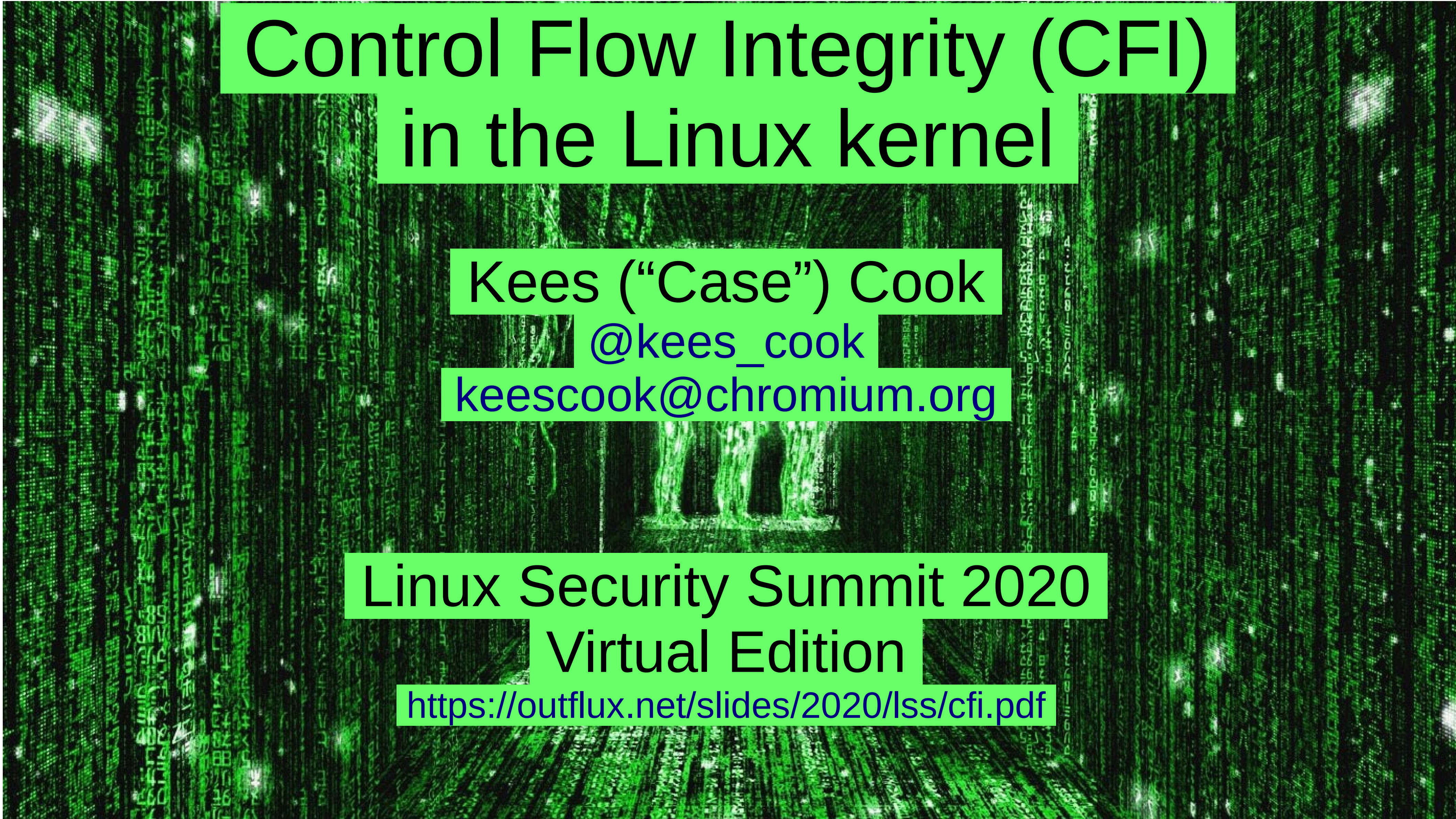


# Control Flow Integrity (CFI) in the Linux kernel

Kees ("Case") Cook
@kees_cook
keescook@chromium.org

Linux Security Summit 2020
Virtual Edition

https://outflux.net/slides/2020/lss/cfi.pdf

# Agenda

- What is kernel Control Flow Integrity (CFI)?
- Clang CFI implementations
- Pixel phones and the Android Ecosystem
- Gotchas
- Upstreaming status
- Do it yourself!

## What is kernel Control Flow Integrity?

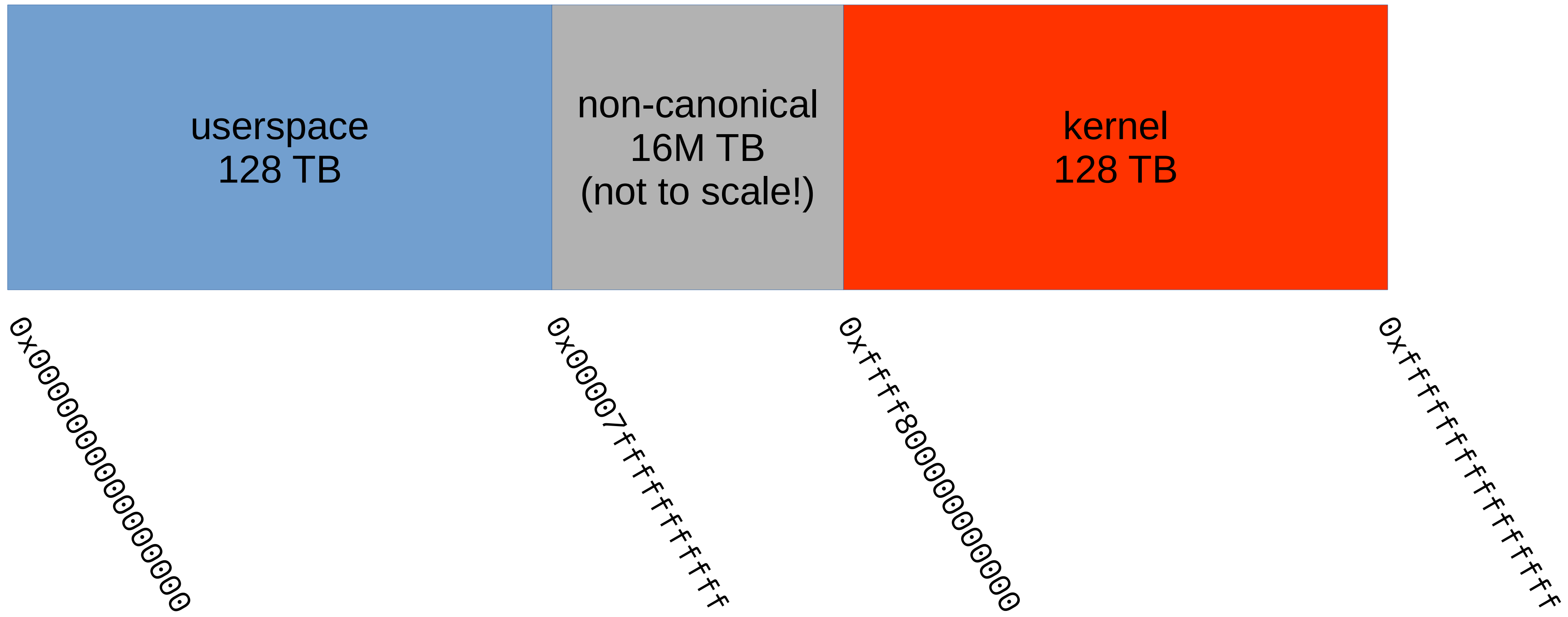
- Why should anyone care about this?
  - Most compromises of the kernel are about gaining execution control, where the initial flaw is some kind of attacker-controlled write to system memory. What can be written to, and how can that be turned into execution control?
- Flaws come in many flavors
  - write 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

