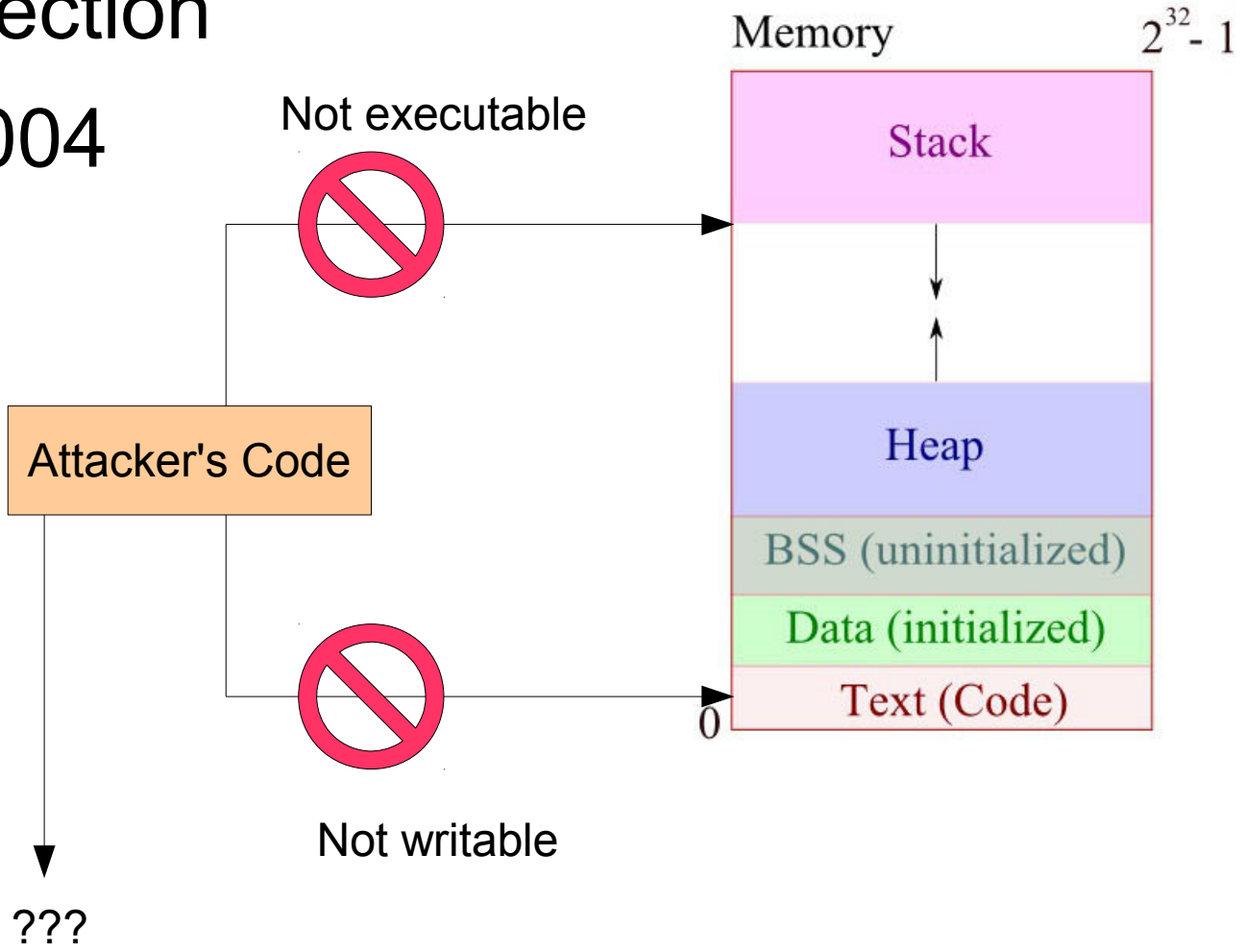
A Little History

- Code Injection



A Little History

- Code Injection
- DEP ~2004

