Hardware-Assisted Fine-Grained Control-Flow Integrity: Adding Lasers to Intel’s CET/IBT

Joao Moreira

<joao@overdrivepizza.com>

<joao.moreira@intel.com>

@lvwr

LINUX SECURITY SUMMIT

intel®

Notices and Disclaimers

Performance varies by use, configuration and other factors.
Learn more at www.Intel.com/PerformanceIndex

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Notices and Disclaimers

This is work in progress!
Existing implementation may have bugs and/or be currently incomplete.

“Adding Lasers” is just a metaphor to making something more accurate.
We are not really using lasers.

Memory Corruption Bugs

Enable control-flow hijacking

A Metaphorical Example

Memory Corruption Bugs

Enable control-flow hijacking

```
<foo>:
…
fptr = &bar;
strcpy(…);
fptr();
…
```

```
<bar>:
if (user_id > 0) return;
do some magic
```

Memory Corruption Bugs

Enable control-flow hijacking

```
<foo>:
…
fptr = &bar;
strcpy(…);
fptr();
…
```

```
<bar>:
if (user_id > 0) return;
do some magic
```

Memory Corruption Bugs

Enable control-flow hijacking

```
<foo>:
…
fptr = &bar;
strcpy(…);
fptr();
…
```

```
<bar>:
if (user_id > 0) return;
do some magic
```

FPTR = 0x&BAR

Memory Corruption Bugs

Enable control-flow hijacking

```
<foo>:
…
fptr = &bar;
strcpy(…);
fptr();
…
```

BUG

```
<bar>:
if (user_id > 0) return;
do some magic
```

FPTR = 0x&BAR + OFFSET

Memory Corruption Bugs

Enable control-flow hijacking

```
<foo>:
…
fptr = &bar;
strcpy(…);
fptr();
…
```

```
<bar>:
if (user_id > 0) return;
do some magic
```

FPTR = 0x&BAR + OFFSET

Memory Corruption Bugs

Enable control-flow hijacking

```
<foo>:
…
fptr = &bar;
strcpy(…);
fptr();
…
```

```
<bar>:
endbr
if (user_id > 0) return;
do some magic
```

Memory Corruption Bugs

Enable control-flow hijacking

```
<foo>:
…
fptr = &bar;
strcpy(…);
fptr();
…
```

```
<bar>:
endbr
if (user_id > 0) return;
do some magic
```

FPTR = 0x&BAR + OFFSET

Coarse-Grained Control-Flow Integrity

Forward-edges

Branches are enforced to function's first instruction by an ABI-like scheme

Coarse-Grained Control-Flow Integrity

Forward-edges

Branches are enforced to function’s first instruction by an ABI-like scheme

Any function can be reached from **any** indirect call or jump

Coarse-Grained Control-Flow Integrity

Forward-edges

Branches are enforced to function’s first instruction by an ABI-like scheme

Any function can be reached from **any** indirect call or jump

Functions can still be used out of context to exploit the system

Coarse-Grained Control-Flow Integrity

Forward-edges

Branches are enforced to function’s first instruction by an ABI-like scheme

Any function can be reached from **any** indirect call or jump

Functions can still be used out of context to exploit the system

Doesn’t fully mitigate control-flow hijacking

Memory Corruption Bugs

Enable control-flow hijacking

Another ~~Metaphorical~~ Example

CVE-2021-3156

Baron Samedit: Heap-based buffer overflow in sudo

CVE-2021-3156

Baron Samedit: Heap-based buffer overflow in sudo (exploit 1)

Corrupts pointer `fn_getenv` in the heap-based struct `sudo_hook_entry`

CVE-2021-3156

Baron Samedit: Heap-based buffer overflow in sudo (exploit 1)

Corrupts pointer **`fn_getenv`** in the heap-based struct **`sudo_hook_entry`**

Corrupted pointer targets **`execve`**/**`execv`** PLT entries in sudoers.so

CVE-2021-3156

Baron Samedit: Heap-based buffer overflow in sudo (exploit 1)