## Attack method: write to kernel 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?

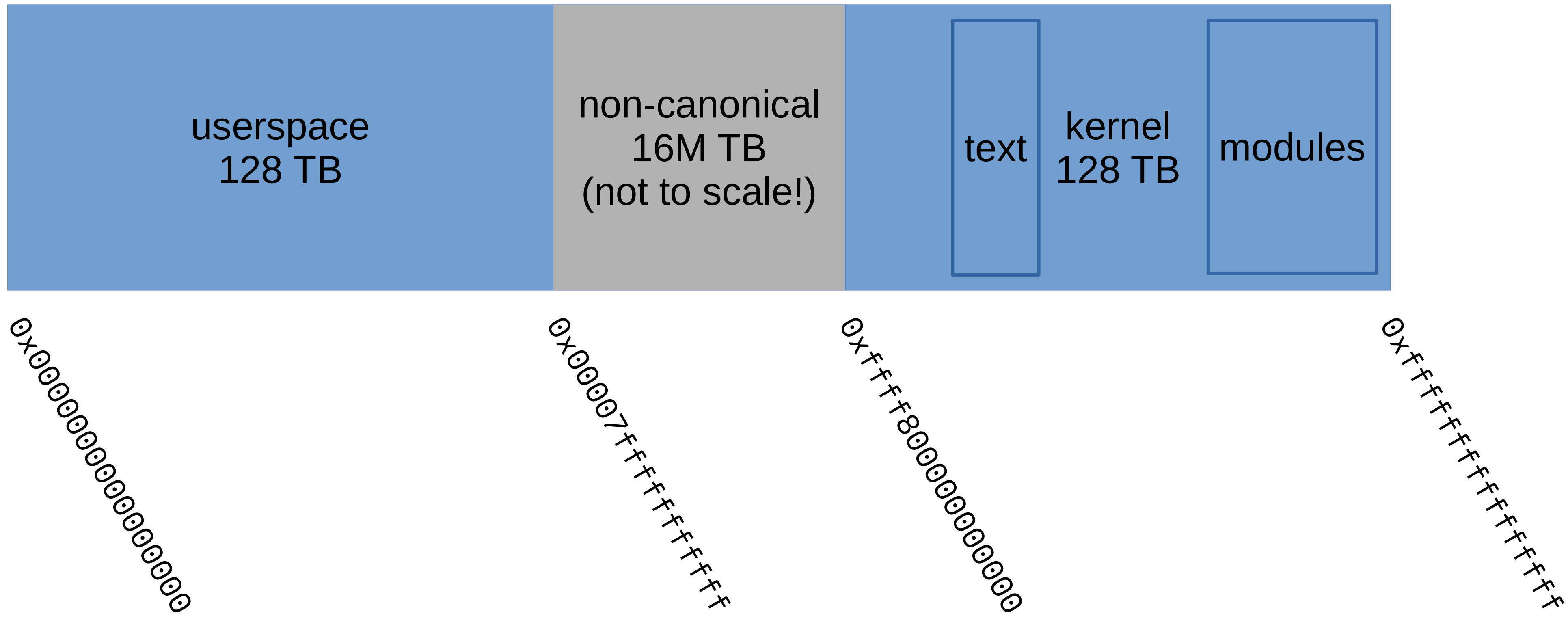
From userspace ...



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

# What is writable and executable?

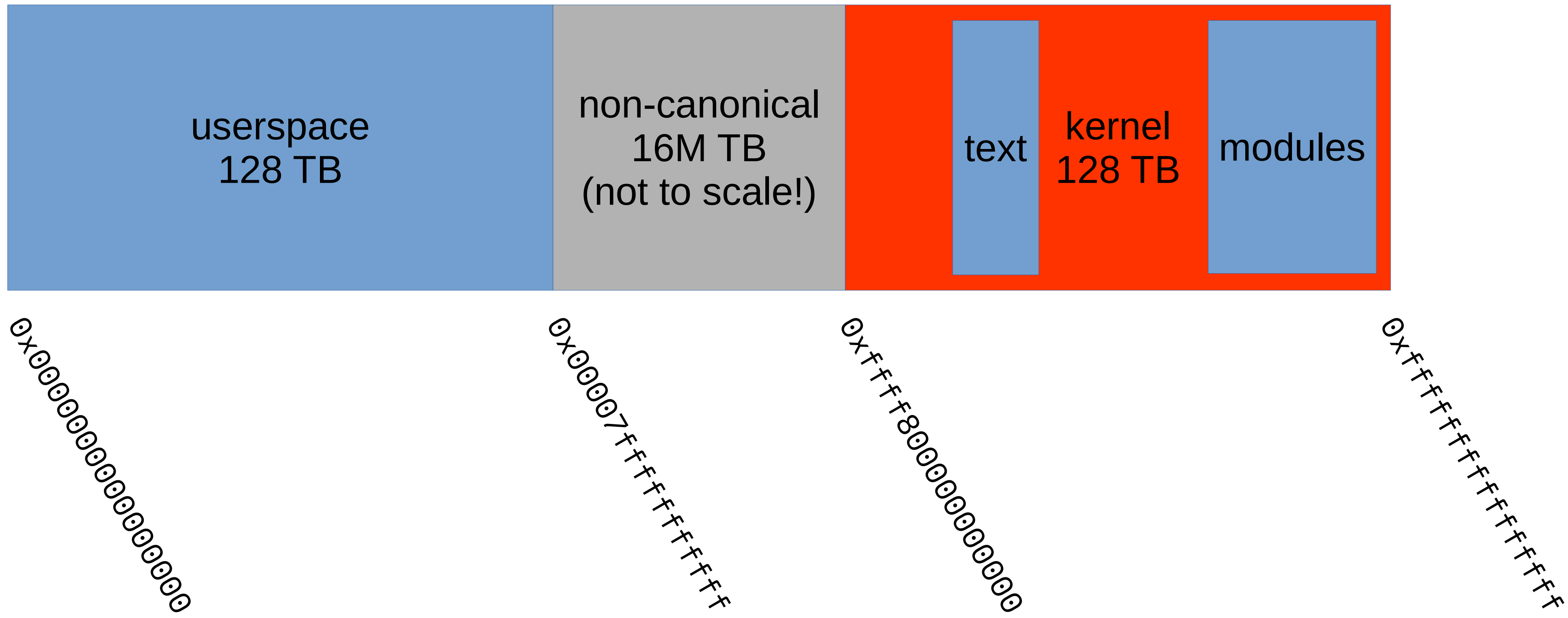
From kernel (ancient, simplified) ...



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

# What is writable and executable?

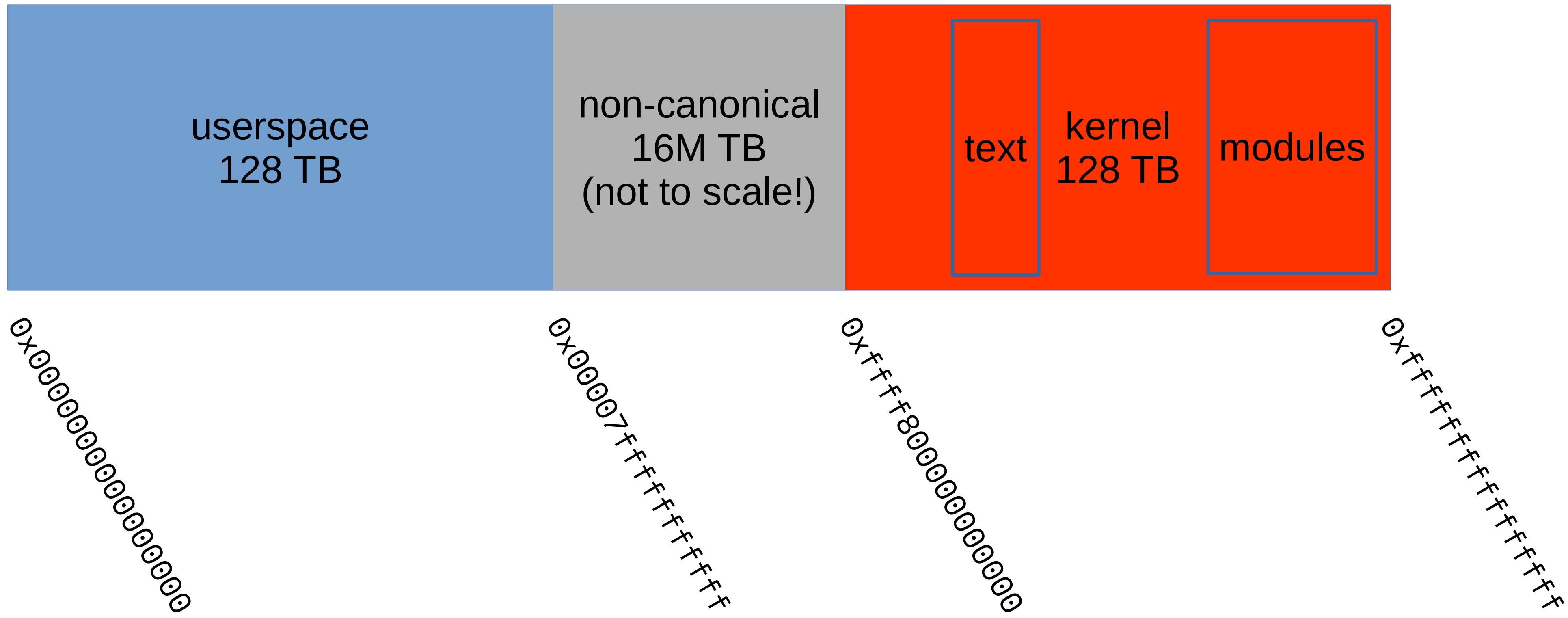
From kernel (NX, simplified) ...