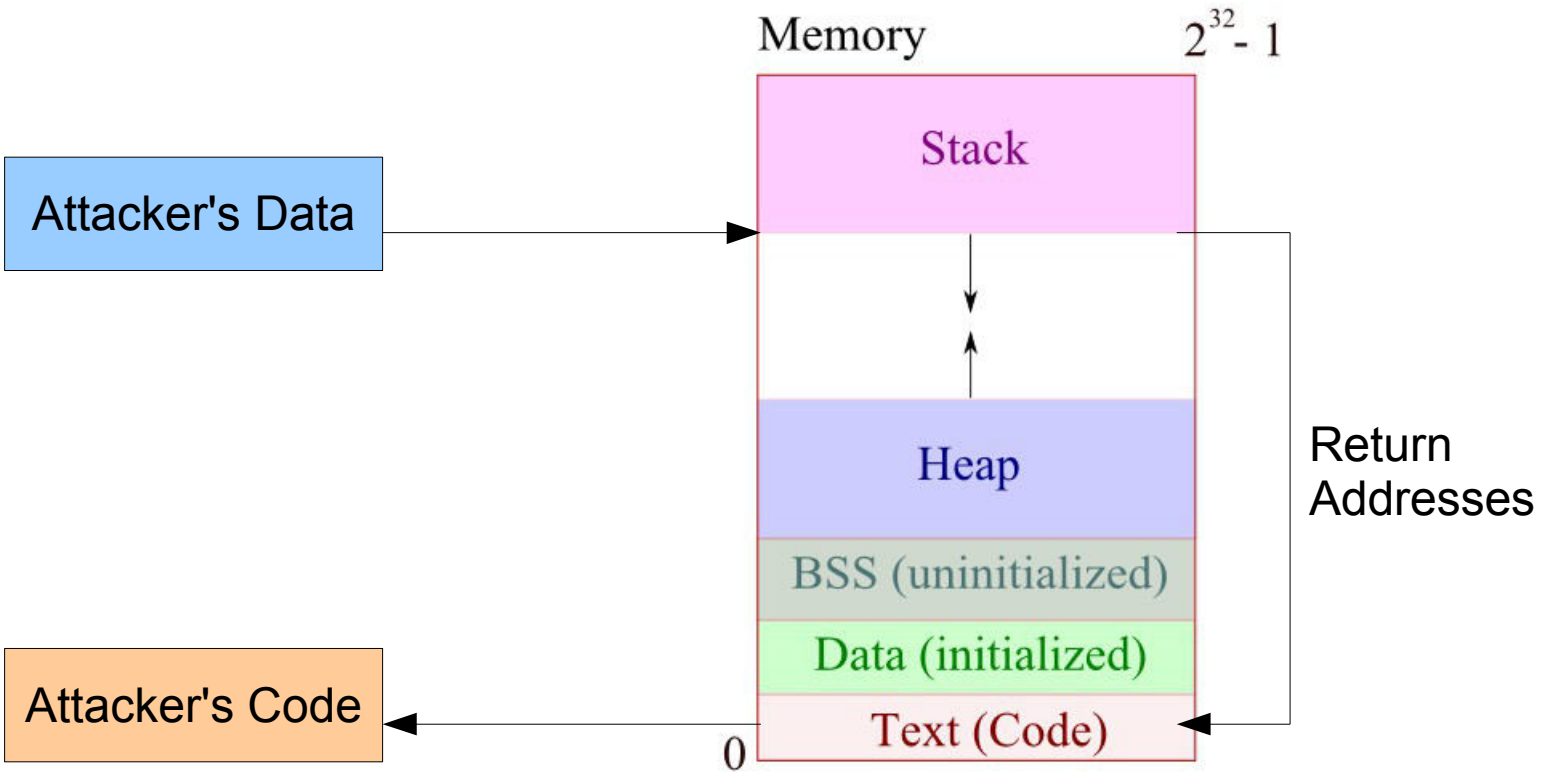


A Little History

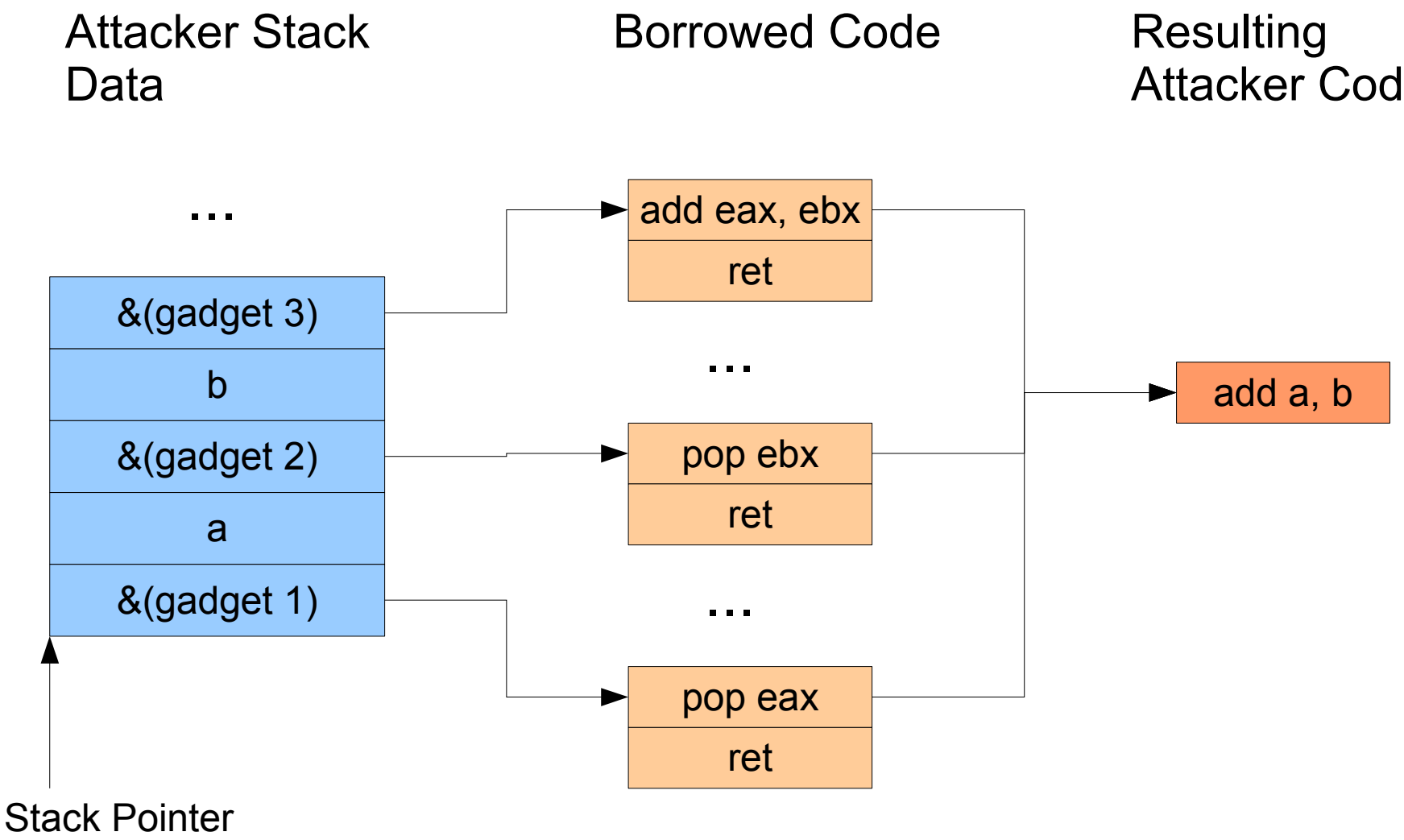
- "The Geometry of Innocent Flesh on the Bone" by Hovav Shacham, 2007
  - Return to Libc by Solar Designer, 1997
- Code Reuse Attack, aka ROP