Corrupts pointer **`fn_getenv`** in the heap-based struct **`sudo_hook_entry`**

Corrupted pointer targets **`execve/execv`** PLT entries in sudoers.so

Exploit succeds despite the **`endbr`** in the PLT entry
(as explicitly dumped in the advisory)

Fine-Grained Control-Flow Integrity

Tightens the CFG with additional rules enforced by the ABI-like scheme

Fine-Grained Control-Flow Integrity

Tightens the CFG with additional rules enforced by the ABI-like scheme

Ideal rule: Indirect branch target is the supposed-to-be target

Fine-Grained Control-Flow Integrity

Tightens the CFG with additional rules enforced by the ABI-like scheme

Ideal rule: Indirect branch target is the supposed-to-be target
In practice: This is statically undecidable

Fine-Grained Control-Flow Integrity

Tightens the CFG with additional rules enforced by the ABI-like scheme

Ideal rule: Indirect branch target is the supposed-to-be target
In practice: This is statically undecidable

Heuristics to the rescue:
Clustering functions and pointers by prototypes

CVE-2021-3156

Baron Samedit: Heap-based buffer overflow in sudo

Function pointer:

```
int (*fn_getenv)(char *, char **, void *)
```

Function prototypes:

```
int execv(char *, char **)

int execve(char *, char **, char **)

int sudoers_hook_getenv(char *, char **, void *)
```

Fine-Grained Control-Flow Integrity

Prototype matching-based implementations

PaX/grsecurity RAP, Clang CFI, Microsoft XFG (and others…)

All software/ABI-like schemes

Other forward-edge CFI schemes exist but are beyond the scope of this talk

Hypotheses

Can we enhance CET/IBT in a way to make it fine-grained?

How much of the hardware-provided perks would it retain?

FineIBT

FineIBT

Hybrid software-hardware implementation

ABI-like scheme

`endbr` instructions anchor control-flow to the beginning of functions

FineIBT

Hybrid software-hardware implementation

ABI-like scheme

`endbr` instructions anchor control-flow to the beginning of functions

Additional checks/policies are performed on function prologue

FineIBT

Hybrid software-hardware implementation

ABI-like scheme

`endbr` instructions anchor control-flow to the beginning of functions

Additional checks/policies are performed on function prologue

Checks emitted by the compiler

FineIBT

Compiler instrumentation

Indirect calls do **hash sets**

FineIBT

Compiler instrumentation

Indirect calls do **hash sets**

Functions prologues do **hash checks**

FineIBT

Compiler instrumentation

Indirect calls do **hash sets**

Functions prologues do **hash checks**

Direct calls incremented with an offset to skip prologue hash checks

FineIBT

Compiler instrumentation

Indirect calls do **hash sets**

Functions prologues do **hash checks**

Direct calls incremented with an offset to skip prologue hash checks

Hashes are generated based on function and pointer prototypes

Regular Assembly Code

```
<foo>:
…
call *rax
…
call <bar>

<bar>:
…
```

IBT Assembly Code

```
<foo>:
…
call *rax
…
call <bar>
```

```
<bar>:
endbr
…
```

FineIBT Assembly Code

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…
call <bar_entry>
```

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT Assembly Code

HASH set

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…
call <bar_entry>
```

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT Assembly Code

HASH SET

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…
call <bar_entry>
```

REGULAR ENDBR

HASH CHECK

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT Assembly Code

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…
call <bar_entry>
```

