Run-time Attacks and Defenses

Prof. Dr.-Ing. Ahmad-Reza Sadeghi
System Security Lab
Technische Universität Darmstadt
Germany
Summer Term 2017
Three Decades of Runtime Attacks

Morris Worm 1988

Code Injection
**AlephOne** 1996

return-into-libc
**Solar Designer** 1997

Return-oriented programming
**Shacham** CCS 2007

Borrowed Code Chunk Exploitation
**Krahmer** 2005

Continuing Arms Race

...
Recent Attacks

**Attacks on Tor Browser [2013]**
FBI Admits It Controlled Tor Servers Behind Mass Malware Attack.

**Cisco Router Exploit [2016]**
Million CISCO ASA Firewalls potentially vulnerable to attacks (IKE buffer overflow)

**Stagefright [Drake, BlackHat 2015]**
These issues in Stagefright code critically expose 95% of Android devices, an estimated 950 million devices

**The Million Dollar Dissident [2016]**
Government targeted human rights defender with a chain of zero-day exploits to infect his iPhone with spyware.
Remote Android Vulnerability Case: Stagefright
Remote Android Vulnerability Case: Stagefright
Remote Android Vulnerability Case: Stagefright

libStagefright: Native Android library that can be used by Apps to process media files

Adversary

Process Memory
Android 4.0.1

- Application
- libStagefright
- Library 1
- Non-randomized Code
Remote Android Vulnerability Case: Stagefright

libStagefright: Native Android library that can be used by Apps to process media files

Example Payloads:
- Connect back to attacker
- Install Malware/Rootkit
- Steal Credentials

No user interaction required
Stagefright – Details

• **Goal: Remote Code Execution**

• **Root Cause**
  • Integer Overflow in Androids media library (called stagefright)
  • More precisely: While parsing the information header of video files

• **Affected Versions**
  • Android 2.2 – 5.1
  • Versions < 4.1 do not fully implement ASLR, hence, no information leak is required to launch an attack

• **Attack Vectors:**
  • MMS (automatically processed by stagefright, requires **NO** user interaction)
  • Internet Browser (via download, user must click on a link)
  • Messaging Apps (WhatsApp, TextSecure...)

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

Industry Efforts on Defenses

- Microsoft EMET includes a ROP detection engine
- Microsoft Control Flow Guard (CFG) in Windows 10
- Google’s compiler extension VTV (Virtual Table Verification)
- Intel’s Hardware Extension CET (Control-flow Enforcement Technology)
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

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

Industry Efforts on Defenses

• Microsoft EMET includes a ROP detection engine
• Microsoft Control Flow Guard (CFG) in Windows 10
• Google’s compiler extension VTV (Virtual Table Verification)
• Intel’s Hardware Extension CET (Control-flow Enforcement Technology)

Hot Topic of Research

• A large body of recent literature on attacks and defenses
Now, the whole story .....
Problem Roots

- There are several observations
  1. A programming error leads to a program-flow deviation
  2. Missing bounds checking
     - Languages like C, C++, or assembler do not automatically enforce bounds checking on data inputs
  3. An adversary can provide inputs that influence the program flow
Runtime Attacks

Code-Injection Attack

Code-Reuse Attack

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)

Adversary

Data flow

Program flow
Runtime Attacks

Code-Injection Attack

- **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)

- **Adversary** inject malicious code

Code-Reuse Attack

- **Adversary**

- **Data flow**
- **Program flow**
Runtime Attacks

**Code-Injection Attack**

- **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)

**Code-Reuse Attack**

- **Adversary**
- **Corrupt code pointer**
- **Inject malicious code**

**Data flow**

**Program flow**
Runtime Attacks

Code-Injection Attack

- 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)

Code-Reuse Attack

- Adversary injects malicious code
- Data Execution Prevention (DEP)

Adversary injects malicious code to corrupt code pointer.
Runtime Attacks

Code-Injection Attack

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)

Code-Reuse Attack

Adversary

Data Execution Prevention

ject malicious code

corrupt code pointer

Data flow

Program flow
Runtime Attacks

Code-Injection Attack

- **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)

- **Adversary** injects malicious code

- **Data Execution Prevention (DEP)**

- **Corrupt code pointer**

- Data flow
- Program flow

Code-Reuse Attack

- **Adversary**

- **Corrupt code pointer**

- Data flow
- Program flow
It started with return-into-libc

Basic idea of return-into-libc

- Redirect execution to functions in shared libraries
- Main target is UNIX C library libc
  - Libc is linked to nearly every Unix program
  - Defines system calls and other basic facilities such as open(), malloc(), printf(), system(), execve(), etc.
- Attack example: `system ("/bin/sh"), exit()`
Limitations of r2libc

- No branching, i.e., no arbitrary code execution
- Critical functions can be eliminated or wrapped
Return-oriented Programing (ROP): Prominent Code-Reuse Attack
ROP: Basic Ideas/Steps

- Use small instruction sequences
- 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
ROP Attack Technique: Overview

Program Stack

Program Code

Sequence 1
- asm_ins
- RET

Sequence 2
- POP REG1
- POP REG2
- RET

Sequence 3
- asm_ins
- RET
ROP Attack Technique: Overview

Corrupt Control Structures

Program Stack
- Return Address 3
- Value 2
- Value 1
- Return Address 2
- Return Address 1

Program Code
- Sequence 1
  - asm_ins
  - RET
- Sequence 2
  - POP REG1
  - POP REG2
  - RET
- Sequence 3
  - asm_ins
  - RET
ROP Attack Technique: Overview

Program Stack:
- Return Address 1
- Value 1
- Value 2
- Return Address 2
- Return Address 3

Program Code:

Sequence 1:
- asm_ins
- RET

Sequence 2:
- POP REG1
- POP REG2
- RET

Sequence 3:
- asm_ins
- RET

SP
REG1:
REG2:
ROP Attack Technique: Overview

Program Stack

- Return Address 3
- Value 2
- Value 1
- Return Address 2
- Return Address 1

SP

Program Code

- Sequence 1
  - asm_ins
  - RET
- Sequence 2
  - POP REG1
  - POP REG2
  - RET
- Sequence 3
  - asm_ins
  - RET

REG1:

REG2:
ROP Attack Technique: Overview

Program Stack
- Return Address 1
- Value 1
- Return Address 2
- Value 2
- Return Address 3

Program Code
- Sequence 1
  - asm_ins
  - RET
- Sequence 2
  - POP REG1
  - POP REG2
  - RET
- Sequence 3
  - asm_ins
  - RET

SP
REG1:
REG2:
ROP Attack Technique: Overview

Program Stack
- Return Address 1
- Return Address 2
- Return Address 3
- Value 1
- Value 2

Program Code
- Sequence 1
  - asm_ins
  - RET
- Sequence 2
  - POP REG1
  - POP REG2
  - RET
- Sequence 3
  - asm_ins
  - RET

SP
- REG1:
- REG2:
ROP Attack Technique: Overview

Program Stack

Return Address 1
Value 1
Return Address 2
Value 2
Return Address 3

Program Code

Sequence 1
asm_ins
RET

Sequence 2
POP REG1
POP REG2
RET

Sequence 3
asm_ins
RET

SP
ROP Attack Technique: Overview

Program Stack

Return Address 1
Value 1
Return Address 2
Value 2
Return Address 3

Program Code

Sequence 1
asm_ins
RET

Sequence 2
POP REG1
POP REG2
RET

Sequence 3
asm_ins
RET

REG1: Value 1
REG2: Value 2
ROP Attack Technique: Overview

Program Stack
- Return Address 3
- Value 2
- Value 1
- Return Address 2
- Return Address 1

Program Code
- Sequence 1
  - asm_ins
  - RET
- Sequence 2
  - POP REG1
  - POP REG2
  - RET
- Sequence 3
  - asm_ins
  - RET

SP
- REG1:
  - Value 1
- REG2:
  - Value 2
ROP Attack Technique: Overview

Program Stack:
- Return Address 1
- Return Address 2
- Return Address 3
- Value 1
- Value 2

Program Code:
- Sequence 1
  - asm_ins
  - RET
- Sequence 2
  - POP REG1
  - POP REG2
  - RET
- Sequence 3
  - asm_ins
  - RET

REG1:
- Value 1

REG2:
- Value 2
ROP Attack Technique: Overview

Program Stack

- Return Address 1
- Return Address 2
- Return Address 3
- Value 1
- Value 2

Program Code

- Sequence 1
  - asm_ins
  - RET
- Sequence 2
  - POP REG1
  - POP REG2
  - RET
- Sequence 3
  - asm_ins
  - RET

SP

REG1:
- Value 1

REG2:
- Value 2
ROP Attack Technique: Overview

ROP shown to be Turing-complete
Code Injection vs. Code Reuse

- **Code Injection** – *Adding a new node to the CFG*
  - Adversary can execute arbitrary malicious code
    - open a remote console (classical shellcode)
    - exploit further vulnerabilities in the OS kernel to install a virus or a backdoor

- **Code Reuse** – *Adding a new path to the CFG*
  - Adversary is limited to the code nodes that are available in the CFG
  - Requires reverse-engineering and static analysis of the code base of a program
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
Defense Implementation Approaches

- **Binary Instrumentation**: Binary analysis is limited → Defense will be incomplete
- **Compiler Extensions**: Requires recompilation of applications → Legacy applications remain vulnerable
- **Code Annotations**: Developers are no security experts → Do everything to “make it just work”
- **New Languages**: Lack of trained developer → Will it still exist in 5-10 years?
EPISODE I

Code Randomization

Make gadgets locations unpredictable
General Idea: Software Diversity
General Idea: Software Diversity
General Idea: Software Diversity

Adversary
General Idea: Software Diversity
General Idea: Software Diversity
Fine-Grained ASLR

Application Run 1

Library (e.g., libc)
Fine-Grained ASLR

Application Run 1

Library (e.g., libc)

Application Run 2

Library (e.g., libc)
Fine-Grained ASLR

Application Run 1

Library (e.g., libc)

- Instruction Sequence 1
  - RET
- Instruction Sequence 2
  - RET
- Instruction Sequence 3
  - RET

Application Run 2

Library (e.g., libc)
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
Key Insight and Observation

- **Goal:** Exploit a memory disclosure
  - Leak of a single address leading to leak of entire memory pages

- **Observations**
  - Leaked address will reside in a 4KB aligned memory page
  - Determine the page boundaries and **disassemble** the 4 KB page
  - Disassembled page contains references to other pages
Key Insight and Observation

- **Goal:** Exploit a memory disclosure
  - Leak of a single address leading to leak of entire memory pages

- **Observations**
  - Leaked address will reside in a 4KB aligned memory page
  - Determine the page boundaries and **disassemble** the 4 KB page
  - Disassembled page contains references to other pages

![Diagram](image-url)
Key Insight and Observation

- Goal: Exploit a memory disclosure
  - Leak of a single address leading to leak of entire memory pages

- Observations
  - Leaked address will reside in a 4KB aligned memory page
  - Determine the page boundaries and **disassemble** the 4 KB page
  - Disassembled page contains references to other pages
Key Insight and Observation

