Policy Agnostic Control-Flow Integrity

SYSTEM FAILURE

Dean Sullivan
University of Central Florida

Orlando Arias
University of Central Florida

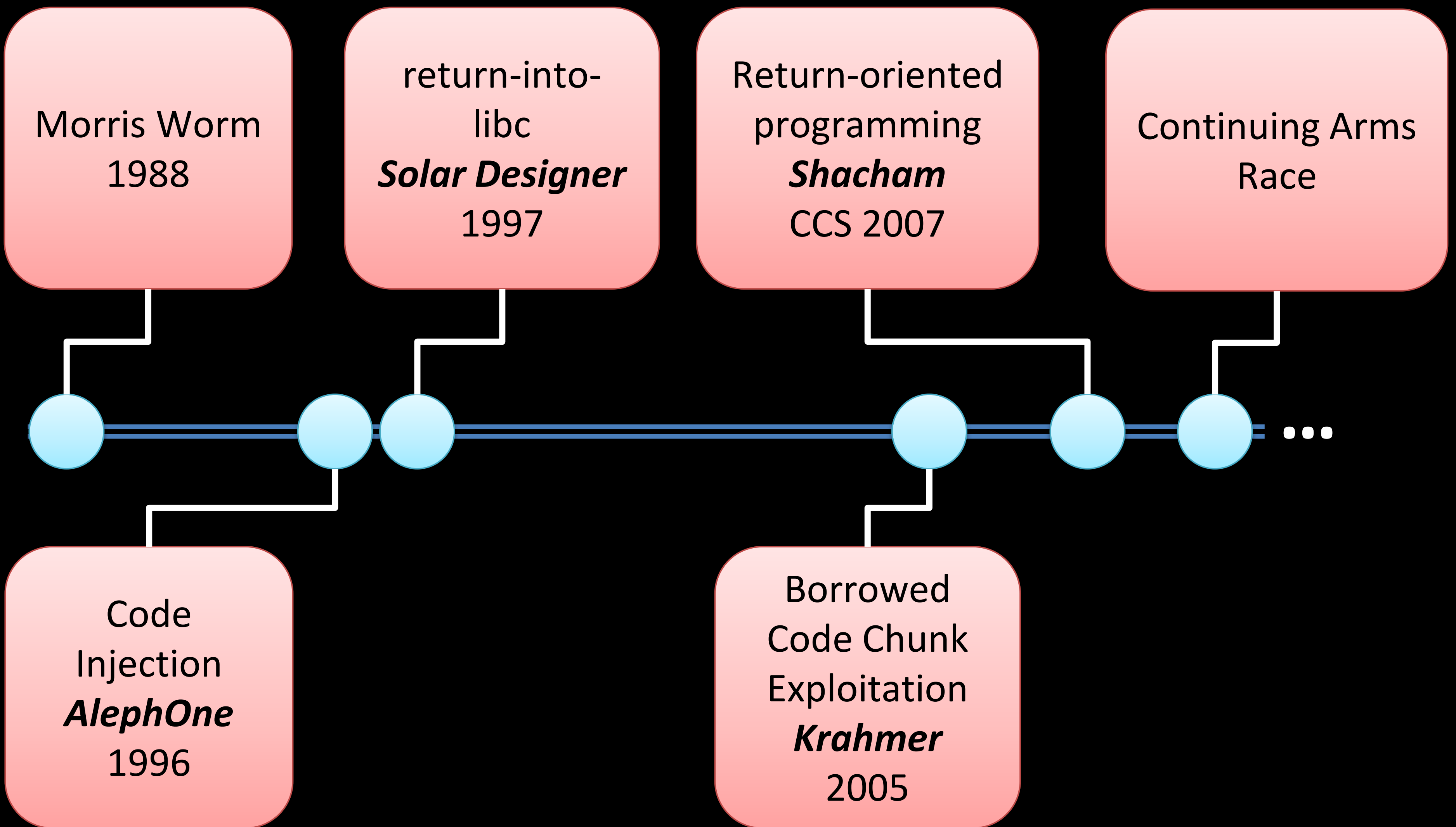
Ahmad-Reza Sadeghi
Technische Universität Darmstadt,
Intel Collaborative Research Institute for Secure Computing, Germany

Lucas Davi
Technische Universität Darmstadt,
Intel Collaborative Research Institute for Secure Computing, Germany

Yier Jin
University of Central Florida,
Cyber Immunity Lab

Motivation

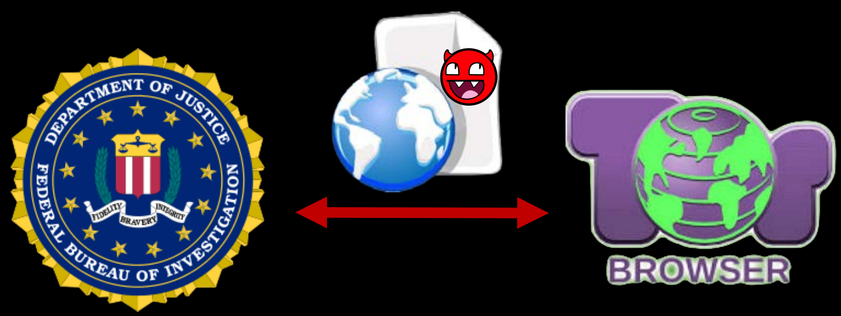
Three Decades of Runtime Attacks



Recent Attacks

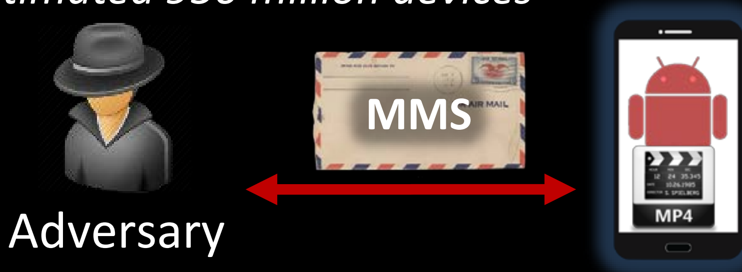
Attacks on Tor Browser [2013]

FBI Admits It Controlled Tor Servers Behind Mass Malware Attack.



Stagefright [Drake, BlackHat 2015]

These issues in Stagefright code critically expose 95% of Android devices, an estimated 950 million devices



Cisco Router Exploit [2016]

Million CISCO ASA Firewalls potentially vulnerable to attacks

The Million Dollar Dissident [2016]

Government targeted human rights defender with a chain of zero-day exploits to infect his iPhone with spyware.

Relevance and Impact

High Impact of Attacks

- Web browsers repeatedly exploited in pwn2own contests
- Zero-day issues exploited in Stuxnet/Duqu [Microsoft, BH 2012]
- iOS jailbreak

Can either be bypassed, or may not be sufficiently effective

[Davi et al, Blackhat2014], [Liebchen et al CCS2015], [Schuster, et al S&P2015]

Hot Topic of Research

- A large body of recent literature on attacks and defenses

Runtime Attacks & Defenses: Continuing Arms Race

Still seeking practical and secure solutions

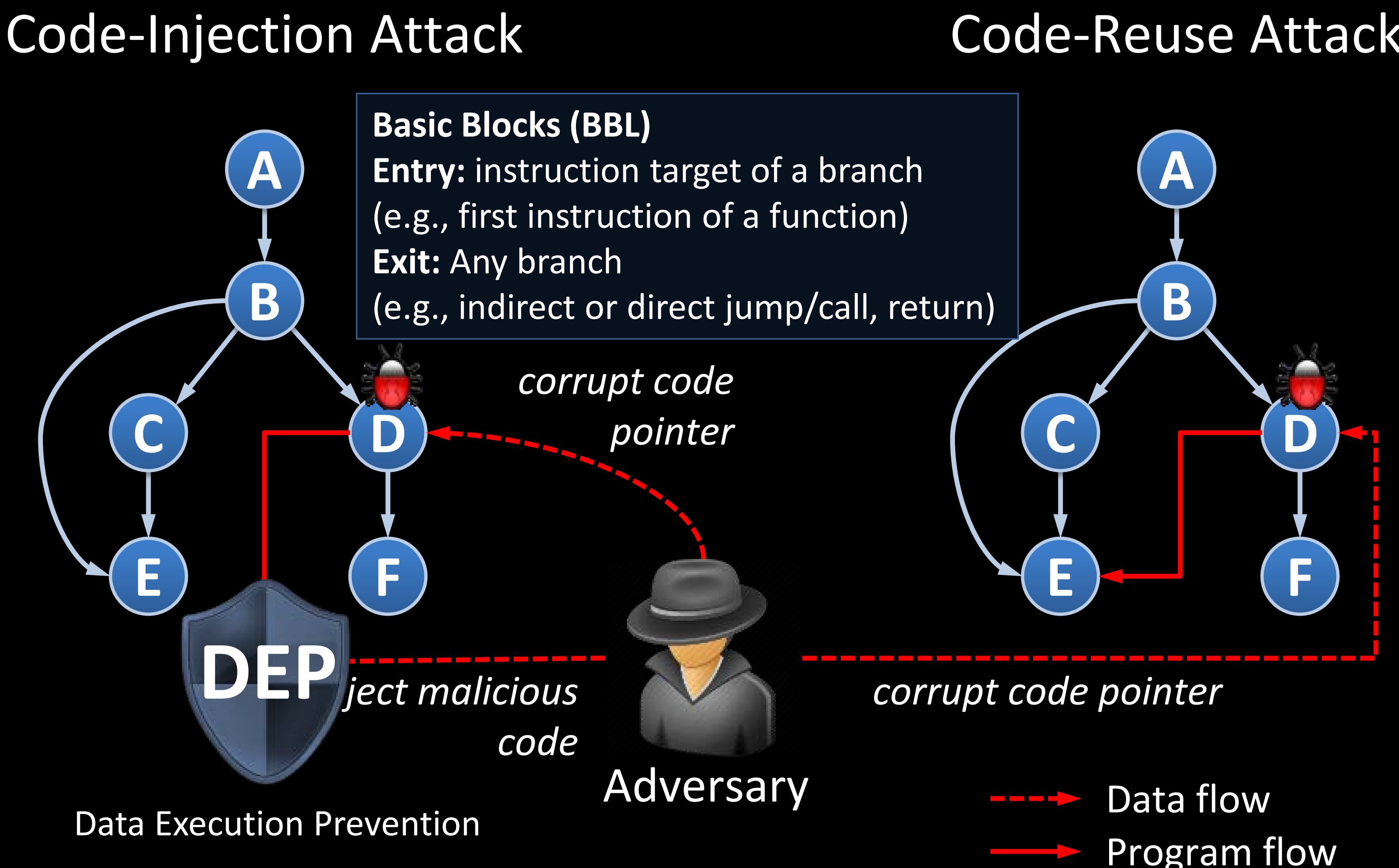
SafeDispatch, MoCFI,
RockJIT, TVip,
StackArmor, CPI/CPS,
Oxymoron, XnR,
Isomeron,
O-CFI,
Readactor,
HAFIX,
...

ROP wo Returns,
Out-of-Control,
Stitching the Gadgets, SROP, JIT-ROP, BlindROP,
COOP, StackDefiler,
"Missing the point(er)"

The whole story .....