HASH SET

DESTROYS R11 AND SETS ZF

REGULAR ENDBR

HASH CHECK

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT Assembly Code

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…
call <bar_entry>
```

HASH SET

DIRECT CALL WITH FIXED OFFSET

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

REGULAR ENDBR

DESTROYS R11 AND SETS ZF

HASH CHECK

FineIBT Cross-DSO support

Presented scheme supports statically-linked binaries (and kernels)

Yet, libraries must be supported:
FineIBT DSOs should not break non-FineIBT DSOs

FineIBT Cross-DSO support

Presented scheme supports statically-linked binaries (and kernels)

Yet, libraries must be supported:
FineIBT DSOs should not break non-FineIBT DSOs

Similar to CET's support in the X86-64 ABI

FineIBT Cross-DSO support

Method 1 — FineIBT Global Flag

FineIBT-enabled DSOs have an ELF bit that flags the feature

FineIBT Cross-DSO support

Method 1 — FineIBT Global Flag

FineIBT-enabled DSOs have an ELF bit that flags the feature

Loader checks all DSOs and sets/unsets global FineIBT flag

FineIBT Cross-DSO support

Method 1 — FineIBT Global Flag

FineIBT-enabled DSOs have an ELF bit that flags the feature

Loader checks all DSOs and sets/unsets global FineIBT flag

All loaded DSOs must be FineIBT-enabled for the policy to be enforced

FineIBT Assembly Code

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…
call <bar_entry>
```

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
testb 0x11, fs:0x48
jne bar_entry
hlt
bar_entry:
…
```

FineIBT Assembly Code

FLAGS

0x01 IBT

0x10 FineIBT

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
testb 0x11, fs:0x48
jne bar_entry
hlt
bar_entry:
…
```

FineIBT Assembly Code

FLAGS

0x01 IBT

0x10 FineIBT

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
testb 0x11, fs:0x48
jne bar_entry
hlt
bar_entry:
…
```

FLAGS SET BY GLIBC

FineIBT Cross-DSO support

Method 1 — FineIBT Global Flag

Special PLT with one 32-byte slot and one 16-byte slot

FineIBT Cross-DSO support

Method 1 — FineIBT Global Flag

Special PLT with one 32-byte slot and one 16-byte slot

First slot is reached by indirect calls, checks hash and jumps to target

Second slot is reached by direct calls, sets hash and jumps to target

FineIBT Cross-DSO support

Method 1 — FineIBT Global Flag

Special PLT with one 32-byte slot and one 16-byte slot

First slot is reached by indirect calls, checks hash and jumps to target

Second slot is reached by direct calls, sets hash and jumps to target

Requires early binding (-z,now) to prevent symbol resolution detours

FineIBT PLT Assembly Code

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…
call <bar@PLT>
```

```
<PLT_ENTRY>:
endbr
cmp 0xdeadbeef, r11
je plt_entry+38
testb 0x11, fs:0x48
jne plt_entry+38
hlt
nops…

<PLT_ENTRY+32>:
mov 0xdeadbeef, r11
jmp target
nops…
```

FineIBT PLT Assembly Code

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…
call <bar@PLT>
```

```
<PLT_ENTRY>:
endbr
cmp 0xdeadbeef, r11
je plt_entry+38
testb 0x11, fs:0x48
jne plt_entry+38
hlt
nops…

