FineIBT

Fine-grain Control-flow Enforcement with Indirect Branch Tracking

Alexander J. Gaidis[1] Joao Moreira[2] Ke Sun[2] Alyssa Milburn[2]
Vaggelis Atlidakis[1] Vasileios P. Kemerlis[1]

[1]Secure Systems Laboratory (SSL)
Department of Computer Science
Brown University

[2]IPAS STORM
Intel

Control-flow Integrity

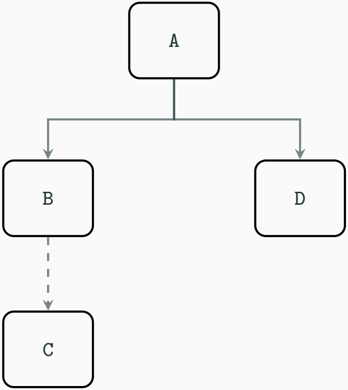
CFI Fundamentals

- CFI defends against control-flow hijacking by:

Control-flow Integrity

CFI Fundamentals

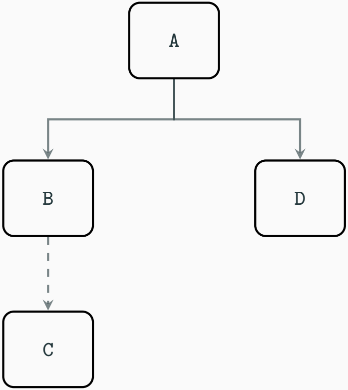
- CFI defends against control-flow hijacking by:
    1. Mapping legal control flows ⇝ CFI policy



Control-flow Integrity

CFI Fundamentals

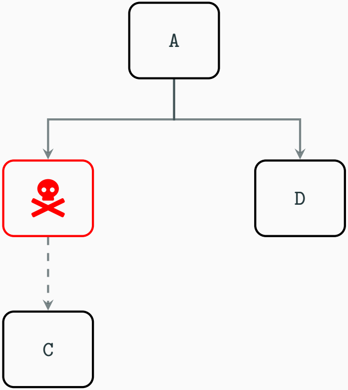
- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows ⇝ CFI policy
  2. Enforcing the policy at run time



Control-flow Integrity

CFI Fundamentals

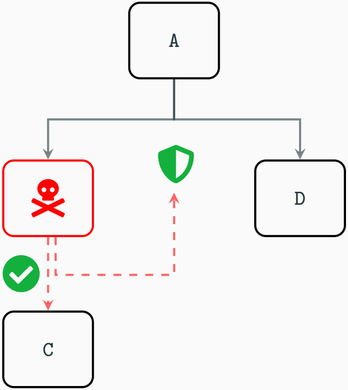
- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows ⇝ CFI policy
  2. Enforcing the policy at run time
- Attackers can tamper with code pointers



Control-flow Integrity

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows ⇝ CFI policy
  2. Enforcing the policy at run time
- Attackers can tamper with code pointers
- Computed control-flow transfers are confined
  e.g., `jmp *%reg`, `call *%reg`, `ret`



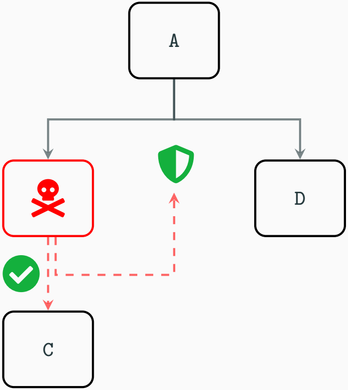
Control-flow Integrity

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows ⇝ CFI policy
  2. Enforcing the policy at run time
- Attackers can tamper with code pointers
- Computed control-flow transfers are confined
  e.g., `jmp *%reg`, `call *%reg`, `ret`

CFI Classification

- Target domain: where the scheme is applicable



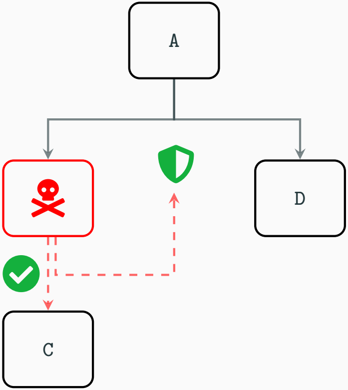
Control-flow Integrity

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows ⇝ CFI policy
  2. Enforcing the policy at run time
- Attackers can tamper with code pointers
- Computed control-flow transfers are confined
  e.g., `jmp *%reg`, `call *%reg`, `ret`

CFI Classification

- Target domain: where the scheme is applicable
- Compatibility: what the scheme requires

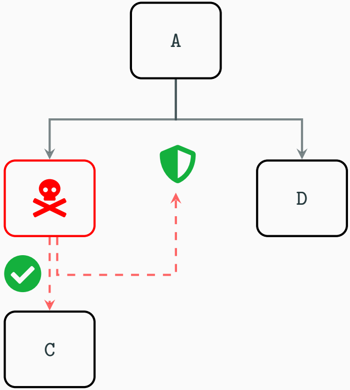


CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows ⇝ CFI policy
  2. Enforcing the policy at run time
- Attackers can tamper with code pointers
- Computed control-flow transfers are confined *e.g.*, `jmp *%reg`, `call *%reg`, `ret`

CFI Classification

- Target domain: where the scheme is applicable
- Compatibility: what the scheme requires
- Coverage: forward edges and/or backward edges



BROWN intel

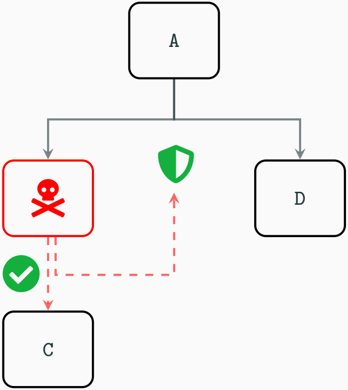
Control-flow Integrity

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows ⇝ CFI policy
  2. Enforcing the policy at run time
- Attackers can tamper with code pointers
- Computed control-flow transfers are confined
  e.g., `jmp *%reg`, `call *%reg`, `ret`

CFI Classification

- Target domain: where the scheme is applicable
- Compatibility: what the scheme requires
- Coverage: forward edges and/or backward edges
- Effectiveness: size of indirect branch target sets



Control-flow Integrity

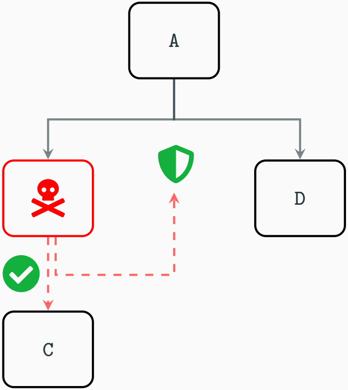
CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows ⇝ CFI policy
  2. Enforcing the policy at run time
- Attackers can tamper with code pointers
- Computed control-flow transfers are confined
  e.g., `jmp *%reg`, `call *%reg`, `ret`