Return Oriented Programming

- ROP "borrows" code from the exploited application to create the attacker code



Return Oriented Programming



Return Oriented Programming

- Takeaway Point:

  Attackers reuse code to circumvent DEP
  - Usually with ROP
  - Also circumvents code signing

Back to Branch Regulation

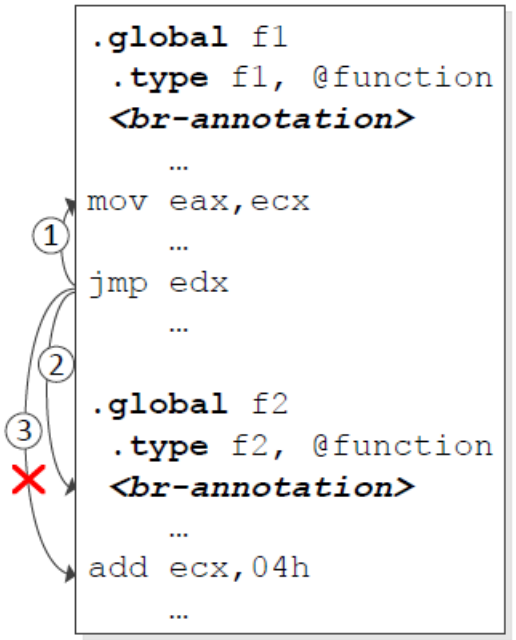
- Prevents code reuse attacks
- Hardware performs a check on every indirect branch
  - "ret" instructions
  - "jmp <blah>" instructions
  - "call <blah>" instructions

Branch Regulation

- Call and ret are simple cases
- On "call <blah>":
  - Verify that <blah> is a valid function entry point
  - Record next instr address (we will return there)
- On "ret":
  - Verify that we are returning to an address recorded by a previous valid call

Branch Regulation

- General indirect jump is hard to regulate
  - Compilers do weird things...
  - Authors chose an OK heuristic
- On "jmp <blah>":



① JMP inside of the function

✓ Function Base < Target < Function Bound

② JMP to a new function

(Function Bound < Target Address)