<PLT_ENTRY+32>:
mov 0xdeadbeef, r11
jmp target
nops…
```

HASH CHECK

FLAGS CHECK

HASH SET

FineIBT PLT

Since early-binding is a requirement

FineIBT PLT

Since early-binding is a requirement

Full RELRO is just one step ahead

FineIBT PLT

Since early-binding is a requirement

Full RELRO is just one step ahead

PLT indirect branches no longer need CFI

FineIBT PLT

Since early-binding is a requirement

Full RELRO is just one step ahead

PLT indirect branches no longer need CFI

RELRO + NO-TRACK CET prefixes can be combined to optimize the PLT

FineIBT PLT

Since early-binding is a requirement

Full RELRO is just one step ahead

PLT indirect branches no longer need CFI

RELRO + NO-TRACK CET prefixes can be combined to optimize the PLT

This is yet to be explored

FineIBT Cross-DSO support

Method 2 — Consulting Shadow Stack (under development)

Leverages CET write-protected shadow stack

FineIBT Cross-DSO support

Method 2 — Consulting Shadow Stack (under development)

Leverages CET write-protected shadow stack

`call` and `ret` instructions implicitly write to the shadow stack

Retrieve caller from shadow stack using `rdssp`

FineIBT Cross-DSO support

Method 2 — Consulting Shadow Stack (under development)

Leverages CET write-protected shadow stack

`call` and `ret` instructions implicitly write to the shadow stack

Retrieve caller from shadow stack using `rdssp`

Enforce policy if caller is within a FineIBT-enabled DSO

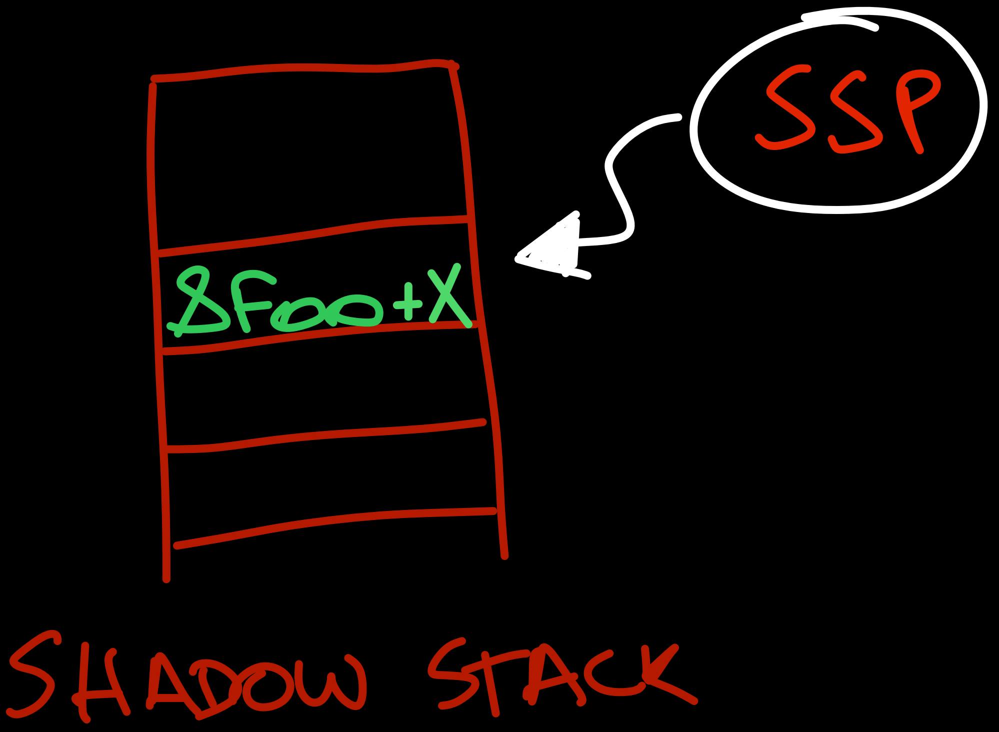
FineIBT Assembly Code

Same DSO enforcement only