Runtime Attacks

Basic Blocks (BBL)
Entry: instruction target of a branch
(e.g., first instruction of a function)
Exit: Any branch
(e.g., indirect or direct jump/call, return)



Return-oriented Programing (ROP): Prominent Code-Reuse Attack

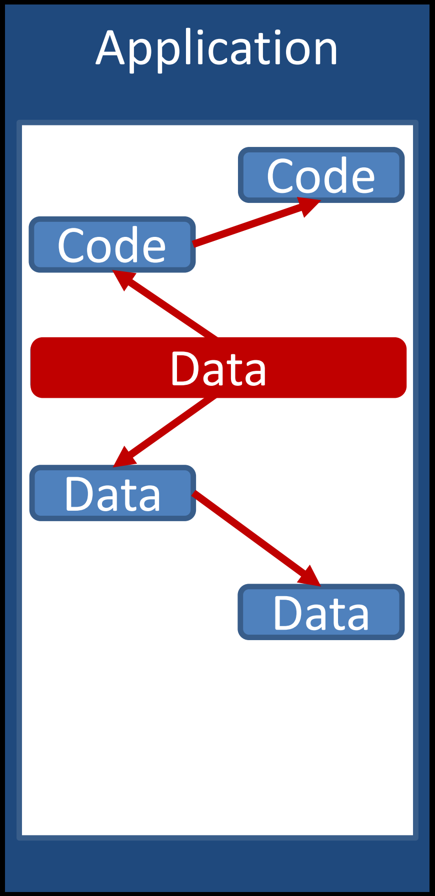
ROP shown to be Turing-complete

ENIGMA
enigma.usenix.org

ROP: Basic Ideas/Steps

- Use small instruction sequences instead of whole functions
- Instruction sequences have length 2 to 5
- All sequences end with a return instruction, or an indirect jump/call
- Instruction sequences chained together as gadgets
- Gadget perform particular task, e.g., load, store, xor, or branch
- Attacks launched by combining gadgets
- Generalization of return-to-libc

Threat Model: Code-reuse Attacks



1. Writable ⊕ Executable
2. Opaque Memory Layout

3. Disclose readable Memory
4. Manipulate writable Memory
5. Computing Engine

Main Defenses against Code Reuse

1. Code Randomization

2. Control-Flow Integrity (CFI)

Randomization vs. CFI

Randomization

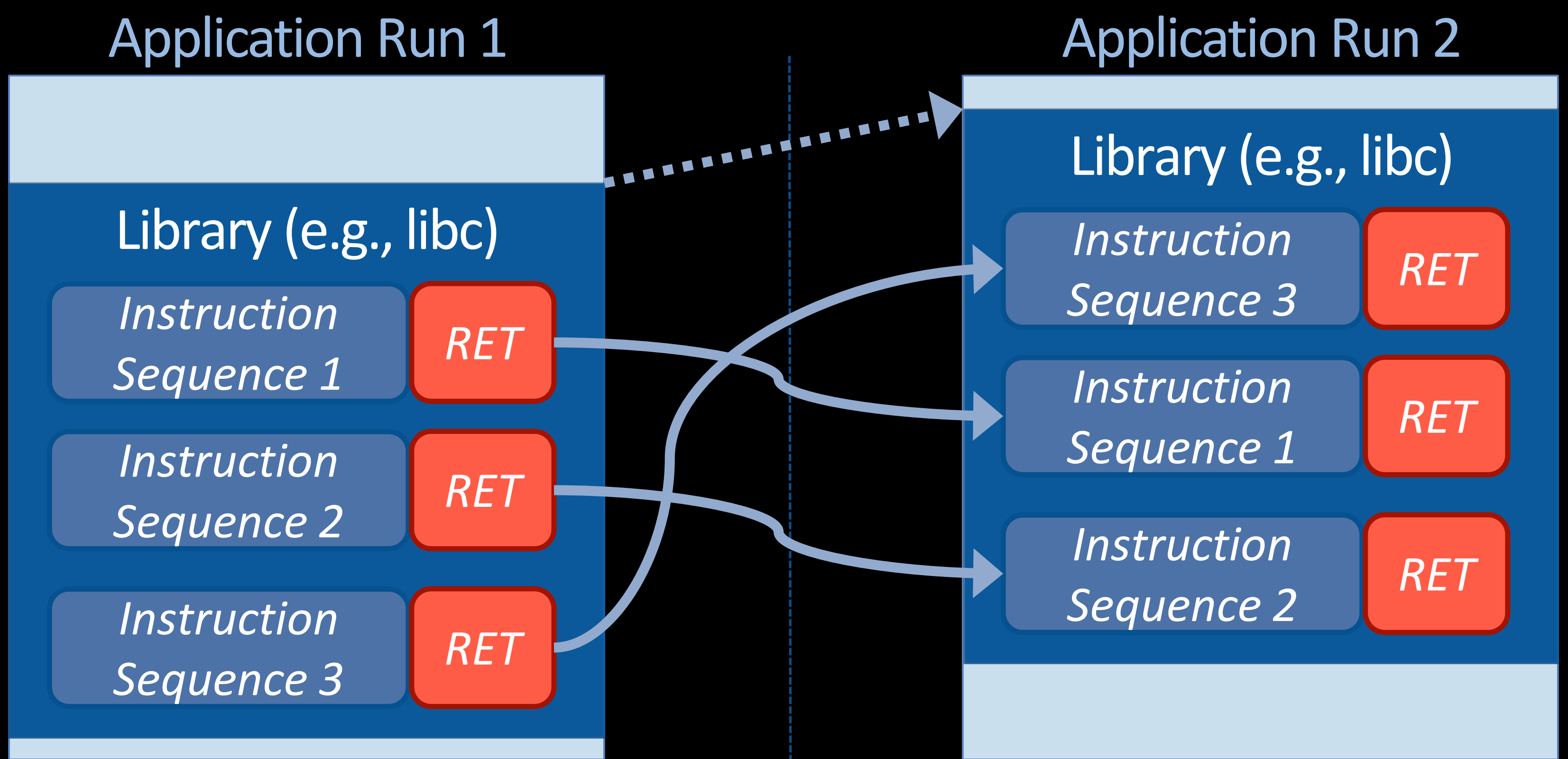
- Low Performance Overhead
- Scales well to complex Software (OS, browser)
- Information Disclosure hard to prevent
- High entropy required

Control-flow Integrity

- Formal Security (Explicit Control Flow Checks)
- Tradeoff: Performance & Security
- Challenging to integrate in complex software, coverage

EPISODE I
Code Randomization
Make gadgets locations unpredictable

Fine-Grained ASLR



- Instruction reordering/substitution within a BBL
  ORP [Pappas et al., IEEE S&P 2012]
- Randomizing each instruction's location:
  ILR [Hiser et al., IEEE S&P 2012]
- Permutation of BBLs:
  STIR [Wartell et al., CCS 2012] & **XIFER** [with Davi et al., AsiaCCS 2013]

Randomization: Memory Leakage Problem

Direct memory disclosure

- Pointer leakage on code pages
- e.g., direct call and jump instruction

Indirect memory disclosure

- Pointer leakage on data pages such as stack or heap
- e.g., return addresses, function pointers, pointers in vTables

JIT-ROP: Bypassing Randomization via Direct Memory Disclosure

Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization

IEEE Security and Privacy 2013, and Blackhat 2013

Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen,
Fabian Monrose, Ahmad-Reza Sadeghi

Just-In-Time ROP: Direct Memory Disclosure

1. Undermines fine-grained ASLR
2. Shows memory disclosures are far more damaging than believed
3. Can be instantiated with real-world exploit

Readactor: Towards Resilience to Memory Disclosure

Readactor:
Practical Code Randomization Resilient to Memory Disclosure
IEEE Security and Privacy 2015

Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, Michael Franz

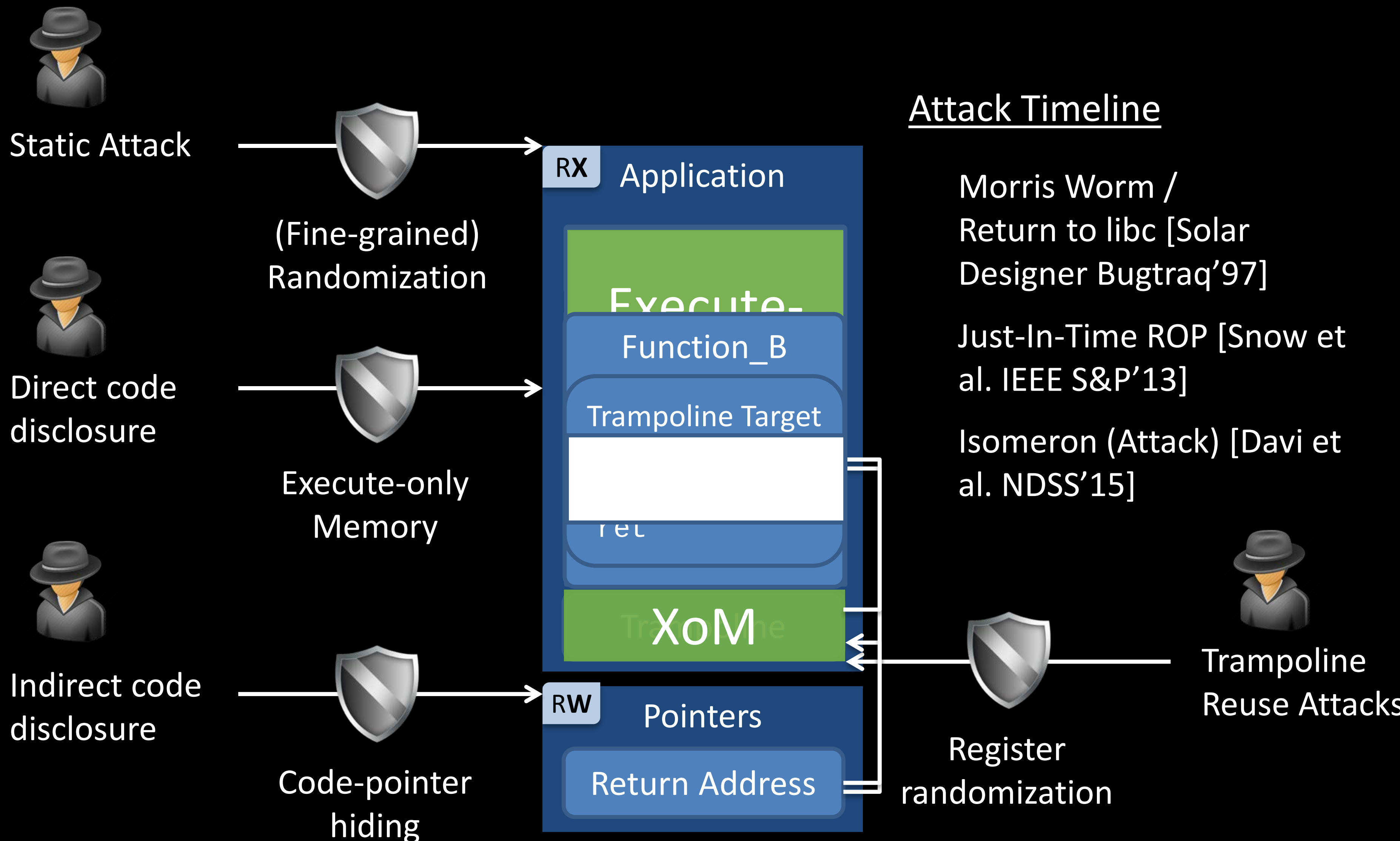
Code Randomization: Attack & Defense Techniques