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

# What is writable and executable?

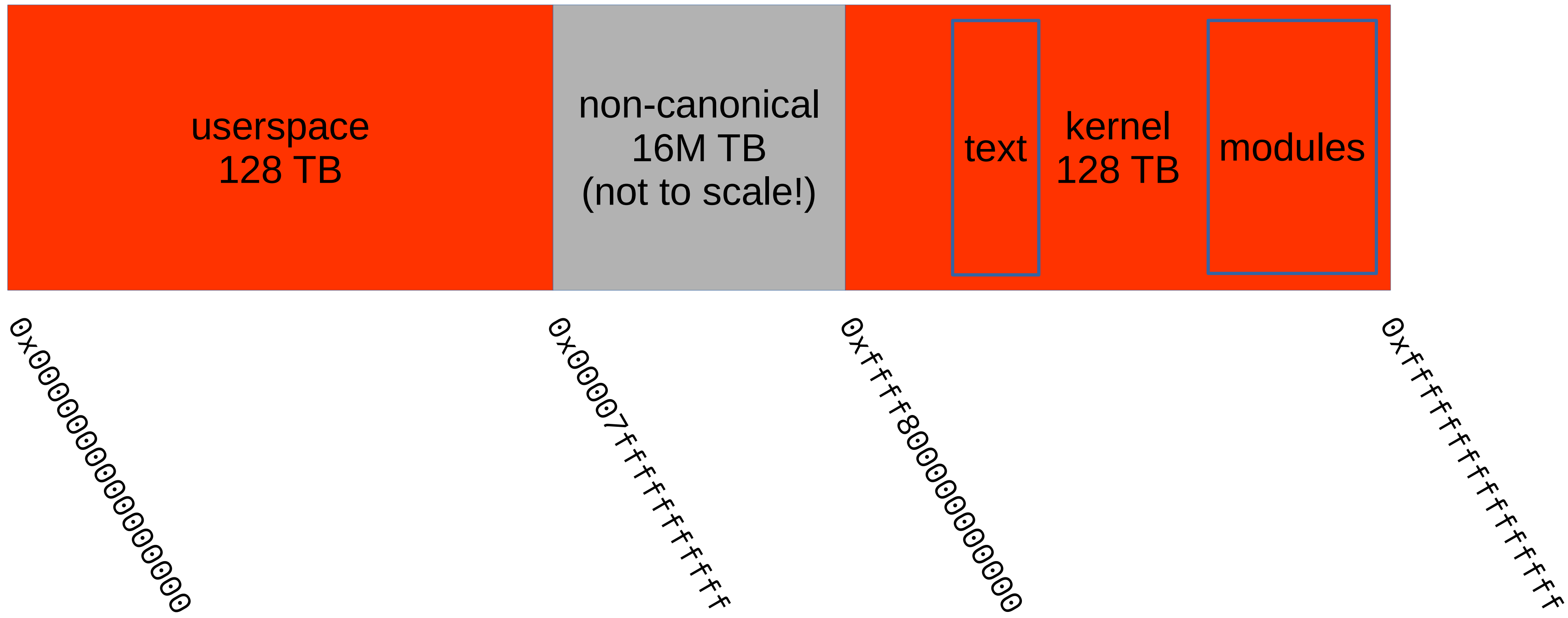
From kernel (NX, RO, simplified) ...



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

# What is writable and executable?

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



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

## Attack method: call into kernel 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);
...
}
```

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

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

# 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**”

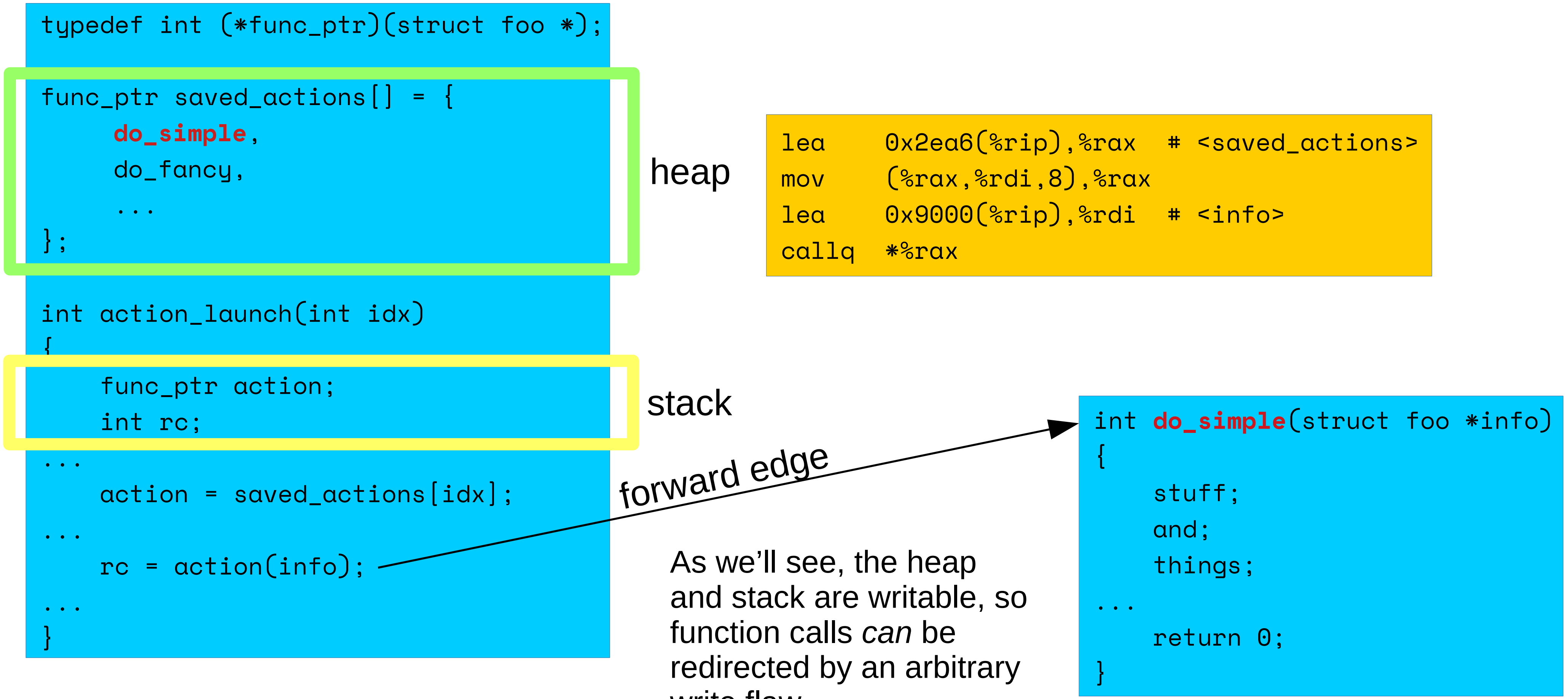
```
typedef int (*func_ptr)(struct foo *);

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
```

forward edge

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

# indirect calls: “**forward-edge**”



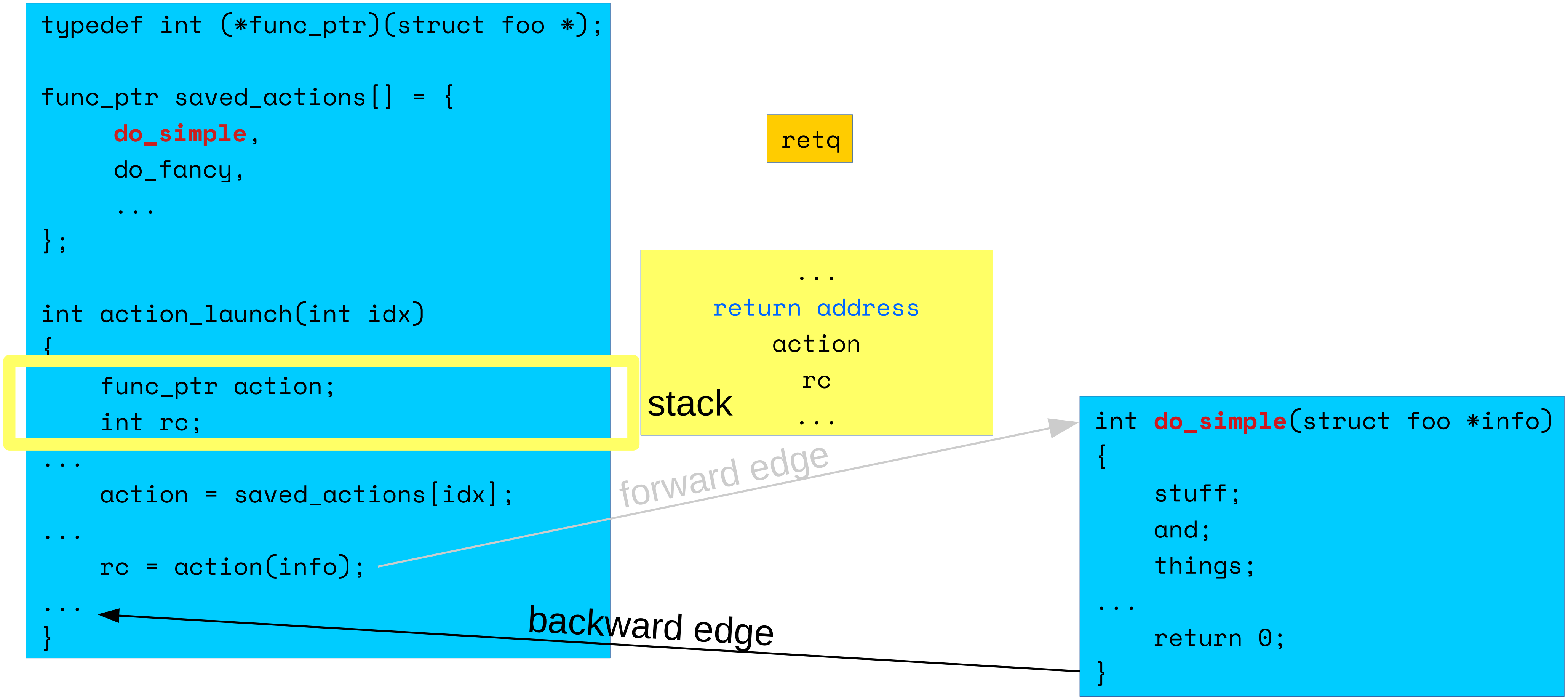
```
typedef int (*func_ptr)(struct foo *);

