NOPoutNG

Improving the Effectiveness of Hardware-assisted Control-flow Integrity via Dynamic Landing Pad Elision

Alexander J. Gaidis
Brown University
agaidis@cs.brown.edu

Jamie Gabbay
Brown University
jamie_gabbay@brown.edu

Joao Moreira
Microsoft
joaomoreira@microsoft.com

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

BROWN

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report $\approx$ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
[†]Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
[‡]The Chromium Projects, "Memory safety," 2023.
[§]Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

BROWN

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report ≈ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

Memory Error Exploitation

- Apps written in memory-/type-unsafe languages (e.g., C) can be exploited via memory-safety violations

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
[†]Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
[‡]The Chromium Projects, "Memory safety," 2023.
[§]Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report ≈ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

Memory Error Exploitation

- Apps written in memory-/type-unsafe languages (e.g., C) can be exploited via memory-safety violations
- By targeting these vulnerabilities, attackers can manipulate control data to hijack execution

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
†Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
‡The Chromium Projects, "Memory safety," 2023.
§Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report ≈ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

Memory Error Exploitation

- Apps written in memory-/type-unsafe languages (e.g., C) can be exploited via memory-safety violations
- By targeting these vulnerabilities, attackers can manipulate control data to hijack execution
- Modern exploits use code-reuse (e.g., ROP, JOP)

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
†Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
‡The Chromium Projects, "Memory safety," 2023.
§Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

BROWN

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report ≈ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

Memory Error Exploitation

- Apps written in memory-/type-unsafe languages (e.g., C) can be exploited via memory-safety violations
- By targeting these vulnerabilities, attackers can manipulate control data to hijack execution
- Modern exploits use code-reuse (e.g., ROP, JOP)

Mitigations

- Modern defenses can be largely grouped into four categories:

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
†Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
‡The Chromium Projects, "Memory safety," 2023.
§Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report $\approx$ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

Memory Error Exploitation

- Apps written in memory-/type-unsafe languages (e.g., C) can be exploited via memory-safety violations
- By targeting these vulnerabilities, attackers can manipulate control data to hijack execution
- Modern exploits use code-reuse (e.g., ROP, JOP)

Mitigations

- Modern defenses can be largely grouped into four categories:
  1. Automated diversification

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
†Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
‡The Chromium Projects, "Memory safety," 2023.
§Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

BROWN

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report ≈ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

Memory Error Exploitation

- Apps written in memory-/type-unsafe languages (e.g., C) can be exploited via memory-safety violations
- By targeting these vulnerabilities, attackers can manipulate control data to hijack execution
- Modern exploits use code-reuse (e.g., ROP, JOP)

Mitigations

- Modern defenses can be largely grouped into four categories:
  1. Automated diversification
  2. Memory isolation

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
†Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
‡The Chromium Projects, "Memory safety," 2023.
§Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

BROWN

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report $\approx$ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

Memory Error Exploitation

- Apps written in memory-/type-unsafe languages (e.g., C) can be exploited via memory-safety violations
- By targeting these vulnerabilities, attackers can manipulate control data to hijack execution
- Modern exploits use code-reuse (e.g., ROP, JOP)

Mitigations

- Modern defenses can be largely grouped into four categories:
  1. Automated diversification
  2. Memory isolation
  3. Data-flow integrity

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
†Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
‡The Chromium Projects, "Memory safety," 2023.
§Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

Memory Safety

Memory Errors

- Modern software is large and complex, often resulting in vulnerabilities
- Memory errors are among the most malefic:
  - They dominate the SANS institute's list of the "*top 25 most dangerous software errors*"*
  - Microsoft and Google report ≈ 70% of all product vulnerabilities stem from memory-safety problems[†‡§]

Memory Error Exploitation

- Apps written in memory-/type-unsafe languages (e.g., C) can be exploited via memory-safety violations
- By targeting these vulnerabilities, attackers can manipulate control data to hijack execution
- Modern exploits use code-reuse (e.g., ROP, JOP)

Mitigations

- Modern defenses can be largely grouped into four categories:
  1. Automated diversification
  2. Memory isolation
  3. Data-flow integrity
  4. Control-flow integrity

*SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," 2023.
†Microsoft Security Response Center, "A proactive approach to more secure code," 2019.
‡The Chromium Projects, "Memory safety," 2023.
§Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," 2021.

BROWN

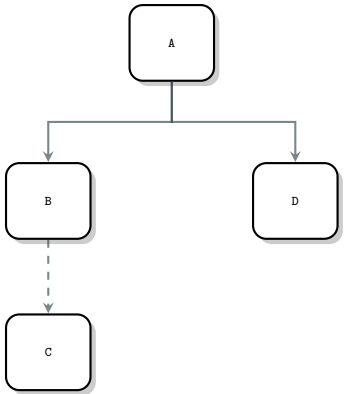
CFI Fundamentals

- CFI defends against control-flow hijacking by:

BROWN

CFI Fundamentals

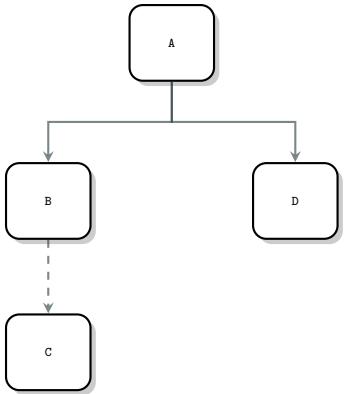
- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows $\rightsquigarrow$ CFI policy



BROWN

CFI Fundamentals

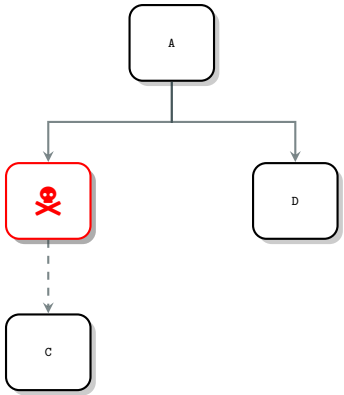
- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows $\rightsquigarrow$ CFI policy
  2. Enforcing the policy at run time



BROWN

CFI Fundamentals

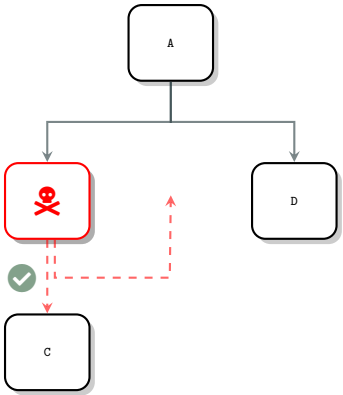
- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows $\rightsquigarrow$ CFI policy
  2. Enforcing the policy at run time
- Attackers can tamper with code pointers



BROWN

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows $\rightsquigarrow$ 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`



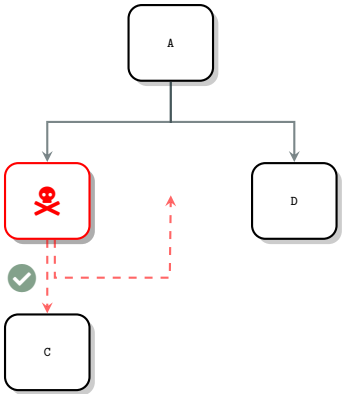
BROWN

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows $\rightsquigarrow$ 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