<u>Attack Timeline</u>

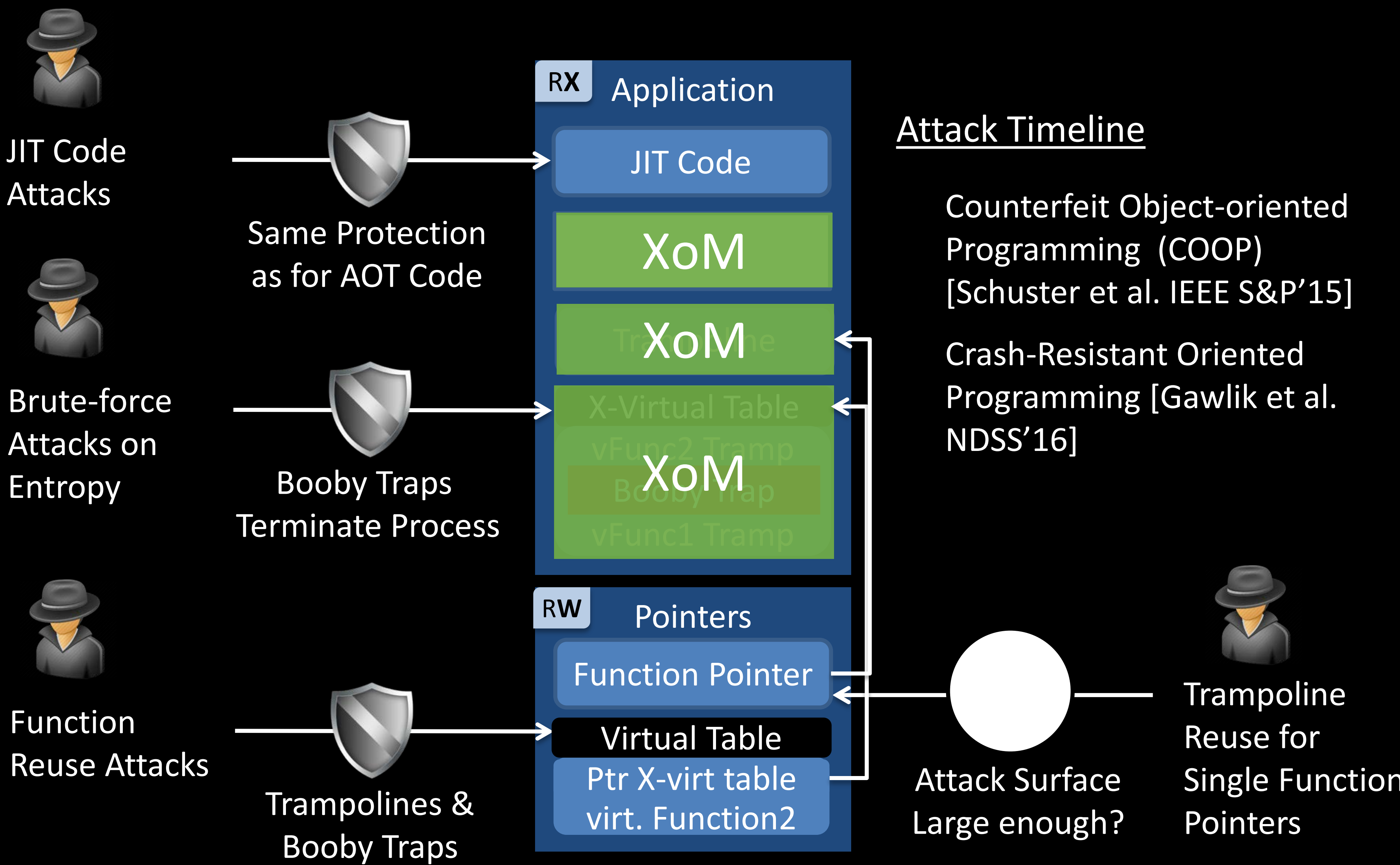
Morris Worm / Return to libc [Solar Designer Bugtraq’97]

Just-In-Time ROP [Snow et al. IEEE S&P’13]

Isomeron (Attack) [Davi et al. NDSS’15]



Code Randomization: Attack & Defense Techniques

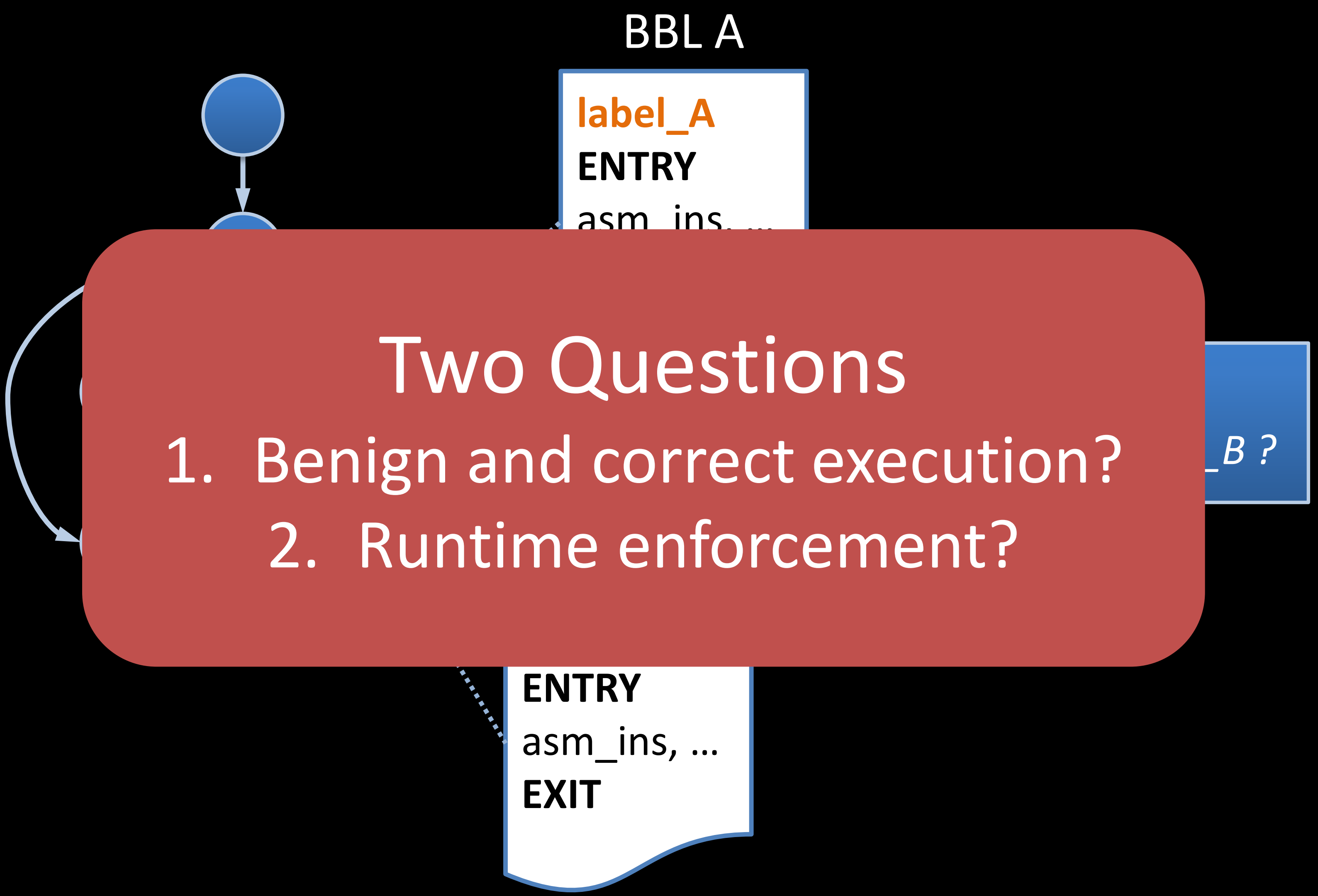


EPISODE II
Control-Flow Integrity (CFI)

Restricting indirect targets
to a pre-defined control-flow graph

Original CFI Label Checking

[Abadi et al., CCS 2005 & TISSEC 2009]



Two Questions

1. Benign and correct execution?
2. Runtime enforcement?

CFI: CFG Analysis and Coverage Problem

CFG Analysis

- Conservative “points-to” analysis
- e.g., over-approximate to avoid breaking the program

CFG Coverage

- Precision of CFG analysis determines security of CFI policy
- e.g., more precise → more secure

Which Instructions to Protect?

Returns

- **Purpose**: Return to calling function
- **CFI Relevance**: Return address located on stack

Indirect Jumps

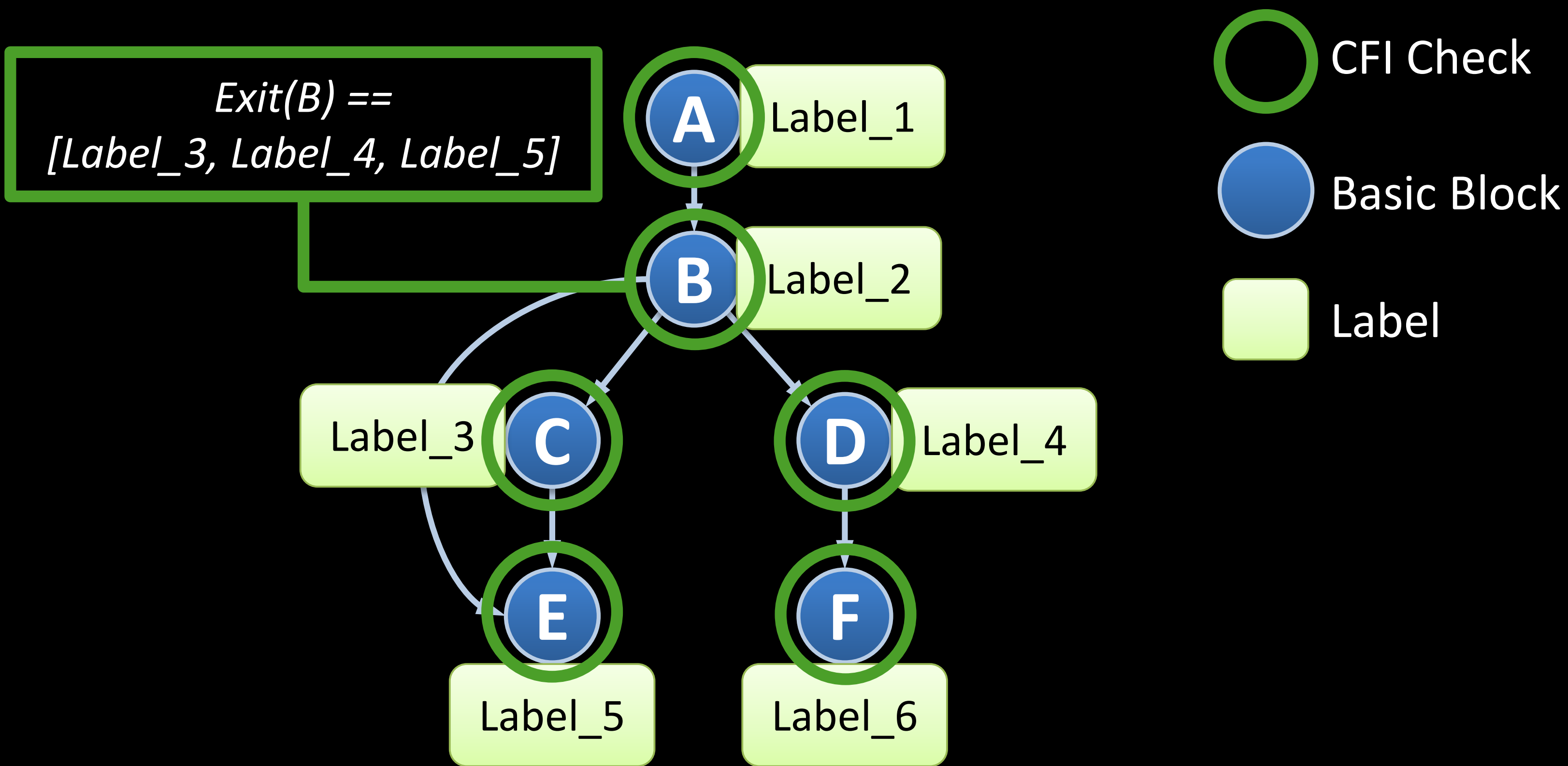
- **Purpose**: switch tables, dispatch to library functions
- **CFI Relevance**: Target address taken from either processor register or memory

Indirect Calls

- **Purpose**: call through function pointer, virtual table calls
- **CFI Relevance**: Target address taken from either processor register or memory