CFI Classification

- Target domain: where the scheme is applicable
- Compatibility: what the scheme requires
- Coverage: forward edges and/or backward edges
- Effectiveness: size of indirect branch target sets

Fewer Targets ←→ More Targets
Fine-grain — Coarse-grain



agaidis@cs.brown.edu (Brown University)

Control-flow Enforcement Technology

CET

- CET is a CFI hardware feature on modern Intel CPUs

Control-flow Enforcement Technology

CET

- CET is a CFI hardware feature on modern Intel CPUs
- Two components: shadow stack, indirect branch tracking

Control-flow Enforcement Technology

CET

- CET is a CFI hardware feature on modern Intel CPUs
- Two components: shadow stack, indirect branch tracking

Indirect Branch Tracking

- New 4-byte instruction: `endbr`

Control-flow Enforcement Technology

CET

- CET is a CFI hardware feature on modern Intel CPUs
- Two components: shadow stack, indirect branch tracking

Indirect Branch Tracking

- New 4-byte instruction: `endbr`
- Indirect branches must target an `endbr`

Control-flow Enforcement Technology

CET

- CET is a CFI hardware feature on modern Intel CPUs
- Two components: shadow stack, indirect branch tracking

Indirect Branch Tracking

- New 4-byte instruction: `endbr`
- Indirect branches must target an `endbr`

Target Domain: Anywhere HW feature is available

Control-flow Enforcement Technology

CET

- CET is a CFI hardware feature on modern Intel CPUs
- Two components: shadow stack, indirect branch tracking

Indirect Branch Tracking

- New 4-byte instruction: `endbr`
- Indirect branches must target an `endbr`

Target Domain: Anywhere HW feature is available

Compatibility: No major requirements (just enable feature and add `endbr`s)

Control-flow Enforcement Technology

CET

- CET is a CFI hardware feature on modern Intel CPUs
- Two components: shadow stack, indirect branch tracking

Indirect Branch Tracking

- New 4-byte instruction: `endbr`
- Indirect branches must target an `endbr`

Target Domain: Anywhere HW feature is available

Compatibility: No major requirements (just enable feature and add `endbrs`)

Coverage: Forward-edges

Control-flow Enforcement Technology

CET

- CET is a CFI hardware feature on modern Intel CPUs
- Two components: shadow stack, indirect branch tracking

Indirect Branch Tracking

- New 4-byte instruction: `endbr`
- Indirect branches must target an `endbr`

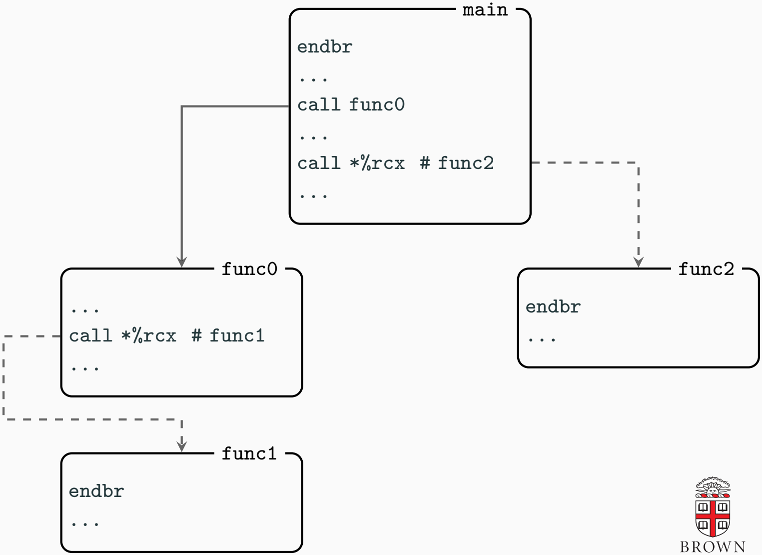
Target Domain: Anywhere HW feature is available

Compatibility: No major requirements (just enable feature and add `endbr`s)

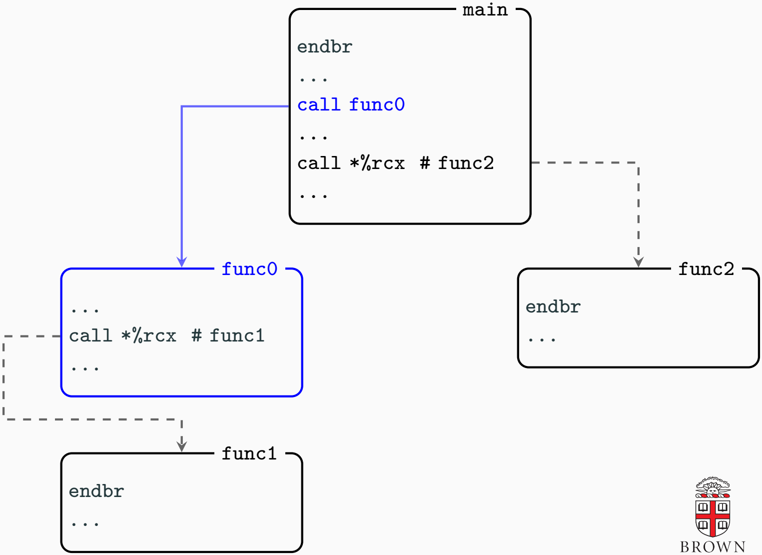
Coverage: Forward-edges

Effectiveness: Coarse-grain

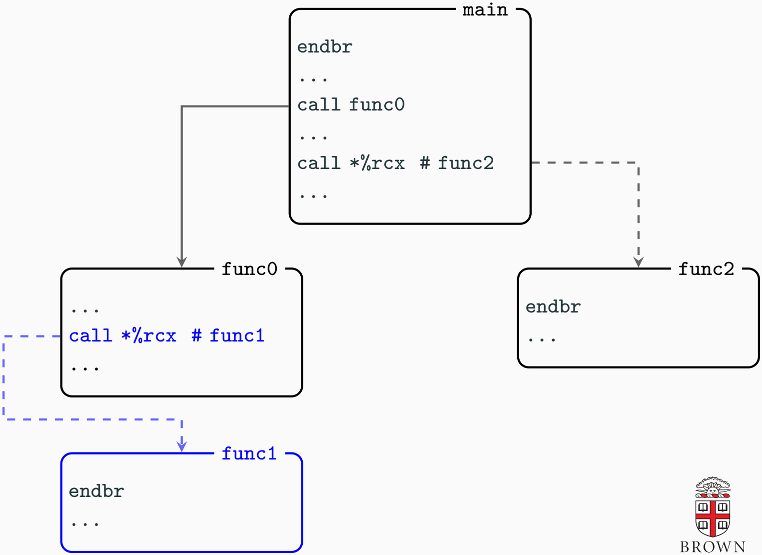
CET → IBT


main
endbr
...
call func0
...
call *%rcx # func2
...
func0
...
call *%rcx # func1
...
func2
endbr
...
func1
endbr
...