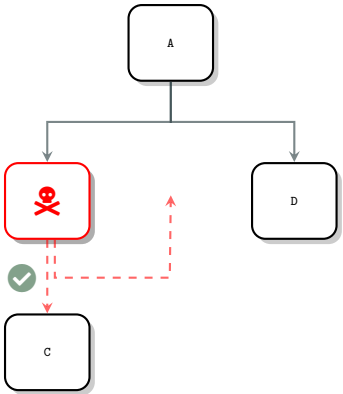
BROWN

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows $\rightsquigarrow$ 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



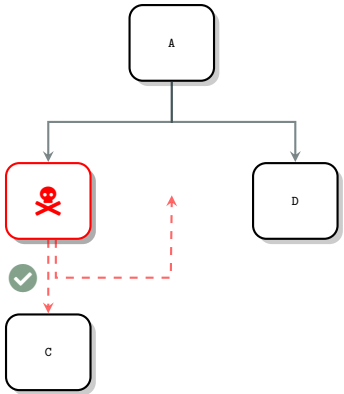
BROWN

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows $\rightsquigarrow$ 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



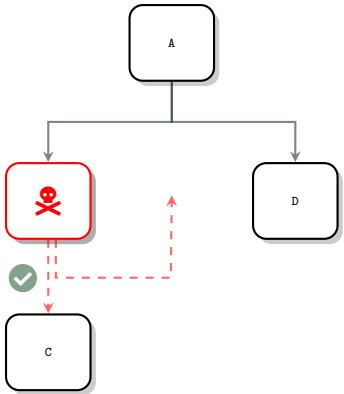
BROWN

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



BROWN

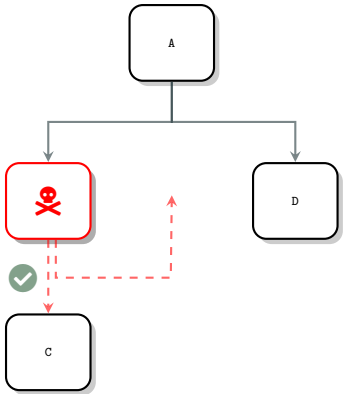
CFI Fundamentals

- 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



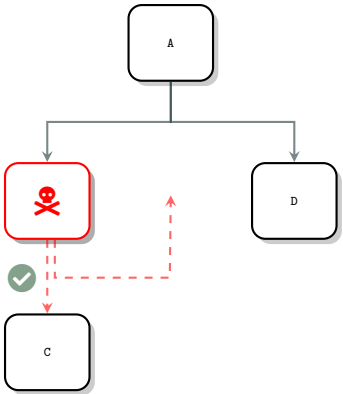
BROWN

CFI Fundamentals

- CFI defends against control-flow hijacking by:
  1. Mapping legal control flows $\rightsquigarrow$ 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 Support

- Hardware support: Intel CET, ARM BTI
- Toolchain support: MSVC, LLVM, GCC
- OS support: GNU/Linux, Windows



BROWN

CET

- A CFI hardware feature on modern Intel CPUs

BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:

BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack

BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

- New 4-byte instruction: `endbr`

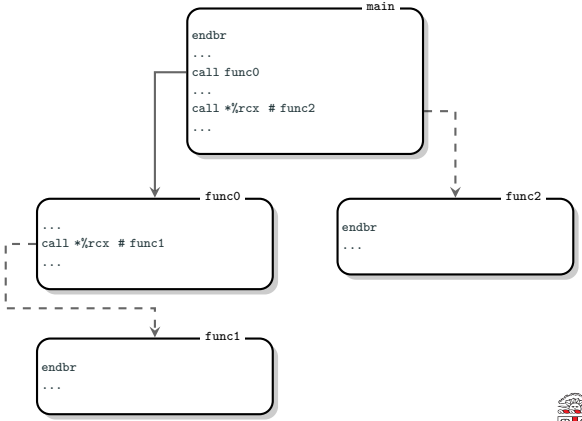
BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

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



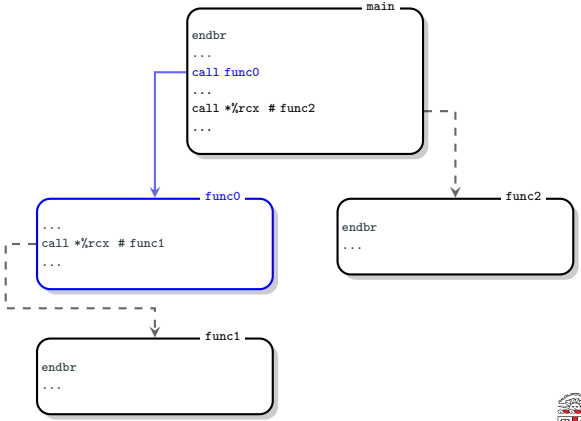
BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

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



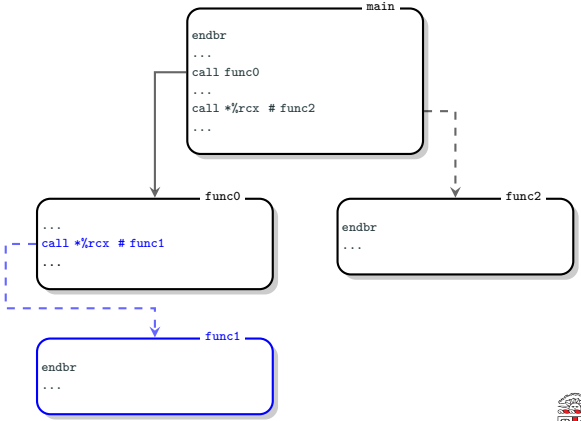
BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

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



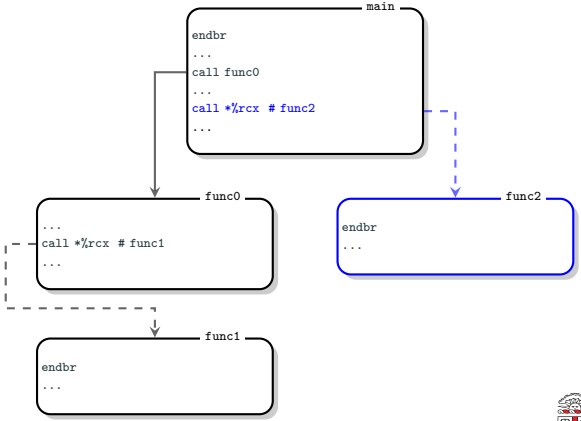
BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