- **Goal:** Exploit a memory disclosure
  - Leak of a single address leading to leak of entire memory pages
- **Observations**
  - Leaked address will reside in a 4KB aligned memory page
  - Determine the page boundaries and **disassemble** the 4 KB page
  - Disassembled page contains references to other pages
Code Randomization: Attack & Defense Techniques

Static Attack

Application

Function_A

Function_B

<instructions>
call Function_A
pop ebx
pop ecx
pop edx
ret

Attack Timeline

- Morris Worm / Return to libc [Solar Designer Bugtraq'97]
Code Randomization: Attack & Defense Techniques

Static Attack

(Fine-grained) Randomization

RX
Application

Attack Timeline

• Morris Worm / Return to libc [Solar Designer Bugtraq’97]
Code Randomization: Attack & Defense Techniques

Static Attack

(Fine-grained) Randomization

Application

Function_A

Function_B
<instructions>
call Function_A
pop ebx
pop ecx
ret

Attack Timeline

- Morris Worm / Return to libc [Solar Designer Bugtraq’97]
- Just-In-Time ROP [Snow et al. IEEE S&P’13]

Direct code disclosure
Code Randomization: Attack & Defense Techniques

- Static Attack
- Direct code disclosure
- (Fine-grained) Randomization
- Execute-only Memory

Application

Execute-only Memory (XoM)

Attack Timeline
- Morris Worm / Return to libc [Solar Designer Bugtraq’97]
- Just-In-Time ROP [Snow et al. IEEE S&P’13]
Code Randomization: Attack & Defense Techniques

- Static Attack
- Direct code disclosure
- (Fine-grained) Randomization
- Execute-only Memory

Application

Execute-only Memory (XoM)

Pointers

Return Address

Attack Timeline

- Morris Worm / Return to libc [Solar Designer Bugtraq’97]
- Just-In-Time ROP [Snow et al. IEEE S&P’13]
Code Randomization: Attack & Defense Techniques

Static Attack

Direct code disclosure

(Fine-grained) Randomization

Execute-only Memory

Indirect code disclosure

Pointers

Return Address

Application

Execute-
Function_B

<instructions>
call Function_A
pop ebx
pop ecx
ret

Attack Timeline

- 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

- Static Attack
- Direct code disclosure
- Indirect code disclosure
- (Fine-grained) Randomization
- Execute-only Memory
- Code-pointer hiding
- Execute-only Memory (XoM)
- Trampoline
- Pointers
- Return Address

Attack Timeline
- 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

- Static Attack
- Direct code disclosure
- Indirect code disclosure

(Fine-grained) Randomization

Execute-only Memory
- (XoM)

Execute-only Memory (XoM)

Pointer hiding

XoM

Return Address

Call Function_A

Pop ebx

Pop ecx

Ret

Attack Timeline

- 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

- Static Attack
- Direct code disclosure
- Indirect code disclosure
- (Fine-grained) Randomization
- Execute-only Memory
- Code-pointer hiding
- Execute-only Memory (XoM)
- XoM
- Pointers
- Return Address

Attack Timeline
- Morris Worm / Return to libc [Solar Designer Bugtraq'97]
- Just-In-Time ROP [Snow et al. IEEE S&P’13]
- Isomer (Attack) [Davi et al. NDSS’15]

Trampoline Reuse Attacks
Code Randomization: Attack & Defense Techniques

- Static Attack
- Direct code disclosure
- Indirect code disclosure
- (Fine-grained) Randomization
- Execute-only Memory
- Code-pointer hiding
- Execute-only
- Trampoline Target
  - pop ebx
  - pop ecx
  - ret
- XoM
- Pointers
  - Return Address

Attack Timeline:
- 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]

Trampoline Reuse Attacks
Code Randomization: Attack & Defense Techniques

- Static Attack
- Direct code disclosure
- Indirect code disclosure
- (Fine-grained) Randomization
- Execute-only Memory
- Code-pointer hiding
- Execute-only Memory
- Return Address
- Pointers
- Trampoline
- XoM

**Attack Timeline**
- 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]

**Register randomization**
**Trampoline Reuse Attacks**
Code Randomization: Attack & Defense Techniques

- Application
- XoM
- XoM
- Trampoline
- Function Pointer

Attack Timeline
Code Randomization: Attack & Defense Techniques

Attack Timeline

XoM

Pointers

Function Pointer

Virtual Table

virt. Function1

virt. Function2
Code Randomization: Attack & Defense Techniques

Attack Timeline

- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]

Function Reuse Attacks

Pointers
- Function Pointer
- Virtual Table
  - virt. Function1
  - virt. Function2
Code Randomization: Attack & Defense Techniques

- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]

Function Reuse Attacks
- Trampolines & Booby Traps

Pointers
- Function Pointer
- Virtual Table

Application
- XoM
- XoM

Attack Timeline
Code Randomization: Attack & Defense Techniques

![Diagram showing code randomization with attack timeline and defense techniques](image)

- **Attack Timeline**
  - Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P'15]

- **Application**
  - XoM
  - X-Virtual Table
    - vFunc2 Tramp
    - vFunc1 Tramp

- **Pointers**
  - Function Pointer
  - Virtual Table
    - Ptr X-virt table

- **Function Reuse Attacks**
  - Trampolines & Booby Traps

- **Pointers & Virtual Table**
  - XoM
Code Randomization: Attack & Defense Techniques

Attack Timeline

- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
Code Randomization: Attack & Defense Techniques

Attack Timeline

- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
Code Randomization: Attack & Defense Techniques

- Trampolines & Booby Traps
  - Brute-force Attacks on Entropy
  - Function Reuse Attacks

