KCFI Comes to GCC

Code-Reuse Defenses for the Linux Kernel

Kees (“Case”) Cook

kees@kernel.org

keescook@google.com

kees@outflux.net

https://hachyderm.io/@kees

https://outflux.net/slides/2026/fossy/gcc-kcfi.pdf

Agenda

- What and why of Control Flow Integrity
- Crash course on KCFI and GCC internals
- Implementation walk-through
- Regression tests and example
- Submission history
- Try it yourself!

What is Control Flow Integrity?

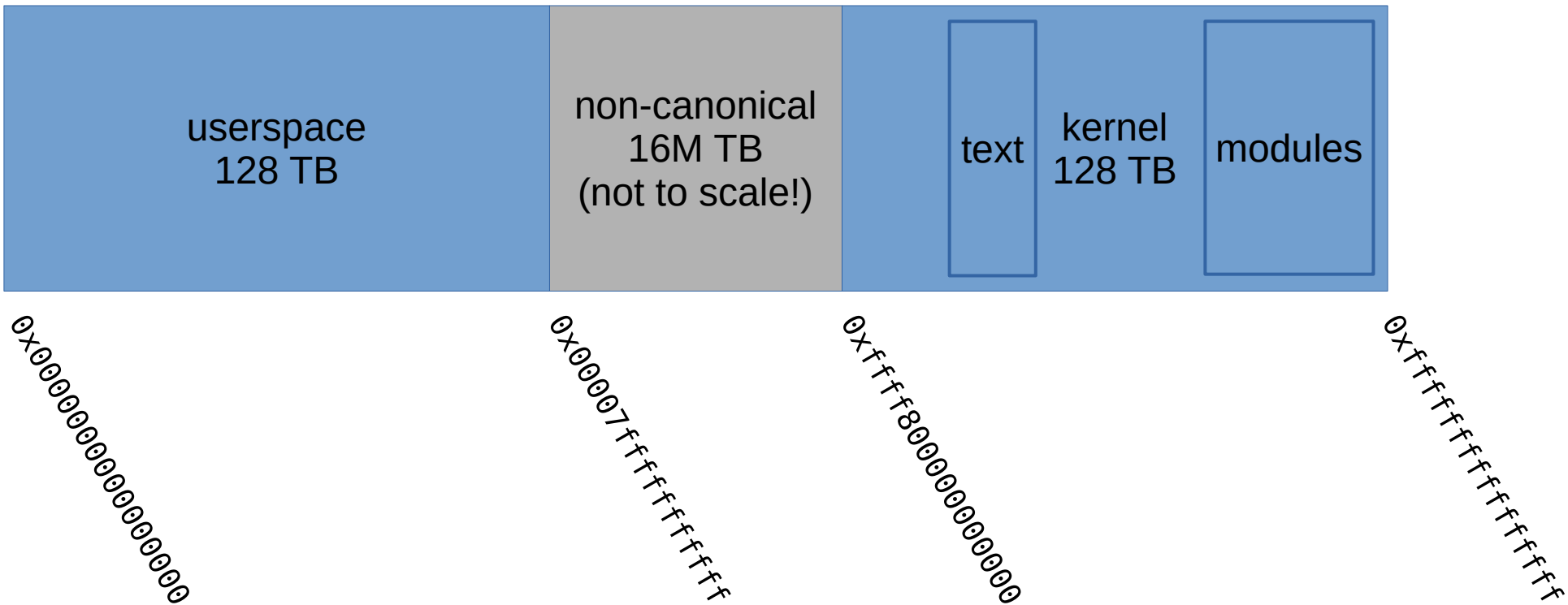
- Why should anyone care about this?
  - Most compromises of the kernel are about gaining execution control, where the initial flaw is some kind of attacker-controlled write to system memory. What can be written to, and how can that be turned into execution control?
- Flaws come in many flavors
  - write only up to a certain amount, only a single zero, only a set of fixed value bytes
  - worst-case is a “write anything anywhere at any time” flaw

Old attack method: write to code!

- Change the kernel code itself, by writing malicious code directly on the kernel! (e.g. ancient rootkits)
- Target must be executable and writable...

What is writable and executable?

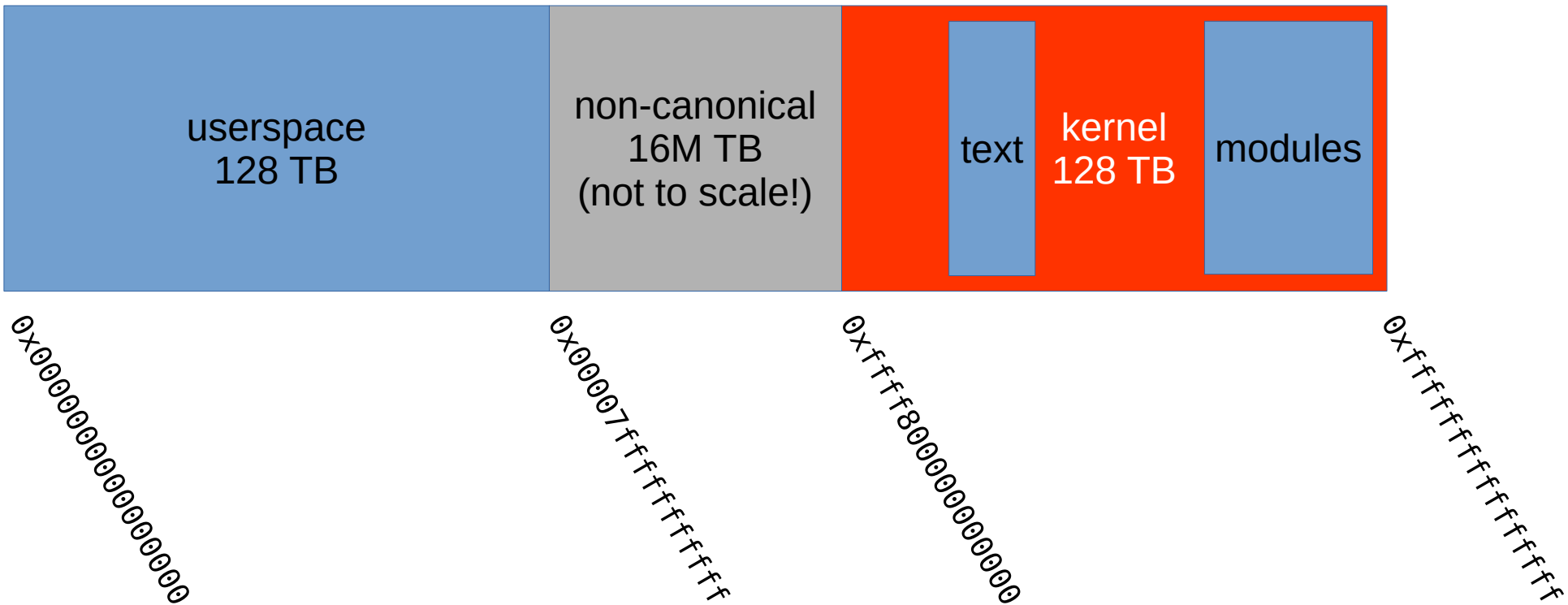
From kernel (ancient, simplified) ...



https://docs.kernel.org/arch/x86/x86_64/mm.html

What is writable and executable?