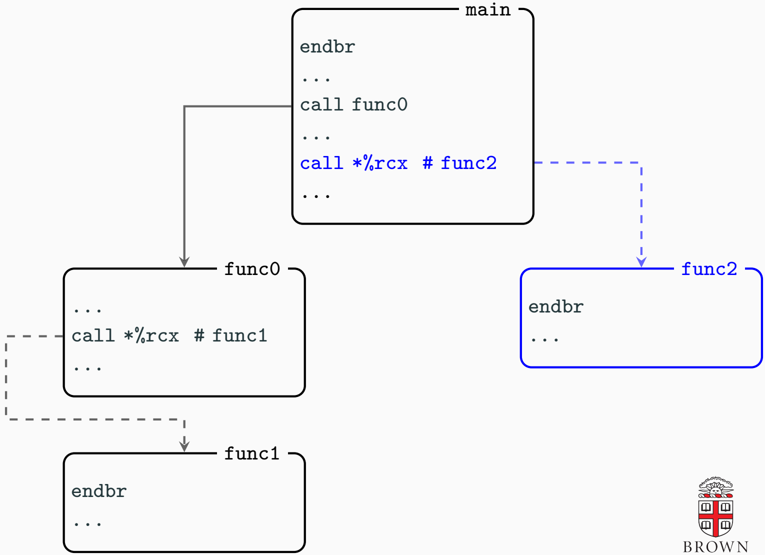
CET → IBT


main
endbr
...
call func0
...
call *%rcx # func2
...
func0
...
call *%rcx # func1
...
func2
endbr
...
func1
endbr
...

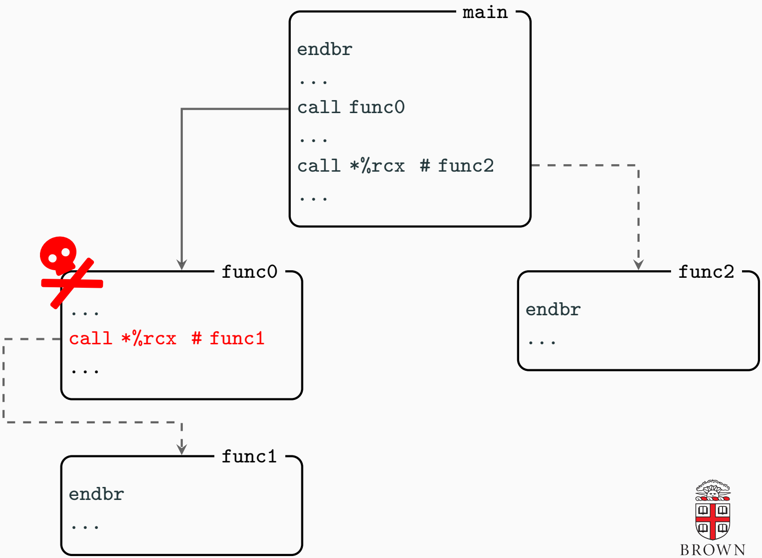
CET → IBT

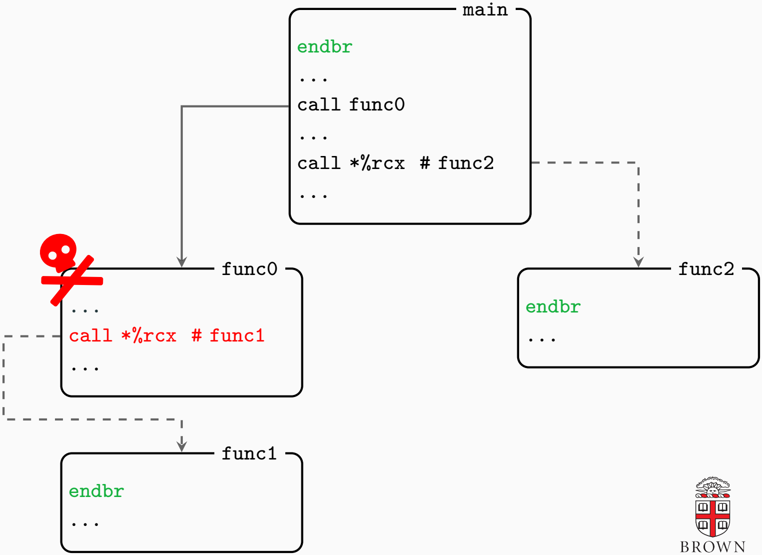

main
endbr
...
call func0
...
call *%rcx # func2
...
func0
...
call *%rcx # func1
...
func2
endbr
...
func1
endbr
...

CET → IBT

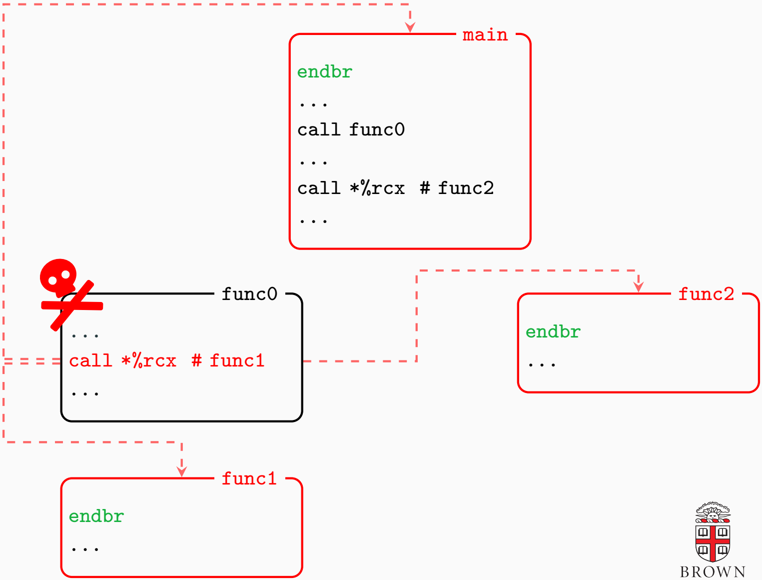

main
endbr
...
call func0
...
call *%rcx # func2
...
func0
...
call *%rcx # func1
...
func1
endbr
...
func2
endbr
...

CET → IBT


main
endbr
...
call func0
...
call *%rcx # func2
...
func0
...
call *%rcx # func1
...
func2
endbr
...
func1
endbr
...


main
endbr
...
call func0
...
call *%rcx # func2
...
func0
...
call *%rcx # func1
...
func2
endbr
...
func1
endbr
...