- Pointer Attacks
  - Function Pointer
  - Virtual Table
  - Ptr X-virt table

- XoM
  - X-Virtual Table
  - vFunc1 Tramp
  - vFunc2 Tramp

- Attack Timeline
  - Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
  - Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]
Code Randomization: Attack & Defense Techniques

Brute-force Attacks on Entropy

Booby Traps Terminate Process

Function Reuse Attacks

Trampolines & Booby Traps

Pointers

Function Pointer

Virtual Table

Ptr X-virt table

XoM

X-Virtual Table

vFunc2 Tramp

vFunc1 Tramp

Attack Timeline

- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
- Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]
Code Randomization: Attack & Defense Techniques

- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
- Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]
Code Randomization: Attack & Defense Techniques

- **Brute-force Attacks on Entropy**
- **Function Reuse Attacks**
- **Trampolines & Booby Traps**
- **Booby Traps Terminate Process**

**Attack Timeline**

- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
- Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]
Code Randomization: Attack & Defense Techniques

JIT Code Attacks

Booby Traps to Terminate Process

Function Reuse Attacks

Trampolines & Booby Traps

Pointers
- Function Pointer
- Virtual Table
- Ptr X-virt table

Application
- JIT Code
- XoM

XoM
- X-Virtual Table
  - vFunc1 Tramp
  - vFunc2 Tramp
  - Booby Trap

Attack Timeline
- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
- Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]
Code Randomization: Attack & Defense Techniques

- **JIT Code Attacks**
  - Same Protection as for AOT Code
  - Booby Traps Terminate Process

- **Brute-force Attacks on Entropy**
  - Trampolines & Booby Traps

- **Function Reuse Attacks**
  - Pointers
    - Function Pointer
    - Virtual Table
    - Ptr X-virt table

**XoM**
- X-Virtual Table
- vFunc1 Tramp
- vFunc2 Tramp

**Attack Timeline**
- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
- Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]
Code Randomization: Attack & Defense Techniques

- **JIT Code Attacks**
- **Brute-force Attacks on Entropy**
- **Function Reuse Attacks**
- **Trampolines & Booby Traps**

**Booby Traps**
- **Terminate Process**

**Function Pointer**
- **Virtual Table**
- **XoM**

**Application**
- **JIT Code**
- **XoM**
- **XoM**
- **XoM**

**Pointers**
- **Function Pointer**
- **Trampoline**
- **Virtual Table**
- **Ptr X-virt table**

**Attack Timeline**
- **Counterfeit Object-oriented Programming (COOP)**
  [Schuster et al. IEEE S&P’15]
- **Crash-Resistant Oriented Programming**
  [Gawlik et al. NDSS’16]
Code Randomization: Attack & Defense Techniques

JIT Code Attacks

Brute-force Attacks on Entropy

Function Reuse Attacks

Trampolines & Booby Traps

Pointers

Trampolines & Booby Traps

Terminate Process

Same Protection as for AOT Code

Booby Traps

Attack Timeline

Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]

Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]

Trampoline Reuse for Single Function Pointers

JIT Code

Application

XoM

XoM

XoM

X-Virtual Table
vFunc2 Tramp
Booby Trap
vFunc1 Tramp

Function Pointer

Virtual Table
Ptr X-virt table
Code Randomization: Attack & Defense Techniques

- JIT Code Attacks
- Brute-force Attacks on Entropy
- Function Reuse Attacks
- Trampolines & Booby Traps

- Same Protection as for AOT Code
- Booby Traps Terminate Process

- XoM
- X-Virtual Table
  - vFunc2 Tramp
  - vFunc1 Tramp

- Function Pointer
- Virtual Table
  - Ptr X-virt table

- Attack Timeline
  - Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
  - Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]

- Attack Surface
  - Large enough?

- Trampoline Reuse for Single Function Pointers
**Code Randomization:**

**Attack & Defense Techniques**

- **JIT Code Attacks**
- **Brute-force Attacks on Entropy**
- **Function Reuse Attacks**

**Attack Timeline**

- Counterfeit Object-oriented Programming (COOP) [Schuster et al. IEEE S&P’15]
- Crash-Resistant Oriented Programming [Gawlik et al. NDSS’16]
- Address Oblivious Code Reuse [Rudd et al. NDSS’17]

**Diagram Elements**

- **Application**
- **JIT Code**
- **XoM**
- **X-Virtual Table**
- **vFunc1 Tramp**
- **vFunc2 Tramp**
- **Booby Trap**
- **Trampoline**
- **Function Pointer**
- **Virtual Table**
- **Ptr X-virt table**

**Additional Notes**

- Terminate Process
- Same Protection as for AOT Code
- Booby Traps
- Trampolines & Booby Traps
- JIT Code Crash-Resistant Oriented Programming
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
Objectives

- secure: prevent code reuse + memory disclosure
- comprehensive: ahead of time + JIT
- practical: real browsers
- fast: Less than 6% overhead
Readactor++: Architecture

Compile time

Compiler

```
#include <stdio.h>
int main(int argc, char **argv) {
    printf("Hello World!");
    return 0;
}
```
Readactor++: Architecture

Compile time

Runtime

Compiler

Application

- Code / Data separation
- Fine-grained code randomization
- Code-pointer hiding
Readactor++: Architecture

Compile time

Compiler

Runtime

Application

• Code / Data separation
• Fine-grained code randomization
• Code-pointer hiding

Operating System
Readactor++: Architecture

## Compile time

- **Compiler**
  - Compile time

## Runtime

- **Application**
  - Code / Data separation
  - Fine-grained code randomization
  - Code-pointer hiding

- **Operating System**
- **Hardware**

### Diagram:

- Compiler flow diagram with compile and runtime stages.
- Application with code/data separation and fine-grained randomization.
- Operating system and hardware connections.
**Readactor++: Architecture**

