# Control-Flow Integrity Kernel Support


Joao Moreira <joao@overdrivepizza.com>
Mark Rutland <mark.rutland@arm.com>
Peter Zijlstra <peterz@infradead.org>
Sami Tolvanen <samitolvanen@google.com>

## 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.

Memory (un)safety bugs enable code pointer corruption

**Control-Flow hijacking:** Arbitrary code execution

W^X, ASLR

Code-reuse, memory disclosure, ret2usr

Strong Address Space Isolation

ROP/JOP/COP/Whatever-OP

WOP reuses (executable) kernel code

**GADGETS**

instruction sequences ended with an indirect-branch

When arbitrarily chained, achieve meaningful (malicious) computation

What if we confine indirect branches to safe,

previously-computed locations?

# Control-Flow Integrity

## Coarse-grained CFI

Targets are either valid or invalid

All indirect branches can go to all valid targets

## Fine-grained CFI

Targets are clustered in sub-groups

An indirect branch goes to specific sub-group

Coarse-grained CFI is known to be **insufficient**

Forward-edge fine-grained CFI requires heuristics to compute target clusters

Pointer and function prototypes must match (Abadi et al)

Already used by other existing CFI schemes out there:

PaX/grsecurity RAP, Clang CFI, Microsoft XFG

# Linux kernel CFI right now…

Existing support:

Intel® Indirect Branch Tracking (IBT): coarse-grained hardware-based CFI

ARM BTI: coarse-grained hardware-based CFI

Clang CFI on ARM: fine-grained software-based CFI; to-be-replaced by kCFI

On the forge:

kCFI: fine-grained full-software CFI

FineIBT:  fine-grained software/IBT-hybrid CFI

# KCFI

A kernel-friendly forward-edge CFI scheme available in the upcoming Clang 16 release. Unlike Clang's other CFI schemes, doesn't require LTO and won't mess up function pointers.

The compiler emits type a hash before each address-taken function, and a check before indirect calls to ensure the target function has the expected type. Always traps if there's a mismatch.

# KCFI cont’d

Assembly functions indirectly called from C code need manual type annotations.

The compiler emits __kcfi_typeid_<functionname> entries to the symbol table to make manual annotations easier.

include/linux/cfi_types.h:

SYM_TYPED_FUNC_START(name)

# KCFI instrumentation

## x86_64

```
<foo>:
…
mov -$hash, %r10d
add -0x4(%rN), %r10d
jz 1f
# loc in .kcfi_traps
ud2
1:
call *%rN/indirect_thunk
…
```

```
<__cfi_bar>:
(padding nops)
mov $hash, eax
<bar>:
…
```

## arm64

```
<foo>:
…
ldur w16, [xN, #-4]
movz w17, #hash
movk w17, #hash,
        lsl #16
cmp w16, w17
b.eq 1f
brk #0x8228   # imm encodes registers
1:
blr xN
…
```

```
.word hash
<bar>:
…
```

# ARM Branch Target Identification (BTI)

Mandatory part of ARMv8.5-A, AArch64-only

AKA “FEAT_BTI” in the ARM Architecture Reference Manual

New BTI instructions (behave as NOPs on existing HW)

Hardware enforces indirect branches land on a compatible BTI

BR -> BTI {J,JC}     BLR -> BTI {C,JC}

New guarded page (GP) page table attribute to enable enforcement

# ARM Branch Target Identification (BTI)

Compiler inserts BTI C or BTI JC in functions possibly called indirectly

… or relies upon AUTIASP being BTI C compatible

Kernel sets GP on kernel code pages

Disabled for GCC (due to GCC bug 106671)

Works with Clang >= 12.0.0

# BTI instrumentation

<foo>
…
blr x0
…

<maybe_called_indirectly>
bti c || bti jc || paciasp
…

<only_called_directly>
…

# Intel® Indirect Branch Tracking

Part of the Intel® Control-Enforcement Technology (CET) extension

New ENDBR instructions which behave as NOPS

Hardware enforces forward indirect branches to land on ENDBRs

Behavior also enforced for speculative execution

NOTRACK prefixes enable relaxing the policy (disabled in Linux)

# Intel® Indirect Branch Tracking

Compiler support emits ENDBRs on prologue of address taken functions

Similarly on JIT/eBPF

OBJTOOL to the rescue

Validates IBT by ensuring that address taken-functions are ENDBR-preceded

Seals (removes) ENDBRs from non-address-taken functions

Also doable through LTO

# IBT instrumentation

```
<foo>
…
call *%rax
…
```

```
<address_taken>
endbr
push…
```

```
<non_address_taken>
push…
```

# FineIBT

IBT does caller-side checks and anchors execution to a large set of valid targets

FineIBT augments callee's prologues with additional prototype checks

Direct calls bypass the prototype checks

# FineIBT

Hot-patched on top of kCFI instrumentation during boot if IBT is available

```
<foo>
…
mov -$hash, %r10d
add -0x4(rN), %r10d
jz 1f
ud2
1:
call *%rN/indirect_thunk
…
```

```
<__cfi_bar>:
(padding nops)
mov $hash, %eax
<bar>
…
```

```
<foo>
…
sub %r11, 16
mov $hash, %r10d
call *%r11
…
```

```
endbr
sub %r10, $hash
jz bar
ud2
<bar>
…
```

FineIBT hash checks don't depend on memory reads

Low latency operations reduce speculation window after the check

Adds security in-depth to IBT speculation hardening

Make it compatible with other mitigations like XOM

Likely to show performance benefits

Prototype Matching relaxation under **FineIBT** is also less disastrous

A relaxed function has no hash check

Can be called from anywhere

Not a big deal if the function is harmless

On **kCFI**, a relaxed indirect call means a call without a hash check

Now, this function pointer can call any function from the address space,

Including critical functions like disable_favorite_security_feature()

Thanks!

Questions?

# Control-Flow Integrity Kernel Support

Joao Moreira <joao@overdrivepizza.com>
Mark Rutland <mark.rutland@arm.com>
Peter Zijlstra <peterz@infradead.org>
Sami Tolvanen <samitolvanen@google.com>