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;
}
```

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

# function returns: “**backward-edge**”

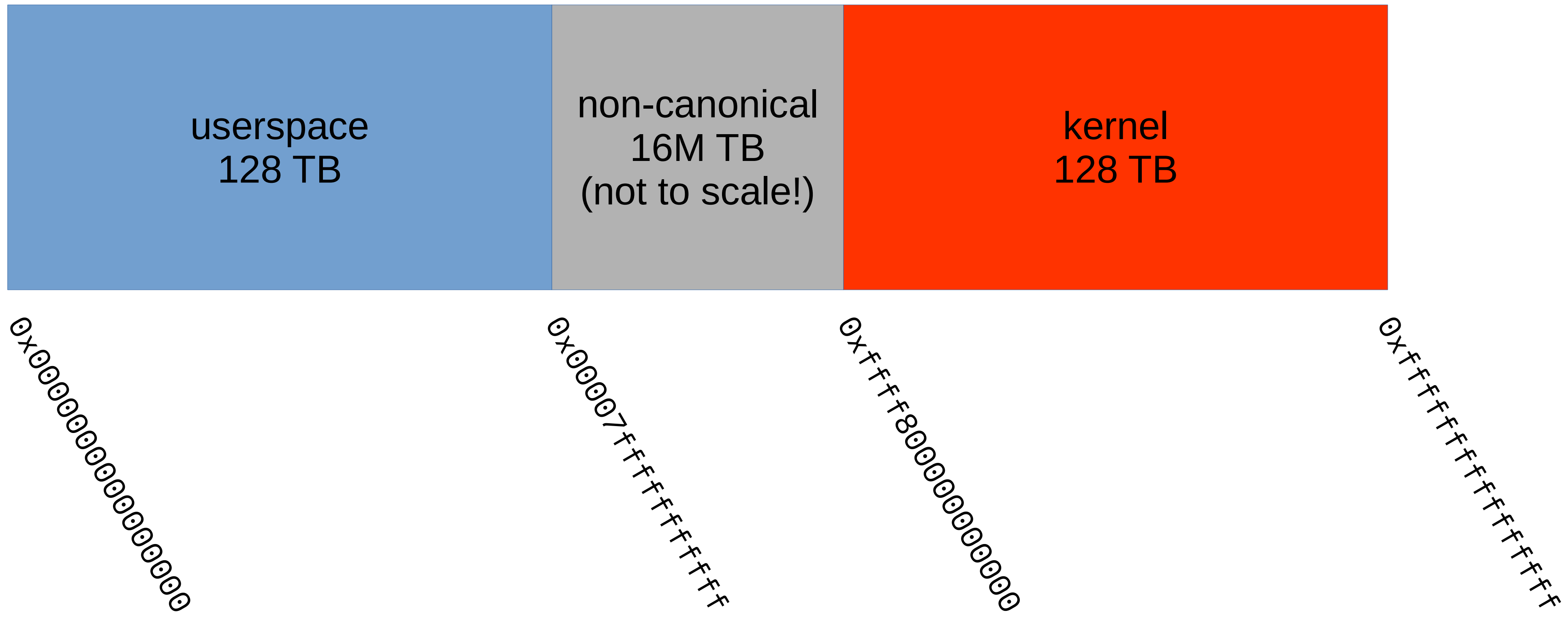


```
typedef int (*func_ptr)(struct foo *);