**Compiler**

Compile time

**Application**

Runtime

- Code / Data separation
- Fine-grained code randomization
- Code-pointer hiding

**Operating System**

- Execute-only support
- Thin-Hypervisor

**Hardware**

- Memory Virtualization
- Extended Page Tables (EPT)
Leakage Resilient Layout Randomization with no HW Support

LR²:
Leakage-Resilient Layout Randomization for Mobile Devices

The Network and Distributed System Security Symposium (NDSS) 2016

Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Ahmad-Reza Sadeghi
LR$^2$: Leakage Resilient Layout Randomization

[Braden et al. NDSS’16]
LR²: Leakage Resilient Layout Randomization

[Braden et al. NDSS’16]

---

**Compiler**
- LR²

**Application**
- Code / Data separation
- Fine-grained code randomization
- Code-pointer hiding

**Operating System**
- Execute-only support

**Hardware**
- Memory Virtualization

**Thin-Hypervisor**
- Extended Page Tables (EPT)
LR²: Leakage Resilient Layout Randomization

[Braden et al. NDSS’16]

Compile time

Runtime

Compiler

Application

- Code / Data separation
- Fine-grained code randomization
- Code-pointer hiding

Code

Data

0x7FFFFFFF

0x00000000

read memory
LR²: Leakage Resilient Layout Randomization
[Braden et al. NDSS’16]

Compile time

Compiler

Runtime

Application

- Code / Data separation
- Fine-grained code randomization
- Code-pointer hiding
- Sandboxing Read-instruction (to prevent read access to the code section)
**LR²: Leakage Resilient Layout Randomization**

[Braden et al. NDSS’16]

### Compiler

- **Code / Data separation**
- **Fine-grained code randomization**
- **Code-pointer hiding**
- **Sandboxing Read-instruction (to prevent read access to the code section)**

### Runtime

- **r1 <- addr**; load target addr in r1
- **r1 = r1 & 0x7FFFFFFF**; ensures r1 is always < 0x80000000 by removing the highest bit
- **r0 <- [r1]**; read memory in r0
Kernel CFI is not enough: Data-Only Attacks against Page Tables

PT-Rand:
Practical Mitigation of Data-only Attacks against Page Tables,
NDSS 2017
Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi
Selfrando: Hardening Tor Browser against De-anonymization Attacks

Selfrando:
Securing the Tor Browser against De-anonymization Exploits,
Privacy Enhancing Technologies Symposium (PETS) 2016
Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, Ahmad-Reza Sadeghi
COOP - Workflow

Main-Loop Gadget:
- For Every C++ Object
- CALL
- 2nd virtual Function

Initial control-flow hijacking

Attacker Controlled Memory
- vTable 1
  - vFunc ptr 1
  - vFunc ptr 2
  - vFunc ptr 3
- vTable 2
  - vFunc ptr 1
  - vFunc ptr 2
  - vFunc ptr 3
  - vFunc ptr 4

vtable ptr
Var 1
Var 2
Var 3
Var 4

0x100000
+0 vtable ptr
+4 Var 1
+8 Var 2
+12 Var 3
+16 Var 4
+20

loop
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]
Original CFI Label Checking
[Abadi et al., CCS 2005 & TISSEC 2009]

BBL A

label_A
ENTRY
asm_ins, ...
EXIT

BBL B

label_B
ENTRY
asm_ins, ...
EXIT
Original CFI Label Checking
[Abadi et al., CCS 2005 & TISSEC 2009]

BBL A

label_A
ENTRY
asm_ins, ...
EXIT

CFI CHECK:
EXIT(A) -> label_B ?

BBL B

label_B
ENTRY
asm_ins, ...
EXIT

C

A

B

C

Original CFI Label Checking
[Abadi et al., CCS 2005 & TISSEC 2009]
Original CFI Label Checking
[Abadi et al., CCS 2005 & TISSEC 2009]

BBL A

CIF CHECK:
EXIT(A) -> label_B ?

BBL B
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
Challenges

Performance

Control-Flow Graph Analysis and Coverage
Label Granularity: Trade-Offs

- Many CFI checks if unique labels assigned per node
Label Granularity: Trade-Offs

- Many CFI checks if unique labels assigned per node
Label Granularity: Trade-Offs

- Many CFI checks if unique labels assigned per node
Label Granularity: Trade-Offs

- Many CFI checks if unique labels assigned per node

Exit(B) == [Label_3, Label_4, Label_5]
Label Granularity: Trade-Offs

- Many CFI checks are required if unique labels are assigned per node
Label Granularity: Trade-Offs

• Many CFI checks are required if unique labels are assigned per node
Label Granularity: Trade-Offs

- Many CFI checks are required if unique labels are assigned per node
Label Granularity: Trade-Offs

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

Exit(B) == [Label_3, Label_4, Label_5]
Label Granularity: Trade-Offs (1/2)

- Many CFI checks are required if unique labels are assigned per node
Many CFI checks are required if unique labels are assigned per node.
Label Granularity: Trade-Offs (1/2)

- Many CFI checks are required if unique labels are assigned per node
Label Granularity: Trade-Offs (1/2)

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

Exit(B) == [Label_3, Label_4, Label_5]
Label Granularity: Trade-Offs (2/2)

- Optimization step: Merge labels to allow single CFI check
- However, this allows for unintended control-flow paths
Label Granularity: Trade-Offs (2/2)

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

Exit(B) == Label_3
Label Granularity: Trade-Offs (2/2)

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

```
Exit(B) == Label_3
Exit(C) == Label_3
```
Label Granularity: Trade-Offs (2/2)