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



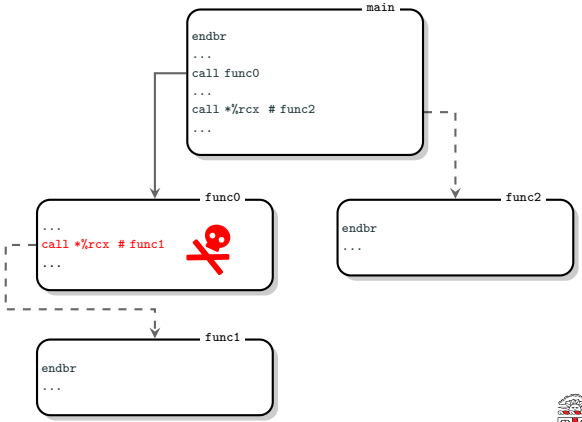
BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

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



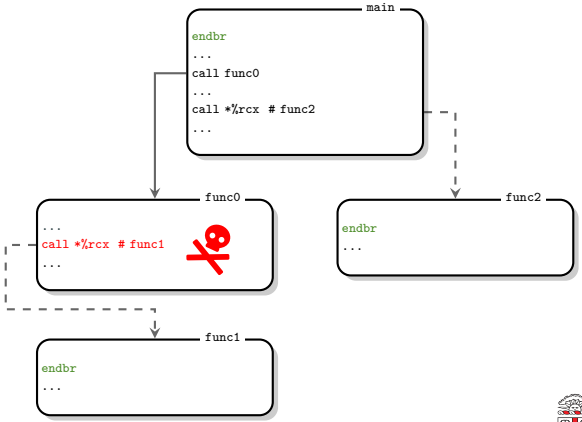
BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

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



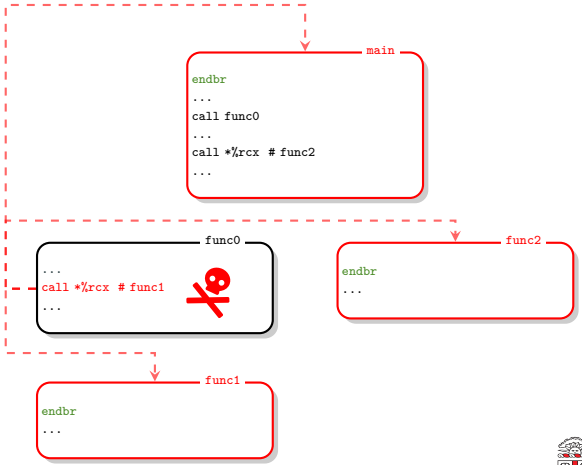
BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

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



BROWN

CET

- A CFI hardware feature on modern Intel CPUs
- Two components:
  1. shadow stack
  2. indirect branch tracking

Indirect Branch Tracking

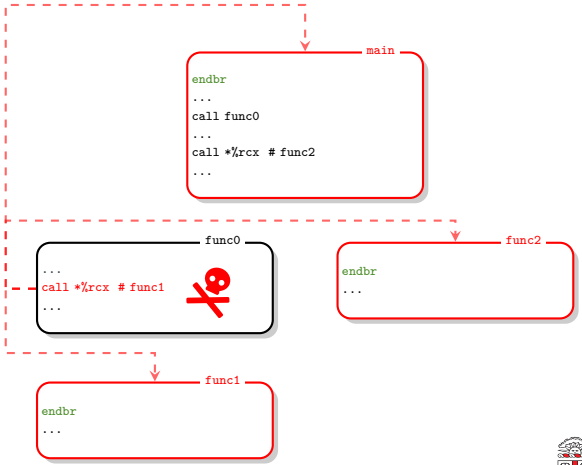
- New 4-byte instruction: `endbr`
- Indirect branches must target an `endbr`

Target Domain: Anywhere HW feature is available

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

Coverage: Forward-edges

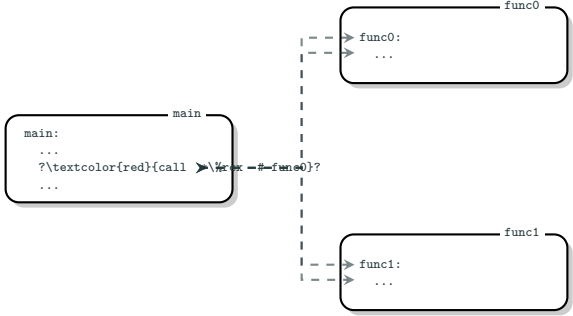
Effectiveness: Coarse-grain



BROWN

FineIBT

- Policy-agnostic mechanism to increase IBT's granularity



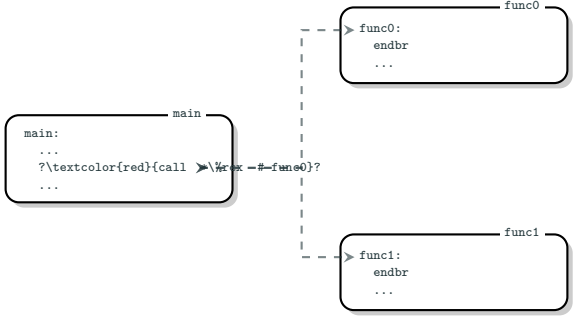
For more details re: FineIBT, see our work published in RAID 2023!

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

BROWN

FineIBT

- Policy-agnostic mechanism to increase IBT's granularity
- Leverages `endbrs` to "pin" indirect branches



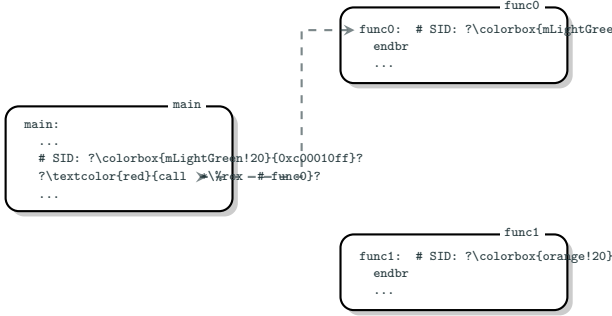
For more details re: FineIBT, see our work published in RAID 2023!

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

BROWN

FineIBT

- Policy-agnostic mechanism to increase IBT's granularity
- Leverages `endbr`s to "pin" indirect branches
- Associates caller/callee pairs with SIDs



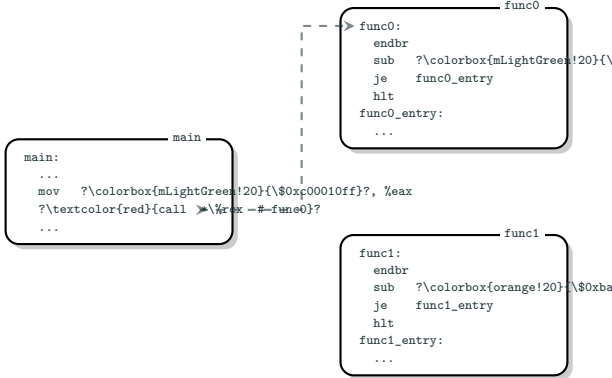
For more details re: FineIBT, see our work published in RAID 2023!