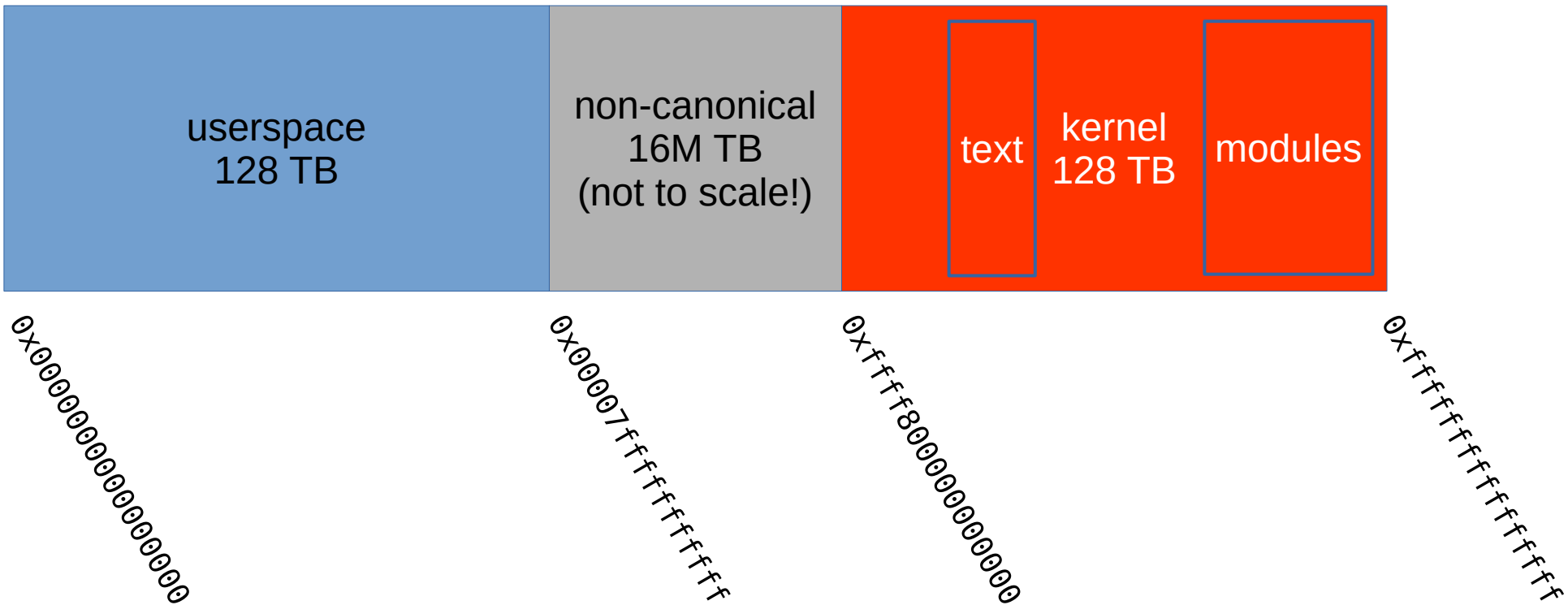
From kernel (NX, simplified) ...



https://docs.kernel.org/arch/x86/x86_64/mm.html

What is writable and executable?

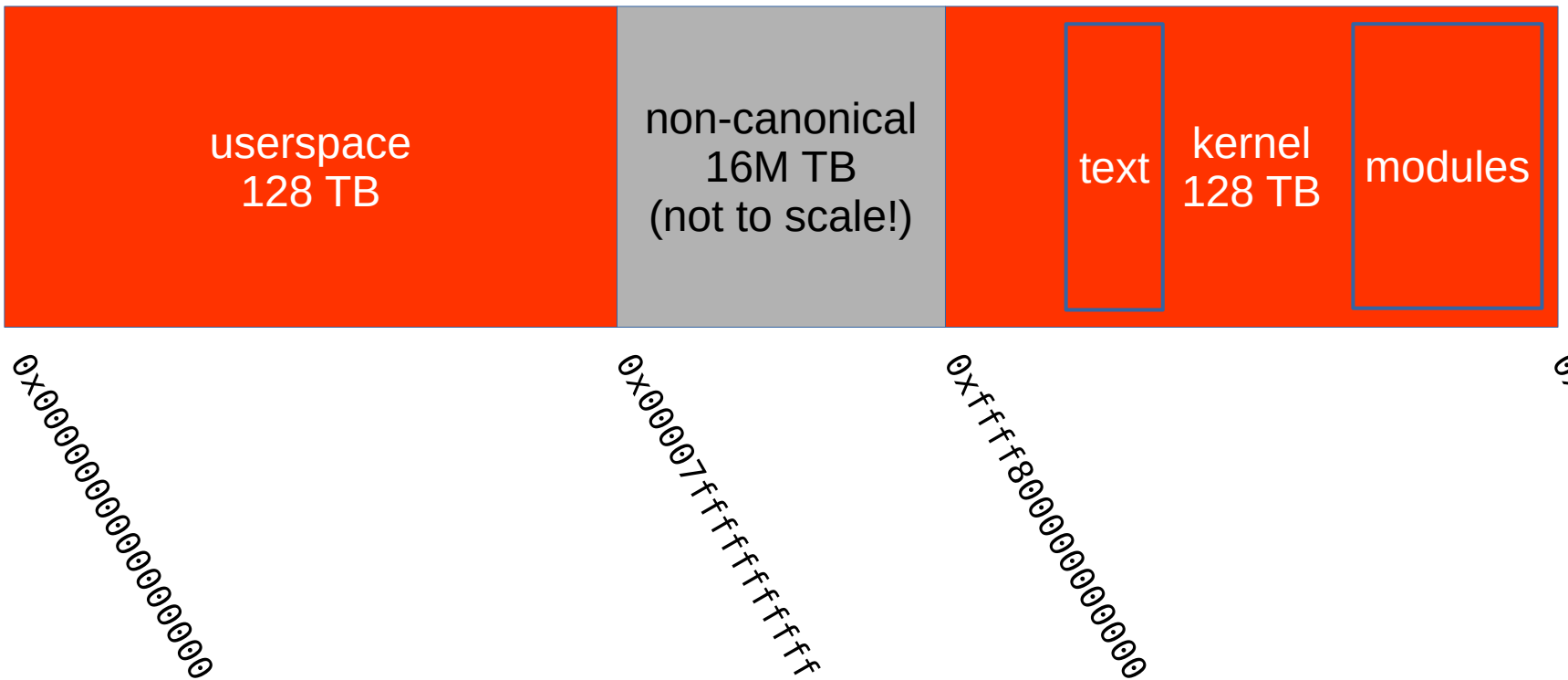
From kernel (NX, RO, simplified) ...



https://docs.kernel.org/arch/x86/x86_64/mm.html

What is writable and executable?

From kernel (NX, RO, SMEP/PXN, simplified) ...



https://docs.kernel.org/arch/x86/x86_64/mm.html

Attack method: reuse existing code!

- Call unexpected kernel code, or with malicious arguments, or in a malicious order, by writing to stored function pointers or arguments.
- Target must be writable and contain function pointers. Attack works by hijacking indirect function calls...