CET → IBT


main
endbr
...
call func0
...
call *%rcx # func2
...
func0
...
call *%rcx # func1
...
func2
endbr
...
func1
endbr
...

IBT → FineIBT

IBT Summary

- ✚ Performant

IBT Summary

- \+ Performant
- − Enforces a coarse-grain policy

IBT → FineIBT

IBT Summary

- + Performant
- − Enforces a coarse-grain policy
- − Libraries greatly increase the number of indirect branch targets

IBT → FineIBT

IBT Summary

- \+ Performant
- − Enforces a coarse-grain policy
- − Libraries greatly increase the number of indirect branch targets

  e.g., GLIBC exports > 2000 functions

IBT → FineIBT

IBT Summary

- + Performant
- − Enforces a coarse-grain policy
- − Libraries greatly increase the number of indirect branch targets

  e.g., GLIBC exports > 2000 functions

FineIBT

BROWN intel

IBT → FineIBT

IBT Summary

+ Performant
− Enforces a coarse-grain policy
− Libraries greatly increase the number of indirect branch targets

  e.g., GLIBC exports $>$ 2000 functions

FineIBT

- Retrofits IBT with light-weight instrumentation

IBT → FineIBT

IBT Summary

+ Performant

– Enforces a coarse-grain policy
– Libraries greatly increase the number of indirect branch targets

  e.g., GLIBC exports > 2000 functions

FineIBT

- Retrofits IBT with light-weight instrumentation
- Provides a mechanism to refine IBT's default policy

IBT Summary

+ Performant
− Enforces a coarse-grain policy
− Libraries greatly increase the number of indirect branch targets

  e.g., GLIBC exports > 2000 functions

FineIBT

- Retrofits IBT with light-weight instrumentation
- Provides a mechanism to refine IBT's default policy
- Supports coarse- and fine-grain CFI policies

IBT → FineIBT

IBT Summary

+ Performant
− Enforces a coarse-grain policy
− Libraries greatly increase the number of indirect branch targets
  e.g., GLIBC exports $>$ 2000 functions

FineIBT

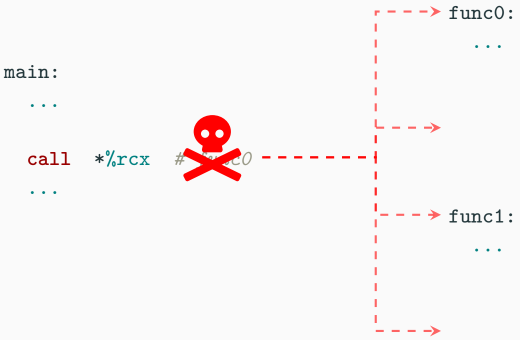
- Retrofits IBT with light-weight instrumentation
- Provides a mechanism to refine IBT's default policy
- Supports coarse- and fine-grain CFI policies
    - Policy agnostic

FineIBT → Foundational Instrumentation

```
main:
  ...
  call  *%rcx  # func0
  ...

func0:
  ...

func1:
  ...
```


main:
...
call *%rcx #
...
func0:
...
func1:
...

FineIBT → Foundational Instrumentation

```
main:
  ...
  call  *%rcx  # func0
  ...
```

```
func0:
  endbr
  ...

func1:
  endbr
  ...
```

```
main:
  ...
  call   *%rcx   # func0
  ...

func0:
  endbr
  ...

func1:
  endbr
  ...
```

FineIBT → Foundational Instrumentation

```
func0:   # SID: 0xc00010ff
  endbr
  ...
```

```
main:
  ...
  # SID: 0xc00010ff
  call  *%rcx  # func0
  ...
```

```
func1:   # SID: 0xbaddcafe
  endbr
  ...
```

FineIBT → Foundational Instrumentation

```
main:
  ...
  mov   $0xc00010ff , %eax
  call  *%rcx   # func0
  ...
```

```
func0:   # SID: 0xc00010ff
  endbr
  ...

func1:   # SID: 0xbaddcafe
  endbr
  ...
```

FineIBT → Foundational Instrumentation

```
main:
  ...
  mov   $0xc00010ff , %eax
  call  *%rcx   # func0
  ...
```

```
func0:
  endbr
  sub   $0xc00010ff , %eax
  je    func0_entry
  hlt
func0_entry:
  ...

func1:
  endbr
  sub   $0xbaddcafe , %eax
  je    func1_entry
  hlt
func1_entry:
  ...
```

FineIBT → Foundational Instrumentation

```
main:
  ...
  mov   $0xc0001   , %eax
  call  *%rcx   # func0
  ...
```

```
func0:
  endbr
  sub   $0xc00010ff , %eax
  je    func0_entry
  hlt
func0_entry:
  ...

func1:
  endbr
  sub   $0xbaddcafe , %eax
  je    func1_entry
  hlt
func1_entry:
  ...
```

FineIBT → Foundational Instrumentation

```
main:
    ...
    mov   $0xc00010ff , %eax
    call  *%rcx    # func0
    ...
```

```
func0:
    endbr
    sub   $0xc00010ff , %eax
    je    func0_entry
    hlt
func0_entry:
    ...

func1:
    endbr
    sub   $0xbaddcafe , %eax
    je    func1_entry
    hlt
func1_entry:
    ...
```

BROWN intel

FineIBT → Foundational Instrumentation

```
main:
  ...
  mov   $0xc00010ff , %eax
  call  *%rcx   # func0
  ...
  call func1_entry
  ...
```

```
func0:
  endbr
  sub   $0xc00010ff , %eax
  je    func0_entry
  hlt
func0_entry:
  ...

func1:
  endbr
  sub   $0xbaddcafe , %eax
  je    func1_entry
  hlt
func1_entry:
  ...
```

FineIBT → Performance Considerations

```
func0:
  endbr
  sub    $0xc00010ff, %eax
  je     func0_entry
  hlt
func0_entry:
  ...
```

Custom Error-handling Code

- hlt is in the IRM's hot-path

FineIBT → Performance Considerations

```
func0:
  endbr
  sub    $0xc00010ff, %eax
  je     func0_entry
  hlt
func0_entry:
  ...
```

Custom Error-handling Code

- hlt is in the IRM's hot-path
- Only executed on SID-check failure

FineIBT → Performance Considerations

```
func0:
  endbr
  sub    $0xc00010ff, %eax
  je     func0_entry
  hlt
func0_entry:
  ...
```

Custom Error-handling Code

- `hlt` is in the IRM's hot-path
- Only executed on SID-check failure
- We can move it outside the hot-path:

FineIBT → Performance Considerations

```
.func0_coldpath:
  hlt
func0:
  endbr
  sub    $0xc00010ff, %eax
  jne    .func0_coldpath
func0_entry:
  ...
```

Custom Error-handling Code

- `hlt` is in the IRM's hot-path
- Only executed on SID-check failure
- We can move it outside the hot-path:
  - ➜ `s/je/jne/`