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

BROWN

FineIBT

- Policy-agnostic mechanism to increase IBT's granularity
- Leverages `endbr`s to "pin" indirect branches
- Associates caller/callee pairs with SIDs
- Extends compile-time instrumentation with fast SID checks



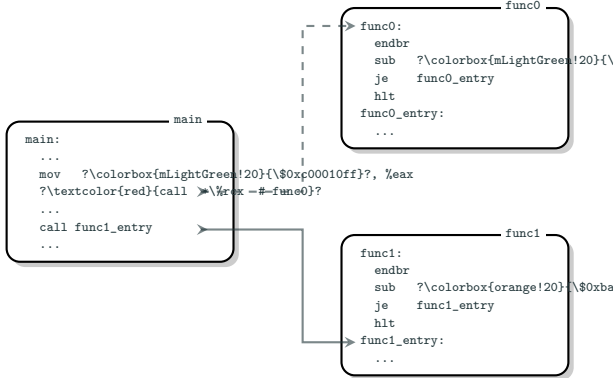
For more details re: FineIBT, see our work published in RAID 2023!

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

BROWN

FineIBT

- Policy-agnostic mechanism to increase IBT's granularity
- Leverages `endbr`s to "pin" indirect branches
- Associates caller/callee pairs with SIDs
- Extends compile-time instrumentation with fast SID checks
- Direct calls bypass all instrumentation, experiencing no overhead



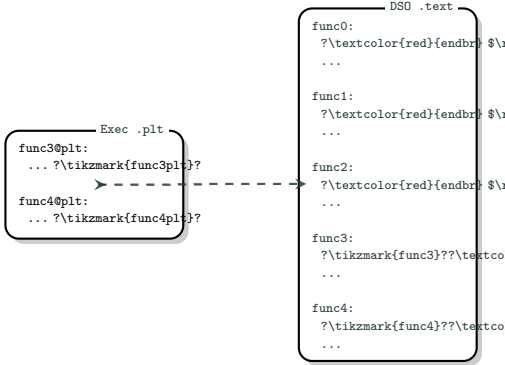
For more details re: FineIBT, see our work published in RAID 2023!

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

BROWN

NOPout

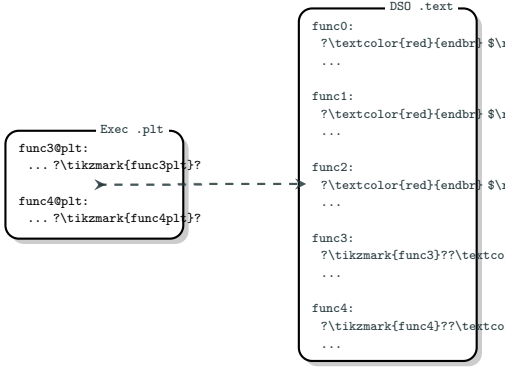
- Load-time feature to *elide* `endbr` from unused functions



BROWN

NOPout

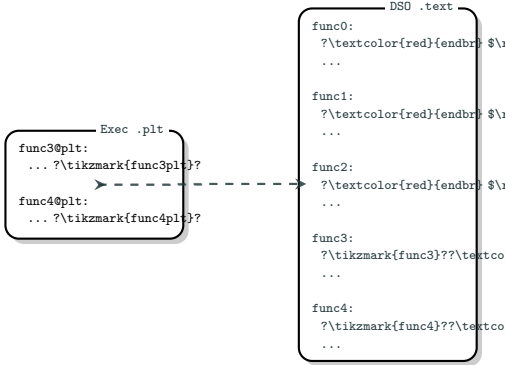
- Load-time feature to *elide* `endbr` from unused functions
- Reduces attack surface irrespective of policy



BROWN

NOPout

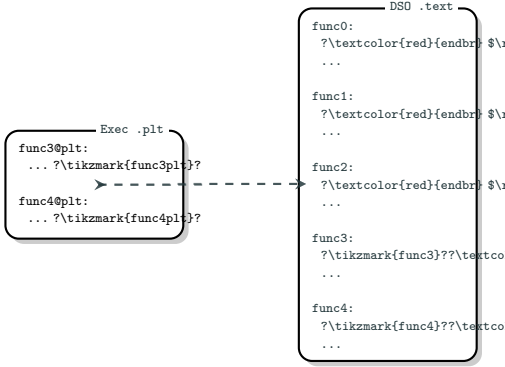
- Load-time feature to *elide* `endbr` from unused functions
- Reduces attack surface irrespective of policy
- During compilation, symbols are collected into `.plt.nopout` that are:



BROWN

NOPout

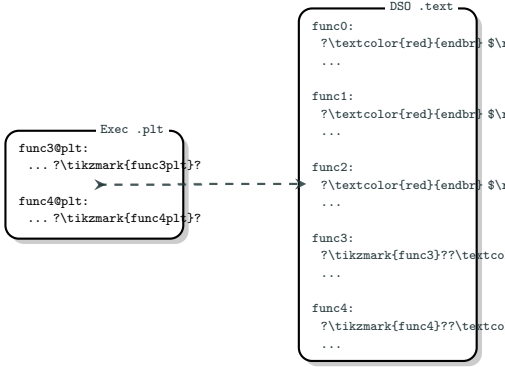
- Load-time feature to *elide* `endbr` from unused functions
- Reduces attack surface irrespective of policy
- During compilation, symbols are collected into `.plt.nopout` that are:
  - Non-address-taken



BROWN

NOPout

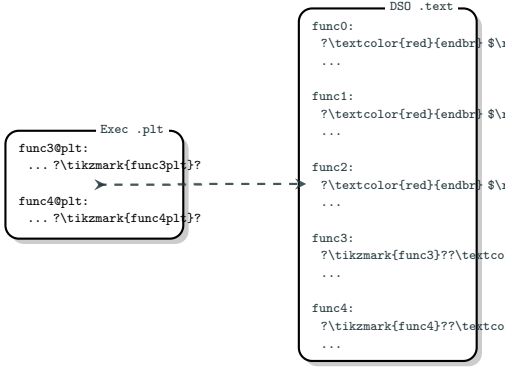
- Load-time feature to *elide* `endbr` from unused functions
- Reduces attack surface irrespective of policy
- During compilation, symbols are collected into `.plt.nopout` that are:
  - Non-address-taken
  - Non-local (i.e., will be visible to other object files)



BROWN

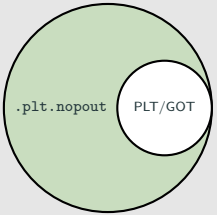
NOPout

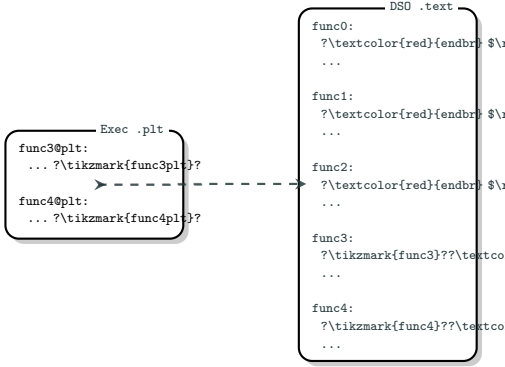
- Load-time feature to *elide* `endbr` from unused functions
- Reduces attack surface irrespective of policy
- During compilation, symbols are collected into `.plt.nopout` that are:
  - Non-address-taken
  - Non-local (i.e., will be visible to other object files)
  - Functions (i.e., `STT_FUNC`s)