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);
...
}
```

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

# What contains writable func ptrs?

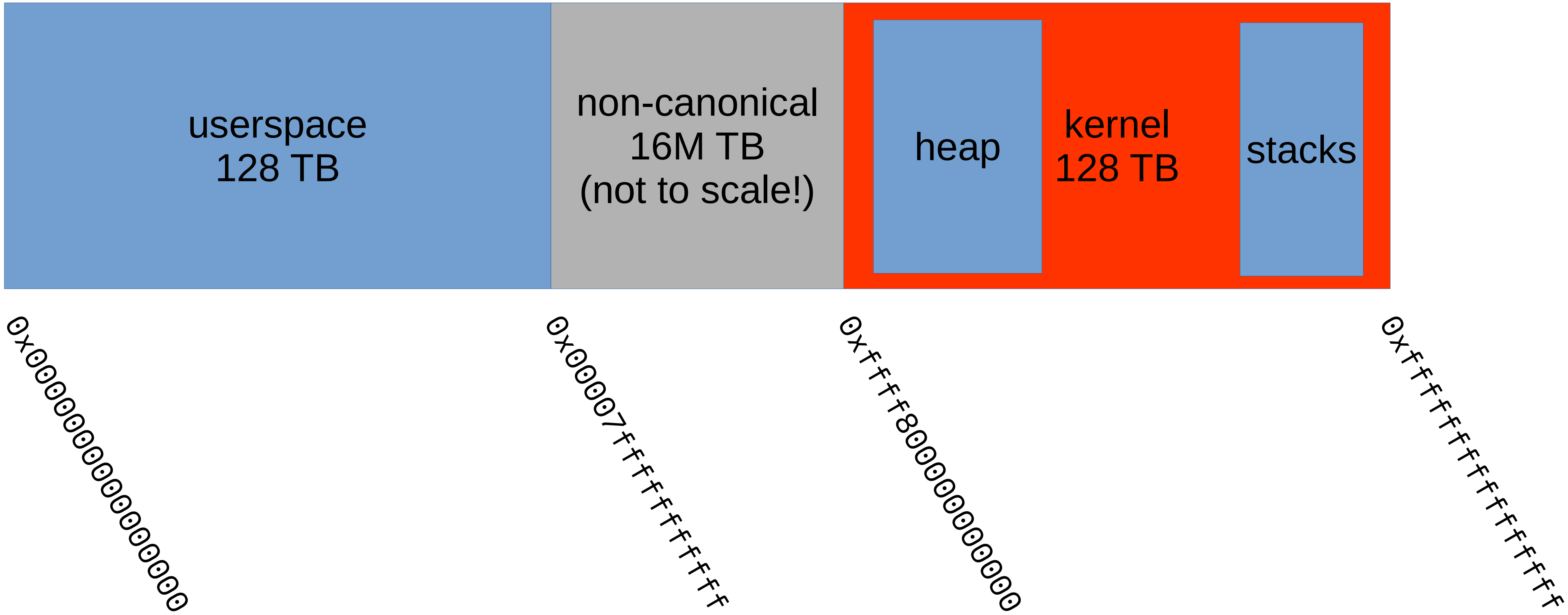
From userspace ...



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

# What contains writable func ptrs?

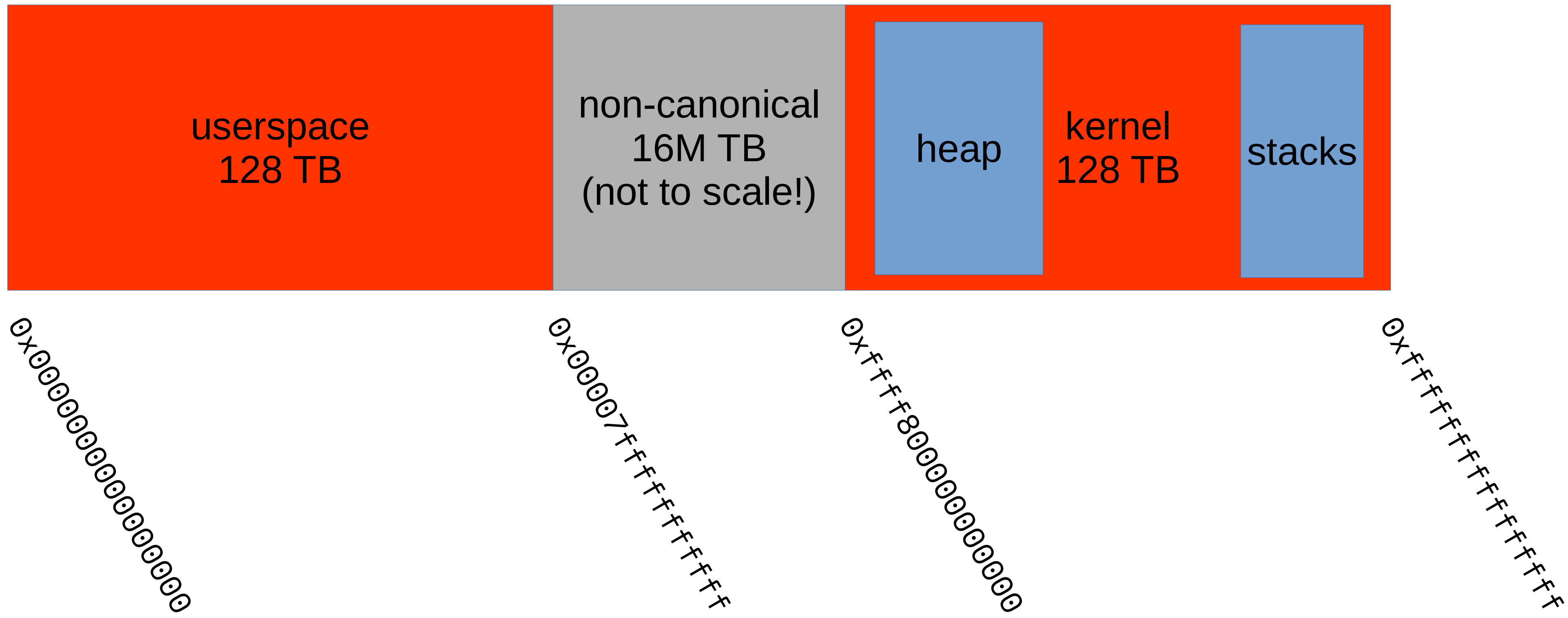
From kernel (simplified) ...



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

# What contains writable func ptrs?

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



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

# What can attacker call?

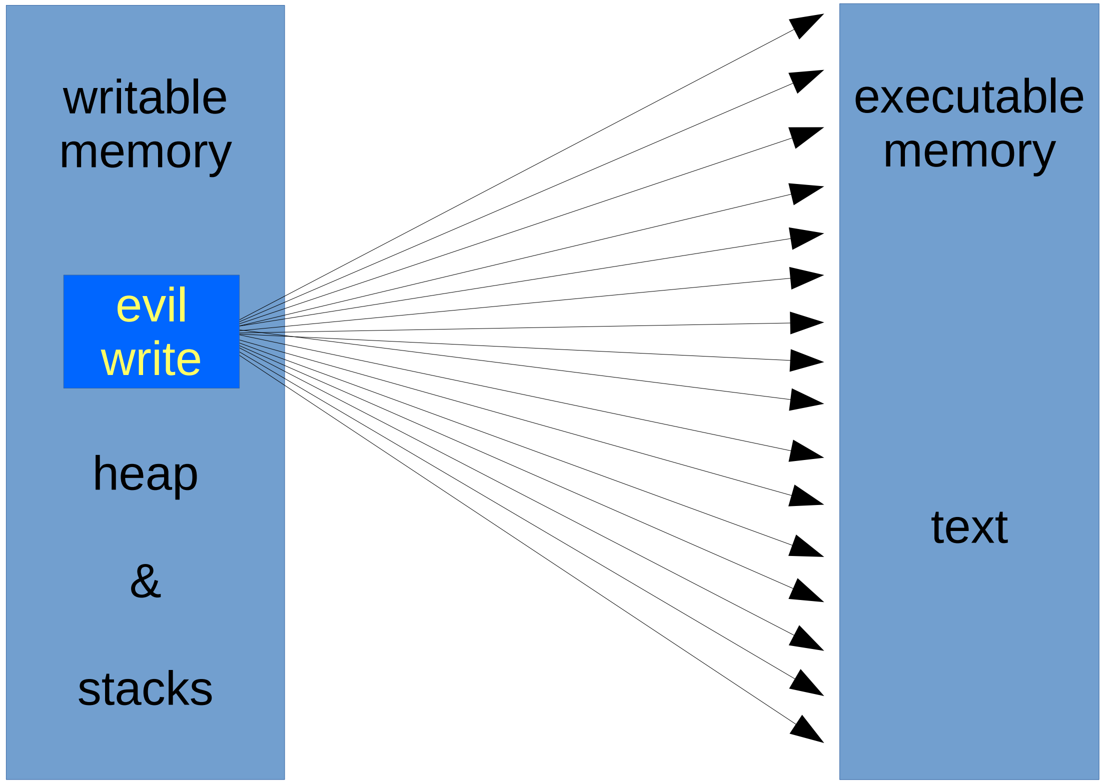
Any executable byte!

forward edge

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

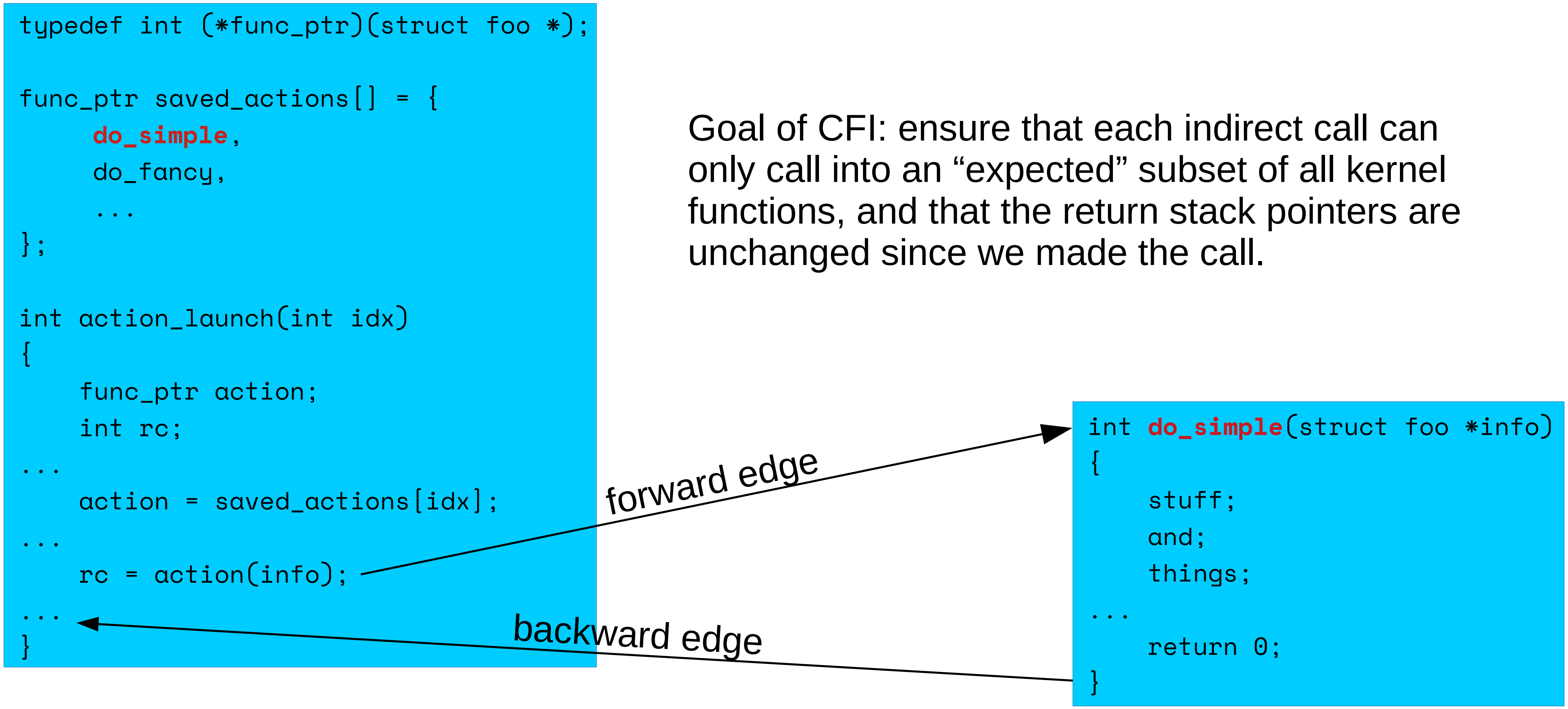
backward edge

```
retq
```



# Control Flow Integrity

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.



```
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);
...
}
```

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

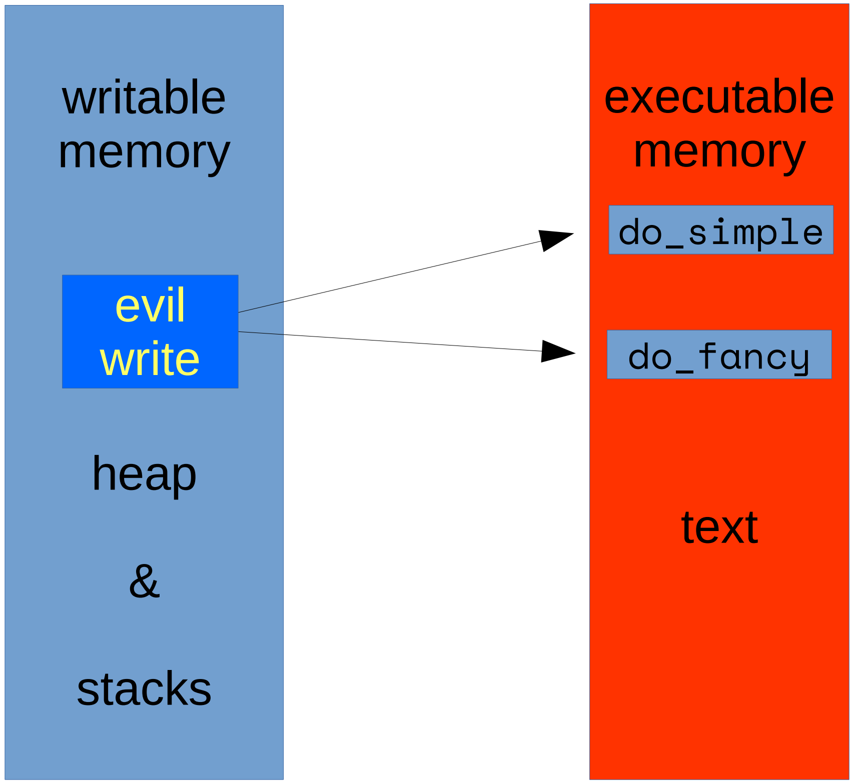
# CFI: forward-edge protection

- validate indirect function pointers at call time
  - some way to indicate “classes” of functions: current research suggests using function prototype (return type, argument types) as “uniqueness” key. 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)
      - int bar(unsigned long)
  - hardware help here has poor granularity (e.g. BTI)

# What can attacker call?

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);
...
}
```