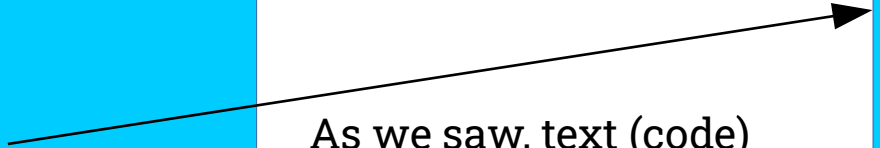
direct function calls

```
lea    0x9000(%rip),%rdi  # <info>
callq  1138 <do_simple>
```

```
int action_launch(int idx)
{
...
    int rc;

...

    rc = do_simple(info);
...
}
```



As we saw, text (code) memory should never be writable (W^X) so calls cannot be redirected by an arbitrary write flaw...

```
int do_simple(struct foo *info)
{
    stuff;
    and;
    things;
...
    return 0;
}
```

indirect function calls

```
typedef int (*func_ptr)(struct foo *);

func_ptr saved_actions[] = {
     do_simple,
     do_fancy,
     ...
};

int action_launch(int idx)
{
    func_ptr action;
    int rc;
...
    action = saved_actions[idx];
...
    rc = action(info);
...
}
```

```
lea    0x2ea6(%rip),%rax  # <saved_actions>
mov    (%rax,%rdi,8),%rax
lea    0x9000(%rip),%rdi  # <info>
callq  *%rax
```

```
int do_simple(struct foo *info)
{
    stuff;
    and;
    things;
...
    return 0;
}
```

indirect calls: “**forward-edge**”

```
typedef int (*func_ptr)(struct foo *);

func_ptr saved_actions[] = {
     do_simple,
     do_fancy,
     ...
};

int action_launch(int idx)
{
    func_ptr action;
    int rc;
...
    action = saved_actions[idx];
...
    rc = action(info);
...
}
```

```
lea    0x2ea6(%rip),%rax  # <saved_actions>
mov    (%rax,%rdi,8),%rax
lea    0x9000(%rip),%rdi  # <info>
callq  *%rax
```


```
int do_simple(struct foo *info)
{
    stuff;
    and;
    things;
...
    return 0;
}
```

indirect calls: “**forward-edge**”

```
typedef int (*func_ptr)(struct foo *);

func_ptr saved_actions[] = {
     do_simple,
     do_fancy,
     ...
};

int action_launch(int idx)
{
     func_ptr action;
     int rc;
...
     action = saved_actions[idx];
...
     rc = action(info);
...
}
```

heap

stack

```
lea    0x2ea6(%rip),%rax  # <saved_actions>
mov    (%rax,%rdi,8),%rax
lea    0x9000(%rip),%rdi  # <info>
callq  *%rax
```

forward edge

```
int do_simple(struct foo *info)
{
     stuff;
     and;
     things;
...
     return 0;
}
```

As we’ll see, the heap and stack are writable, so function calls *can* be redirected by an arbitrary write flaw

function return: “**backward-edge**”

```
typedef int (*func_ptr)(struct foo *);

func_ptr saved_actions[] = {
     do_simple,
     do_fancy,
     ...
};

int action_launch(int idx)
{
    func_ptr action;
    int rc;
...
    action = saved_actions[idx];
...
    rc = action(info);
...
}
```

retq

stack

```
...
return address
action
rc
...
```

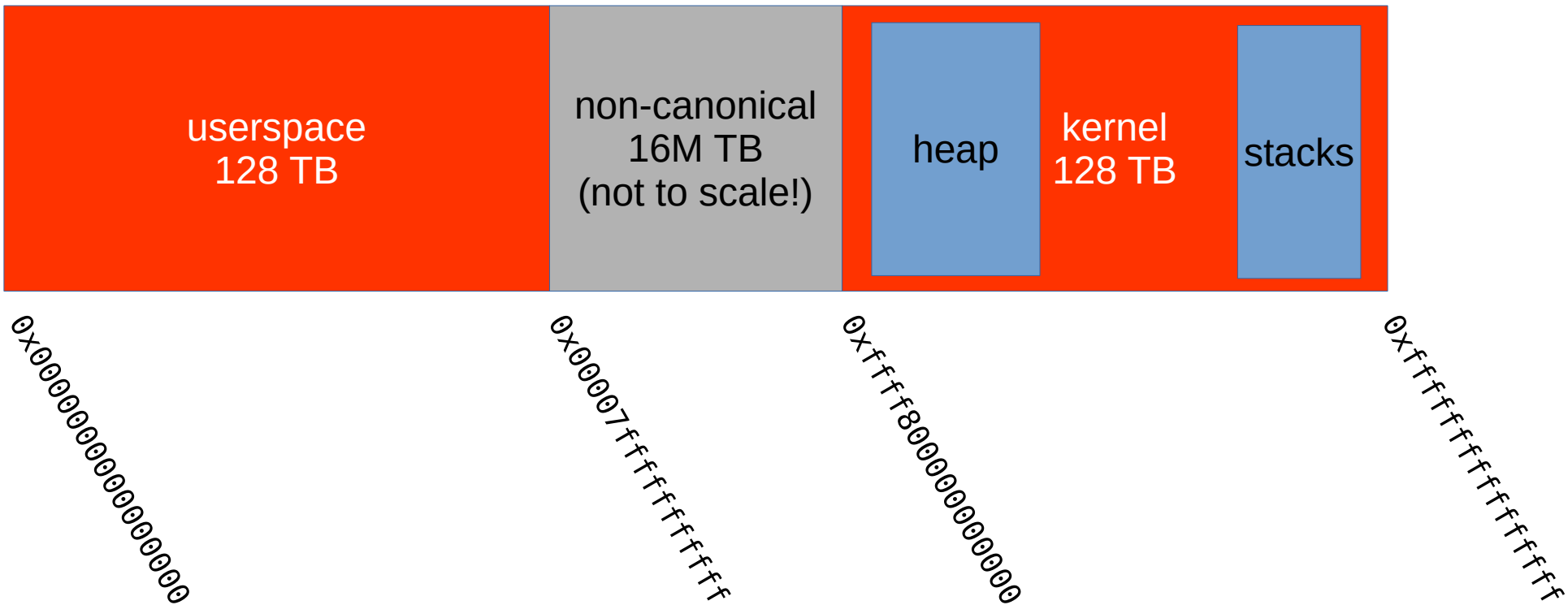
forward edge

backward edge

```
int do_simple(struct foo *info)
{
    stuff;
    and;
    things;
...
    return 0;
}
```

What contains writable func ptrs?

From kernel (SMAP/PAN, simplified) ...



https://docs.kernel.org/arch/x86/x86_64/mm.html

What can attacker call?