NOPout

- Load-time feature to *elide* `endbr` from unused functions
- Reduces attack surface irrespective of policy
- During compilation, symbols are collected into `.plt.nopout` that are:
  - Non-address-taken
  - Non-local (i.e., will be visible to other object files)
  - Functions (i.e., `STT_FUNC`s)
- At load-time, `libnopout.so` compares `.plt.nopout` to PLT/GOT slots
- If an entry is in `.plt.nopout` but not in a PLT/GOT slot, its 4-byte `endbr` is replaced with a 4-byte `nop`





BROWN

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime

NOPoutNG Solutions

BROWN

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults

NOPoutNG Solutions

BROWN

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects

NOPoutNG Solutions

BROWN

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime

BROWN

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects

BROWN

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes
   - Increased disk/memory footprint

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes
   - Increased disk/memory footprint

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects
2. No Extra Metadata: Leverages existing data

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes
   - Increased disk/memory footprint

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects
2. No Extra Metadata: Leverages existing data
   - No standard changes needed

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes
   - Increased disk/memory footprint

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects
2. No Extra Metadata: Leverages existing data
   - No standard changes needed
   - No additional disk/memory footprint

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes
   - Increased disk/memory footprint
3. Dynamic Loading: Loose description, no evaluation

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects
2. No Extra Metadata: Leverages existing data
   - No standard changes needed
   - No additional disk/memory footprint

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes
   - Increased disk/memory footprint
3. Dynamic Loading: Loose description, no evaluation

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects
2. No Extra Metadata: Leverages existing data
   - No standard changes needed
   - No additional disk/memory footprint
3. Dynamic Loading: Implementation and evaluation

BROWN

NOPout vs. NOPoutNG

NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes
   - Increased disk/memory footprint
3. Dynamic Loading: Loose description, no evaluation
4. Limited Evaluation: No performance measurements

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects
2. No Extra Metadata: Leverages existing data
   - No standard changes needed
   - No additional disk/memory footprint
3. Dynamic Loading: Implementation and evaluation

BROWN

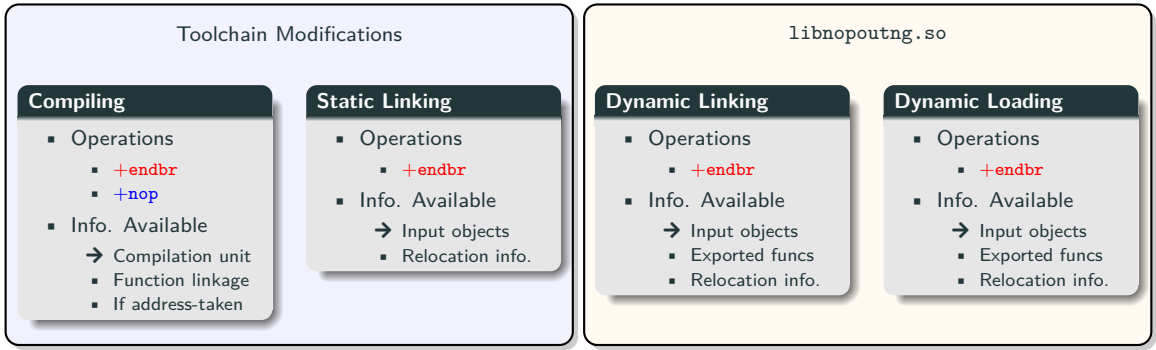
NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
   - Insecure defaults
   - Increased CoW effects
2. Extraneous Metadata: Requires new ELF section
   - Needs ELF standard changes
   - Increased disk/memory footprint
3. Dynamic Loading: Loose description, no evaluation
4. Limited Evaluation: No performance measurements

NOPoutNG Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
   - Secure defaults
   - Minimized CoW effects
2. No Extra Metadata: Leverages existing data
   - No standard changes needed
   - No additional disk/memory footprint
3. Dynamic Loading: Implementation and evaluation
4. Thorough Evaluation: Performance measurements

BROWN


Toolchain Modifications
Compiling
Operations
+endbr
+nop
Info. Available
Compilation unit
Function linkage
If address-taken
Static Linking
Operations
+endbr
Info. Available
Input objects
Relocation info.
libnopoutng.so
Dynamic Linking
Operations
+endbr
Info. Available
Input objects
Exported funcs
Relocation info.
Dynamic Loading
Operations
+endbr
Info. Available
Input objects
Exported funcs
Relocation info.

BROWN

Information Available

- Linkage and partial address-taken property of functions within compilation unit

Inserting `endbrs`

- `NOPoutNG` inserts `endbr` into functions that:
  - Are address-taken

Inserting `nops`

- `NOPoutNG` inserts `nops` into functions that:
  - Aren't local (i.e., `STB_BIND != STB_LOCAL`)
  - Aren't address-taken locally

a.c

```
1  int main(int c, char **v) {
2    ((c > 1) ? &f1 : &f2)();
3    f3();
4  }
```

Compile
+endbr
+nop

a.o

```
?<?main?>:?
   ?0:? ?\textcolor{\nopcolor}{nopl   (\%ra
   ?\minteddots?
   ?c:? mov  0x0(%rip),%rax  # ?<?f2?>?
     # -> R_X86_64_REX_GOTPCRELX
   ?13:? mov  0x0(%rip),%rcx  # ?<?f1?>?
     # -> R_X86_64_REX_GOTPCRELX
   ?\minteddots?
   ?2b:? callq  *%rax
   ?2d:? callq  32  # ?<?f3?>?
     # -> R_X86_64_PLT32
   ?\minteddots?
```

b.c

```
1  void f1() { puts("f1\n"); }
2  void f2() { puts("f2\n"); }
3  void f3() {
4    void (*fp)() = &f2;
5    fp();
6  }
```

Compile
+endbr
+nop

b.o

```
?<?f1?>:?
   ?0:? ?\textcolor{\nopcolor}{nopl   (\%ra
   ?\minteddots?
   ?f:? callq  14  # ?<?puts?>?
     # -> R_X86_64_PLT32
   ?\minteddots?
?<?f2?>:?
   ?20:? ?\textcolor{\endbrcolor}{endbr64}?
   ?\minteddots?
   ?2f:? callq  34  # ?<?puts?>?
     # -> R_X86_64_PLT32
   ?\minteddots?
?<?f3?>:?
   ?40:? ?\textcolor{\nopcolor}{nopl   (\%ra
   ?\minteddots?
   ?4c:? mov  0x0(%rip),%rax  # ?<?f2?>?
     # -> R_X86_64_REX_GOTPCRELX
   ?\minteddots?
   ?59:? callq  *-0x8(%rbp)
   ?\minteddots?
```

Information Available

- Relocation info. for all input object files

Inserting `endbrs`

- Exported functions address-taken locally already have `endbrs`
- `R_X86_64_REX_GOTPCRELX` relocations for `f1` and `f2` so we add `endbrs`

Maintaining `nop`s

- The call to `f3` is relaxed and doesn't turn into a call via the PLT → `nop` stays