✓ Target = <br-annotation>

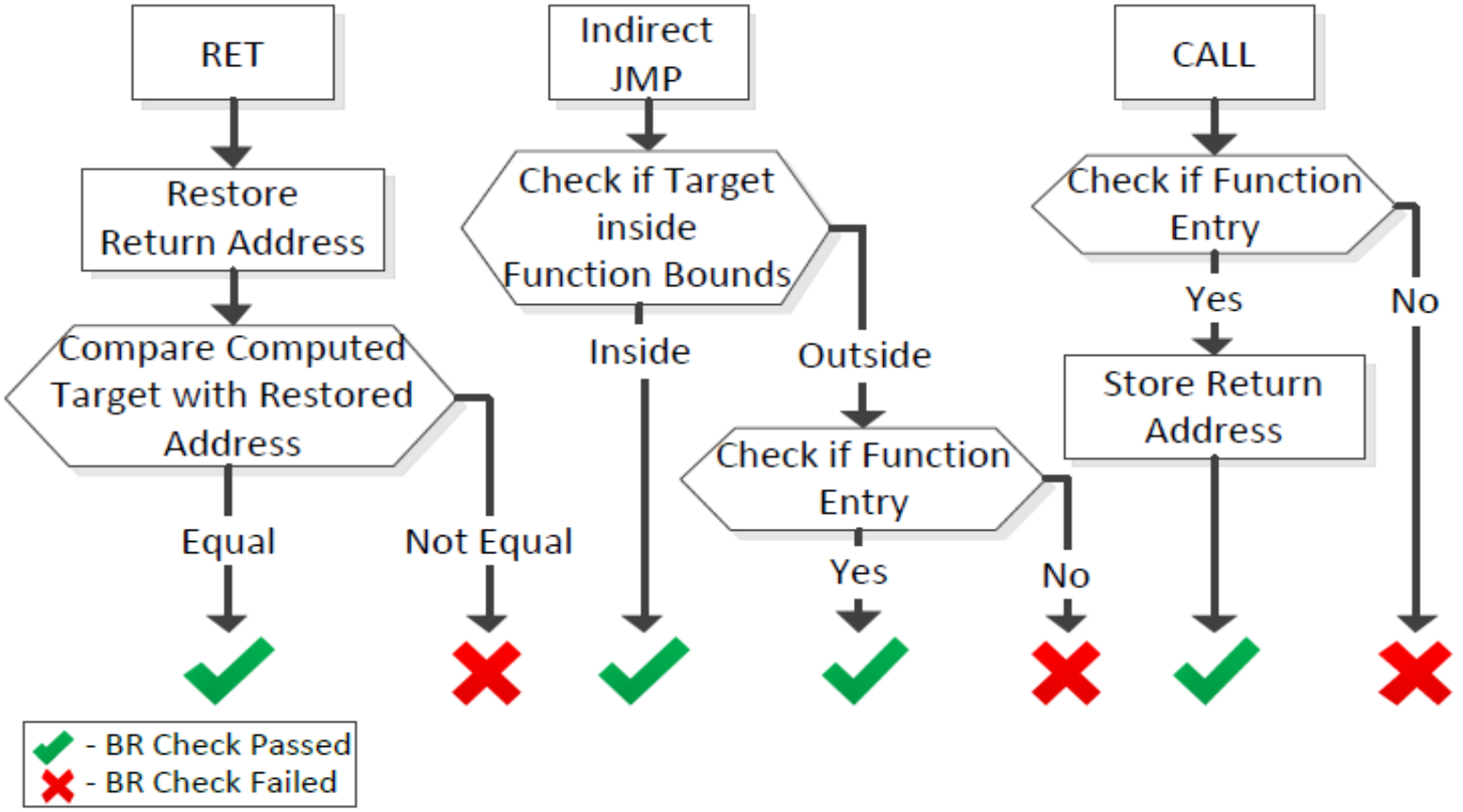
③ JMP to middle of another function

(Function Bound < Target Address)