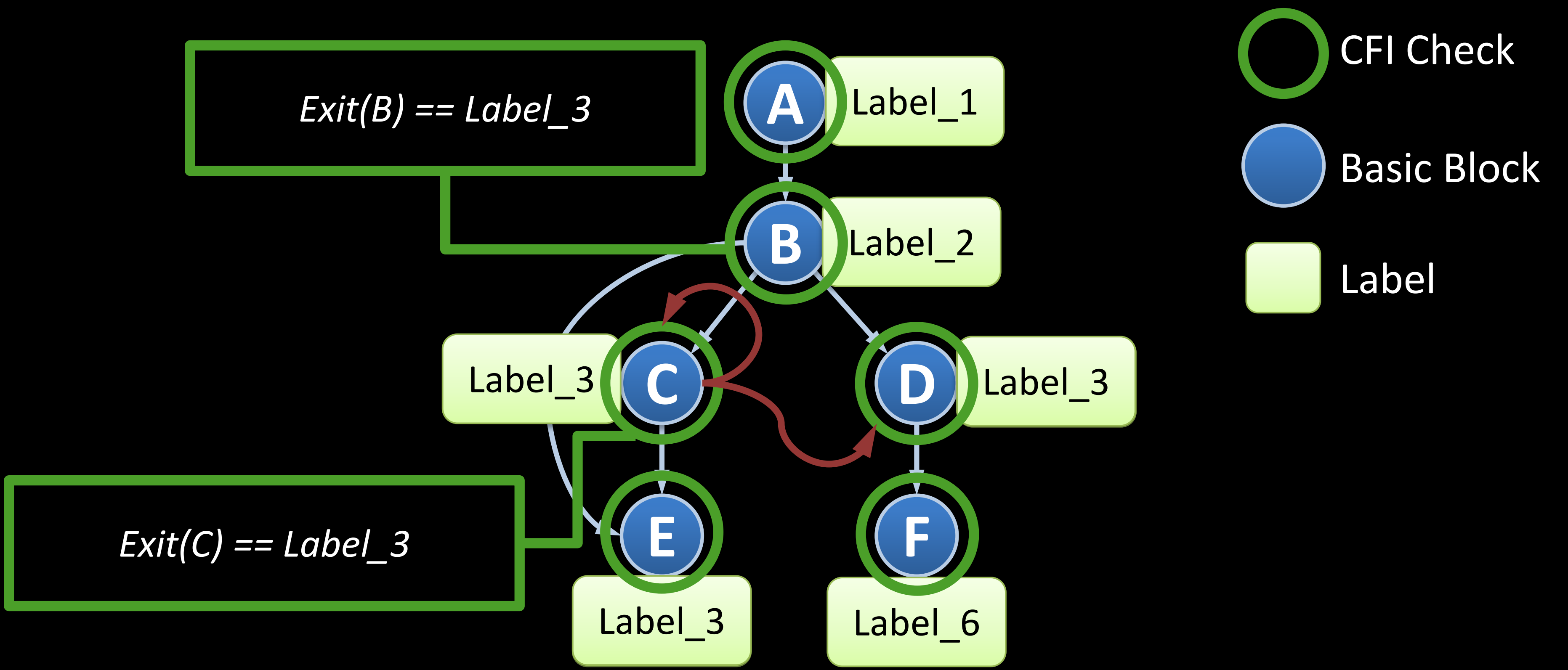
Label Granularity: Trade-Offs (1/2)

- Many CFI checks are required if unique labels are assigned per node



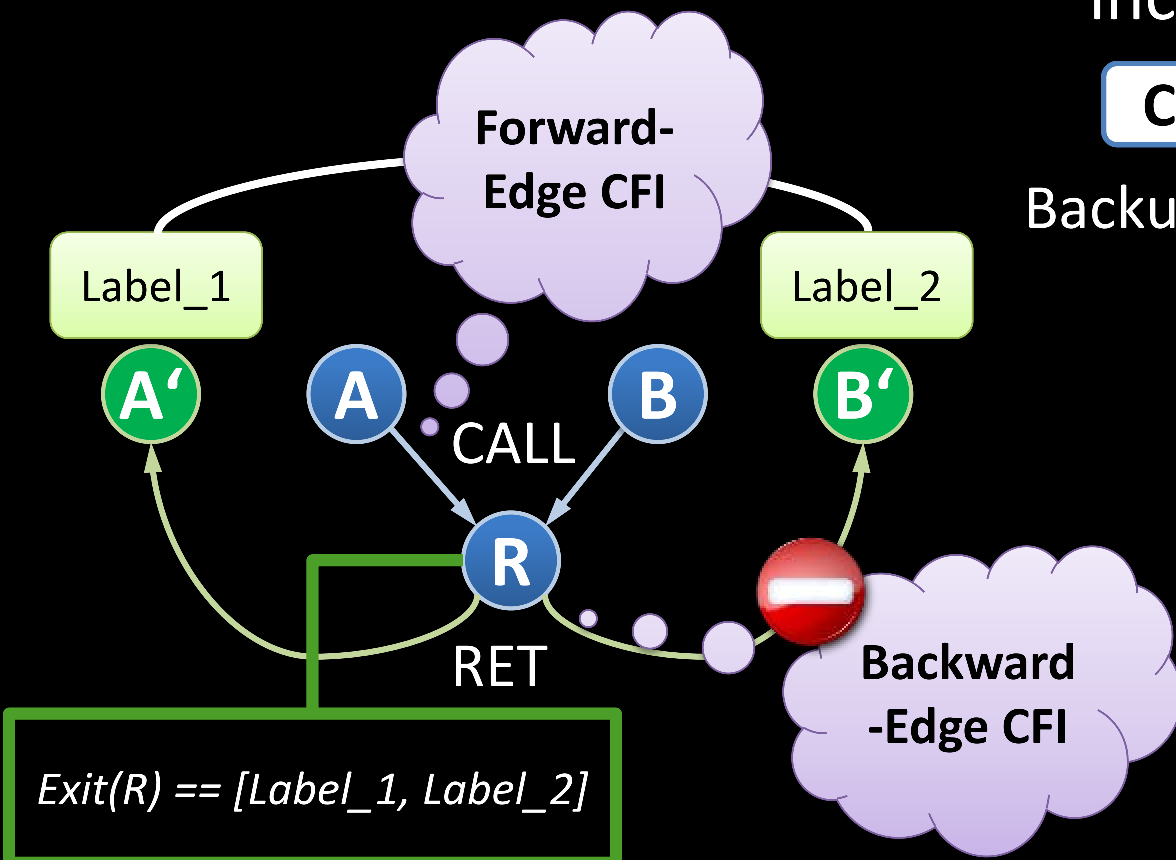
Label Granularity: Trade-Offs (2/2)

- Optimization step: Merge labels to allow single CFI check
- However, this allows for unintended control-flow paths

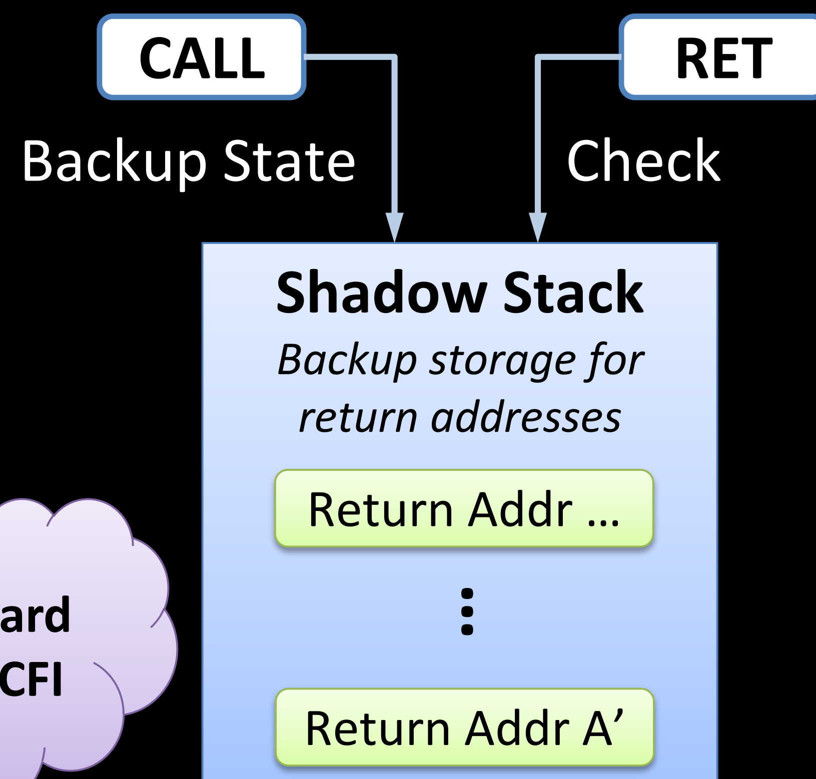


Label Problem for Returns

- Static CFI label checking leads to coarse-grained protection for returns



- Shadow stack allows for fine-grained return address protection but incurs higher overhead



Forward- vs. Backward-Edge

- Some CFI schemes consider only forward-edge CFI
  - Google's VTV and IFCC [Tice et al., USENIX Sec 2015]
  - SAFEDISPATCH [Jang et al., NDSS 2014]
  - And many more: TVIP, VTint, vfguard
- Assumption: Backward-edge CFI through stack protection
- Problems of stack protections:
  - Stack Canaries: memory disclosure of canary
  - ASLR (base address randomization of stack): memory disclosure of base address
  - Variable reordering (memory disclosure)

StackDefiler
Protecting Stack is Hard!

Losing Control:
On the Effectiveness of Control-Flow Integrity under Stack Attacks
ACM CCS 2015

Christopher Liebchen, Marco Negro, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Stephen Crane, Mohaned Qunaibit, Michael Franz, Mauro Conti

StackDefiler

- Goal:
  - Bypass fine-grained Control-Flow Integrity
  - IFCC & VTV (CFI implementations by Google for GCC and LLVM)
- Approach:
  - Due to optimization by compiler critical CFI pointer is spilled on the stack
  - StackDefiler discloses the stack address and overwrites the spilled CFI pointer
  - At restoring of spilled registers a malicious CFI pointer is used for future CFI checks
  - No stack-based vulnerability needed

Bypassing (Coarse-grained) CFI

Stitching the Gadgets
USENIX Security 2014
Lucas Davi, Daniel Lehmann,
Ahmad-Reza Sadeghi, Fabian Monrose

COOP
IEEE S&P 2015
Felix Schuster, Thomas Tendyck,
Christopher Liebchen, Lucas Davi,
Ahmad-Reza Sadeghi, Thorsten Holz

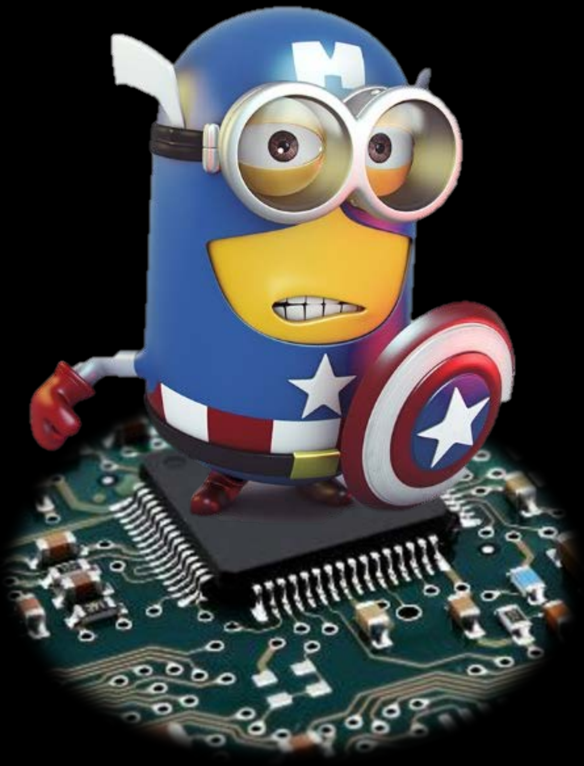
Coarse-grained CFI: Lessons Learned

1. Too many
   → Restrict ... adow stack)
2. Heuristics
   → Adjuste ... false positive
3. Too many ... s
   • Resolvi ... -trivial
   → Compro

Control-Flow Integrity

Out of control [Göktas et al., IEEE S&P 2014]	**Control-Flow Bending** [Carlini et al., USENIX Sec. 2015]
Stitching the gadgets [Davi et al., USENIX Sec. 2014]	**FlowStich** [Hu et al., USENIX Sec. 2015]
ROP is still dangerous [Carlini et al., USENIX Sec. 2014]	**Control Jujutsu** [Evans et al., CCS 2015]
Size does matter [Göktas et al., USENIX Sec. 2014]	**StackDefiler** [Conti et al., CCS 2015]
COOP [Schuster et al., IEEE S&P 2015]	**Signal-oriented Programming (SROP)** [Bosman et al., IEEE S&P 2014]

Hardware CFI



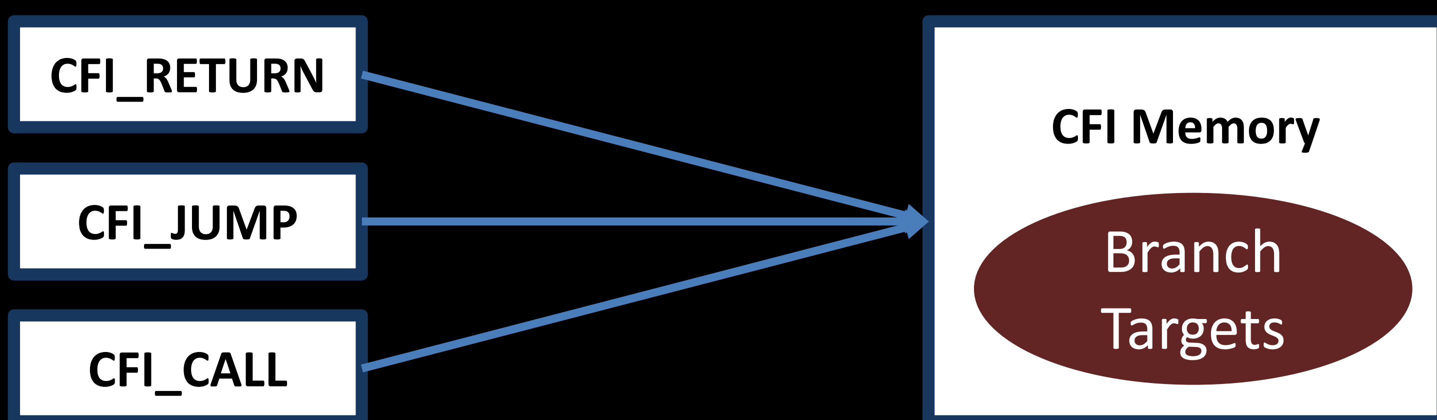
Why Leveraging Hardware for CFI ?