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

**Diagram**

- Exit(B) == Label_3
- Exit(C) == Label_3
Label Granularity: Trade-Offs (2/2)

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

Exit(B) == Label_3

Exit(C) == Label_3
Label Problem for Returns

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

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

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

\[ \text{Exit}(R) == \{\text{Label}_1, \text{Label}_2\} \]
Label Problem for Returns

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

```
Exit(R) == [Label_1, Label_2]
```
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

\[
\text{Exit}(R) = \{\text{Label}_1, \text{Label}_2\}
\]
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

Exit(R) == [Label_1, Label_2]
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

Exit(R) == [Label_1, Label_2]
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

\[
\text{Exit}(R) = [\text{Label}_1, \text{Label}_2]
\]
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-Edge CFI**

**Backup State**

- **Backup storage for return addresses**

**Backward-Edge CFI**

**Exit(R) == [Label_1, Label_2]**
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)
Practical CFI for Commercial-Off-The-Shelf Binaries
“Practical” (coarse-grained) Control Flow Integrity (CFI)

Recently, many solutions proposed

- CCFIR [IEEE S&P’13]
- MS BlueHat Prize
- kBouncer [USENIX Sec’13]
- ROPecker [NDSS’14]
- ROPGuard [Microsoft EMET]
- CFI for COTS Binaries [USENIX Sec’13]

Main Coarse-Grained CFI Policies

- CFI Policy 1: Call-Preceded Sequences
  - Returns need to target a call-preceded instruction

Application

| CALL A | asm_ins | asm_ins | CALL B | asm_ins | asm_ins | CALL C | asm_ins |
Main Coarse-Grained CFI Policies

- CFI Policy 1: Call-Preceded Sequences
  - Returns need to target a call-preceded instruction

Application

```
CALL A
asm_ins
asm_ins
CALL B
asm_ins
CALL C
asm_ins
```
Main Coarse-Grained CFI Policies

- CFI Policy 1: Call-Preceded Sequences
  - Returns need to target a call-preceded instruction

![Diagram of Application with CALL A, CALL B, CALL C, and RET nodes connected by arrows indicating call and return sequences.]
Main Coarse-Grained CFI Policies

- CFI Policy 1: Call-Preceded Sequences
  - Returns need to target a call-preceded instruction

```
Application
CALL A
asm_ins
asm_ins
CALL B
asm_ins
CALL C
asm_ins

RET
```
Main Coarse-Grained CFI Policies

- **CFI Policy 1: Call-Preceded Sequences**
  - Returns need to target a call-preceded instruction

- **CFI Policy 2: Behavioral-Based Heuristics**
  - Prohibit a chain of $N$ short sequences each consisting of less than $S$ instructions

---

**Application**

```
CALL A
asm_ins
asm_ins
CALL B
asm_ins
CALL C
asm_ins
```

RET

1 < S
2 < S
... < S
Main Coarse-Grained CFI Policies

- **CFI Policy 1: Call-Preceded Sequences**
  - Returns need to target a call-precended instruction

- **CFI Policy 2: Behavioral-Based Heuristics**
  - Prohibit a chain of \( N \) short sequences each consisting of less than \( S \) instructions

Application:

```
CALL A
asm_ins
asm_ins
CALL B
asm_ins
CALL C
asm_ins
```

RET

\[
\begin{align*}
1 & \quad < S \\
2 & \quad < S \\
\ldots & \quad < S \\
N & \quad < S \\
\end{align*}
\]

\[
\begin{align*}
> S & \quad > S \\
\end{align*}
\]
Main Coarse-Grained CFI Policies

• CFI Policy 1: Call-Preceded Sequences
  • Returns need to target a call-preceded instruction

• CFI Policy 2: Behavioral-Based Heuristics
  • Prohibit a chain of \( N \) short sequences each consisting of less than \( S \) instructions
Main Coarse-Grained CFI Policies

- **CFI Policy 1: Call-Preceded Sequences**
  - Returns need to target a call-preceded instruction

- **CFI Policy 2: Behavioral-Based Heuristics**

  ![Threshold Setting]

  - **kBouncer**: (N=7; S<=20)
  - **ROPecker**: (N=11; S<=6)

Application

- CALL A
- asm_ins
- CALL B
- asm_ins
- CALL C
- asm_ins

RET
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
# Most Restrictive Coarse-Grained CFI

## CFI Policy

<table>
<thead>
<tr>
<th>CFI Policy</th>
<th>kBouncer</th>
<th>ROPecker</th>
<th>ROPGuard</th>
<th>EMET</th>
<th>CFI for COTS Binaries</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CFI Policy 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Call-Preceded Sequences</em></td>
<td><img src="checkmark.png" alt="Yes" /></td>
<td><img src="cross.png" alt="No" /></td>
<td><img src="checkmark.png" alt="Yes" /></td>
<td><img src="checkmark.png" alt="Yes" /></td>
<td><img src="checkmark.png" alt="Yes" /></td>
</tr>
<tr>
<td><strong>CFI Policy 2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Behavioral-Based Heuristics</em></td>
<td><img src="checkmark.png" alt="Yes" /></td>
<td><img src="checkmark.png" alt="Yes" /></td>
<td><img src="cross.png" alt="No" /></td>
<td><img src="cross.png" alt="No" /></td>
<td><img src="checkmark.png" alt="Yes" /></td>
</tr>
<tr>
<td><strong>Time of CFI Check</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WinAPI</td>
<td>2 Pages Sliding Window / Critical Functions</td>
<td>WinAPI/Critical Functions</td>
<td>Indirect Branch</td>
<td><img src="checkmark.png" alt="Yes" /></td>
</tr>
</tbody>
</table>

![No Restriction](cross.png) ![CFI Policy](checkmark.png)

*Here only core policies shown. Other policies available in our analysis*
CFI Defense Literature

2002
- Program Shepherding
  Kiriansky et al. (USENIX Sec.)