```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
rdssp r11
mov *r11, r11
sub <DSO_start>, r11
cmp r11, <DSO_length>
jb bar_entry
hlt
bar_entry:
…
```

FineIBT Assembly Code

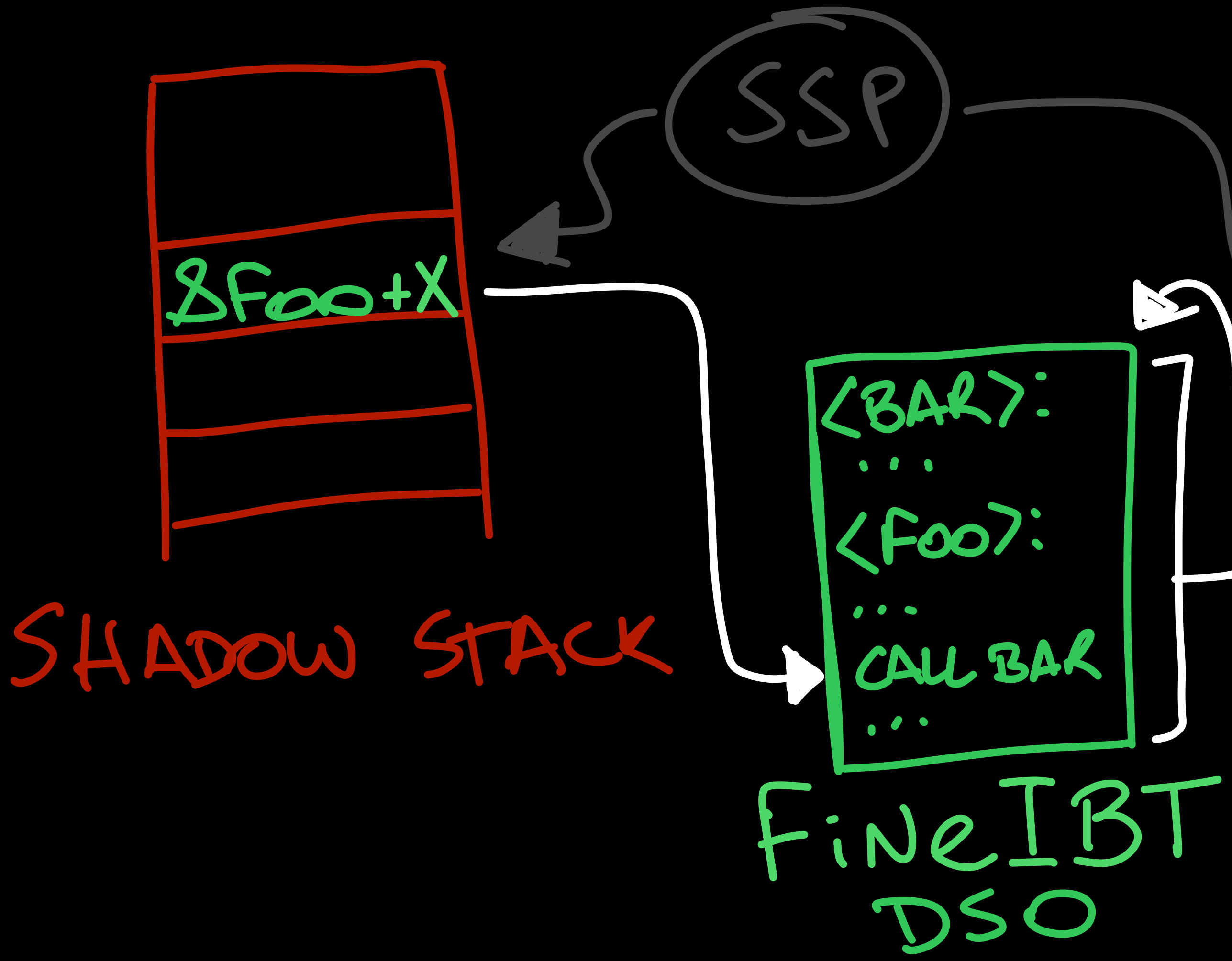
Same DSO enforcement only




```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
rdssp r11
mov *r11, r11
sub <DSO_start>, r11
cmp r11, <DSO_length>
jb bar_entry
hlt
bar_entry:
…
```

FineIBT Assembly Code

Same DSO enforcement only