Efficiency

Security

Dedicated CFI instructions

Isolated CFI storage



Why CFI Processor Support?

CFI Processor Support based on Instruction set architecture (ISA) extensions

- Dedicated CFI instructions
- Avoids offline training phase
- Instant attack detection
- CFI control state:
  Binding CFI data to CFI state and instructions

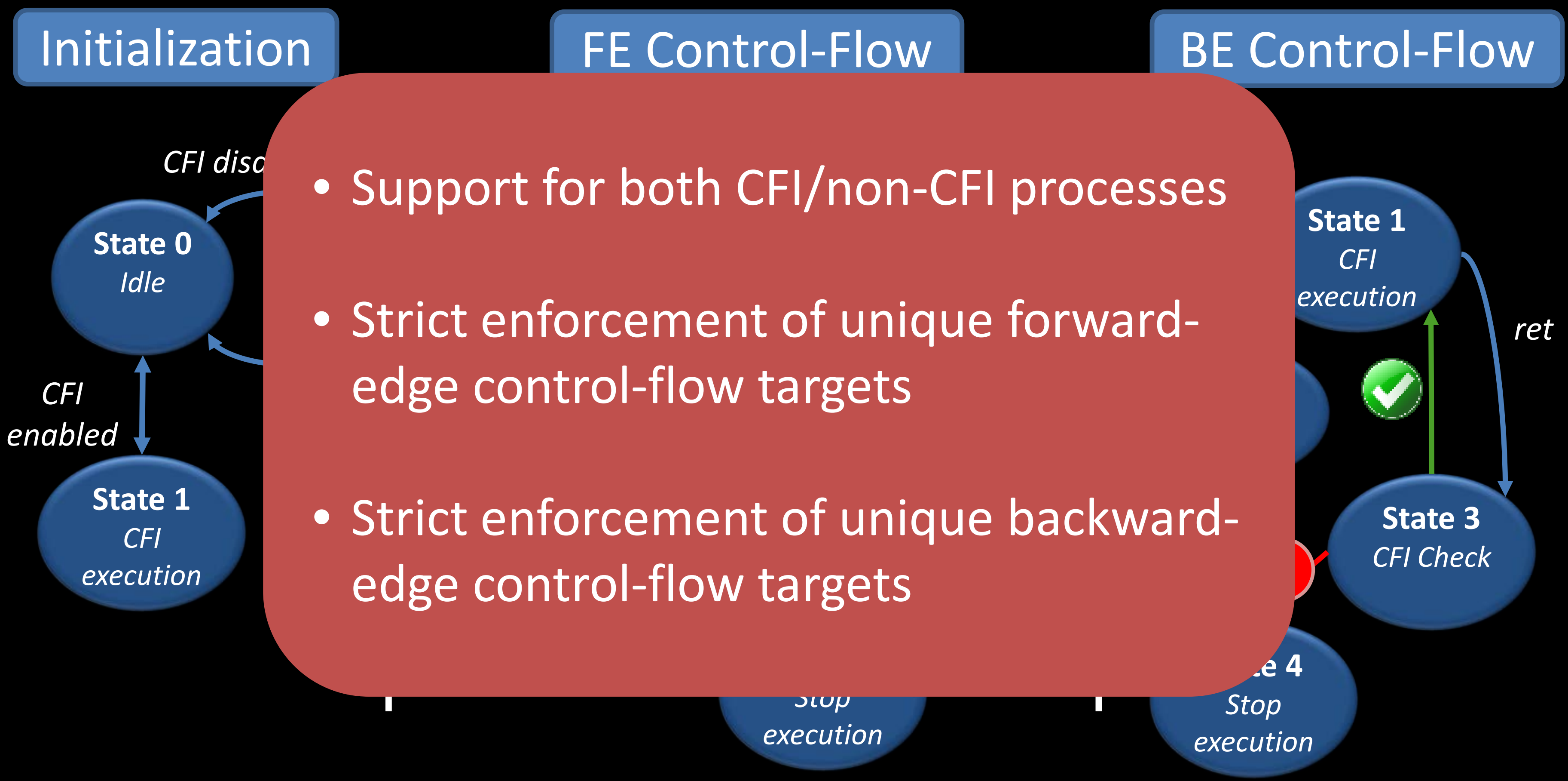
HAFIX++

Strategy Without Tactics:
Policy-Agnostic Hardware-Enhanced Control-Flow Integrity
Design Automation Conference (DAC 2016)
Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen,
Ahmad-Reza Sadeghi, Yier Jin

Objectives

Backward-Edge and Forward-Edge CFI	Stateful, CFI policy agnostic
No burden on developer	No code annotations/changes
Security	Hardware protection On-Chip Memory for CFI Data No unintended sequences
High performance	< 3% overhead
Enabling technology	All applications can use CFI features Support of Multitasking
Compatibility to legacy code	CFI and non-CFI code on same platform

HAFIX++ Fine-Grained CFI State Model



- Support for both CFI/non-CFI processes
- Strict enforcement of unique forward-edge control-flow targets
- Strict enforcement of unique backward-edge control-flow targets

HAFIX++ ISA Extensions

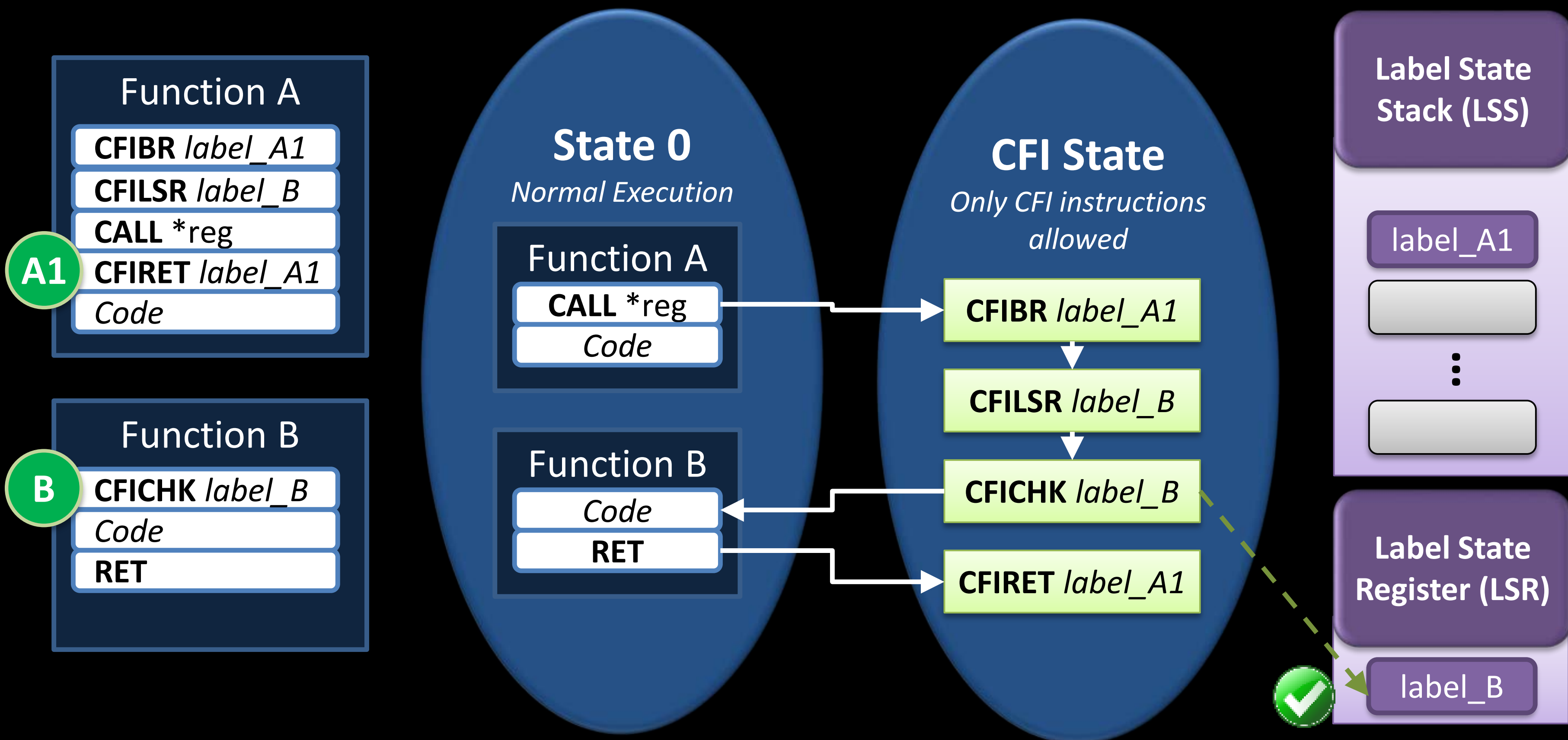
cfibr	Issued at call site → setup Backward (BW) Edge
cfiret	Issue at return site → check BW Edge
cfiprc	Issued at call site → setup call target
cfiprj	Issued at jump site → setup jump target
cfichk	Issued at call/jmp target → check Forward (FW) Edge

- Fine-grained forward edge control-flow policy
  - Separation of call/jump
  - Unique label per target
- Fine-grained backward edge control-flow policy
  - Return to only most recently issued return label