FineIBT → Performance Considerations

```
.func0_coldpath:
  hlt
func0:
  endbr
  sub    $0xc00010ff, %eax
  jne    .func0_coldpath
func0_entry:
  ...
```

Custom Error-handling Code

- `hlt` is in the IRM's hot-path
- Only executed on SID-check failure
- We can move it outside the hot-path:
  - → `s/je/jne/`
- The `hlt` can also be swapped for custom error-handlers without affecting performance

FineIBT → Performance Considerations

```
.func0_coldpath:
  ...  /* set arg0, ..., argn */
  call  __fineibt_chk_fail@PLT
func0:
  endbr
  sub   $0xc00010ff, %eax
  jne   .func0_coldpath
func0_entry:
  ...
```

Custom Error-handling Code

- `hlt` is in the IRM's hot-path
- Only executed on SID-check failure
- We can move it outside the hot-path:
  - ➜ `s/je/jne/`
- The `hlt` can also be swapped for custom error-handlers without affecting performance

FineIBT → Performance Considerations

```
main:
  ...
  mov   $0xc00010ff, %eax
  call  *%rcx   # func0
  ...
  call func0_entry
  ...
.func0_coldpath:
  hlt
func0:
  endbr
  sub   $0xc00010ff, %eax
  jne   .func0_coldpath
func0_entry:
  ...
```

CPU Front-end and I-Cache

- Direct calls bypass FineIBT's instrumentation

FineIBT → Performance Considerations

```
main:
  ...
  mov   $0xc00010ff, %eax
  call  *%rcx   # func0
  ...
  call func0_entry
  ...
.func0_coldpath:
  hlt
func0:
  endbr
  sub   $0xc00010ff, %eax
  jne   .func0_coldpath
func0_entry:
  ...
```

CPU Front-end and I-Cache

- Direct calls bypass FineIBT's instrumentation
- 4 instructions for fine-grain CFI
  - Different policies, same overhead

BROWN intel

FineIBT → Performance Considerations

```
main:
  ...
  mov   $0xc00010ff, %eax
  call  *%rcx   # func0
  ...
  call func0_entry
  ...
.func0_coldpath:
  hlt
func0:
  endbr
  sub   $0xc00010ff, %eax
  jne   .func0_coldpath
func0_entry:
  ...
```

CPU Front-end and I-Cache

- Direct calls bypass FineIBT's instrumentation
- 4 instructions for fine-grain CFI
  - Different policies, same overhead
- FineIBT's hot-path uses 16 bytes in total:

	5 bytes	per indirect branch site
+	11 bytes	per indirect branch target
	16 bytes	

BROWN intel

FineIBT → Performance Considerations

```
main:
  ...
  mov   $0xc00010ff, %eax
  call  *%rcx   # func0
  ...
  call func0_entry
  ...
.func0_coldpath:
  hlt
func0:
  endbr
  sub   $0xc00010ff, %eax
  jne   .func0_coldpath
func0_entry:
  ...
```

CPU Front-end and I-Cache

- Direct calls bypass FineIBT's instrumentation
- 4 instructions for fine-grain CFI
  - Different policies, same overhead
- FineIBT's hot-path uses 16 bytes in total:

	5 bytes	per indirect branch site
+	11 bytes	per indirect branch target
	16 bytes	

- Clang-CFI's hot-path uses 25 bytes in total:

	20 bytes	per indirect branch site
+	5 bytes	per trampoline entry
	25 bytes	

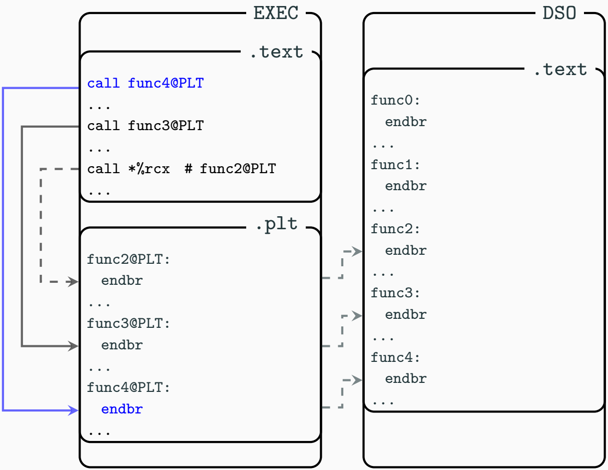
```
.func0_coldpath:
  hlt
func0:
  endbr
  sub    $0xc00010ff, %eax
  jne    .func0_coldpath
func0_entry:
  ...
```

Instruction Selection

- SID-checking instructions:

Insn	Clears SID	Macro-fused (w/ jne/je)
cmp	✗	✓
sub	✓	✓
xor	✓	✗

FineIBT → Security Considerations


EXEC
.text
call func4@PLT
...
call func3@PLT
...
call *%rcx # func2@PLT
...
.plt
func2@PLT:
endbr
...
func3@PLT:
endbr
...
func4@PLT:
endbr
...
DSO
.text
func0:
endbr
...
func1:
endbr
...
func2:
endbr
...
func3:
endbr
...
func4:
endbr
...