✗ Target Address ≠ <br-annotation>

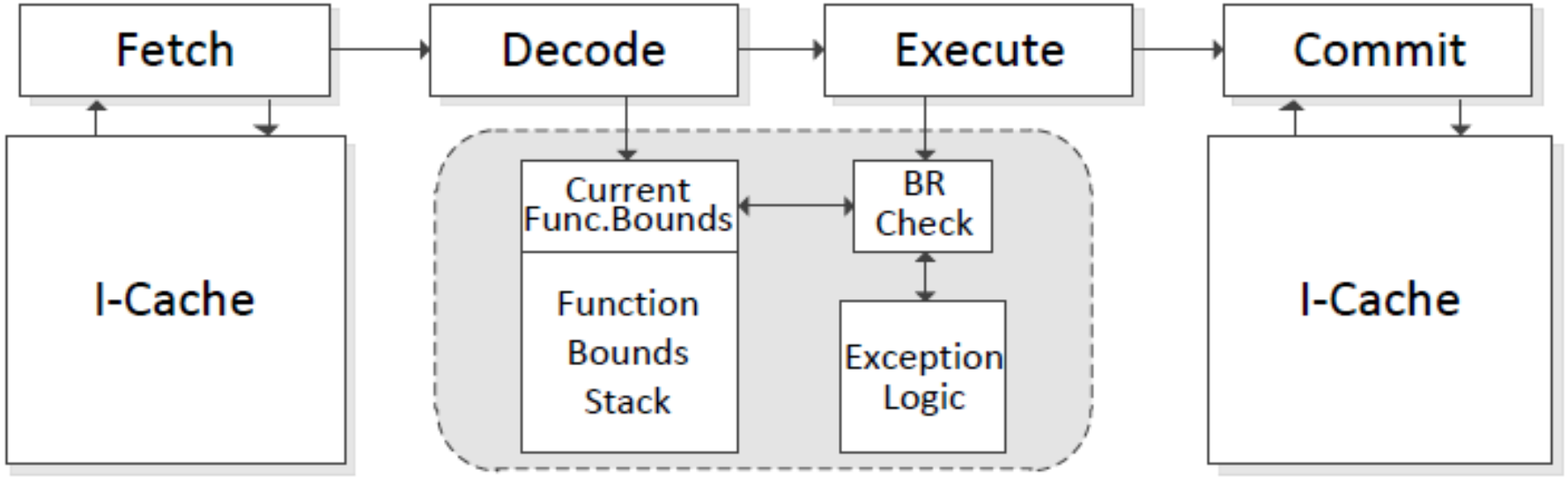
Putting It Together

- Need Secure Call Stack
- Need Function Boundary Annotations



How?

- Special Hardware in Pipeline

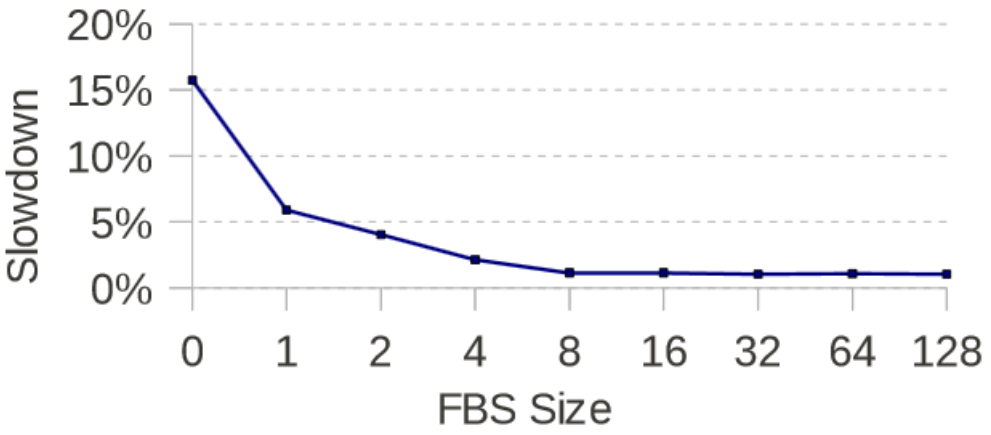


- Function Bounds Stack =

  Secure Call Stack + Function Start / End

Performance

- Performance Overhead: 1-2%
  - Cuz it executes in parallel!
  - Measured by simulation
- Foundation Bound Stack Size: Only 16 Entries



Effectiveness

- Constrain RET targets: gadgets can't chain
  - 99% reduction in available gadgets
- Effectively stops ROP
  - ... in the 5 binaries the authors looked at
- Should *slow* ROP regardless
  - ROP programming goes from Hard to Infeasible

Security Analysis

- Paper Makes Assumptions
  - All Exploits Use a Syscall
  - All Exploits Need a "Dispatcher" Gadget
- Not as good as full Control Flow Integrity (CFI)
  - Why not take that extra step?
    - Because CFI requires compiler-level static analysis
- Extra Note:

  The "security" of a system is difficult to measure

Other Paper: kBouncer

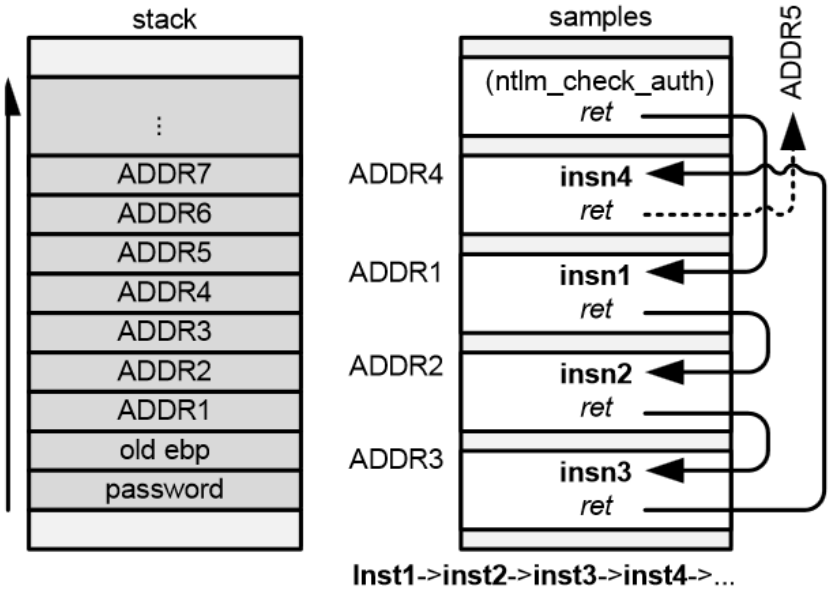
- Uses Last Branch Recording (LBR) Registers
  - Existing Hardware
- Checks LBR for ROP on Syscalls
- Runtime Overhead: ~1%
- Limitations:
  - User Space Unprotected
  - Syscall Boundary Can Be Fooled

Other Paper: CFIMon

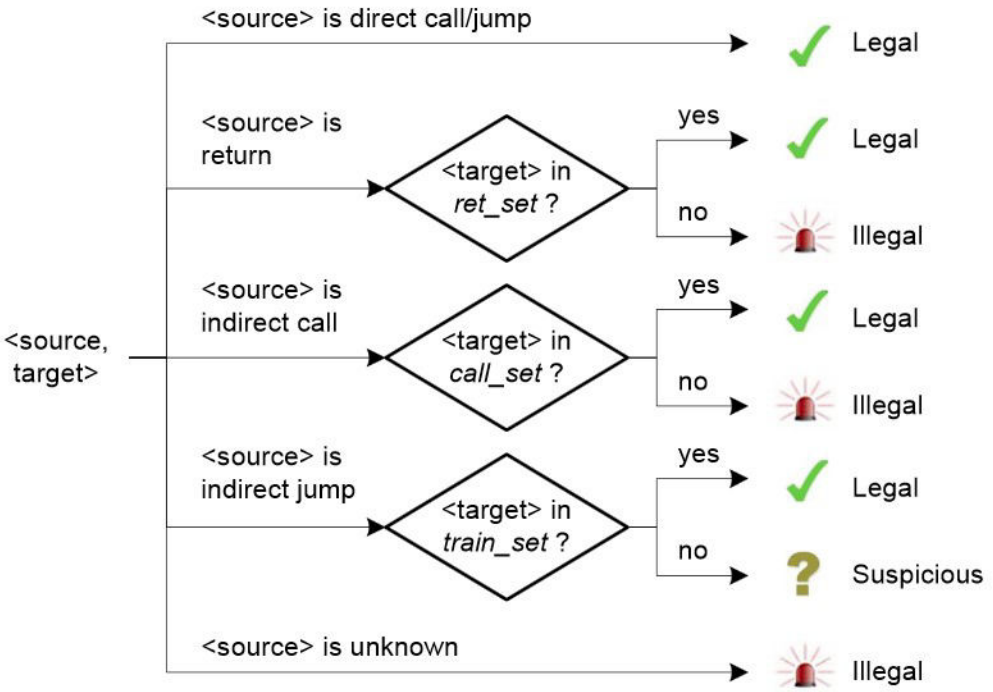
- Uses Branch Trace Store (BTS)
  - Existing Hardware
- Trains branch data based on normal application runs
- Flags branches taken as "suspicious" when witnessing abnormal behavior
- Runtime Overhead: ~6%
- Limitations:
  - Some Fale Positives

CFIMon Example

ROP Attack



CFIMon Checks



- CFIMon catches this attack at the first "ret"

Done

- Any Questions?