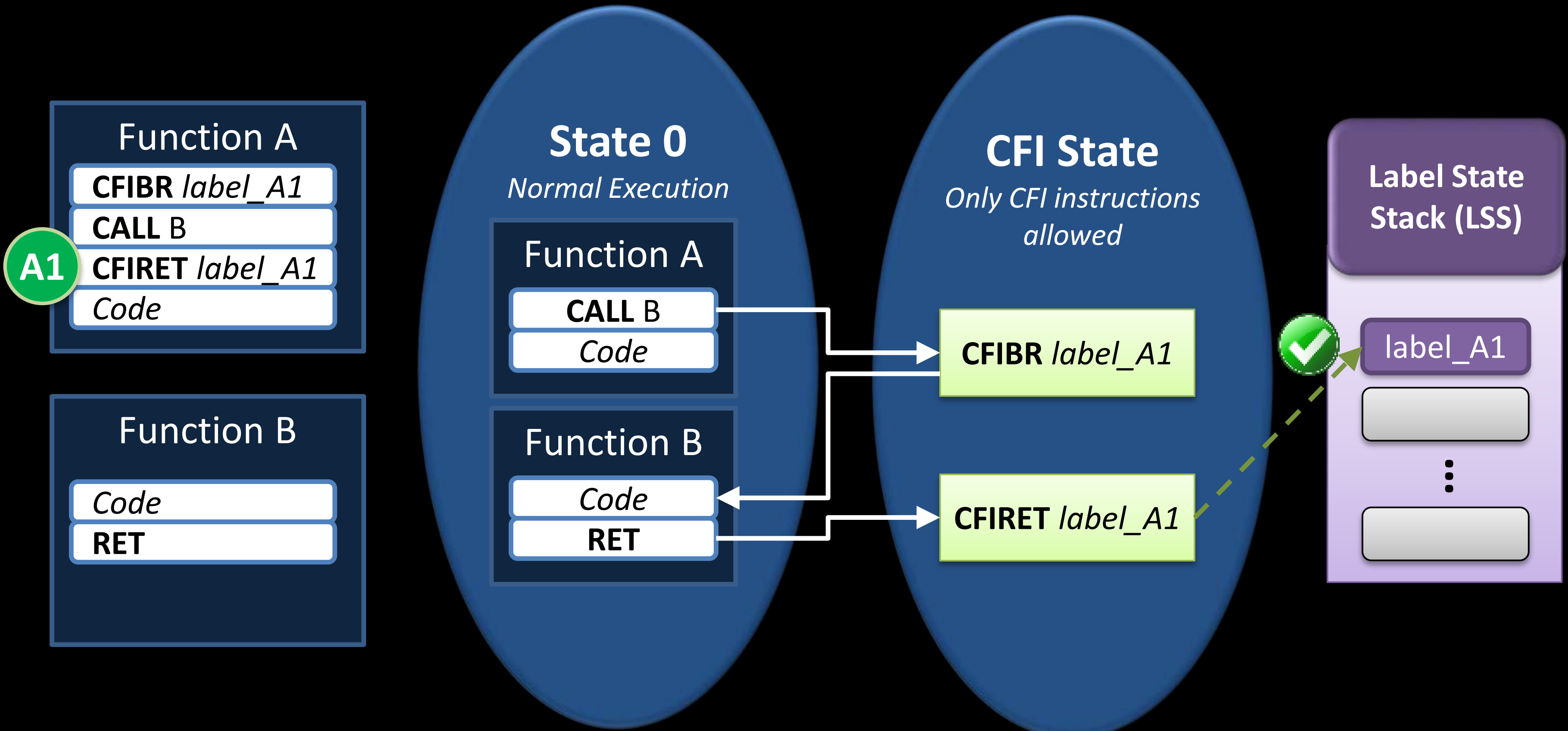
Label State Stack (LSS)

Label State Register (LSR)

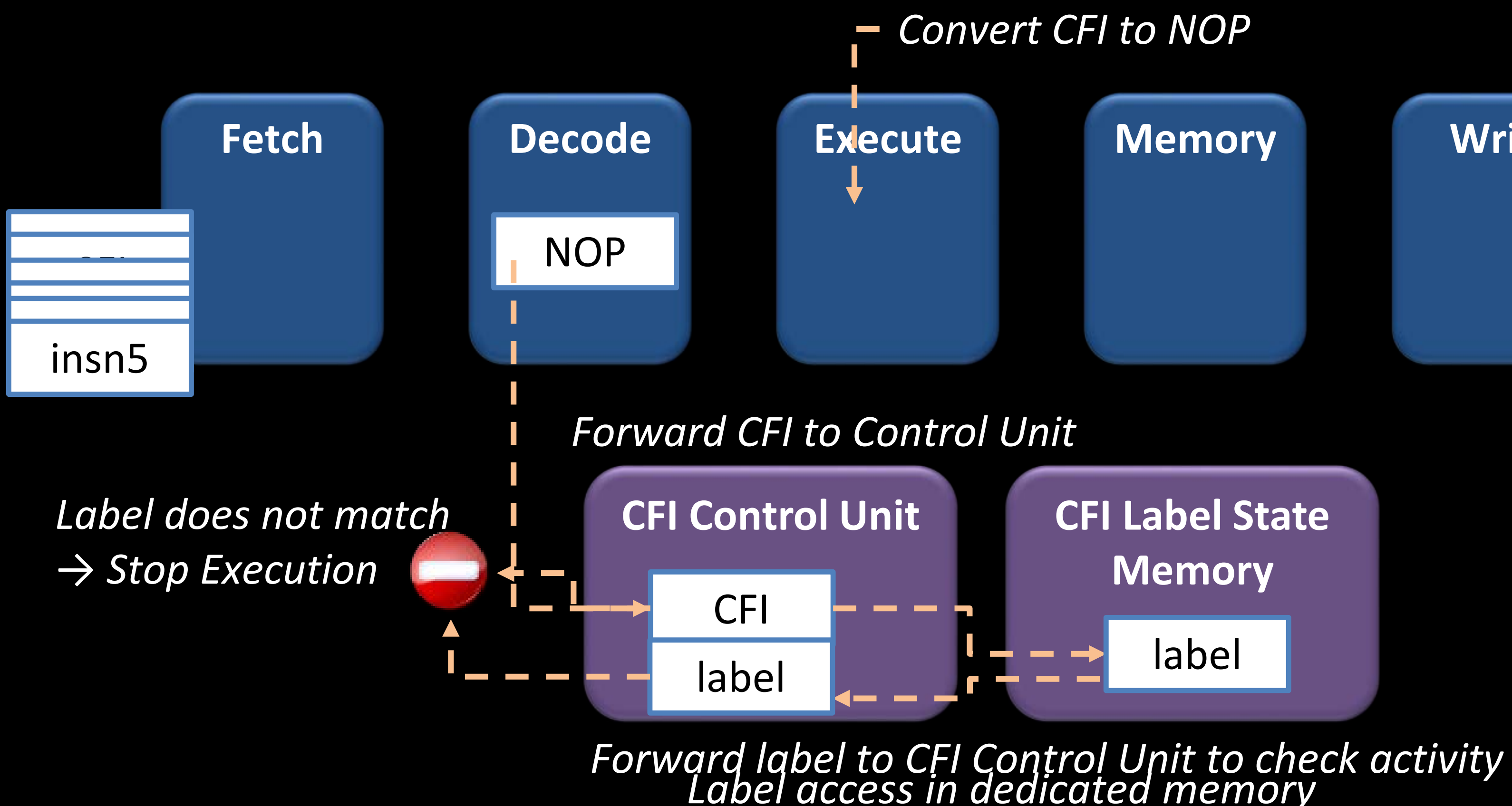
Indirect Call Policy

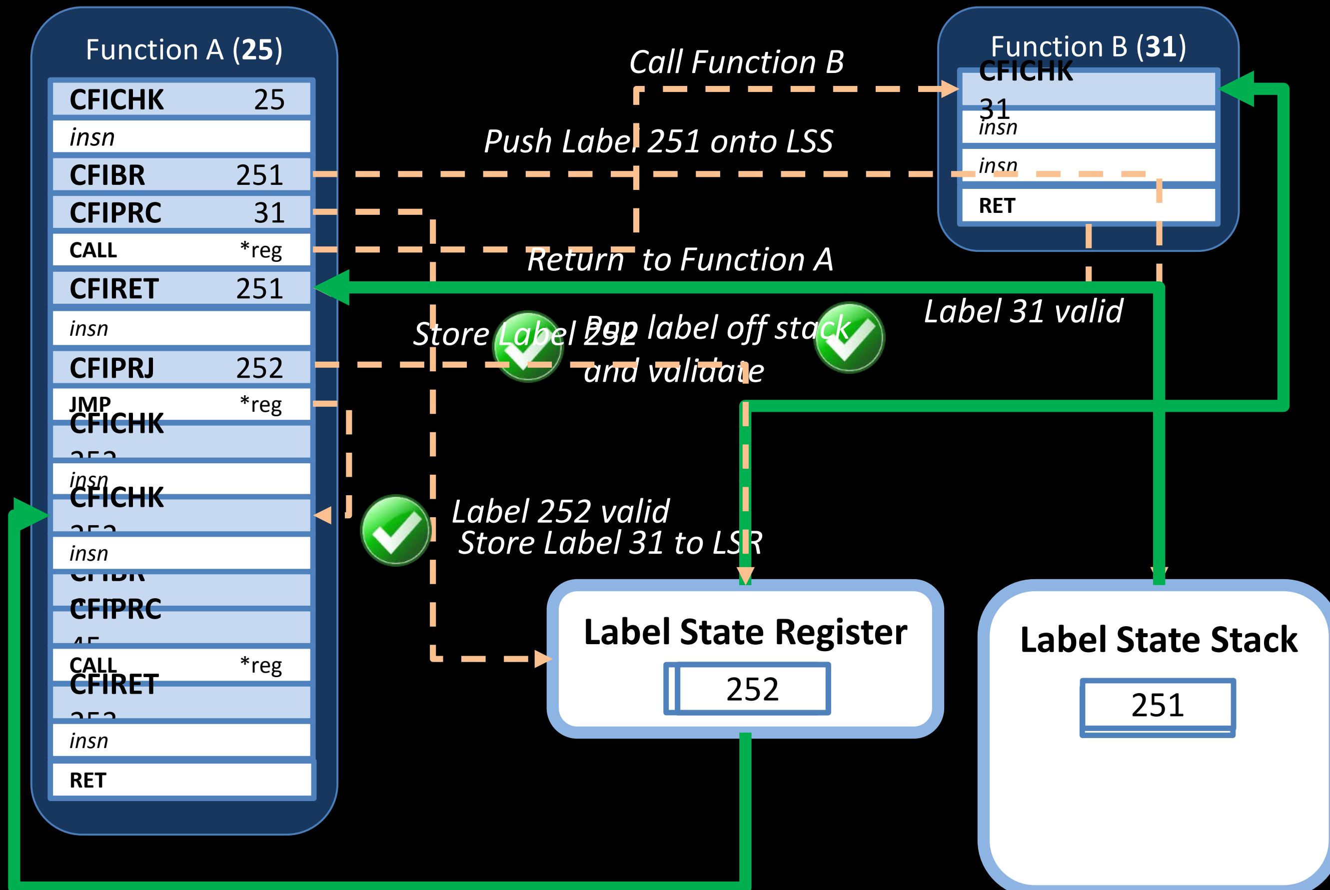


Function Return Policy



HAFIX++ Pipeline





Function A (25)
CFICHK 25
insn
CFIBR 251
CFIPRC 31
CALL *reg
CFIRET 251
insn
CFIPRJ 252
JMP *reg
CFICHK
insn
CFICHK
insn
CFIPRC
CALL *reg
CFIRET
insn
RET
Function B (31)
CFICHK
31
insn
insn
RET
Call Function B
Push Label 251 onto LSS
Return to Function A
Store Label 252
Pop label off stack and validate
Label 31 valid
Label 252 valid
Store Label 31 to LSR
Label State Register
252
Label State Stack
251

Challenges …

Architectural Issues

- Runtime overhead caused by CFI instrumentation
  - Initializing and validating the CFI state upon every FW/BW edge
  - I-cache pressure during instruction fetch
  - Effective CPI
- Runtime overhead and problems caused by hardware
  - Branch instruction occur about every 3-5 instructions
  - CFI instructions/operations around every one of them
  - Memory access for CFI metadata is slow
  - CFI metadata could be corrupted if considered data (StackDefiler)
  - CFI metadata could be a bottleneck if placed in code