```
<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
rdssp r11
mov *r11, r11
sub <DSO_start>, r11
cmp r11, <DSO_length>
jb bar_entry
hlt
bar_entry:
…
```

FineIBT Cross-DSO support

Method 2 — Consulting the Shadow Stack (under development)

As described, enforces intra-DSO policy

FineIBT Cross-DSO support

Method 2 — Consulting the Shadow Stack (under development)

As described, enforces intra-DSO policy

Can be extended to support multiple DSOs with heterogeneous granularity

FineIBT Cross-DSO support

Method 2 — Consulting the Shadow Stack (under development)

As described, enforces intra-DSO policy

Can be extended to support multiple DSOs with heterogeneous granularity

Probably has more overhead than the global bit scheme

FineIBT Cross-DSO support

Method 2 — Consulting the Shadow Stack (under development)

As described, enforces intra-DSO policy

Can be extended to support multiple DSOs with heterogeneous granularity

Probably has more overhead than the global bit scheme

Overheads only in calls from coarse-grained into fine-grained DSOs

FineIBT

Other perks

Intel(R) 64 IA-32 Architecture SDM, Session 18.3.8:

"When the CET tracker is in the WAIT_FOR_ENBRANCH state, instruction execution will be limited or blocked, even speculatively, if the next instruction is not an ENDBRANCH."

Confines speculation to coarse CFG

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…

<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT

Other perks

Low latency speculation window:
depends only on quick r11 / imm ops

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…

<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT

Other perks

Low latency speculation window:
depends only on quick r11 / imm ops

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…

<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT

Other perks

Low latency speculation window:
depends only on quick r11 / imm ops

We couldn't use this branch as a
transient execution attack gadget

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…

<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT

Other perks

Low latency speculation window: depends only on quick r11 / imm ops

We couldn’t use this branch as a transient execution attack gadget

Confines speculation to refined CFG

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…

<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT

Other perks

Low latency speculation window: depends only on quick r11 / imm ops

We couldn’t use this branch as a transient execution attack gadget

Confines speculation to refined CFG

Statically-linked scenario evaluation only (Shoutout to Ke Sun for this analysis)