Information Available

- Exported functions in all loaded binaries
- Dynamic relocations from all loaded binaries

Metadata Collection

- `NOPoutNG` searches the `.dynsym` of all loaded objects, collecting:
  - All functions required (i.e., `SHN_UNDEF`, `STT_FUNC`)
  - All functions provided (i.e., defined, `STT_FUNC`)

Inserting `endbrs`

- Required functions provided get `endbrs`

Maintaining `nops`

- Provided functions **not** required keep `nops`

Information Available

- Exported functions in all loaded binaries
- Dynamic relocations from all loaded binaries

Metadata Collection

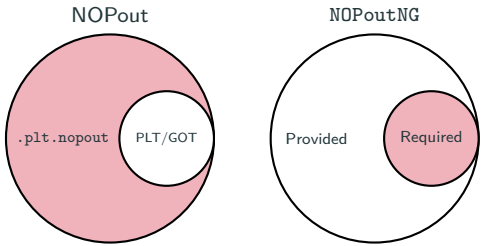
- `NOPoutNG` searches the `.dynsym` of all loaded objects, collecting:
  - All functions required (i.e., `SHN_UNDEF`, `STT_FUNC`)
  - All functions provided (i.e., defined, `STT_FUNC`)

Inserting `endbrs`

- Required functions provided get `endbrs`

Maintaining `nops`

- Provided functions **not** required keep `nops`



⚓ Hooking `dlopen` and `dlsym`

- Record the addresses of `dlopen` and `dlsym` in `libc.so` while iterating through its symbols
- Preempt `libc.so`'s symbols with `dlopen` and `dlsym` hooks provided by `NOPoutNG`
- The hooks call the original `dlopen` and `dlsym` functions and do `endbr` insertion before returning

Hooking `dlopen` and `dlsym`

- Record the addresses of `dlopen` and `dlsym` in `libc.so` while iterating through its symbols
- Preempt `libc.so`'s symbols with `dlopen` and `dlsym` hooks provided by `NOPoutNG`
- The hooks call the original `dlopen` and `dlsym` functions and do `endbr` insertion before returning

Live-Patching Scheme

- Naively patching `endbrs` into code would allow an attacker to corrupt code pages while writable
- We implement a secure live-patching scheme based on NOPout's originally proposed design
- This allows `s/nop/endbr/` safely at runtime

Hooking `dlopen` and `dlsym`

- Record the addresses of `dlopen` and `dlsym` in `libc.so` while iterating through its symbols
- Preempt `libc.so`'s symbols with `dlopen` and `dlsym` hooks provided by `NOPoutNG`
- The hooks call the original `dlopen` and `dlsym` functions and do `endbr` insertion before returning

Live-Patching Scheme

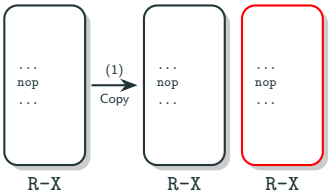
- Naively patching `endbr`s into code would allow an attacker to corrupt code pages while writable
- We implement a secure live-patching scheme based on NOPout's originally proposed design
- This allows `s/nop/endbr/` safely at runtime

Hooking `dlopen` and `dlsym`

- Record the addresses of `dlopen` and `dlsym` in `libc.so` while iterating through its symbols
- Preempt `libc.so`'s symbols with `dlopen` and `dlsym` hooks provided by `NOPoutNG`
- The hooks call the original `dlopen` and `dlsym` functions and do `endbr` insertion before returning

Live-Patching Scheme

- Naively patching `endbr`s into code would allow an attacker to corrupt code pages while writable
- We implement a secure live-patching scheme based on NOPout's originally proposed design
- This allows `s/nop/endbr/` safely at runtime

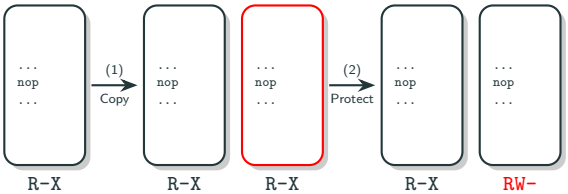


Hooking `dlopen` and `dlsym`

- Record the addresses of `dlopen` and `dlsym` in `libc.so` while iterating through its symbols
- Preempt `libc.so`'s symbols with `dlopen` and `dlsym` hooks provided by `NOPoutNG`
- The hooks call the original `dlopen` and `dlsym` functions and do `endbr` insertion before returning

Live-Patching Scheme

- Naively patching `endbr`s into code would allow an attacker to corrupt code pages while writable
- We implement a secure live-patching scheme based on NOPout's originally proposed design
- This allows `s/nop/endbr/` safely at runtime

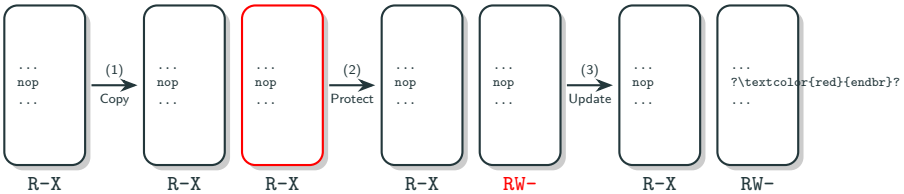


Hooking `dlopen` and `dlsym`

- Record the addresses of `dlopen` and `dlsym` in `libc.so` while iterating through its symbols
- Preempt `libc.so`'s symbols with `dlopen` and `dlsym` hooks provided by `NOPoutNG`
- The hooks call the original `dlopen` and `dlsym` functions and do `endbr` insertion before returning

Live-Patching Scheme

- Naively patching `endbr`s into code would allow an attacker to corrupt code pages while writable
- We implement a secure live-patching scheme based on NOPout's originally proposed design
- This allows `s/nop/endbr/` safely at runtime

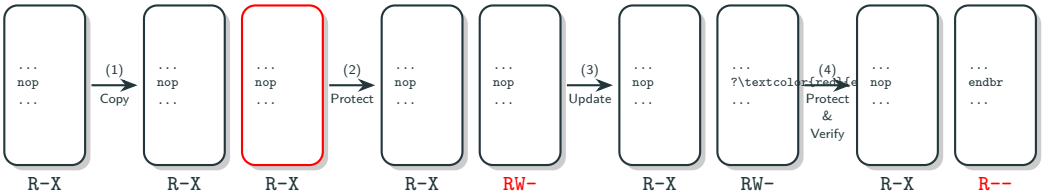


Hooking `dlopen` and `dlsym`

- Record the addresses of `dlopen` and `dlsym` in `libc.so` while iterating through its symbols
- Preempt `libc.so`'s symbols with `dlopen` and `dlsym` hooks provided by `NOPoutNG`
- The hooks call the original `dlopen` and `dlsym` functions and do `endbr` insertion before returning

Live-Patching Scheme

- Naively patching `endbr`s into code would allow an attacker to corrupt code pages while writable
- We implement a secure live-patching scheme based on NOPout's originally proposed design
- This allows `s/nop/endbr/` safely at runtime

