Motivation

PROTECT

ALL THE NATIVE CODES
Three Decades of Runtime Attacks

- Morris Worm 1988
- Code Injection *AlephOne* 1996
- return-into-libc *Solar Designer* 1997
- Borrowed Code Chunk Exploitation *Krahmer* 2005
- Return-oriented programming *Shacham* CCS 2007
- Continuing Arms Race

...
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


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

Vire, VGuard,
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

Code-Injection Attack

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

Table:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
</tr>
</tbody>
</table>

Data Execution Prevention (DEP)

Adversary

Corrupt code pointer

Inject malicious code

Data flow

Program flow
Return-oriented Programming (ROP): Prominent Code-Reuse Attack

ROP shown to be Turing-complete
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 $\oplus$ 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]
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

Static Attack

Direct code disclosure

Indirect code disclosure

(Fine-grained) Randomization

Execute-only Memory

Code-pointer hiding

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]

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

Trampoline Reuse Attacks

Register randomization
Code Randomization: Attack & Defense Techniques

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

- Same Protection as for AOT Code
- Booby Traps Terminate Process
- Attack Surface Large enough?
- Trampoline Reuse for Single Function Pointers

JIT Code
XoM
Virtual Table
Ptr X-virt table
virt. Function2
Function Pointer

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

Attack Surface

Code Randomization: Attack & Defense Techniques
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 \(\rightarrow\) 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

\[ \text{Exit}(B) = [\text{Label}_3, \text{Label}_4, \text{Label}_5] \]
Label Granularity: Trade-Offs (2/2)

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

```
Exit(B) == Label_3
```

```
Exit(C) == Label_3
```
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]
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 call sites available
   → Restrict returns to their actual caller (shadow stack)

2. Heuristics are ad-hoc and ineffective
   → Adjusted sequence length leads to high false positive

3. Too many indirect jump and call targets
   • Resolving indirect jumps and calls is non-trivial
   → Compromise: Compiler support

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfibr</td>
<td>Issued at call site → setup Backward (BW) Edge</td>
</tr>
<tr>
<td>cfiret</td>
<td>Issue at return site → check BW Edge</td>
</tr>
<tr>
<td>cfiprc</td>
<td>Issued at call site → setup call target</td>
</tr>
<tr>
<td>cfiprj</td>
<td>Issued at jump site → setup jump target</td>
</tr>
<tr>
<td>cfichk</td>
<td>Issued at call/jmp target → check Forward (FW) Edge</td>
</tr>
</tbody>
</table>

- 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)
- Register (LSR)
Indirect Call Policy

State 0
Normal Execution

CFIBR label_A1
CFILSR label_B
CALL *reg
CFIRET label_A1
Code

Function A

Function B

CFICHK label_B
Code
RET

Function B

CFIBR label_A1
CFILSR label_B
CFICHK label_B
CFIRET label_A1

CFI State
Only CFI instructions allowed

Label State
Stack (LSS)

Label State
Register (LSR)

label_A1
label_B
Function Return Policy

State 0
Normal Execution

CFIBR label_A1
CALL B
CFIRET label_A1
Code

Function A

Function B
Code
RET

Function A
CALL B
Code

CFIRET label_A1

Function B
Code
RET

CFIBR label_A1

CFI State
Only CFI instructions allowed

Label State Stack (LSS)

label_A1

A1
HAFIX++ Pipeline

- Fetch
- Decode
  - NOP
  - insn5
  - CFI
  - insn2
  - insn3
  - insn4
  - insn5
- Execute
- Memory
- Write

- CFI Control Unit
  - CFI label
  - Forward CFI to Control Unit
  - Convert CFI to NOP
  - Label does not match
  - → Stop Execution
- CFI Label State Memory
  - label
  - Forward label to CFI Control Unit to check activity
  - Forward label to CFI Control Unit to check activity

Label access in dedicated memory
Function A (25)

- CFICHK 25
- insn
- CFIBR 251
- CFIPRC 31
- CALL *reg
- CFIRET 251
- insn
- CFIPRJ 252
- JMP CFICHK 252
- insn
- CFICHK 252
- insn
- CFIBR
- CFIPRC 45
- CALL *reg
- CFIRET 253
- insn
- RET

Function B (31)

- CFICHK 31
- insn
- insn
- RET

Label State Stack

- 251

Label State Register

- 252

Label 31 valid

Push Label 251 onto LSS

Call Function B

Return to Function A

Store Label 252 off stack and validate

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
  o Initializing and validating the CFI state upon every FW/BW edge
  o I-cache pressure during instruction fetch
  o Effective CPI

• Runtime overhead and problems caused by hardware
  o Branch instruction occur about every 3-5 instructions
  o CFI instructions/operations around every one of them
  o Memory access for CFI metadata is slow
  o CFI metadata could be corrupted if considered data (StackDefiler)
  o CFI metadata could be a bottleneck if placed in code
The Multiple Callers Problem

- We cannot 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

Process 1
- Label State Stack:
  - 0003
  - 7932
  - 3589
  - 9265
  - 1415
  - 1618
- Label State Register
- This is running

Process 2
- Label State Stack:
  - 0287
  - 3536
  - 0452
  - 8459
  - 8182
  - 7182
  - 5772
- Label State Register
- This is being scheduled
The Scheduling Issue