```
<foo>:
…
mov 0xdeadbeef, r11
call *rax
…

<bar>:
endbr
xor 0xdeadbeef, r11
je bar_entry
hlt
bar_entry:
…
```

FineIBT

Other perks

Does not depend on LTO

Compiler embeds hash information into objects for creating PLTs

FineIBT Implementation

Basic support on LLVM/LLD 12.0

GLIBC support on top of GRTE branch

MUSL support on 1.2.0 and (non-ABI compliant)

FineIBT Implementation

Source code

FineIBT LLVM:

https://github.com/intel/fineibt_llvm

FineIBT GLIBC:

https://github.com/intel/fineibt_glibc

FineIBT Testing:

https://github.com/intel/fineibt_testing

Performance

Test Sets

Custom benchmark

Designed to take the worst out of CFI instrumentation

DUMMY loop that performs indirect calls to an empty function
FIBONACCI sequence calculation with an indirect recursive call
BUBBLE-sort with an indirect swap function

Test Sets

Custom benchmark

Designed to take the worst out of CFI instrumentation

DUMMY loop that performs indirect calls to an empty function
FIBONACCI sequence calculation with an indirect recursive call
BUBBLE-sort with an indirect swap function

Two versions of each application:

1: Global function pointer whose relocation is resolved direct into target function
2: Local function pointer which will go through PLT entries

Test Sets

Custom benchmark

Designed to take the worst out of CFI instrumentation

DUMMY loop that performs indirect calls to an empty function
FIBONACCI sequence calculation with an indirect recursive call
BUBBLE-sort with an indirect swap function

Two versions of each application:

1: Global function pointer whose relocation is resolved direct into target function
2: Local function pointer which will go through PLT entries

Numbers used are an average of 10 runs computed by perf

Test Sets

SPEC CPU 2017 (nc)

Type-casts within SPEC would cause unintended violations:

FineIBT using the same tag for every prototype

Functions added to the ignore-list in Clang CFI setups only

Test Sets

SPEC CPU 2017 (nc)

Type-casts within SPEC would cause unintended violations:

FineIBT using the same tag for every prototype

Functions added to the ignore-list in Clang CFI setups only

600.PERLBENCH: Suggested by Clang CFI documentation

Ignore-list: Perl_leave_scope, qsort, Perl_sv_setsv_flags, Perl_mg_get

Test Sets

SPEC CPU 2017 (nc)

Type-casts within SPEC would cause unintended violations:

FineIBT using the same tag for every prototype

Functions added to the ignore-list in Clang CFI setups only

600.PERLBENCH: Suggested by Clang CFI documentation

Ignore-list: Perl_leave_scope, qsort, Perl_sv_setsv_flags, Perl_mg_get

625.X264: Picked at random

Ignore-list: spec_qsort, med3

Test Sets

SPEC CPU 2017 (nc)

Type-casts within SPEC would cause unintended violations:

FineIBT using the same tag for every prototype

Functions added to the ignore-list in Clang CFI setups only

600.PERLBENCH: Suggested by Clang CFI documentation

Ignore-list: Perl_leave_scope, qsort, Perl_sv_setsv_flags, Perl_mg_get

625.X264: Picked at random

Ignore-list: spec_qsort, med3

Numbers picked by the benchmark out of 3 runs

Test Sets

More details

11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60GHz / 32GB RAM
Linux Fedora 33 5.10.11-200.1.cet.fc33.x86_64
CET support ON. Turbo boost, ASLR and SMT off

Setup-equivalent MUSL, same compiler, same arguments (except for CFI scheme)

FineIBT global flag check was hard-coded as nops, ensuring policy is always enforced

Compilation arguments in backup-slides

Test Sets

Setups

NO CFI: No CFI instrumentation

COARSE: Native IBT only

FINE: Native IBT + FineIBT compiler instrumentation

CLANG CFI: Clang CFI

CLANG CFI NC: Clang CFI without Cross-DSO support

Custom Benchmark

Performance overheads using NO CFI instrumentation as baseline

SETUP	COARSE	FINE	CLANG CFI
DUMMY	-0.85%	1.18%	57.42%
FIBO	1.8%	13.77%	32.78%
BSORT	0.73%	1.78%	5.12%

Custom Benchmark

Performance overheads using NO CFI instrumentation as baseline

Calls forced through PLT entries			
SETUP	COARSE	FINE	CLANG CFI
DUMMY	-15.8%	0.03%	13.99%
FIBO	8.12%	12.83%	17.15%
BSORT	13.27%	15.65%	18.54%

SPEC CPU 2017 (nc)

Performance overheads using no CFI instrumentation as baseline

	COARSE	FINE	CLANG CFI
600.PERLBENCH	0%	1.66%	3.56%
625.X264	0.77%	1.54%	1.54%

Space Overheads

Space overheads using no CFI instrumentation as baseline

SETUP	COARSE	FINE	CLANG CFI	CLANG CFI NC