The Multiple Callers Problem

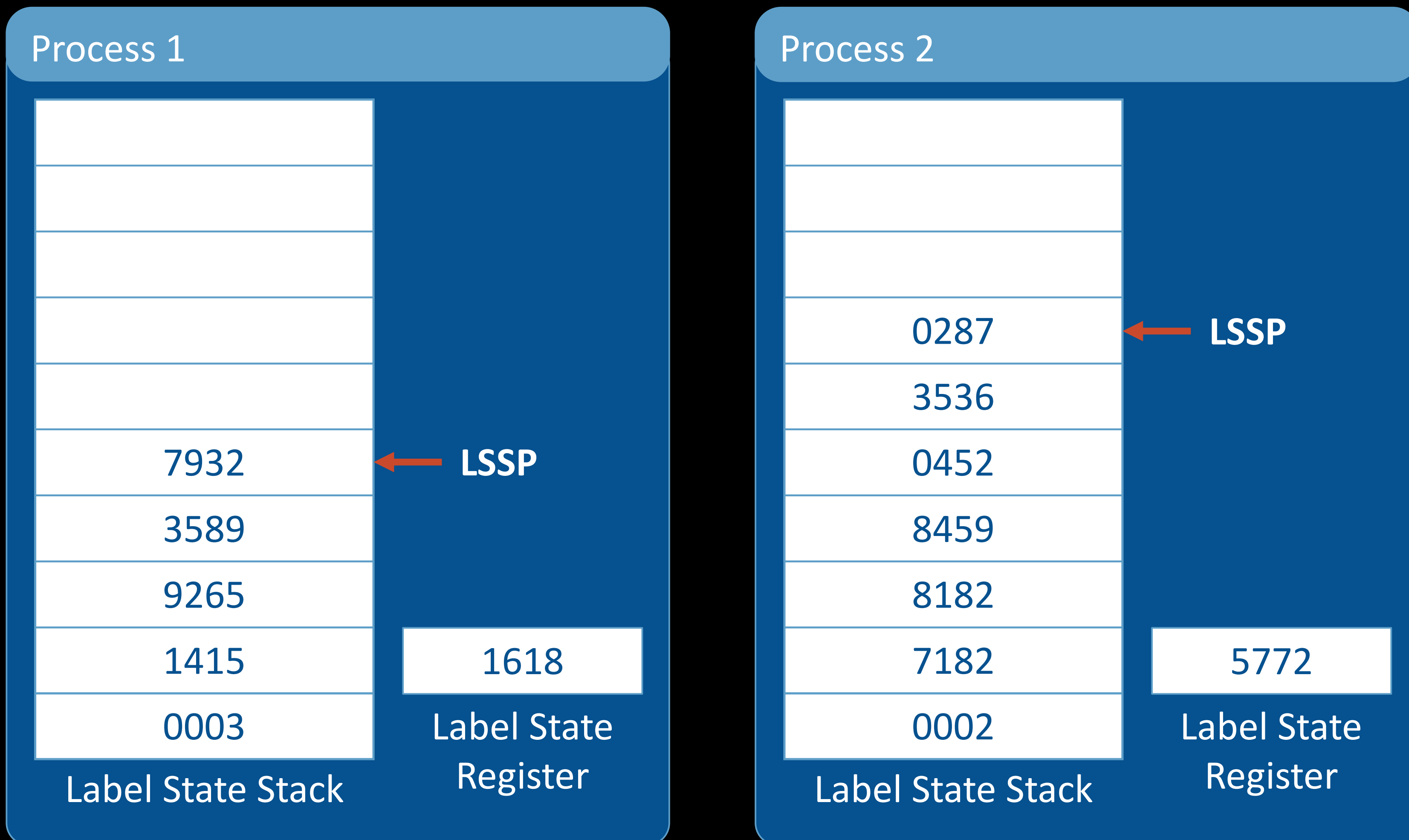
- We can not assign both 45 and 33 at the same time.
- We could assign a common label to all targets
  - Introduces erroneous edges in the Control Flow Graph

  → Call targets must be disjointed! Use a trampoline!

System Challenges

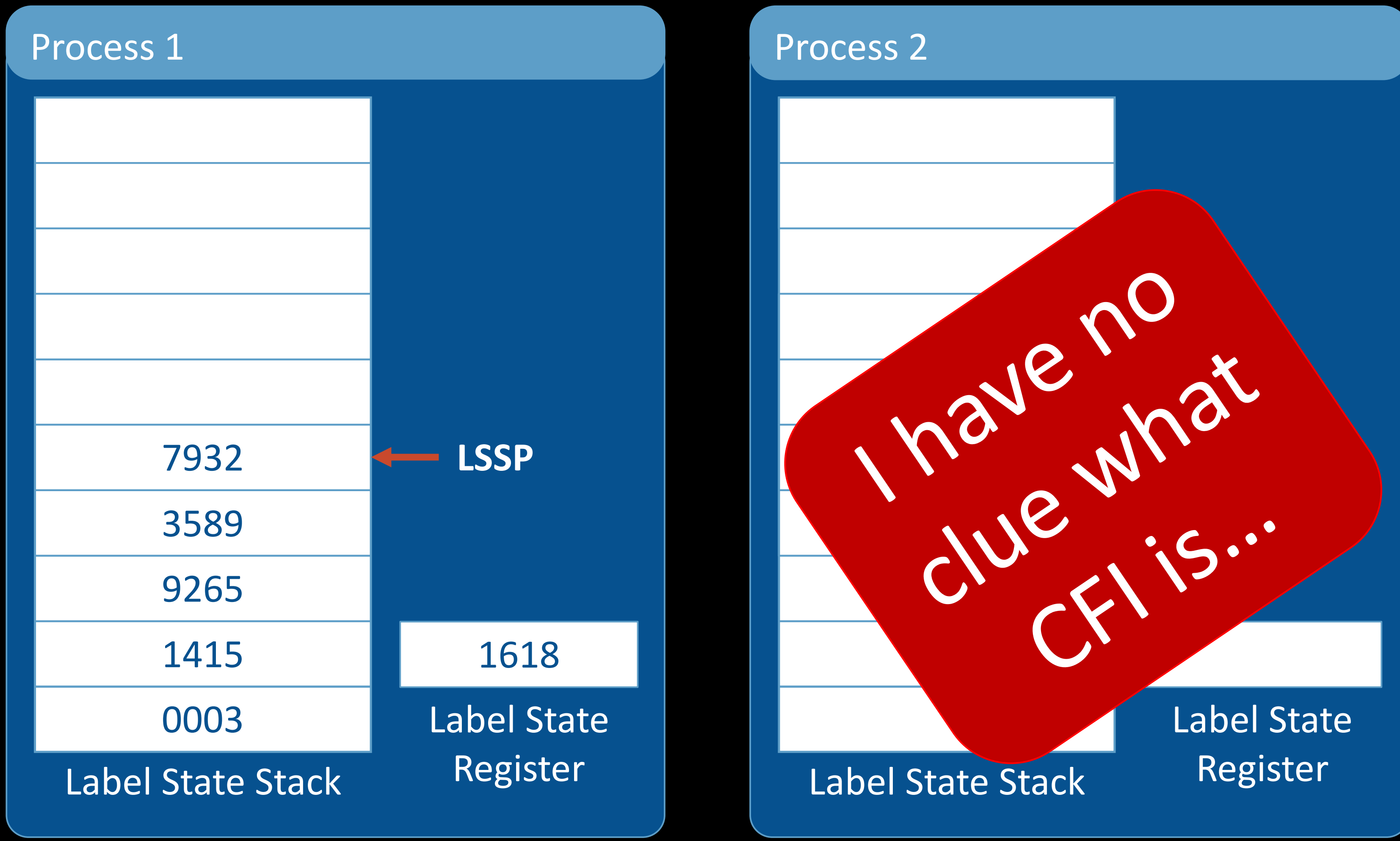
1. Sharing CFI subsystem resources
2. Separation of process states
3. Handling CFI Module Exceptions
4. Handling of legacy code

The Scheduling Issue



This is running **This is being scheduled**

The Scheduling Issue



This is running

This is being scheduled

The Stack Issue

2884	
7950	
3832	
2643	
3846	
7932	← LSSP
3589	
9265	
1415	
0003	

Label State Stack

We ran out of stack space! What do we do?

The Process Control Block

- Representation of a process to the kernel
- In Linux, look for `task_struct` in `include/linux/sched.h`
- Information contains:
  - Execution state (runnable, suspended, zombie…)
  - Virtual memory allocations
  - Process owner
  - Process group
  - Process id
  - I/O status information
  - CPU context state