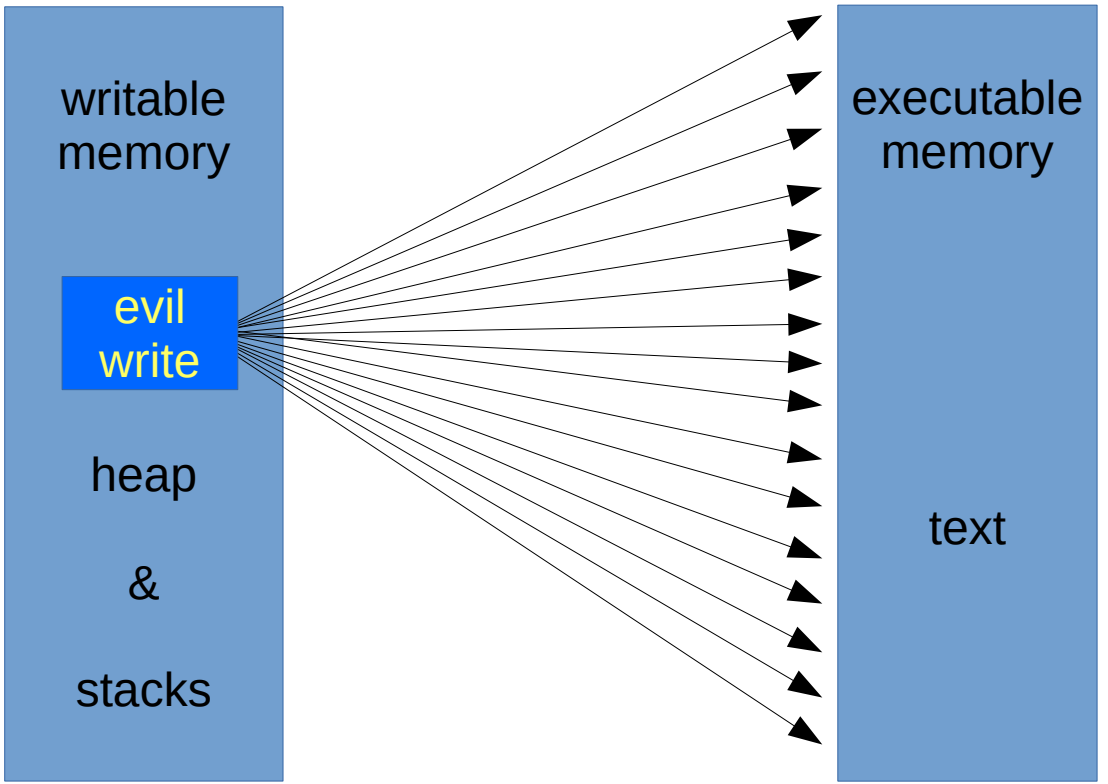
Any executable byte!

forward edge

```
mov     (%rax,%rdi,8),%rax
callq   *%rax
```

backward edge

```
retq
```



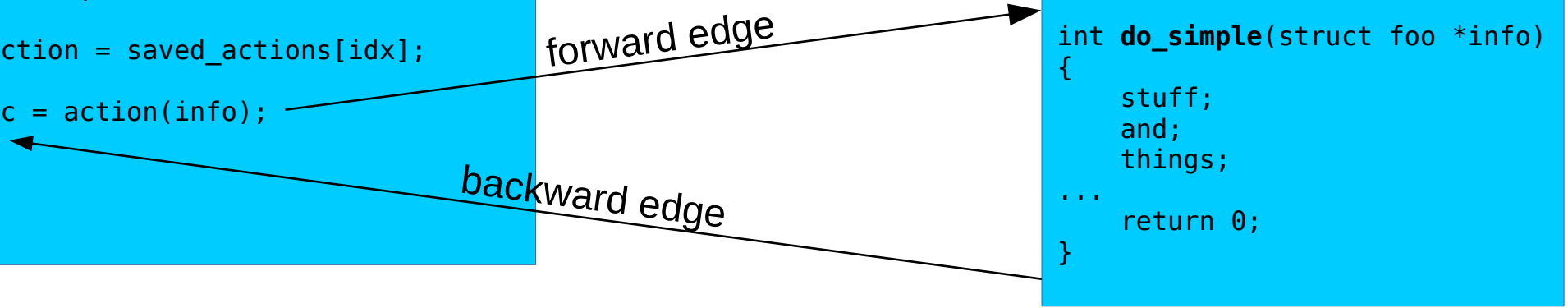
Control Flow Integrity

```
typedef int (*func_ptr)(struct foo *);

func_ptr saved_actions[] = {
     do_simple,
     do_fancy,
     ...
};

int action_launch(int idx)
{
    func_ptr action;
    int rc;
...
    action = saved_actions[idx];
...
    rc = action(info);
...
}
```

Goal of CFI: ensure that each indirect call can only call into an “expected” subset of all kernel functions, and that the return stack pointers are unchanged since we made the call.




```
int do_simple(struct foo *info)
{
    stuff;
    and;
    things;
...
    return 0;
}
```

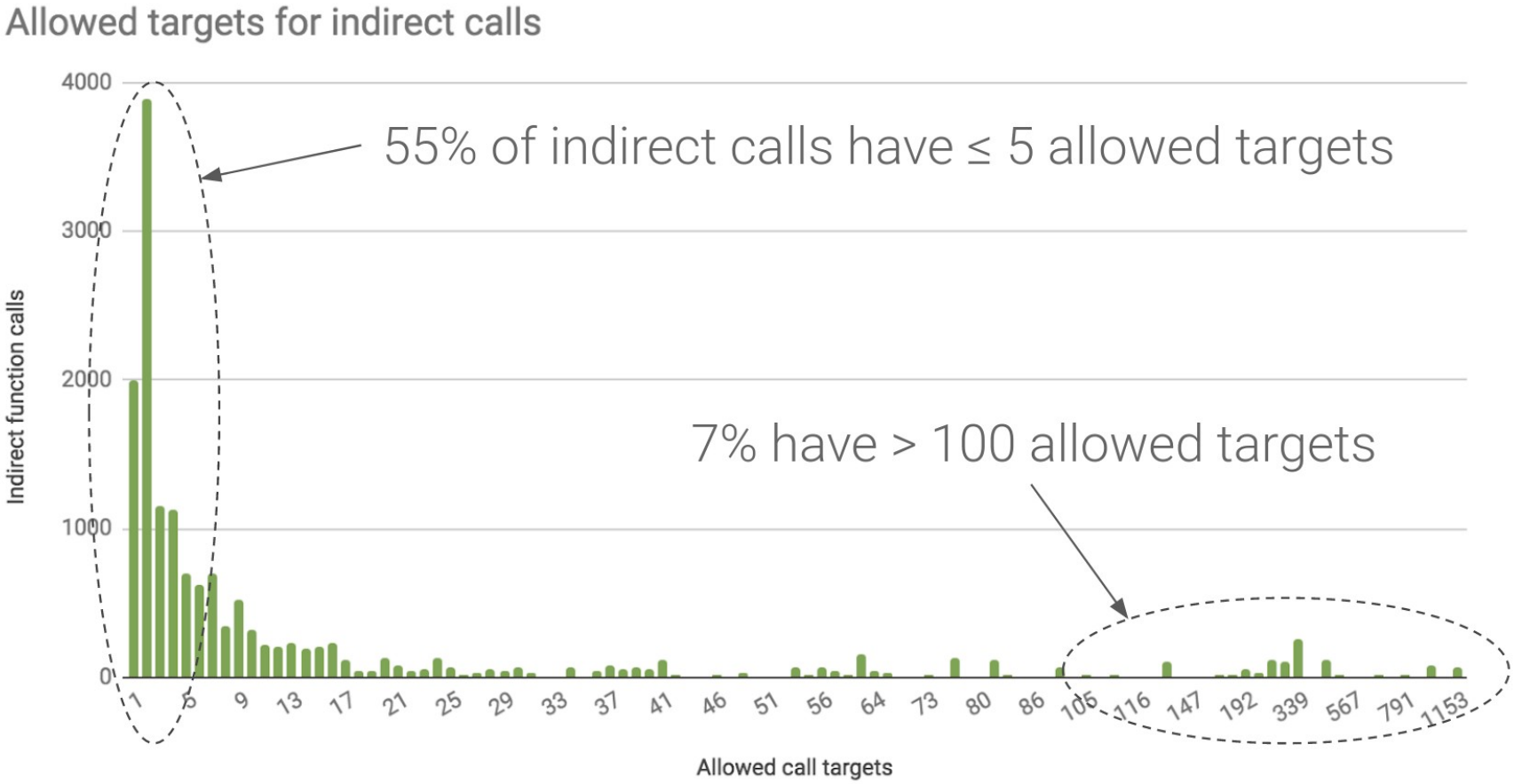
CFI: backward-edge protection

- Maintain integrity of saved return addresses
  - some way to have “trusted” stack of actual return addresses (e.g. dedicated register for a separate stack for returns: “Shadow Call Stack”)
  - honestly, best done in hardware (x86: CET, arm64: Pointer Authentication)
- Out of scope for this presentation …