## Forward-edge protection in Clang

- Needs global call site visibility, so Link Time Optimization (LTO) becomes a prerequisite:
  - This needs a fair bit of build script changes because `.o` files aren’t actually object files any more, they’re LLVM IR, so standard (bfd) binutils don’t work on them any more (need LLVM tools instead)
  - some symbols get weird due to LTO’s aggressive inlining and other optimizations
- Functions with same prototype collected into jump tables and checks added at each call site

# Stock: without Clang CFI

```
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq
...
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq
```

```
<action_launch>:
  201890: push   %rbx
  201891: movslq %edi,%rax
  201894: mov    0x200550(,%rax,8),%rax




  20189c: mov    $0x203b44,%edi
  2018a1: callq  *%rax
...
```

# Protected: with Clang CFI

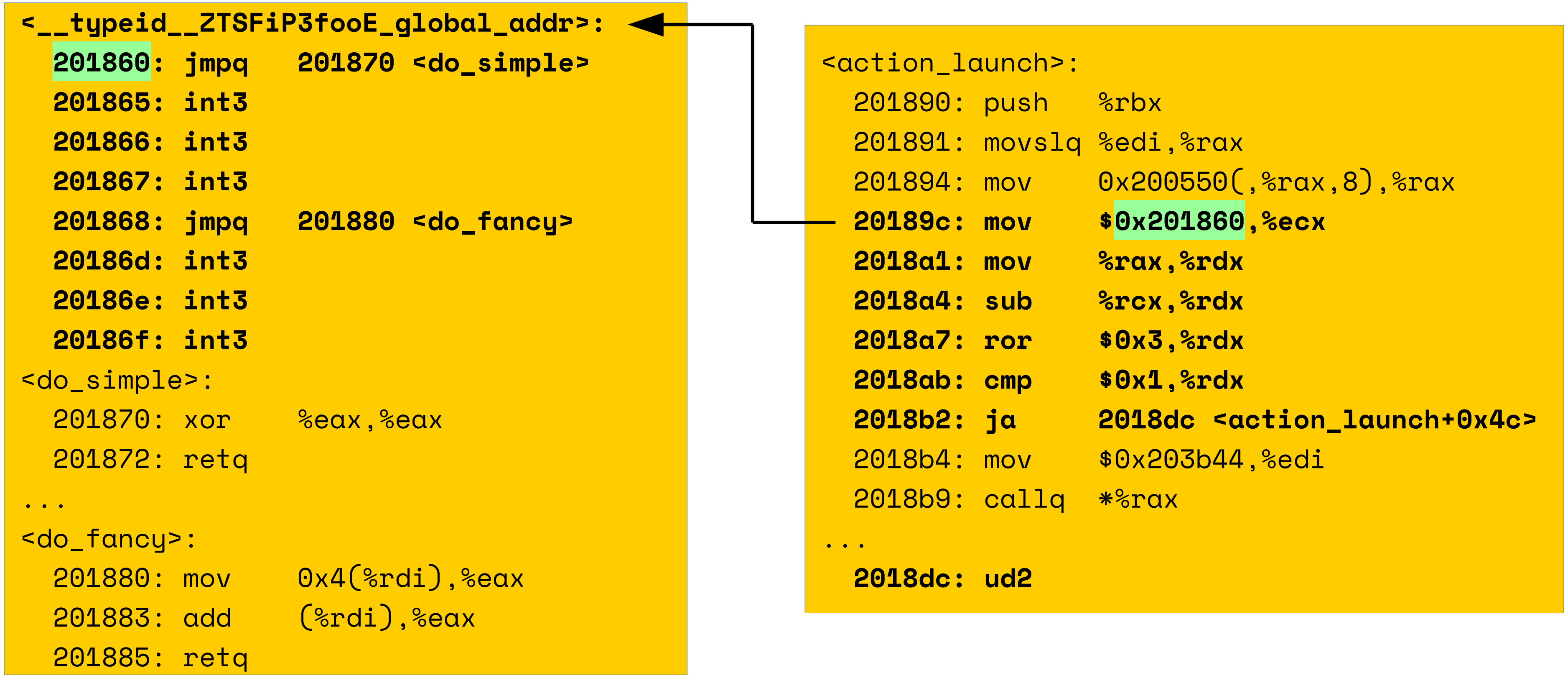
```
clang -fuse-ld=lld -flto -fvisibility=default \
      -fsanitize=cfi -fno-sanitize-cfi-canonical-jump-tables
```

```
<__typeid__ZTSFiP3fooE_global_addr>:
  201860: jmpq   201870 <do_simple>
  201865: int3
  201866: int3
  201867: int3
  201868: jmpq   201880 <do_fancy>
  20186d: int3
  20186e: int3
  20186f: int3
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq
...
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq
```

```
<action_launch>:
  201890: push   %rbx
  201891: movslq %edi,%rax
  201894: mov    0x200550(,%rax,8),%rax
  20189c: mov    $0x201860,%ecx
  2018a1: mov    %rax,%rdx
  2018a4: sub    %rcx,%rdx
  2018a7: ror    $0x3,%rdx
  2018ab: cmp    $0x1,%rdx
  2018b2: ja     2018dc <action_launch+0x4c>
  2018b4: mov    $0x203b44,%edi
  2018b9: callq  *%rax
...
  2018dc: ud2
```

# Protected: with Clang CFI

```
clang -fuse-ld=lld -flto -fvisibility=default \
      -fsanitize=cfi -fno-sanitize-cfi-canonical-jump-tables
```



```
<__typeid__ZTSFiP3fooE_global_addr>:
  201860: jmpq   201870 <do_simple>
  201865: int3
  201866: int3
  201867: int3
  201868: jmpq   201880 <do_fancy>
  20186d: int3
  20186e: int3
  20186f: int3
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq
...
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq
```

```
<action_launch>:
  201890: push   %rbx
  201891: movslq %edi,%rax
  201894: mov    0x200550(,%rax,8),%rax
  20189c: mov    $0x201860,%ecx
  2018a1: mov    %rax,%rdx
  2018a4: sub    %rcx,%rdx
  2018a7: ror    $0x3,%rdx
  2018ab: cmp    $0x1,%rdx
  2018b2: ja     2018dc <action_launch+0x4c>
  2018b4: mov    $0x203b44,%edi
  2018b9: callq  *%rax
...
  2018dc: ud2
```

# Protected: with Clang CFI

```
clang -fuse-ld=lld -flto -fvisibility=default \
      -fsanitize=cfi -fno-sanitize-cfi-canonical-jump-tables
```

```
<__typeid__ZTSFiP3fooE_global_addr>:
  201860: jmpq   201870 <do_simple>
  201865: int3
  201866: int3
  201867: int3
  201868: jmpq   201880 <do_fancy>
  20186d: int3
  20186e: int3
  20186f: int3
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq
...
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq
```

```
<action_launch>:
  201890: push   %rbx
  201891: movslq %edi,%rax
  201894: mov    0x200550(,%rax,8),%rax
  20189c: mov    $0x201860,%ecx
  2018a1: mov    %rax,%rdx
  2018a4: sub    %rcx,%rdx
  2018a7: ror    $0x3,%rdx
  2018ab: cmp    $0x1,%rdx
  2018b2: ja     2018dc <action_launch+0x4c>
  2018b4: mov    $0x203b44,%edi
  2018b9: callq  *%rax
...
  2018dc: ud2
```

# Protected: with Clang CFI

```
clang -fuse-ld=lld -flto -fvisibility=default \
      -fsanitize=cfi -fno-sanitize-cfi-canonical-jump-tables
```

```
<__typeid__ZTSFiP3fooE_global_addr>:
  201860: jmpq   201870 <do_simple>
  201865: int3
  201866: int3
  201867: int3
  201868: jmpq   201880 <do_fancy>
  20186d: int3
  20186e: int3
  20186f: int3
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq
...
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq
```