Process 1
This is running

<table>
<thead>
<tr>
<th>Label State Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7932</td>
</tr>
<tr>
<td>3589</td>
</tr>
<tr>
<td>9265</td>
</tr>
<tr>
<td>1415</td>
</tr>
<tr>
<td>0003</td>
</tr>
</tbody>
</table>

LSSP 1618
Label State Register

Process 2
This is being scheduled

I have no clue what CFI is...
The Stack Issue

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

Label State Stack

<table>
<thead>
<tr>
<th>Label State Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>2884</td>
</tr>
<tr>
<td>7950</td>
</tr>
<tr>
<td>3832</td>
</tr>
<tr>
<td>2643</td>
</tr>
<tr>
<td>3846</td>
</tr>
<tr>
<td>7932</td>
</tr>
<tr>
<td>3589</td>
</tr>
<tr>
<td>9265</td>
</tr>
<tr>
<td>1415</td>
</tr>
<tr>
<td>0003</td>
</tr>
</tbody>
</table>

LSSP
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

CFI Subsystem

<table>
<thead>
<tr>
<th>Label State Stack</th>
<th>CFI Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>0287</td>
<td></td>
</tr>
<tr>
<td>3536</td>
<td></td>
</tr>
<tr>
<td>0452</td>
<td></td>
</tr>
<tr>
<td>8459</td>
<td></td>
</tr>
<tr>
<td>8182</td>
<td></td>
</tr>
<tr>
<td>7182</td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>LSSP</td>
<td></td>
</tr>
</tbody>
</table>

Process 1 -- PCB

<table>
<thead>
<tr>
<th>TASK_RUNNING</th>
<th>CFI Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSSP</td>
<td></td>
</tr>
</tbody>
</table>

Process 2 -- PCB

<table>
<thead>
<tr>
<th>TASK_RUNNING</th>
<th>CFI Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSSP</td>
<td></td>
</tr>
</tbody>
</table>
The Scheduling Issue Resolved

CFI Subsystem

Label State Stack

LSSP

1415
0003

Label State Register

Disabled

Process 1 -- PCB

TASK_RUNNING

CFI Context

CFI_ON

...

Process 2 -- PCB

TASK_RUNNING

CFI Context

CFI_OFF

...

0969
2774
3589
1618
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 Subsystem

<table>
<thead>
<tr>
<th>2884</th>
<th>LSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>7950</td>
<td></td>
</tr>
<tr>
<td>3832</td>
<td></td>
</tr>
<tr>
<td>2643</td>
<td></td>
</tr>
<tr>
<td>3846</td>
<td></td>
</tr>
<tr>
<td>7932</td>
<td></td>
</tr>
<tr>
<td>3589</td>
<td></td>
</tr>
<tr>
<td>9265</td>
<td></td>
</tr>
<tr>
<td>1415</td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td></td>
</tr>
</tbody>
</table>

Label State Stack

Process 1 -- PCB

<table>
<thead>
<tr>
<th>TASK_RUNNING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI_CONTEXT</td>
</tr>
</tbody>
</table>

CFI Context

1618

Label State Register
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]

Function A
- \( A_1 \): call [D₁]
- \( A_2 \): call [D₂]
- \( A_3 \): ins
- \( A_4 \): return

Function B
- \( B_1 \): ENDBRANCH
- \( B_2 \): push eax
- \( B_3 \): pop eax
- \( B_4 \): return

Function C
- \( C_1 \): ENDBRANCH
- \( C_2 \): ...
- \( C_3 \): return

Shadow Stack
- \( A_2 \)

Stack
- \( A_3 \)

Data
- \( D_1 \)
- \( B_1 \)
- \( C_1 \)
- \( D_2 \)
- \( B_1 \)
- \( C_1 \)
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: \texttt{branchend}
  - \textit{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++

<table>
<thead>
<tr>
<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>Coarse</td>
<td>3.75%</td>
</tr>
<tr>
<td>HAFIX</td>
<td>✓</td>
<td>-</td>
<td>Coarse</td>
<td>2%</td>
</tr>
<tr>
<td>LandHere</td>
<td>-</td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Intel CET</td>
<td>✓</td>
<td>-</td>
<td>Coarse</td>
<td>N/A</td>
</tr>
<tr>
<td>HAFIX++</td>
<td>✓</td>
<td>✓</td>
<td>Fine</td>
<td>1.75%</td>
</tr>
</tbody>
</table>

- **XFI** (Budiu et al., ASID 2006)
  - BE-Support: No
  - FE-Support: Yes
  - Shared library & Multitasking: Yes
  - Granularity: Coarse
  - Overhead: 3.75%

- **HAFIX** (Davi et al., DAC 2015)
  - BE-Support: Yes
  - FE-Support: No
  - Shared library & Multitasking: No
  - Granularity: Coarse
  - Overhead: 2%

- **LandHere** (http://landhere.galois.com)
  - BE-Support: No
  - FE-Support: No
  - Shared library & Multitasking: Yes
  - Granularity: N/A
  - Overhead: N/A

- **Intel CET**
  - BE-Support: Yes
  - FE-Support: No
  - Shared library & Multitasking: Yes
  - Granularity: Coarse
  - Overhead: N/A

- **HAFIX++** (Sullivan et al., DAC 2016)
  - BE-Support: Yes
  - FE-Support: Yes
  - Shared library & Multitasking: Yes
  - Granularity: Fine
  - Overhead: 1.75%

---

- **Can branch to any call/jump target with *endbranch* inst.**
- **Architectural dependent optimizations**