CFI: forward-edge protection

- validate indirect function pointers at call time
  - some way to indicate similar of functions: current research suggests using function prototype (return type, argument types) as the “uniqueness” key, which tracks existing C type checking. For example:
    - if the same prototype, call site can choose any matching function:
      - `int do_fast_path(unsigned long, struct file *file)`
      - `int do_slow_path(unsigned long, struct file *file)`
    - if different prototypes, calls cannot be mixed:
      - `void foo(unsigned long)` vs `int bar(unsigned long)`
      - `void foo(struct task_struct *task)` vs `void bar(struct file *file)`
  - hardware (e.g. IBT/BTI) has poor granularity (all functions callable)

Function prototype uniqueness



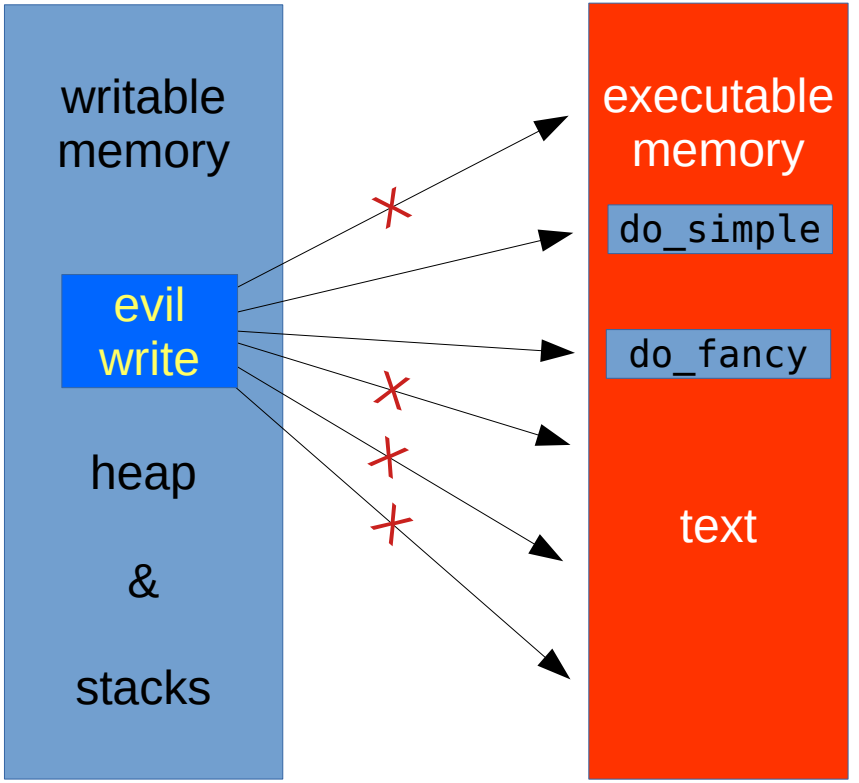
https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html

What can attacker call under CFI?

With forward-edge CFI, call sites encode a single function prototype they are allowed to call. Everything else is rejected.

```
int do_simple(struct foo *info);
int do_fancy(struct foo *info);

int action_launch(int idx)
{
    int (*action)(struct foo *);
...
    rc = action(info);
...
}
```



kCFI forward-edge protection

- Indirectly called functions have a hash value based on their function prototype (“type id”) inserted before function entry.
- Indirect call sites check for this type id before performing the call. (Loading the register is not separated from the check nor the call.)
- Implemented in Clang in 2022, Linux v6.1.

Stock, without kCFI

```
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq


<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq


<mismatched>:
  201890: xor    %eax,%eax
  201892: retq
```

```
<action_launch>:
  2018a0: push   %rax
  2018a1: movslq %edi,%rax
  2018a4: lea    0x200550(%rip),%rcx
  2018ab: mov    %rsi,%rdi
  2018ae: mov    (%rcx,%rax,8),%r11



  2018c0: callq  *%r11
...
```

Stock, without kCFI

```
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq


<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq


<mismatched>:
  201890: xor    %eax,%eax
  201892: retq
```

function pointer (%r11) can point anywhere

```
<action_launch>:
  2018a0: push   %rax
  2018a1: movslq %edi,%rax
  2018a4: lea    0x200550(%rip),%rcx
  2018ab: mov    %rsi,%rdi
  2018ae: mov    (%rcx,%rax,8),%r11



  2018c0: callq  *%r11
...
```

Protected with kCFI

`$CC -fsanitize=kcfi`

```
<__cfi_do_simple>:
  20186b: mov    $0x5d558549,%eax
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq

<__cfi_do_fancy>:
  20186b: mov    $0x5d558549,%eax
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq

<__cfi_mismatch>:
  20188b: mov    $0x36b1c5a6,%eax
<mismatched>:
  201890: xor    %eax,%eax
  201892: retq
```

```
<action_launch>:
  2018a0: push   %rax
  2018a1: movslq %edi,%rax
  2018a4: lea    0x200550(%rip),%rcx
  2018ab: mov    %rsi,%rdi
  2018ae: mov    (%rcx,%rax,8),%r11
  2018b2: mov    $0xa2aa7ab7,%r10d
  2018b8: add    -0x4(%r11),%r10d
  2018bc: je     0x2018c0
  2018be: ud2
  2018c0: callq  *%r11
...
```

Protected with kCFI

`$CC -fsanitize=kcfi`

```
<__cfi_do_simple>:
  20186b: mov    $0x5d558549,%eax
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq

<__cfi_do_fancy>:
  20186b: mov    $0x5d558549,%eax
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq

<__cfi_mismatch>:
  20188b: mov    $0x36b1c5a6,%eax
<mismatched>:
  201890: xor    %eax,%eax
  201892: retq
```

load inverted type id hash into register

```
<action_launch>:
  2018a0: push   %rax
  2018a1: movslq %edi,%rax
  2018a4: lea    0x200550(%rip),%rcx
  2018ab: mov    %rsi,%rdi
  2018ae: mov    (%rcx,%rax,8),%r11
  2018b2: mov    $0xa2aa7ab7,%r10d
  2018b8: add    -0x4(%r11),%r10d
  2018bc: je     0x2018c0
  2018be: ud2
  2018c0: callq  *%r11
...
```

Protected with kCFI

`$CC -fsanitize=kcfi`

```
<__cfi_do_simple>:
  20186b: mov    $0x5d558549,%eax
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq

<__cfi_do_fancy>:
  20186b: mov    $0x5d558549,%eax
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq

<__cfi_mismatch>:
  20188b: mov    $0x36b1c5a6,%eax
<mismatched>:
  201890: xor    %eax,%eax
  201892: retq
```