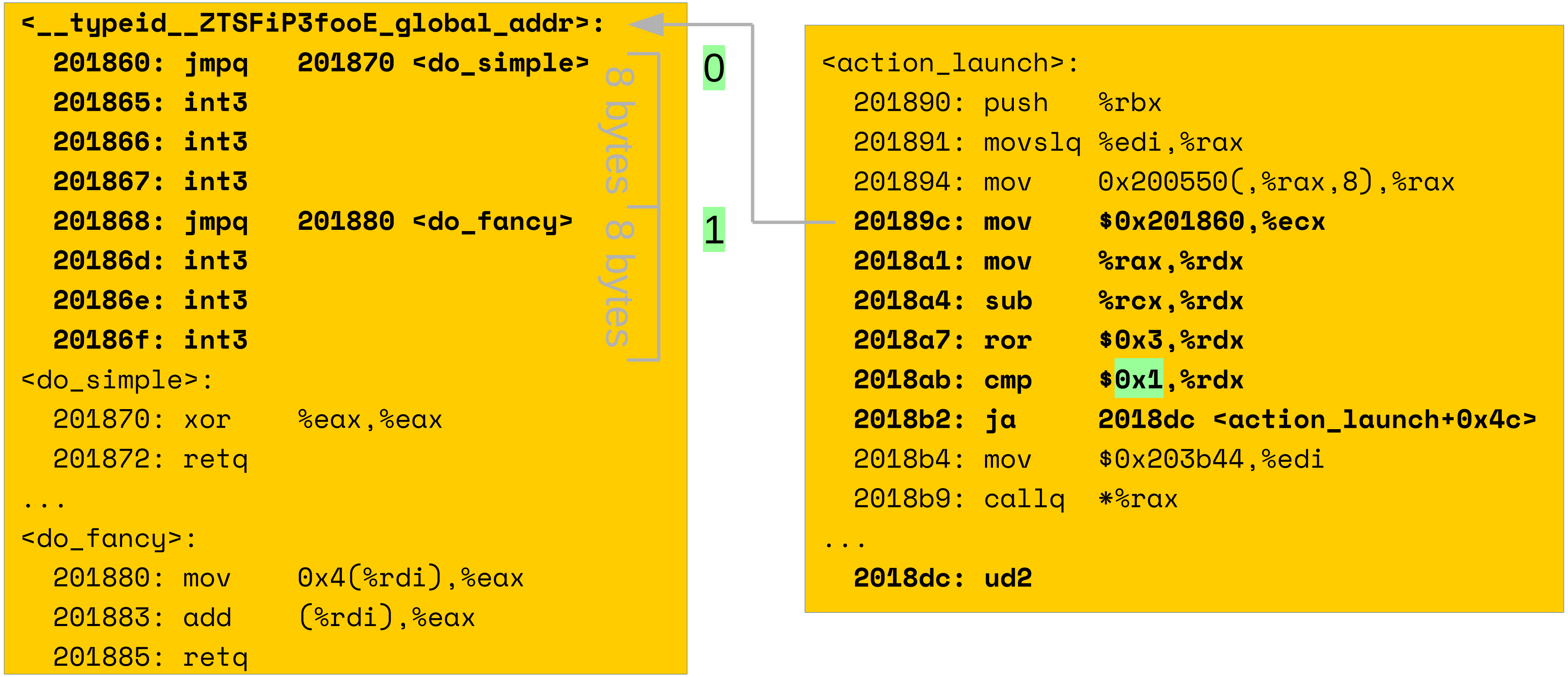
8 bytes

8 bytes

```
<action_launch>:
  201890: push   %rbx
  201891: movslq %edi,%rax
  201894: mov    0x200550(,%rax,8),%rax
  20189c: mov    $0x201860,%ecx
  2018a1: mov    %rax,%rdx
  2018a4: sub    %rcx,%rdx
  2018a7: ror    $0x3,%rdx
  2018ab: cmp    $0x1,%rdx
  2018b2: ja     2018dc <action_launch+0x4c>
  2018b4: mov    $0x203b44,%edi
  2018b9: callq  *%rax
...
  2018dc: ud2
```

# Protected: with Clang CFI

```
clang -fuse-ld=lld -flto -fvisibility=default \
      -fsanitize=cfi -fno-sanitize-cfi-canonical-jump-tables
```



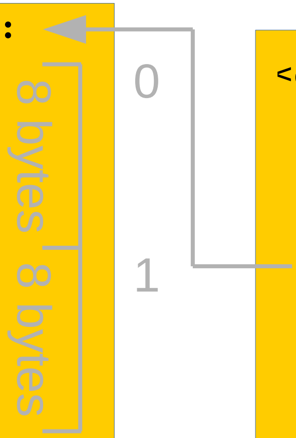
```
<__typeid__ZTSFiP3fooE_global_addr>:
  201860: jmpq    201870 <do_simple>
  201865: int3
  201866: int3
  201867: int3
  201868: jmpq    201880 <do_fancy>
  20186d: int3
  20186e: int3
  20186f: int3
<do_simple>:
  201870: xor     %eax,%eax
  201872: retq
...
<do_fancy>:
  201880: mov     0x4(%rdi),%eax
  201883: add     (%rdi),%eax
  201885: retq
```

```
<action_launch>:
  201890: push    %rbx
  201891: movslq  %edi,%rax
  201894: mov     0x200550(,%rax,8),%rax
  20189c: mov     $0x201860,%ecx
  2018a1: mov     %rax,%rdx
  2018a4: sub     %rcx,%rdx
  2018a7: ror     $0x3,%rdx
  2018ab: cmp     $0x1,%rdx
  2018b2: ja      2018dc <action_launch+0x4c>
  2018b4: mov     $0x203b44,%edi
  2018b9: callq   *%rax
...
  2018dc: ud2
```

# Protected: with Clang CFI

```
clang -fuse-ld=lld -flto -fvisibility=default \
      -fsanitize=cfi -fno-sanitize-cfi-canonical-jump-tables
```

```
<__typeid__ZTSFiP3fooE_global_addr>:
  201860: jmpq   201870 <do_simple>
  201865: int3
  201866: int3
  201867: int3
  201868: jmpq   201880 <do_fancy>
  20186d: int3
  20186e: int3
  20186f: int3
<do_simple>:
  201870: xor    %eax,%eax
  201872: retq
...
<do_fancy>:
  201880: mov    0x4(%rdi),%eax
  201883: add    (%rdi),%eax
  201885: retq
```



```
<action_launch>:
  201890: push   %rbx
  201891: movslq %edi,%rax
  201894: mov    0x200550(,%rax,8),%rax
  20189c: mov    $0x201860,%ecx
  2018a1: mov    %rax,%rdx
  2018a4: sub    %rcx,%rdx
  2018a7: ror    $0x3,%rdx
  2018ab: cmp    $0x1,%rdx
  2018b2: ja     2018dc <action_launch+0x4c>
  2018b4: mov    $0x203b44,%edi
  2018b9: callq  *%rax
...
  2018dc: ud2
```

## Jump tables and type mangling

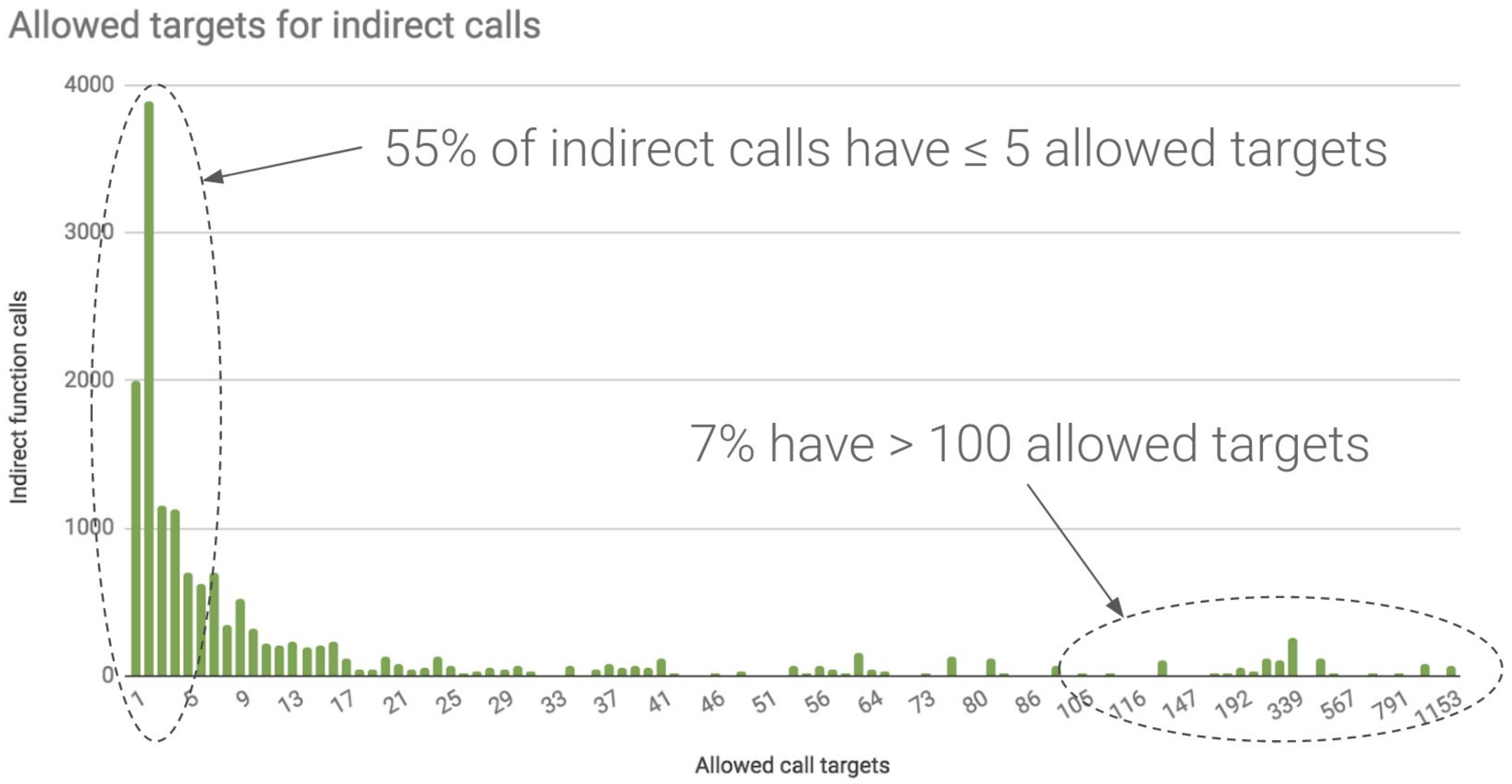
```
<__typeid__ZTSFiP3fooE_global_addr>:
  201860: jmpq   201870 <do_simple>
  201865: int3
  201866: int3
  201867: int3
  201868: jmpq   201880 <do_fancy>
  20186d: int3
  20186e: int3
  20186f: int3
```

```
$ llvm-cxxfilt _ZTSFiP3fooE
typeinfo name for int (foo*)
```

# Better implementation ideas?

- For improved speed, instead of jump tables and DSO check functions, add hash bytes before function start/return destinations, and check for matches at call and return sites (e.g. as done by the last public version of PaX Team's RAP). This isn't compatible with execute-only memory, though.
- After function prototype bucketizing, perform finer grained analysis for function reachability to avoid having a large number of similar functions callable from each call site (e.g. as done by João Moreira, et al.'s kCFI). For example, the kernel has really a lot of `void foo(void)` functions...