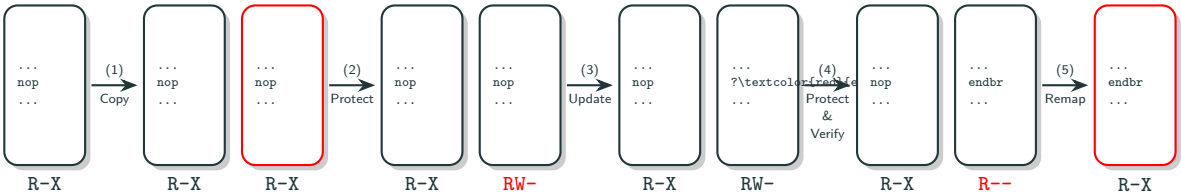


Hooking `dlopen` and `dlsym`

- Record the addresses of `dlopen` and `dlsym` in `libc.so` while iterating through its symbols
- Preempt `libc.so`'s symbols with `dlopen` and `dlsym` hooks provided by `NOPoutNG`
- The hooks call the original `dlopen` and `dlsym` functions and do `endbr` insertion before returning

Live-Patching Scheme

- Naively patching `endbr`s into code would allow an attacker to corrupt code pages while writable
- We implement a secure live-patching scheme based on NOPout's originally proposed design
- This allows `s/nop/endbr/` safely at runtime



Toolchain Modifications

- Modified LLVM v12
  - Adds ≈100 C++ LoC to compiler
  - Adds ≈100 C++ LoC to (static) linker

Runtime Library

- Self-contained shared library, `libnopoutng.so`, ≈1250 C LoC
- File-system footprint ≈42KB
- Memory footprint ≈19KB
- `LD_PRELOAD` library to activate protection

BROWN

Machine Description

- CPU: Intel Core i7-11800H @ 2.3GHz
- Memory: 16GB of DDR4 RAM
- OS: Debian v11 with Linux kernel v5.13 (patched to support user-space IBT)
- DVFS disabled
- Forced C-state to be `C0`
- Single user mode enabled for "quiet" system

Software Description

- Hardened *all* benchmark applications and their dependencies
- Software built:
  - position independent
  - `-O2` optimization
  - full `RELRO` (incl. eager binding)
- `musl libc` used

BROWN

SPEC CPU 2017

- Motivation
  - Industry standard benchmark
  - Runtime reports include program startup time
- Setup
  - 8 C benchmarks from SPECspeed Int/FP suites
  - 5 runs of the `ref` workload
- Results
  - IBT overhead is minimal, ranging from ≈0%–5.03%
  - `NOPoutNG` adds negligible overhead to IBT

Benchmark	IBT	IBT + `NOPoutNG`
`600.perlbench`	5.03%	5.03%
`602.gcc`	0.77%	0.77%
`605.mcf`	0.06%	0.06%
`625.x264`	0.08%	0.15%
`657.xz`	0.43%	0.43%
`619.lbm`	≈0%	3.67%
`638.imagick`	0.08%	0.08%
`644.nab`	0.45%	0.45%

BROWN

Real-world Applications

- Motivation
  - Variety of real-world use-cases
- Setup
  - Networking performed over `lo`
  - Benchmarks were tuned to maximize CPU utilization
  - 10 runs of each benchmark
- Results
  - `NOPoutNG` adds negligible overhead

Benchmark	IBT	IBT + `NOPoutNG`
Nginx (1KB)	0.07%	0.07%
Nginx (100KB)	0.25%	0.25%
Nginx (1MB)	0.30%	1.56%
Redis (`GET`)	≈0%	≈0%
Redis (`SET`)	≈0%	≈0%
SQLite	≈0%	0.06%

BROWN

Dynamic Loading μBenchmark

- Motivation
  - Study the performance of our `dlopen`/`dlsym` implementation
- Setup
  - Compression sizes derived from the Calgary corpus*
  - Data was non-uniform and constant across runs
  - 100 benchmark iterations
- Results
  - `NOPoutNG` yields an average overhead of 13.23%

Dynamic Loading μBenchmark Design

- Allocate buffers *etc.*
- Start timing
  - `dlopen zlib`
  - `dlsym compress, uncompress, and crc32`
  - Checksum, compress, and uncompress 4 buffers
- Stop timing
- Free buffers *etc.*

BROWN

*T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for Text Compression," ACM Computing Surveys (CSUR), vol. 21, no. 4, pp. 557–591, 1989.

Effectiveness

endbr Reduction

- Larger reduction equates to more security and a reduced attack surface.
- Across the board, we are able to reduce the number of endbr dynamically by 33.49%–86.40%.

Application	# endbr	
	IBT	NOPoutNG
Nginx	3244	1466 (-54.47%)
Redis	5413	3600 (-33.49%)
SQLite	2366	1111 (-53.04%)
600.perlbench	3758	1341 (-64.32%)
602.gcc	11512	4505 (-60.87%)
605.mcf	1746	403 (-76.92%)
625.x264	2141	646 (-69.83%)
657.xz	2057	519 (-74.77%)
619.lbm	1726	397 (-77.00%)
638.imagick	3639	495 (-86.40%)
644.nab	1891	429 (-77.31%)

BROWN

Comparison with NOPout

- The order of operations in `NOPoutNG` significantly reduces the amount of CoW work that needs to be performed.
- This increases the "shareability" of libraries.

Application	NOPout	NOPoutNG	
	+nop	+endbr	Pages
Nginx	1602 (92.44%)	131 (7.56%)	42 (172KB)
Redis	1656 (91.34%)	157 (8.66%)	53 (217KB)
SQLite	1101 (94.67%)	62 (5.33%)	25 (102KB)
`600.perlbench`	1089 (93.56%)	75 (6.44%)	27 (111KB)
`602.gcc`	1115 (95.46%)	53 (4.54%)	20 (82KB)
`605.mcf`	1156 (99.40%)	7 (0.60%)	5 (20KB)
`625.x264`	1158 (99.40%)	7 (0.60%)	4 (16KB)
`657.xz`	1150 (98.88%)	13 (1.12%)	9 (37KB)
`619.lbm`	1158 (99.57%)	5 (0.43%)	4 (16KB)
`638.imagick`	1156 (99.40%)	7 (0.60%)	4 (16KB)
`644.nab`	1138 (97.51%)	29 (2.49%)	18 (74KB)

Conclusion

! NOPout Problems

1. Order of Operations: `s/endbr/nop/` at runtime
2. Extraneous Metadata: Requires new ELF section
3. Dynamic Loading: Loose description, no evaluation
4. Limited Evaluation: No performance measurements

✓ `NOPoutNG` Solutions

1. Order of Operations: `s/nop/endbr/` at runtime
2. No Extra Metadata: Leverages existing data
3. Dynamic Loading: Implementation and evaluation
4. Thorough Evaluation: Performance measurements

Prototype Repository
https://gitlab.com/brown-ssl/nopoutng

BROWN

Backup

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

Example

```
 1 int
 2 main(int argc, char *argv[])
 3 {
 4   int (*fptr)(void) = &foo;
 5   char arr[10];
 6   ...
 7   strcpy(arr, argv[1]);
 8   ...
 9   fptr();
10   ...
11 }
```