Kernel Scheduler Additions

read `current` CFI awareness

if CFI is enabled

    backup CFI state for `current`

read `next` CFI awareness

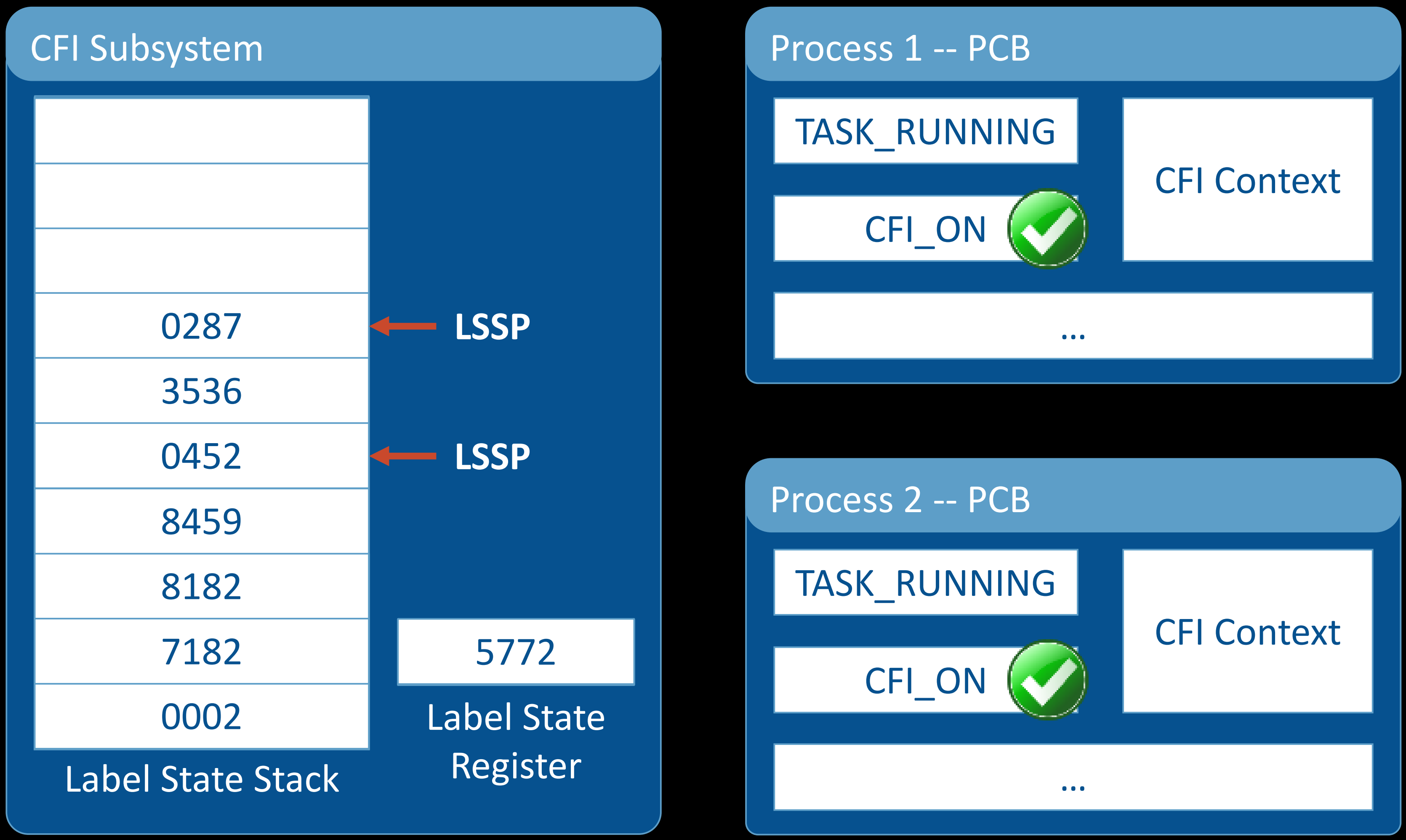
if CFI is enabled

    restore CFI state for `next`

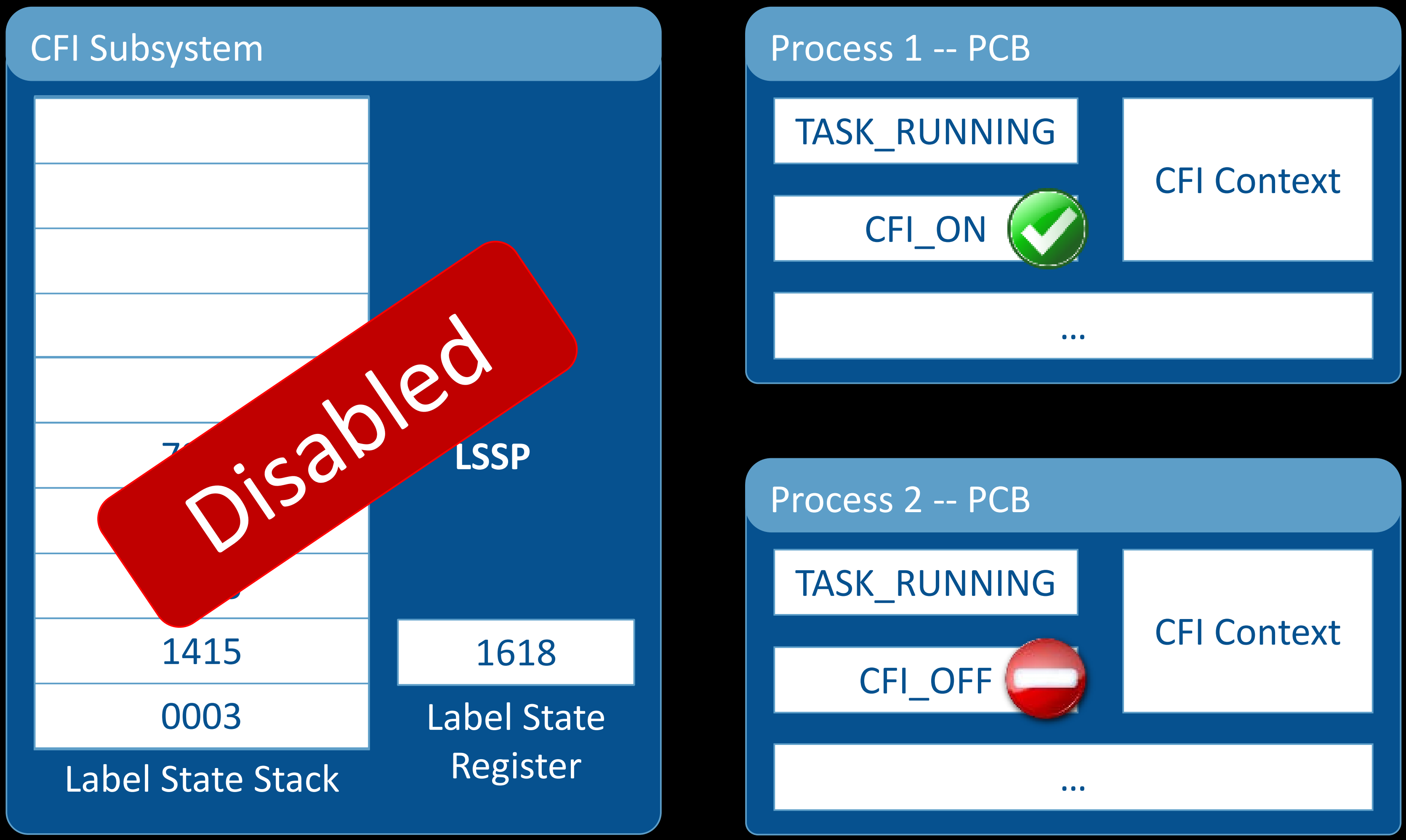
else

    disable CFI subsystem

The Scheduling Issue Resolved



The Scheduling Issue Resolved



Your stack still overflows

or underflows for that matter

- We use the PCB already, add things there

on overflow:

    copy bottom half of `current`'s LSS to PCB

    move top half of LSS to bottom

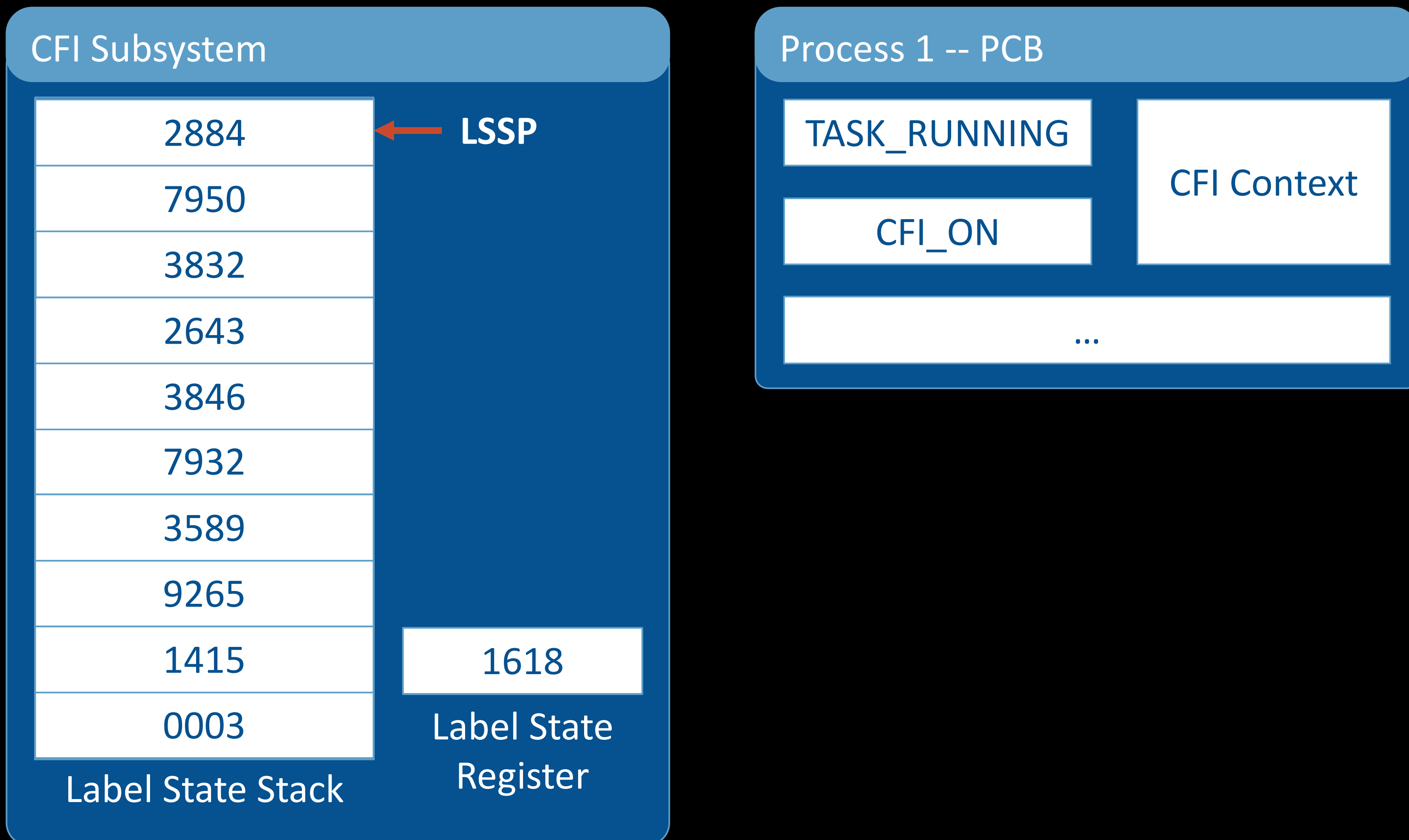
    set LSSP to new location

on underflow:

    get bottom half of `current`'s LSS from PCB

    set LSSP to new location

The Stack Issue Resolved



CFI Faults

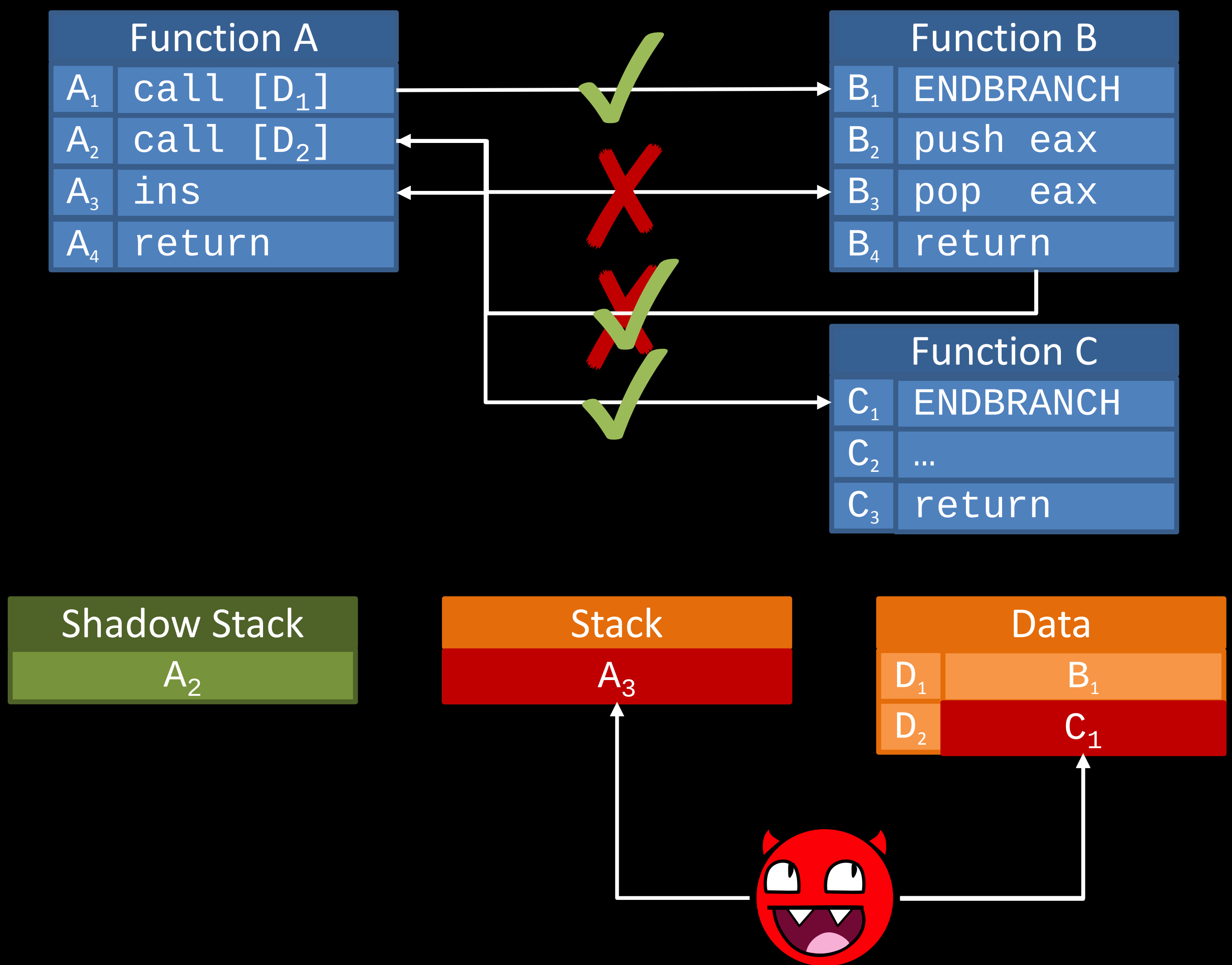
- The CFI subsystem detected a CFI violation
- Add kernel log entry with CFI fault information
- Send `SIGKILL` to offending process
  - This kills the process with no chance of a signal handler running

Related Works

- **HCFI:**
  - New instructions to track control flow
  - *Combines and relocates instructions into pipeline bubble slots*
  - *Single threaded, embedded applications only*
- **Intel CET:**
  - Shadow stack for return addresses
  - New register **`ssp`** for the shadow stack
  - Conventional move instructions cannot be used in shadow stack
  - New instructions to operate on shadow stack
  - New instruction for indirect call/jump targets: **`branchend`**
  - *Any indirect call/jump can target any valid indirect branch target*

Control-flow Enforcement Technology

[Intel 2016]



Control-flow Enforcement Technology

[Intel 2016]

- Backward edge:
  - Shadow stack detects return-address manipulation
  - Shadow stack protected, cannot be accessed by attacker
  - New register **`ssp`** for the shadow stack
  - Conventional move instructions cannot be used in shadow stack
  - New instructions to operate on shadow stack
- Forward edge:
  - New instruction for indirect call/jump targets: **`branchend`**
  - *Any indirect call/jump can target any valid indirect branch target*
  - Could be combined with fine-grained compiler-based CFI (LLVM CFI)

Comparison with HAFIX++

	BE-Support	FE-Support	Shared library & Multitasking	Granularity	Overhead
XFI Budiu et al, ASID 2006	−	✓	✓	Coarse	3.75%
HAFIX Davi et al., DAC 2015	✓	−	−	[illegible]	2%
LandHere [illegible]	−	−	✓	[illegible]	N/A
[illegible]	[illegible]	✓	−	Fine	1%
Intel CET https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf	✓	−	✓	Coarse	N/A
HAFIX++ Sullivan et al., DAC 2016	✓	✓	✓	Fine	1.75%

Architectural dependent optimizations

Can branch to any call/jump target with *endbranch* inst.