DUMMY	7.27%	13.74%	5672.86%	1.96%
FIBO	6.94%	12.7%	6439.79%	1.83%
BSORT	7.06%	12.61%	6427.85%	1.7%
PERLBENCH	0.21%	0.51%	11.83%	0.82%
X264	0.27%	0.45%	19.43%	0.9%

Conclusions

Hypotheses

Can we enhance CET/IBT in a way to make it fine-grained?

How much of the hardware-related perks would it retain?

Hypotheses

Can we enhance CET/IBT in a way to make it fine-grained?

Yes

How much of the hardware-related perks would it retain?

Good performance

Improves transient execution mitigation

Reasonable space overheads

Next

TODO List

Security Validation
What are the weak spots and possible bypasses?
Can we fix it?

Benchmarking: it *IS* Rocket Science
How can we benchmark CFI instrumentation properly?
How can we measure the security guarantees provided?

TODO List

Integrate cross-DSO support into compiler runtime libraries
Can we integrate cross-DSO support into compiler-rt libs?
If yes, can we upstream FineIBT into LLVM?

Improve cross-DSO support methods
How to properly evaluate the current designs?
Can we think of better designs?

TODO List

Enable kernel support for FineIBT

Can we help with IBT support merge?

What is the best way to handle assembly compatibility?

Brainstorm about C++ vtables

Can FineIBT be useful in the vtables context?

Is there any benefit in doing so?

How would it handle polymorphism?

TODO List

Benchmarking shows a heavy toll on FineIBT PLTs

How to optimize the PLT with and without RELRO?

Can we safely make it non-dependent on early-binding?

Important People

Thank you!

Prof. Vasilis Kemerlis (Brown University)
Alex Gaidis (Brown University)

Michael LeMay (Intel)
HJ Lu (Intel)
Ke Sun (Intel)
Henrique Kawakami (Intel)
Alyssa Milburn (Intel)

Jared Candelaria (Former Intel)
Vedvyas Shanbhogue (Former Intel)

THANKS!

Hardware-Assisted Fine-Grained Control-Flow Integrity: Adding Lasers to Intel's CET/IBT

Joao Moreira
<joao@overdrivepizza.com>
<joao.moreira@intel.com>
@lvwr

LINUX SECURITY SUMMIT

intel®

Compilation Arguments

MUSL

NOCFI

-O2 -flto -Wl,-z,relro,-z,now —rtlib=compiler-rt -fuse-ld=lld

COARSE

-O2 -flto -fcf-protection=coarse -Wl,-z,relro,-z,now,-z,force-ibt —rtlib=compiler-rt -fuse-ld=lld

FINE

-O2 -flto -fcf-protection=fine -Wl,-z,relro,-z,now,-z,force-ibt,-z,force-fine-ibt -fuse-ld=lld —rtlib=compiler-rt"

CLANG CFI

-O2 -flto -fsanitize=cfi-icall -fvisibility=default —rtlib=compiler -fsanitize-blacklist=$ROOT/extras/cfi_blacklist.txt -fuse-ld=lld -fno-sanitize-cfi-canonical-jump-tables -Wl,-z,relro,-z,now

Compilation Arguments

Custom benchmark

NOCFI

-O0 -lfto -Wl,-z,relro,-z,now,-dynamic-linker=$ROOT/musl_nocfi/lib/ld-musl-x86_64.so.1 -fuse-ld=lld —rtlib=compiler-rt

COARSE

-O0 -lfto -fcf-protection=branch -Wl,-z,relro,-z,now,-z,force-ibt,-dynamic-linker=$ROOT/musl_coarse/lib/ld-musl-x86_64.so.1 -fuse-ld=lld —rtlib=compiler-rt

FINE

-O0 -lfto -fcf-protection=fine -Wl,-z,relro,-z,now,-z,force-ibt,-z,force-fine-ibt,-dynamic-linker=$ROOT/musl_fine/lib/ld-musl-x86_64.so.1 -fuse-ld=lld —rtlib=compiler-rt

Compilation Arguments

Custom benchmark

CLANG CFI CROSS DSO

-O0 -flto -fsanitize=cfi-icall -fsanitize-cfi-cross-dso -fvisibility=default —rtlib=compiler-rt -fsanitize-blacklist=$ROOT/extras/cfi_blacklist.txt -fno-sanitize-cfi-canonical-jump-tables -Wl,-z,relro,-z,now,-dynamic-linker=$ROOT/install/musl_llvm/lib/ld-musl-x86_64.so.1 -fuse-ld=lld

CLANG CFI

-O0 -flto -fsanitize=cfi-icall -fvisibility=default —rtlib=compiler-rt -fsanitize-blacklist=$ROOT/extras/cfi_blacklist.txt -fno-sanitize-cfi-canonical-jump-tables -Wl,-z,relro,-z,now,-dynamic-linker=$ROOT/install/musl_llvm/lib/ld-musl-x86_64.so.1 -fuse-ld=lld

References

FineIBT Technical Reference:
https://openwall.com/lists/kernel-hardening/2021/02/11/1

Intel CET:
Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. Shanbhogue et al. 2019.
https://dl.acm.org/doi/10.1145/3337167.3337175

References

Clang CFI:
clang.llvm.org/docs/ControlFlowIntegrityDesign.html

PaX/grsecurity RAP:
https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

Microsoft XFG:
query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE37dMC

References

Qualys Security Advisory:
Baron Samedit: Heap-Based buffer overflow in Sudo (CVE-2021-3156)
https://qualys.com/2021/01/26/cve-2021-3156/baron-samedit-heap-based-overflow-sudo.txt

GLIBC / GRTE branch:
https://sourceware.org/git/?p=glibc.git;a=shortlog;h=refs/heads/google/grte/v6-2.29/master