2005
- Control-Flow Integrity (CFI)
  Abadi et al. (CCS 2005)

2006
- XFI
  Abadi et al. (OSDI)

2010
- EMET
  Microsoft

2011
- Control-Flow Locking
  Bletch et al. (ACSAC)

2012
- Branch Regulation
  Kayaalp et al. (ISCA)

2013
- Control-Flow Restrictor
  Pewny et al. (ACSAC)

2014
- Forward-Edge CFI
  Tice et al. (USENIX Sec.)

2015
- Control-Flow Guard
  Microsoft

2016
- Vtrust
  Zhang et al. (NDSS)
Hardware CFI
Why Leveraging Hardware for CFI?

- **Efficiency**
  - Dedicated CFI instructions
    - CFI_RETURN
    - CFI_JUMP
    - CFI_CALL

- **Security**
  - Isolated CFI storage
    - CFI Memory
      - Branch Targets
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:
Hardware-Assisted Flow Integrity Extension
Design Automation Conference (DAC 2015), Best Paper Award
Orlando Arias, Lucas Davi, Matthias Hanreich, Yier Jin, Patrick Koeberl,
Debayan Paul, Ahmad-Reza Sadeghi, Dean Sullivan
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
Overview on HAFIX

- **Contributions**
  - Efficient CFI hardware implementation for Intel Siskiyou Peak and SPARC-LEON3
  - Dedicated CFI instructions and memory

- **HAFIX Policies**
  1. Function *returns* only allowed to target active call sites or the last active call site
  2. Function *calls* need to target a valid function entry

- **Limitations**
  - No policy enforcement for indirect jumps
  - Coarse-grained policy for indirect calls
<table>
<thead>
<tr>
<th>Objectives</th>
<th>Backward-Edge and Forward-Edge CFI</th>
<th>Stateful, CFI policy agnostic</th>
</tr>
</thead>
<tbody>
<tr>
<td>No burden on developer</td>
<td>Stateful, CFI policy agnostic</td>
<td>No code annotations/changes</td>
</tr>
<tr>
<td>Security</td>
<td>Security</td>
<td>Hardware protection</td>
</tr>
<tr>
<td>High performance</td>
<td>High performance</td>
<td>On-Chip Memory for CFI Data</td>
</tr>
<tr>
<td>Enabling technology</td>
<td>Enabling technology</td>
<td>No unintended sequences</td>
</tr>
<tr>
<td>Compatibility to legacy code</td>
<td>Compatibility to legacy code</td>
<td>&lt; 3% overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All applications can use CFI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>features</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Support of Multitasking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CFI and non-CFI code on same</td>
</tr>
<tr>
<td></td>
<td></td>
<td>platform</td>
</tr>
</tbody>
</table>
HAFIX++ ISA Extensions

- 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)
Function Return Policy

State 0
Normal Execution

CFI State
Only CFI instructions allowed

Function A
- **CFIBR label_A1**
- **CALL B**
- **CFIRET label_A1**
- **Code**

Function B
- **Code**
- **RET**
Function Return Policy

State 0
Normal Execution

Function A
- CFIBR label_A1
- CALL B
- CFIRET label_A1
- Code

Function B
- Code
- RET

CFI State
Only CFI instructions allowed

Function A
- CALL B
- Code

Function B
- Code
- RET
Function Return Policy

State 0
Normal Execution

CFI State
Only CFI instructions allowed

Function A
CFIBR label_A1
CALL B
CFIRET label_A1
Code

Function B
Code
RET

Function A
CALL B
Code

Function B
Code
RET

CFIBR label_A1

Label State Stack (LSS)
### Function Return Policy

#### State 0
**Normal Execution**

- **Function A**
  - `CFIBR label_A1`
  - `CALL B`
  - `CFIRET label_A1`
  - `Code`

- **Function B**
  - `Code`
  - `RET`

#### CFI State
**Only CFI instructions allowed**

- **Function A**
  - `CALL B`
  - `Code`
  - `CFIBR label_A1`

- **Function B**
  - `Code`
  - `RET`

- **Label State Stack (LSS)**
  - `label_A1`
  - `...`

---

**Legend:**
- `CFIBR`: Control Flow Integrity Branch
- `CFIRET`: Control Flow Integrity Return
Function Return Policy

State 0
Normal Execution

CFI State
Only CFI instructions allowed

Function A
- CALL B
- CFIRET label_A1
- Code

Function B
- Code
- RET

Label State Stack (LSS)
- label_A1
- ...

Function A
- CALL B
- Code
- CFIBR label_A1

Function B
- Code
- RET
- CFIBR label_A1

A1
Function Return Policy

**State 0**
Normal Execution

- Function A
  - CFIBR `label_A1`
  - CALL B
  - CFIRET `label_A1`
  - Code

- Function B
  - Code
  - RET

**CFI State**
Only CFI instructions allowed

- Function A
  - CALL B
  - Code

- Function B
  - Code
  - RET

- CFIBR `label_A1`

**Label State Stack (LSS)**

- `label_A1`
- ...
Function Return Policy

State 0
Normal Execution

Function A
- CFIBR label_A1
- CALL B
- CFIRET label_A1
- Code

Function B
- Code
- RET

CFI State
Only CFI instructions allowed

Function A
- CALL B
- Code

Function B
- Code
- RET
- CFIRET label_A1

Label State Stack (LSS)
- label_A1
- ...