FineIBT → Security Considerations

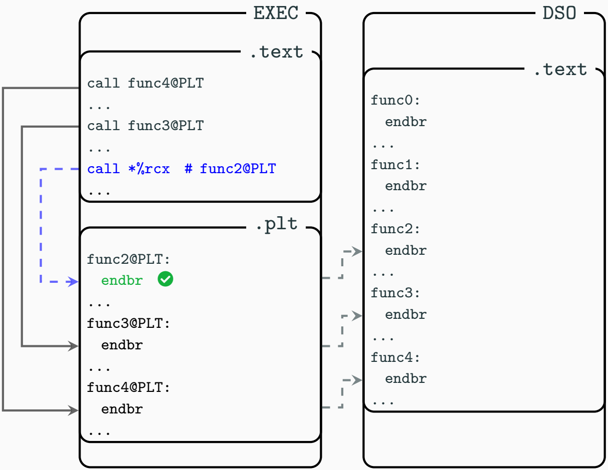
FineIBT → Security Considerations

FineIBT → Security Considerations

IBT PLT

- Every entry prefixed with endbr in case they are AT

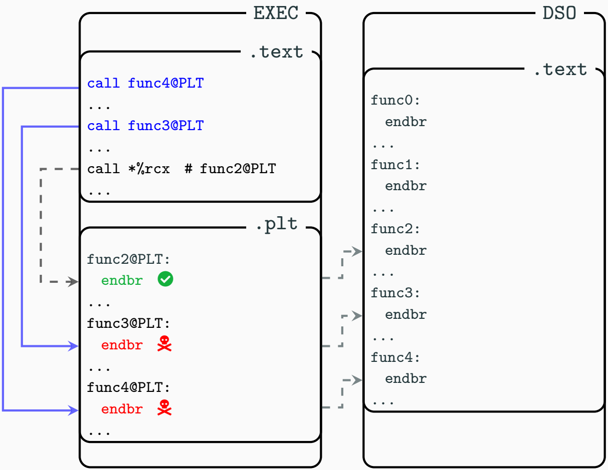
FineIBT → Security Considerations



IBT PLT

- Every entry prefixed with endbr in case they are AT

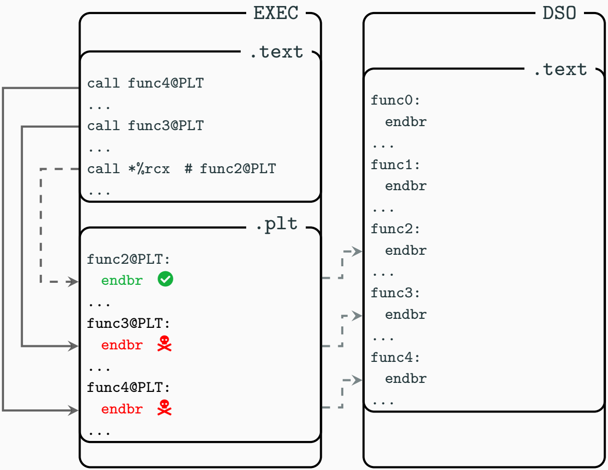
FineIBT → Security Considerations



IBT PLT

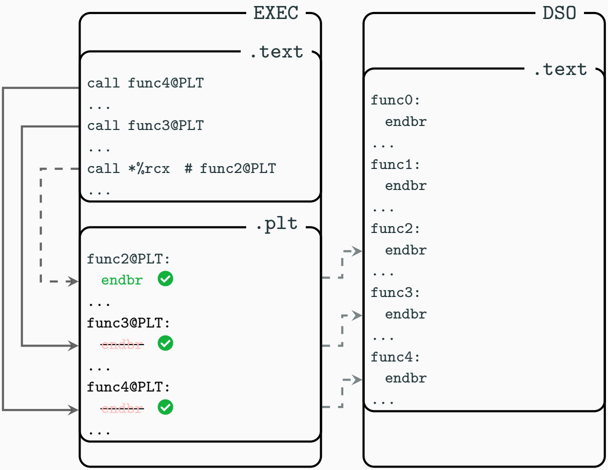
- Every entry prefixed with endbr in case they are AT

FineIBT → Security Considerations



IBT PLT

- Every entry prefixed with endbr in case they are AT



IBT PLT

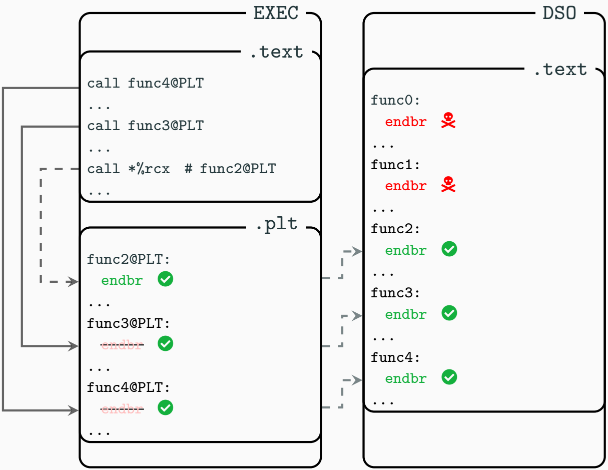
- Every entry prefixed with endbr in case they are AT

FineIBT PLT

- Only AT entries prefixed with endbr
- Entries hardened with FineIBT IRM

BROWN intel

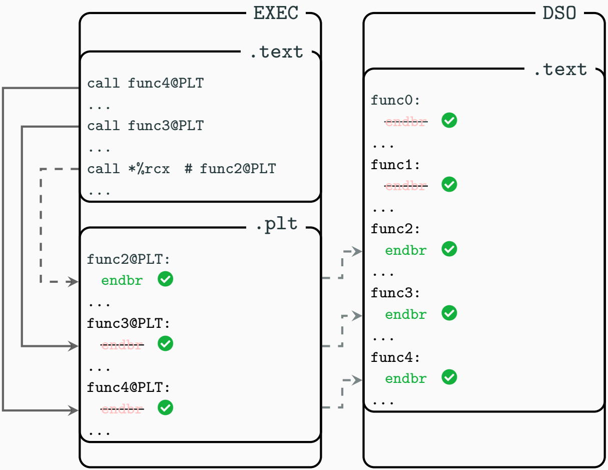
FineIBT → Security Considerations



Exported Functions

- Unused (exported) DSO functions still contain endbrs

FineIBT → Security Considerations



Exported Functions

- Unused (exported) DSO functions still contain endbrs

NOPout

- Runtime library that refines indirect branch targets by eliding endbrs in unused functions

FineIBT → Summary

Features

Features

- Compatible with existing defenses (*e.g.*, XOM)

FineIBT → Summary

Features

- Compatible with existing defenses (*e.g.*, XOM)
- Supports custom error-handling

Features

- Compatible with existing defenses (*e.g.*, XOM)
- Supports custom error-handling
- Supports secure cross-DSO function calls

FineIBT → Summary

Features

- Compatible with existing defenses (*e.g.*, XOM)
- Supports custom error-handling
- Supports secure cross-DSO function calls
- Provides a tool (`NOPout`) to dynamically refine indirect branch target sets

Features

- Compatible with existing defenses (*e.g.*, XOM)
- Supports custom error-handling
- Supports secure cross-DSO function calls
- Provides a tool (`NOPout`) to dynamically refine indirect branch target sets

Classification

FineIBT → Summary

Features

- Compatible with existing defenses (*e.g.*, XOM)
- Supports custom error-handling
- Supports secure cross-DSO function calls
- Provides a tool (`NOPout`) to dynamically refine indirect branch target sets

Classification

- **Target Domain**: works in multiple domains (*e.g.*, user/kernel space)

BROWN intel

FineIBT → Summary

Features

- Compatible with existing defenses (*e.g.*, XOM)
- Supports custom error-handling
- Supports secure cross-DSO function calls
- Provides a tool (NOPout) to dynamically refine indirect branch target sets

Classification

- **Target Domain**: works in multiple domains (*e.g.*, user/kernel space)
- **Compatibility**: uses source code to increase precision, but can be applied to binaries

BROWN intel

FineIBT → Summary

Features

- Compatible with existing defenses (*e.g.*, XOM)
- Supports custom error-handling
- Supports secure cross-DSO function calls
- Provides a tool (`NOPout`) to dynamically refine indirect branch target sets

Classification

- **Target Domain**: works in multiple domains (*e.g.*, user/kernel space)
- **Compatibility**: uses source code to increase precision, but can be applied to binaries
- **Coverage**: forward-edge protection (we assume a shadow stack for backward edges)