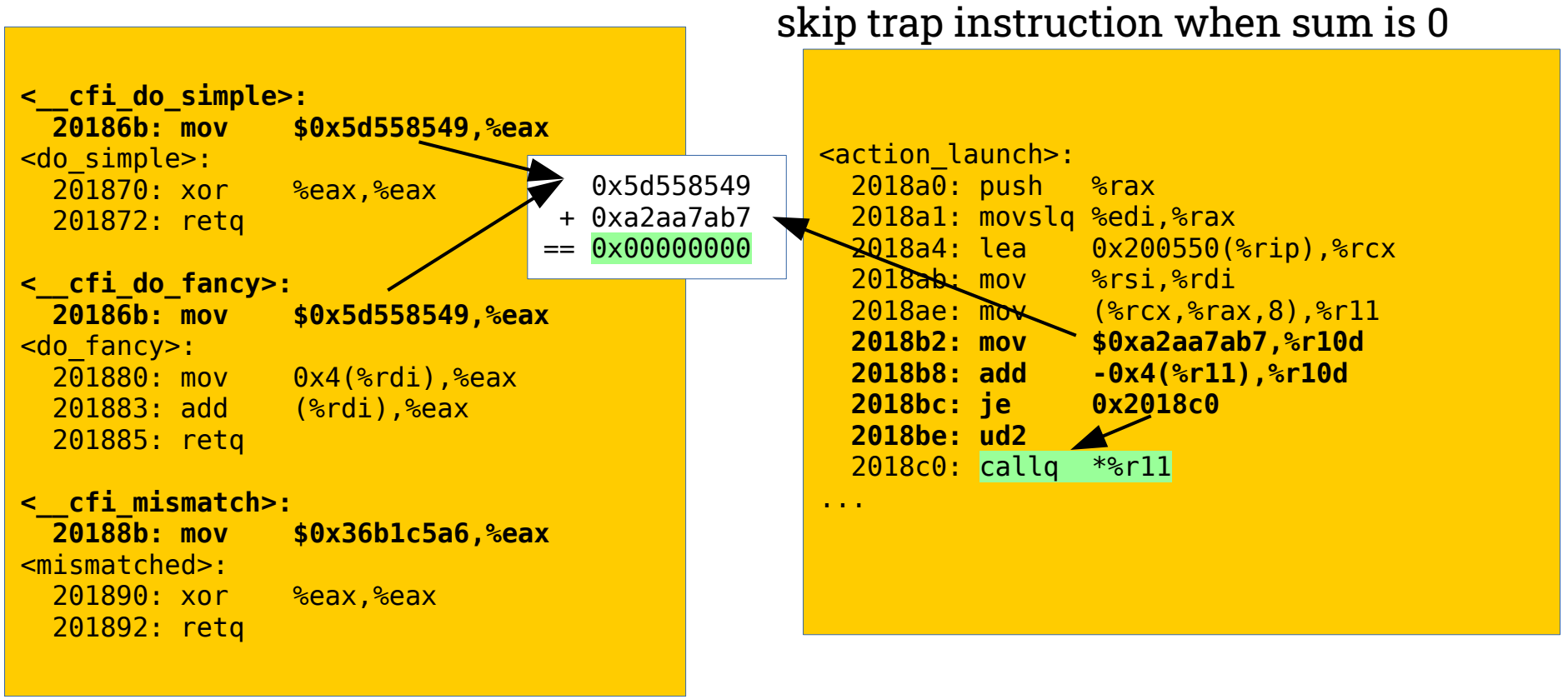
add hash found 4 bytes before the function

```
<action_launch>:
  2018a0: push   %rax
  2018a1: movslq %edi,%rax
  2018a4: lea    0x200550(%rip),%rcx
  2018ab: mov    %rsi,%rdi
  2018ae: mov    (%rcx,%rax,8),%r11
  2018b2: mov    $0xa2aa7ab7,%r10d
  2018b8: add    -0x4(%r11),%r10d
  2018bc: je     0x2018c0
  2018be: ud2
  2018c0: callq  *%r11
...
```

Protected with kCFI

`$CC -fsanitize=kcfi`

skip trap instruction when sum is 0



Protected with kCFI

`$CC -fsanitize=kcfi`

execute trap instruction when sum is **not** 0

```
<__cfi_do_simple>:
  20186b: mov    $0x5d558549,%eax
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq

<__cfi_do_fancy>:
  20186b: mov    $0x5d558549,%eax
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq

<__cfi_mismatch>:
  20188b: mov    $0x36b1c5a6,%eax
<mismatched>:
  201890: xor    %eax,%eax
  201892: retq
```

```
  0x36b1c5a6
+ 0xa2aa7ab7
== 0xd95c405d
```

```
<action_launch>:
  2018a0: push   %rax
  2018a1: movslq %edi,%rax
  2018a4: lea    0x200550(%rip),%rcx
  2018ab: mov    %rsi,%rdi
  2018ae: mov    (%rcx,%rax,8),%r11
  2018b2: mov    $0xa2aa7ab7,%r10d
  2018b8: add    -0x4(%r11),%r10d
  2018bc: je     0x2018c0
  2018be: ud2
  2018c0: callq  *%r11
...
```

What do failures look like?

```
# CONFIG_CFI_PERMISSIVE is not set

Kernel panic - not syncing: CFI failure (target: lkdtm_increment)
...
Call Trace:
 __cfi_check+0x4ec77/0x50780
...
 do_syscall_64+0x72/0xa0
 entry_SYSCALL_64_after_hwframe+0x49/0xbe


CONFIG_CFI_PERMISSIVE=y

CFI failure (target: lkdtm_increment+0x0/0x10):
WARNING: CPU: 3 PID: 2806 at kernel/cfi.c:29 __cfi_check_fail+0x38/0x40
...
Call Trace:
...
```

My crash course on GCC internals

- Front end (e.g. parsing)
  - command line flags, C syntax, builtins
- Middle end (e.g. optimization)
  - GIMPLE, IPA
  - RTL expansion
- Back end (e.g. code generation)
  - machine instruction definitions
  - register selection
  - assembler output

Map of kCFI to GCC internals

- Front end
  - knows attributes and C types
- Middle end
  - turns C types into type ids (hashes)
  - wraps indirect calls with type id
- Back end (architecture specific)
  - turns wrapped calls into inseparable check-and-call instruction set
  - inserts type id function preambles
  - annotates kCFI traps

SIMPLIFIED

kCFI Front end

- Adds `-fsanitize=kcfi` option handling
- Checks for fixed registers, `nocf_check`, and arch feature collisions
- Handles `no_sanitize("kcfi")` attribute
- Tracks inline functions
- Adds `__builtin_linux_abi_kcfi_{hash,name}`

kCFI Type Mangling (for type id)

- Convert function prototype into a string and hash it
  - must match Clang's kCFI mangler (Itanium C++ Mangling)
  - `int foo(char *arg, struct bar *p)` → "`FiPcP3barE`" → `0x48837ff8`
- Changed the hash algo in Clang to match proposed FNV-1a in GCC patches. Clang's xxHash is not in GCC and Clang has deprecated it. (Hash doesn't need to be crypto secure.)
- New builtins for sensible testing and Linux's future type-based allocation isolation work:
  - `__builtin_linux_abi_kcfi_name(type)` returns `const char *`
  - `__builtin_linux_abi_kcfi_hash(type)` returns `unsigned int`

kCFI Middle end

- RTL wrapper for new type of “CALL”
  - call equivalency is tied to type id
  - register liveness testing
- inline handling
  - check Front End’s marking for kCFI being enabled/disabled
- sibcall optimization handling
  - second set of kCFI RTL check-and-sibcall

kCFI Back end (4 architectures)