A1
Function Return Policy

State 0
Normal Execution

Function A
- CFIBR label_A1
- CALL B
- CFIRET label_A1
- Code

Function B
- Code
- RET

CFI State
Only CFI instructions allowed

Function A
- CALL B
- Code

Function B
- Code
- RET

CFIBR label_A1

Label State Stack (LSS)
- label_A1
- ...
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)
Control-flow Enforcement Technology

[Intel 2016]

State 0
Normal Execution

WAIT_FOR_ENDBRANCH
Only CFI instructions allowed

Function A
- CALL B
- Code

Function B
- BRANCHEND
- Code
- RET
Control-flow Enforcement Technology

[Intel 2016]

**State 0**
Normal Execution

- **Function A**
  - CALL B
  - Code

- **Function B**
  - BRANCHEND
  - Code
  - RET

**WAIT_FOR_ENDBRANCH**
Only CFI instructions allowed

- **Function A**
  - CALL B
  - Code

- **Function B**
  - BRANCHEND
  - Code
  - RET

- **Function B**
  - BRANCHEND
  - Code
  - RET

- **Shadow Stack**

- **Regular Stack**
Control-flow Enforcement Technology

[Intel 2016]

- **State 0**
  - Normal Execution
  - Function A: CALL B, Code
  - Function B: BRANCHEND, Code, RET

- **WAIT_FOR_ENDBRANCH**
  - Only CFI instructions allowed
  - Function A: CALL B, Code
  - Function B: BRANCHEND, Code, RET

- **Shadow Stack**
- **Regular Stack**
Control-flow Enforcement Technology
[Intel 2016]

State 0
Normal Execution

Function A
CALL B
Code

Function B
BRANCHEND
Code
RET

Function A
CALL B
Code

Function B
BRANCHEND
Code
RET

Function B
BRANCHEND
Code
RET

WAIT_FOR_ENDBRANCH
Only CFI instructions allowed

Shadow Stack

Regular Stack
Control-flow Enforcement Technology

[Intel 2016]
Control-flow Enforcement Technology

[Intel 2016]

Function A
- CALL B
- Code

Function B
- BRANCHEND
- Code
- RET

State 0
Normal Execution

Function A
- CALL B
- Code

Function B
- BRANCHEND
- Code
- RET

WAIT_FOR_ENDBRANCH
Only CFI instructions allowed

Function B
- BRANCHEND
- Code
- RET

Shadow Stack
- Return Address

Regular Stack
- Return Address
Control-flow Enforcement Technology

[Intel 2016]

Function A
- CALL B
- Code

Function B
- BRANCHEND
- Code
- RET

State 0
Normal Execution

Function A
- CALL B
- Code

Function B
- BRANCHEND
- Code
- RET

Function B
- BRANCHEND
- Code
- RET

WAIT_FOR_ENDBRANCH
Only CFI instructions allowed

Shadow Stack
- Return Address

Regular Stack
- Return Address
Control-flow Enforcement Technology

[Intel 2016]

Function A
CALL B
Code

Function B
BRANCHEND
Code
RET

State 0
Normal Execution

Function A
CALL B
Code

Function B
BRANCHEND
Code
RET

Function B
BRANCHEND
Code
RET

WAIT_FOR_ENDBRANCH
Only CFI instructions allowed

Shadow Stack
Return Address

Regular Stack
Return Address
Control-flow Enforcement Technology
[Intel 2016]

State 0
Normal Execution
- Function A
  - CALL B
  - Code

Function B
- BRANCHEND
- Code
- RET

WAIT_FOR_ENDBRANCH
Only CFI instructions allowed
- Function A
  - CALL B
  - Code
- BRANCHEND
  - Code
  - RET

- Function B
  - BRANCHEND
  - Code
  - RET

Equal?

Shadow Stack
- Return Address

Regular Stack
- Return Address
Control-flow Enforcement Technology

[Intel 2016]

Function A
- CALL B
- Code

Function B
- BRANCHEND
- Code
- RET

State 0
Normal Execution

Function A
- CALL B
- Code

Function B
- BRANCHEND
- Code
- RET

WAIT_FOR_ENDBRANCH
Only CFI instructions allowed

Function B
- BRANCHEND
- Code
- RET

Shadow Stack
- Return Address

Regular Stack
- Return Address

Equal?
Control-flow Enforcement Technology
[Intel 2016]

State 0
Normal Execution

WAIT_FOR_ENDBRANCH
Only CFI instructions allowed

Function A
CALL B
Code

Function A
CALL B
Code

Function B
BRANCHEND
Code
RET

Function B
BRANCHEND
Code
RET

Function B
BRANCHEND
Code
RET

Regular Stack

Shadow Stack
## Hardware-Based Solutions

<table>
<thead>
<tr>
<th></th>
<th>BE-Support</th>
<th>FE-Support</th>
<th>Shared library &amp; Multitasking</th>
<th>Granularity</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>XFI</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>Coarse</td>
<td>3.75%</td>
</tr>
<tr>
<td>Budiu et al., ASID 2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAFIX</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Coarse</td>
<td>2%</td>
</tr>
<tr>
<td>Davi et al., DAC 2015</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LandHere</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>Coarse</td>
<td>N/A</td>
</tr>
<tr>
<td><a href="http://langalois.com">http://langalois.com</a></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HCFI</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Fine</td>
<td>1%</td>
</tr>
<tr>
<td>Christoulakis et al., CODASPY 2016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel CET</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Coarse</td>
<td>N/A</td>
</tr>
<tr>
<td>Intel Tech Review</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAFIX++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Fine</td>
<td>1.75%</td>
</tr>
<tr>
<td>Sullivan et al., DAC 2016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Architectural dependent optimizations*
Thank you!
Talk to me:
www.trust.cased.de