BROWN intel

FineIBT → Summary

Features

- Compatible with existing defenses (*e.g.*, XOM)
- Supports custom error-handling
- Supports secure cross-DSO function calls
- Provides a tool (`NOPout`) to dynamically refine indirect branch target sets

Classification

- **Target Domain**: works in multiple domains (*e.g.*, user/kernel space)
- **Compatibility**: uses source code to increase precision, but can be applied to binaries
- **Coverage**: forward-edge protection (we assume a shadow stack for backward edges)
- **Effectiveness**: policy agnostic, supports coarse- and fine-grain CFI policies

FineIBT → Implementation

FineIBT

- FineIBT is a set of modifications to the LLVM v12 toolchain
  - ≈700 C++ LOC added to the LLVM compiler (llvm-{gcc, g++})
  - ≈200 C++ LOC added to the LLVM linker (lld)

FineIBT → Implementation

FineIBT

- FineIBT is a set of modifications to the LLVM v12 toolchain
  - ≈700 C++ LOC added to the LLVM compiler (`llvm-{gcc, g++}`)
  - ≈200 C++ LOC added to the LLVM linker (`lld`)

NOPout

- NOPout is a DSO (`libnopout.so`) implemented in ≈800 C LOC

FineIBT → Implementation

FineIBT

- FineIBT is a set of modifications to the LLVM v12 toolchain
  - ≈700 C++ LOC added to the LLVM compiler (llvm-{gcc, g++})
  - ≈200 C++ LOC added to the LLVM linker (lld)

NOPout

- NOPout is a DSO (libnopout.so) implemented in ≈800 C LOC

CFI Policy

- FineIBT is policy agnostic
- Demonstrated FineIBT's policy flexibility by testing different policies:
  - Arity-based CFI
  - Type-based CFI
  - MLTA-based CFI

SPEC CPU 2017

Benchmark	IBT	FineIBT !EH	FineIBT EH	Clang-CFI
600.perlbench	1.46%	1.51%	≈0%	1.17%
602.gcc	0.03%	0.12%	0.05%	0.6%
605.mcf	0.51%	3.67%	1.94%	2.04%
620.omnetpp	≈0%	0.91%	≈0%	6.57%
623.xalancbmk	≈0%	0.99%	≈0%	7.89%
625.x264	0.05%	0.29%	≈0%	0.93%
631.deepsjeng	0.03%	0.17%	≈0%	≈0%
641.leela	≈0%	≈0%	≈0%	≈0%
657.xz	≈0%	≈0%	≈0%	0.06%

- FineIBT exhibited ≈0%–1.94% slowdown
- Results are on-par with IBT (≈0%–1.46%)
- FineIBT outperforms Clang-CFI

SPEC CPU 2017

Benchmark	IBT	FineIBT !EH	FineIBT EH	Clang-CFI
600.perlbench	1.46%	1.51%	≈0%	1.17%
602.gcc	0.03%	0.12%	0.05%	0.6%
605.mcf	0.51%	3.67%	1.94%	2.04%
620.omnetpp	≈0%	0.91%	≈0%	6.57%
623.xalancbmk	≈0%	0.99%	≈0%	7.89%
625.x264	0.05%	0.29%	≈0%	0.93%
631.deepsjeng	0.03%	0.17%	≈0%	≈0%
641.leela	≈0%	≈0%	≈0%	≈0%
657.xz	≈0%	≈0%	≈0%	0.06%

- + FineIBT exhibited ≈0%–1.94% slowdown
- + Results are on-par with IBT (≈0%–1.46%)
- + FineIBT outperforms Clang-CFI

BROWN intel

SPEC CPU 2017

Benchmark	IBT	FineIBT !EH	FineIBT EH	Clang-CFI
600.perlbench	1.46%	1.51%	≈0%	1.17%
602.gcc	0.03%	0.12%	0.05%	0.6%
605.mcf	0.51%	3.67%	1.94%	2.04%
620.omnetpp	≈0%	0.91%	≈0%	6.57%
623.xalancbmk	≈0%	0.99%	≈0%	7.89%
625.x264	0.05%	0.29%	≈0%	0.93%
631.deepsjeng	0.03%	0.17%	≈0%	≈0%
641.leela	≈0%	≈0%	≈0%	≈0%
657.xz	≈0%	≈0%	≈0%	0.06%

- FineIBT exhibited ≈0%–1.94% slowdown
- Results are on-par with IBT (≈0%–1.46%)
- FineIBT outperforms Clang-CFI

Real-world Applications

Application	IBT	FineIBT (EH)
Nginx (1KB)	≈0%	≈0%
Nginx (100KB)	0.77%	1.92%
Nginx (1MB)	≈0%	0.11%
Redis (GET)	0.39%	1.17%
Redis (SET)	0.39%	1.17%
MariaDB	0.55%	0.60%
SQLite	≈0%	0.36%

- FineIBT exhibited ≈0%–1.92% tput/runtime slowdown
- Real-world results confirmed those of SPEC CPU 2017

NOPout

- + Reduced valid targets by up to 17291
- – Additional memory required ranges between 184KB–7.84MB

NOPout

- + Reduced valid targets by up to 17291
- − Additional memory required ranges between 184KB–7.84MB

ConFIRM

✓ Passed 17/18 compatibility/security tests in ConFIRM

▶ Only failed jit because JIT engine was not FineIBT-aware

Conclusion

Takeaways

- CFI enforcement mechanism that is performant, effective, and compatible
- Policy agnostic: supports coarse- and fine-grain policies
- Run-time target refinement via `NOPout` library
- A version of `FineIBT` was recently added to Linux kernel v6.2

Availability: https://gitlab.com/brown-ssl/fineibt

BROWN intel

Backup Slides

Memory Safety

- Programs written in memory unsafe languages (*e.g.*, C/C++, ASM) are prone to memory errors

Example

```c
int
main(int argc, char *argv[])
{
  int (*fptr)(void) = &foo;
  char arr[10];
  ...
  strcpy(arr, argv[1]);
  ...
  fptr();
  ...
}
```

Repercussions

- Attackers can corrupt memory to perform control-flow hijacking
- Control-flow hijacking can be escalated to achieve arbitrary code execution

Control-flow Integrity Scheme Classifications

- CFI schemes can be classified according to 4 properties:

Effectiveness

Coverage

Compatibility

Target Domain

BROWN intel

Control-flow Integrity Scheme Classifications

- CFI schemes can be classified according to 4 properties:

Effectiveness

Coverage

Compatibility

Target Domain

Effectiveness

- How many (code) locations an indirect branch can target

Fewer Locations ←	→ More Locations
Fine-grain	Coarse-grain

☠ Coarse-grain schemes are overly permissive

✔ Fine-grain schemes refine the set of allowed targets

Control-flow Integrity Scheme Classifications

- CFI schemes can be classified according to 4 properties:

Effectiveness

Coverage

Compatibility

Target Domain

Coverage

- The type of control-flow transfers covered
  - Forward edges
    e.g., indirect calls/jumps
  - Backward edges
    e.g., returns

Control-flow Integrity Scheme Classifications

- CFI schemes can be classified according to 4 properties:

`Effectiveness`

`Coverage`

`Compatibility`

`Target Domain`

Compatibility

- What the scheme requires
  - Source code provides useful information
  - Binary data avoids recompilation
  - Different languages require different considerations

BROWN intel

Control-flow Integrity Scheme Classifications

- CFI schemes can be classified according to 4 properties:

Effectiveness

Coverage

Compatibility

Target Domain

Target Domain

- Where the scheme is applicable
  - User space
  - Kernel space
  - Embedded systems

BROWN intel

ConFIRM Results (Full)

Test	Result	Description
callback	✓	Callbacks support
code_coop	✓	COOP attack resilience
convention	✓	Different x86 calling conv. support
cppeh	✓	C++ exception handling support
data_symbl	✓	Import/export data sym. handling
fptr	✓	Indirect function call support
jit	✗	Runtime-generated code support
load_time_dynlnk	✓	Load-time function resolution
mem	✓	Memory mgmt. API support
multithreading	✓	Concurrent thread exec. support
pic	✓	PIC/PIE support
ret	✓	Return-address validation
run_time_dynlnk	✓	Run-time function resolution
signal	✓	Signal handling support
switch	✓	switch-based CF support
tail_call	✓	Tail-call optimizations
unmatched_pair	✓	Unmatched call/ret pairs
vtbl_call	✓	Virtual function support

BROWN intel

NOPout Results (Full)

NOPout

Application	AT-elided	Pages	KB
redis-server	858 (19.05%)	206	844
sqlite	896 (34.07%)	165	676
nginx	2873 (19.33%)	606	2482
mariadbd	17291 (44.98%)	1915	7844
600.perlbench	942 (35.19%)	246	1008
602.gcc	3646 (67.71%)	665	2724
605.mcf	83 (4.57%)	45	184
620.omnetpp	5623 (71.03%)	405	1659
623.xalancbmk	8023 (77.78%)	644	2638
625.x264	302 (14.82%)	76	311
631.deepsjeng	1690 (42.45%)	189	774
641.leela	1722 (42.91%)	194	795
657.xz	176 (9.17%)	59	242

- \+ Reduces valid targets by up to 17291
- − Additional memory required ranges between 184KB–7.84MB

Clang-CFI Instrumentation

```
0x401140 <func>:
  ...
  40116a:  mov    $0x401250,%ecx
  40116f:  mov    %rax,%rdx
  401172:  sub    %rcx,%rdx
  401175:  rol    $0x3d,%rdx
  401179:  cmp    $0x2,%rdx
  40117d:  jae    4011c5
  40117f:  callq  *%rax
  ...
  4011a3:  callq  4011d0 <f0.cfi>
  ...
  4011c5:  ud2
  ...
0x4011d0 <f0.cfi>:
  ...
0x4011f0 <f1.cfi>:
  ...
0x401250 <f0>:
  401250:  jmpq   4011d0 <f0.cfi>
  401255:  int3
  401256:  int3
  401257:  int3
0x401258 <f1>:
  401258:  jmpq   4011f0 <f1.cfi>
  40125d:  int3
  40125e:  int3
  40125f:  int3
  ...
```

Non-IBT PLT

```
PLT0: pushq    GOT+8(%rip)         /*  GOT[1]   */
      jmp      *GOT+16(%rip)       /*  GOT[2]   */
      nopl     0x0(%rax)           /*   PAD     */
...
PLT3: jmp      *fsym3@GOT(%rip)    /*  GOT[5]   */
      pushq    $0x2
      jmp      PLT0
PLT4: jmp      *fsym4@GOT(%rip)    /*  GOT[6]   */
      pushq    $0x3
      jmp      PLT0
...
PLTn: jmp      *fsymn@GOT(%rip)    /* GOT[n+2]  */
      pushq    $0xn-1
      jmp      PLT0
```

IBT PLT

```
PLT0: pushq    GOT+8(%rip)          /*  GOT[1]  */
      jmp      *GOT+16(%rip)        /*  GOT[2]  */
      nopl     0x0(%rax)            /*   PAD    */
...
PLT3: endbr64
      pushq    $0x2
      jmp      PLT0
      xchg     %ax,%ax              /*   PAD    */
PLT4: endbr64
      pushq    $0x3
      jmp      PLT0
      xchg     %ax,%ax              /*   PAD    */
...
PLTn: endbr64
      pushq    $0xn-1
      jmp      PLT0
      xchg     %ax,%ax              /*   PAD    */

...
SPLT3: endbr64
       jmp      *fsym3@GOT(%rip)    /*  GOT[5]  */
       nopw     0x0(%rax,%rax,1)    /*   PAD    */
SPLT4: endbr64
       jmp      *fsym4@GOT(%rip)    /*  GOT[6]  */
       nopw     0x0(%rax,%rax,1)    /*   PAD    */
...
SPLTn: endbr64
       jmp      *fsymn@GOT(%rip)    /* GOT[n+2] */
       nopw     0x0(%rax,%rax,1)    /*   PAD    */
```

BROWN intel

FineIBT PLT

```
PLT0:    shl    $0x20, %rax
         or     $SID, %rax
         pushq  GOT+8(%rip)          /* GOT[1] */
         jmp    *GOT+16(%rip)        /* GOT[2] */
         nopw   %cs:0x0(%rax,%rax,1) /*  PAD   */
...
PLT4:    endbr64
         cmp    $SID, %eax
         pushq  $0x3
         xchg   %ax, %ax             /*  PAD   */
         je     PLT0
         hlt
         nopw   0x0(%rax,%rax,1)     /*  PAD   */
...
FPLT4:   mov   $SID, %eax
         jmp   *fsym4@GOT(%rip)      /* GOT[6] */
         nopl  0x0(%rax,%rax,1)      /*  PAD   */
...
ATFPLT4: endbr64
         sub   $SID, %eax
         je    FPLT4
         hlt
         data16 nopw %cs:0x0(%rax,%rax,1) /*PAD*/
         nopl  (%rax)                /*  PAD   */
```

ARM Instrumentation (FineBTI)

```
main:                          /* caller */
  ...
  movz w9, #0x3a, lsl #16     /* SID = 0x3a0000 */
  blr  x0                     /* x0 = &func */
  ...
  bl   func_entry
  ...
.func_finebti_coldpath:
  ...                         /* arg0, ..., argn */
  bl   __finebti_chk_fail@PLT
func:                         /* callee */
  bti  c
  subs w9, w9, #0x3a0, lsl #12 /* SID=0x3a0000 */
  bne  .func_finebti_coldpath
func_entry:
  ...
```