- some common code
  - function preambles (depends on arch's NOP size)
  - trap annotation section (used by x86_64 and riscv64)
- i386 (x86_64), aarch64, arm, riscv
  - specific machine instruction layout for kCFI check-and-call RTL
  - trap encoding (for aarch64 and arm)

Walk-through: Front-end

```
// an address-taken target
int target(struct foo *p) { return p->val + 1; }
int dispatch(int (*func)(struct foo *), struct foo *p) {
    return func(p);      // the KCFI-checked indirect call
}
```

- `toplev.cc: -fsanitize=kcfi` sets `SANITIZE_KCFI` if `targetm.kcfi.supported()` and in C mode.
- (parser builds the `FUNCTION_TYPE` tree for `int(struct foo *)`)
- `c-attribs.cc:` Check for colliding attributes (`no_sanitize("kcfi")`, `nocf_check`)
- (now we have GIMPLE)

Walk-through: Middle-end

- `tree-inline.cc`: marks any call inlined from a `no_sanitize("kcfi")` function, checked later via `gimple_call_inlined_from_kcfi_nosantize_p`
- GIMPLE → RTL expansion pass, for indirect calls (`REG_P (XEXP (fnaddr, 0))`) which ends up being per-arch:
  - calculate type id (**`int(struct foo *)`** → **`FiP3fooE`** → **`0x963e877a`**):

    ```
    rtx kcfi_type_rtx = kcfi_get_type_id_for_expanding_gimple_call ()
    ```

  - wrap call in KCFI RTL, via `rtl.def`'s `DEF_RTL_EXPR(KCFI, "kcfi", "ee")`:

    ```
    call = gen_rtx_KCFI (VOIDmode, call, kcfi_type_rtx)
    ```

- Now call equivalence is by type id (via `rtx_equal_p`'s new awareness of KCFI)
- `df-scan.cc` and `rtlanal.cc` can examine inside `KCFI` RTL to see registers, etc.

Walk-through: Back-end

- Call-side emission (for **dispatch**'s calls of **int(struct foo *)**):
  - per-arch KCFI `define_insn`'s output template (e.g. `i386.md`) emits:

```
  movl  $0x69c17886, %r10d    ; inverse type id (0 - 0x963e877a)
  addl  -4(%rax), %r10d       ; load target's preamble id
  je    .Lkcfi_callN          ; jump if sum==0
.Lkcfi_trapN:
  ud2                         ; mismatch traps
.Lkcfi_callN:
  call *%rax                  ; call indirect
```

  (Also adds annotation section)

Walk-through: Back-end

- Target-side emission, function preamble (for **target** itself):
  - `assemble_start_function` calls `kcfi_emit_preamble` using `targetm.kcfi.emit_type_id` (e.g. `i386`) and emits:

    ```
    __cfi_target:
        movl  $0x963e877a, %rax
    ```

    (Also deals with Patchable Function Entry alignment NOPs…)
- External address-taken decl emission (for decls of **target**):
  - `assemble_external_real` calls `kcfi_emit_typeid_symbol` and emits:

    ```
    .weak   __kcfi_typeid_target
    .set    __kcfi_typeid_target, $0x963e877a
    ```

Regression testing (half of series!)

- Implementation requirements, e.g.:
  - type id hash correctness
  - function entry padding, type id offsets
  - trap instruction encoding and annotation
  - fixed registers, retpoline
- Bugs encountered during development, e.g.:
  - cross call merging, mov/call adjacency
  - inlining, sibcall (tail call), and LTO correctness

Regression: cross call merging

```
struct G {
    int (*do_apples)(struct apple *, int);
    int (*do_oranges)(struct orange *, int);
    struct apples *apples;
    struct orange *oranges;
} *g;

if (g->do_apples)
    return g->do_apples(g->apples, 7);
if (g->do_oranges)
    return g->do_oranges(g->oranges, 7);
...
```

```
  movl do_apples, %r11      ; load function pointer
  test %r11, %r11
  je   check_oranges        ; if NULL, move on
  movl apples, %rdi         ; %rdi is arg 1
call_via_r11:
  movl 7, %rsi              ; %rsi is arg 2
  call *%r11                ; which type_id? ;)
  retq
check_oranges:
  movl do_oranges, %r11     ; load function ptr
  test %r11, %r11
  je   out                  ; if NULL, move on
  movl oranges, %rdi
  goto call_via_r11
out: ...
```

GCC implementation attempts

- My initial attempts didn't understand how to use an RTL wrapper to create the distinct "KCFI" instruction, so it was plagued with one corner case failure after another. (E.g. see cross call merging above.)
- Once I understood how to create the "KCFI check and call" RTL instruction, everything feel into place.
- Everything since then has mostly been cleanups, removing early work-arounds, and tweaking per-architecture back-end implementations.
- Hard to get review across the entire series since regular GCC development patches don't span the entire stack. Review has tended to focus on a single architecture, or just the type mangling, or just tests, middle end, front end, etc. It seems like there has be sufficient review on every area now. I hope!

History: v1 – v10

v1 (RFC)	2025-08-21	20250821064202.work.893-kees@kernel.org
v2	2025-09-05	20250905001157.it.269-kees@kernel.org
v3	2025-09-13	20250913231256.make.519-kees@kernel.org
v4	2025-09-26	20250926023737.it.616-kees@kernel.org
v5	2025-10-22	20251022181345.do.256-kees@kernel.org
v6	2025-11-04	20251104165430.it.619-kees@kernel.org
v7	2025-11-17	20251117201219.makes.617-kees@kernel.org
v8	2025-11-20	20251120222105.us.687-kees@kernel.org
v9	2025-12-10	20251210022025.harder.803-kees@kernel.org
v10	2026-01-07	20260107200301.better.465-kees@kernel.org
	2026-04-30	GCC 16 release