Repercussions

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

CFI Support

Toolchain Support

- Microsoft Visual C++ (MSVC)
  - Control Flow Guard (CFG) → forward-edge scheme that validates control-flow transfers via a bitmap
  - Extended Flow Guard (XFG) → forward-edge scheme that builds on CFG with a type-based policy
  - Intel CET (Shadow Stacks)
- LLVM/Clang
  - Clang CFI → forward-edge scheme with a type-based policy
  - ShadowCallStack (SCS) → backward-edge scheme to isolate return addresses
  - SafeStack → backward-edge scheme that splits the stack into safe and unsafe parts
  - Intel CET (Shadow Stacks and IBT)
- GCC
  - Virtual Table Verification (VTV) → for C++, verifies at runtime that vtable pointer and vtable object types match
  - Intel CET (Shadow Stacks and IBT)

Hardware Support

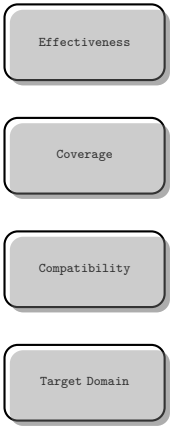
- Intel CET starting with "Tiger Lake" CPUs
- ARM BTI starting with ARMv8.5-A

Operating System Support

- Linux Kernel
  - kCFI → forward-edge scheme implemented in software that uses a type-based policy
  - IBT → HW-assisted, coarse-grain, forward-edge scheme
  - FineIBT → HW-assisted, fine-grain, forward-edge scheme
  - Userspace Shadow Stacks
- Windows Kernel
  - KCFG (Kernel Control Flow Guard) → kernel implementation of CFG

Control-flow Integrity Scheme Classifications

- CFI schemes can be classified according to 4 properties:



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

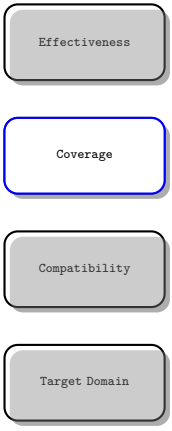
Coarse-grain schemes are overly permissive

Fine-grain schemes refine the set of allowed targets

BROWN

Control-flow Integrity Scheme Classifications

- CFI schemes can be classified according to 4 properties:



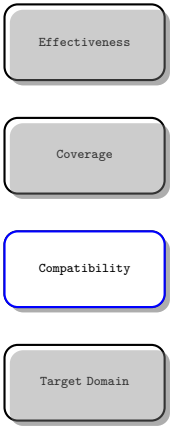
Coverage

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

BROWN

Control-flow Integrity Scheme Classifications

- CFI schemes can be classified according to 4 properties:



Compatibility

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

BROWN

- CFI schemes can be classified according to 4 properties:

Target Domain

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

BROWN

Notable Relocation Types

Relocation	Calculation
`R_X86_64_64`	$value + addend$
`R_X86_64_PC32`	$value + addend - offset$
`R_X86_64_PC64`	$value + addend - offset$
`R_X86_64_GOTPCREL`	$GOT.offset + \&GOT + addend - offset$
`R_X86_64_GOTPCRELX`	$GOT.offset + \&GOT + addend - offset$
`R_X86_64_REX_GOTPCRELX`	$GOT.offset + \&GOT + addend - offset$

Non-IBT PLT

```
1  PLT0: pushq    GOT?+?8(%rip)         /*  GOT[1]  */
2        jmp      *GOT?+?16(%rip)       /*  GOT[2]  */
3        nopl     0x0(%rax)           /*   PAD    */
4  ...
5  PLT3: jmp      *fsym3@GOT(%rip)    /*  GOT[5]  */
6        pushq    $0x2
7        jmp      PLT0
8  PLT4: jmp      *fsym4@GOT(%rip)    /*  GOT[6]  */
9        pushq    $0x3
10       jmp      PLT0
11 ...
12 PLTn: jmp      *fsymn@GOT(%rip)    /* GOT[n+2] */
13       pushq    $0xn-1
14       jmp      PLT0
```

BROWN

IBT PLT

```
1  PLT0: pushq    GOT?+?8(%rip)         /*  GOT[1]  */
2        jmp      *GOT?+?16(%rip)       /*  GOT[2]  */
3        nopl     0x0(%rax)             /*   PAD    */
4  ...
5  PLT3: endbr64
6        pushq    $0x2
7        jmp      PLT0
8        xchg     %ax,%ax               /*   PAD    */
9  PLT4: endbr64
10       pushq    $0x3
11       jmp      PLT0
12       xchg     %ax,%ax               /*   PAD    */
13 ...
14 PLTn: endbr64
15       pushq    $0xn-1
16       jmp      PLT0
17       xchg     %ax,%ax               /*   PAD    */
18
19 ...
20 SPLT3: endbr64
21        jmp      *fsym3@GOT(%rip)    /*  GOT[5]  */
22        nopw     0x0(%rax,%rax,1)    /*   PAD    */
23 SPLT4: endbr64
24        jmp      *fsym4@GOT(%rip)    /*  GOT[6]  */
25        nopw     0x0(%rax,%rax,1)    /*   PAD    */
26 ...
27 SPLTn: endbr64
28        jmp      *fsymn@GOT(%rip)    /* GOT[n+2] */
29        nopw     0x0(%rax,%rax,1)    /*   PAD    */
```

BROWN

FineIBT PLT

```
1  PLT0:    shl    $0x20, %rax
2           or     $SID, %rax
3           pushq  GOT?+?8(%rip)          /* GOT[1] */
4           jmp    *GOT?+?16(%rip)        /* GOT[2] */
5           nopw   %cs:0x0(%rax,%rax,1) /*  PAD   */
6  ...
7  PLT4:    endbr64
8           cmp    $SID, %eax
9           pushq  $0x3
10          xchg   %ax, %ax              /*  PAD   */
11          je     PLT0
12          hlt
13          nopw   0x0(%rax,%rax,1)      /*  PAD   */
14 ...
15 FPLT4:   mov    $SID, %eax
16          jmp    *fsym4@GOT(%rip)      /* GOT[6] */
17          nopl   0x0(%rax,%rax,1)      /*  PAD   */
18 ...
19 ATFPLT4: endbr64
20          sub    $SID, %eax
21          je     FPLT4
22          hlt
23          data16 nopw %cs:0x0(%rax,%rax,1) /*PAD*/
24          nopl   (%rax)                /*  PAD   */
```

BROWN

ARM Instrumentation (FineBTI)

```
1  main:                        /* caller */
2    ...
3    movz w9, #0x3a, lsl #16    /* SID = 0x3a0000 */
4    blr  x0                    /* x0 = &func */
5    ...
6    bl   func_entry
7    ...
8  .func_finebti_coldpath:
9    ...                        /* arg0, ..., argn */
10   bl   __finebti_chk_fail@PLT
11 func:                        /* callee */
12   bti  c
13   subs w9, w9, #0x3a0, lsl #12 /* SID=0x3a0000 */
14   bne  .func_finebti_coldpath
15 func_entry:
16   ...
```

BROWN