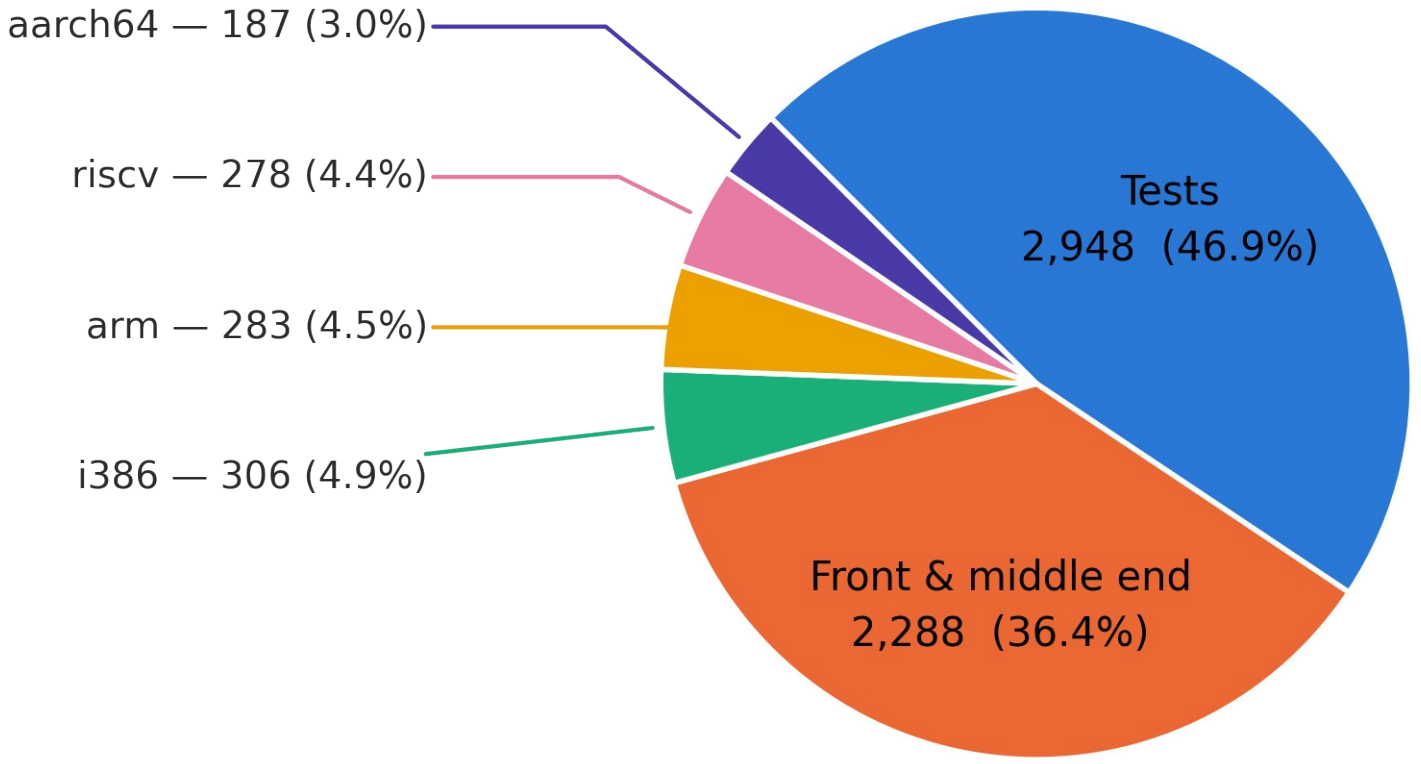
Roughly 1-2 weeks between each version; told at v10 to “wait until GCC 16 releases”

History: v11 – v14

	2026-04-30	GCC 16 release
v11	2026-05-11	20260511194847.faster.180-kees@kernel.org
v12	2026-05-15	20260515161551.stronger.641-kees@kernel.org
v13	2026-06-18	20260618204530.work.910-kees@kernel.org
v14	2026-07-16	20260716005527.it.950-kees@kernel.org

Roughly a month between each version, after the 4 month pause waiting for GCC 16 and the initial rebase.

kCFI v14 diffstat



85 files changed, 6234 insertions(+), 56 deletions(-)

Reviews

- Thank you to all my reviewers!

  Andrea Pinski, Qing Zhao, Martin Uecker, Richard Biener, Jakub Jelinek, Jeff Law, Uroš Bizjak, Ard Biesheuvel, Peter Zijlstra, and Sam James.

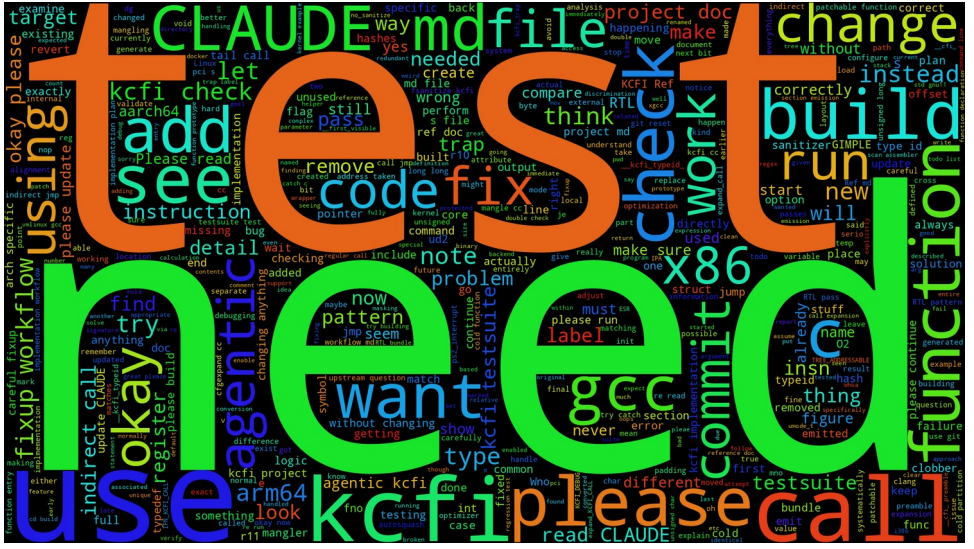
- I hope to get this into distro snapshots for wider testing soon...

Try it yourself!

- Linux kernel has had compiler-agnostic kCFI support since v6.18 (October 2025)
- Build GCC with kCFI series
  - `PI=https://inbox.sourceware.org/gcc-patches`
  - `wget -O v14.mbox $PI/20260716005527.it.950-kees@kernel.org/raw`
  - `git am v14.mbox`
  - `mkdir kcfi && cd kcfi && ../configure [your favorite options]`
  - `make -j$ALL_YOUR_CPUS install`
- Build Linux
  - `export CC=/path/to/your/gcc`
  - `make [your favorite config target]`
  - `./scripts/config -e CONFIG_CFI`
  - `make -s -j$ALL_YOUR_CPUS`

LLM Usage

- semantic grep
- build automation
- crash debugging
- reproducer reduction



- reproducer conversions to regression tests
- patch review (Linux's Sashiko worked well)

How to use `Assisted-by:` tag?

https://gcc.gnu.org/ai-policy.html

> “… any contribution of LLM-generated content must include an “Assisted-by:” tag.”

Most of the KCFI patches have (human-written) code along with associated (LLM-massaged) test cases. This tag doesn’t distinguish them. Should I explain this in the commit log (as Linux kernel policy for LLMs suggests), or split the patch? But it was all also reviewed by LLMs, so … put tag on all patches? I’d love better guidance for following the policy!

Questions/Comments/Thoughts?

Kees (“Case”) Cook
kees@kernel.org
keescook@google.com
kees@outflux.net
https://hachyderm.io/@kees

https://outflux.net/slides/2026/fossy/gcc-kcfi.pdf

https://kspp.github.io/

Bonus Slides

Actually, there’s just one, so … Bonus Slide

Backward-edge protection in hw

- Intel CET: hardware-based read-only shadow call stack
  - Implicit use of otherwise read-only shadow stack during `call` and `ret` instructions
- ARM v8.3a Pointer Authentication (“signed return address”)
  - New instructions: `paciasp` and `autiasp`
  - Clang and gcc: `-msign-return-address`

```
stp     x29, x30, [sp, #-48]!
mov     x29, sp
str     w0, [x29, #28]
...
mov     w0, #0x0
ldp     x29, x30, [sp], #48

ret
```

```
paciasp
stp     x29, x30, [sp, #-48]!
mov     x29, sp
str     w0, [x29, #28]
...
mov     w0, #0x0
ldp     x29, x30, [sp], #48
autiasp
ret
```