# Why finer grained bucketizing?



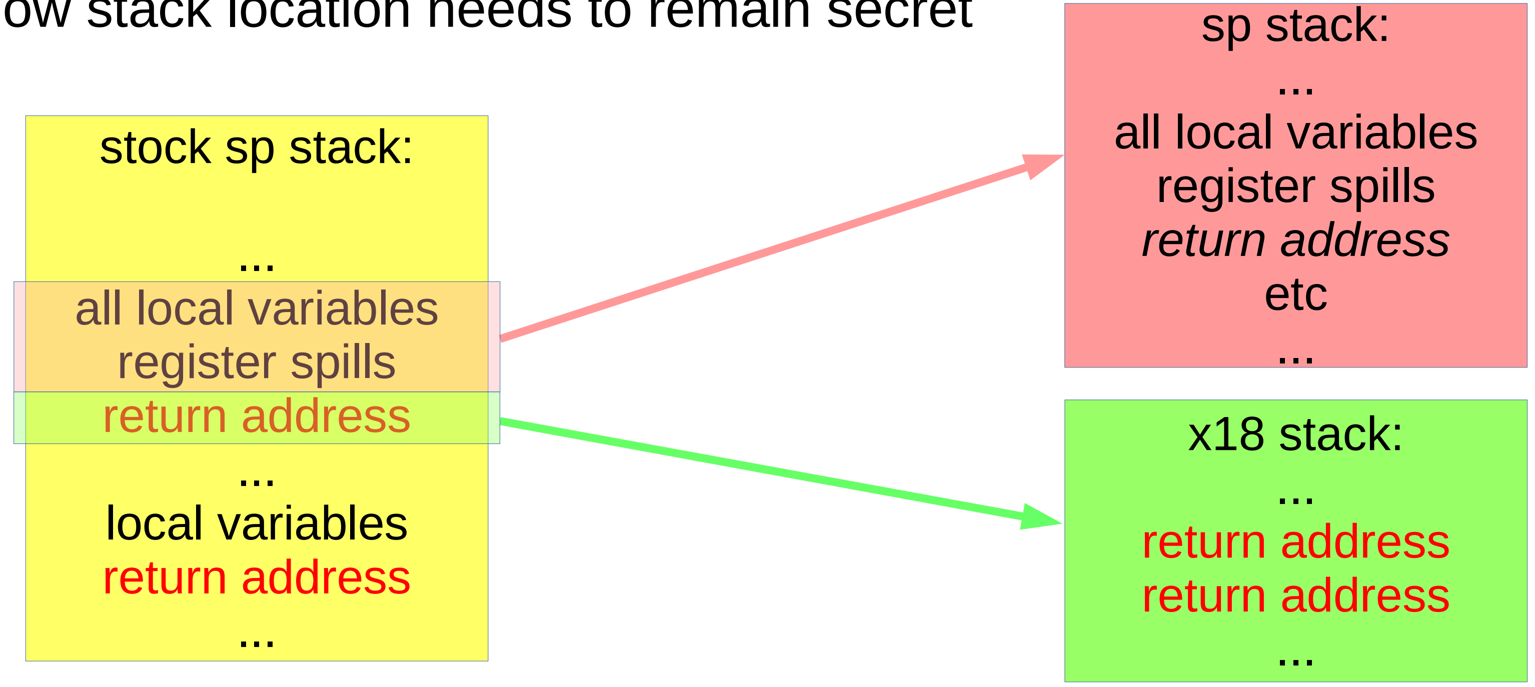
https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html

## 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

# Backward-edge protection in Clang

- x86 implementation removed because it was slow and had race conditions :(
- arm64 can reserve a register (x18) for all shadow stack manipulations
- Shadow stack location needs to remain secret

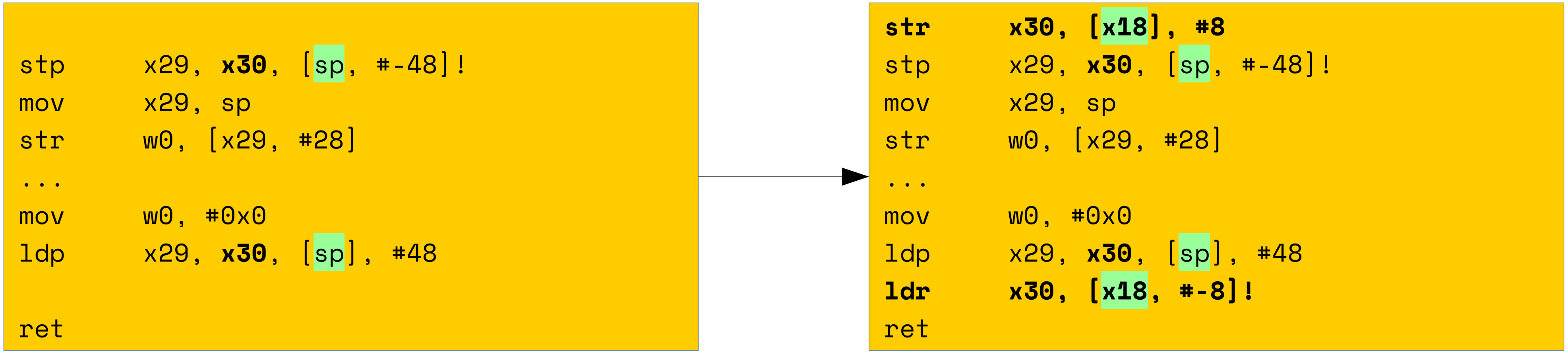


# Clang Shadow Call Stack (SCS)

`-ffixed-x18 -fsanitize=shadow-call-stack`

Results in two stack registers: `sp` and unspilled `x18`

Only loads of the return address (link) register from shadow stack (pointed to by `x18`) are used for return.



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

ret
```

```
str     x30, [x18], #8
stp     x29, x30, [sp, #-48]!
mov     x29, sp
str     w0, [x29, #28]
...
mov     w0, #0x0
ldp     x29, x30, [sp], #48
ldr     x30, [x18, #-8]!
ret
```

# Backward-edge protection in hardware

- 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
```

# Pixel phones & Android ecosystem

- Sami Tolvanen and others have been doing a giant amount of work to enable LTO, CFI, and SCS in the Android and upstream kernels
- Pixel (3 and later) as well as any other vendors enabling the feature:
  - Q3 2018: added forward-edge protection ("CFI")
  - Q3 2019: added backward-edge protection ("SCS")
- Android Compatibility Definition Document (CDD) says:

  "[C-SR] Are STRONGLY RECOMMENDED to enable control flow integrity (CFI) in the kernel to provide additional protection against code-reuse attacks (e.g. CONFIG_CFI_CLANG and CONFIG_SHADOW_CALL_STACK).

  [C-SR] Are STRONGLY RECOMMENDED not to disable Control-Flow Integrity (CFI), Shadow Call Stack (SCS) or Integer Overflow Sanitization (IntSan) on components that have it enabled."

  https://source.android.com/compatibility/10/android-10-cdd#9_7_security_features

# Gotchas

- massive LTO linking times
  - final linking step under LTO was very slow, so switched to ThinLTO (`-flto=thin`).
- assembly code
  - jump tables only built for C code, so Peter Collingbourne extended Clang to generate jump table entries for all `extern` functions (`-fno-sanitize-cfi-canonical-jump-tables`).
- relative addresses
  - exception tables: calculated as delta from true function address, ignored jump table address, so disable CFI checks for exception tables (which are hard-coded).
- linker aliases
  - ftrace made unusual calls to differing prototypes, but linker aliases satisfied CFI.
- Kernel Page Table Isolation (KPTI)
  - jump tables were outside mapped entry stub, so had to also map the jump tables.

# Upstreaming status

- Clang: done (?) as of unreleased LLVM 11
  - Nick Desaulniers and many other folks have been steadily adding features and fixing bugs specific to building Linux with Clang. And while not strictly CFI, the recent massive work on asm-goto made x86 possible at all.
  - https://github.com/ClangBuiltLinux/linux/issues
- Kernel: consistent progress
  - Clang Shadow Call Stack support (19 patches, in Linus's tree, expected for v5.8)
  - On-going: function pointer prototype corrections (e.g. `-Wfunction-cast`)
  - Clang Link Time Optimization (22 patches: crossing our fingers!)
    - Mostly tricky mechanical build script and Kconfig changes
    - Earlier questions about memory ordering and visibility changes seem(?) to be settled
  - Clang Control Flow Integrity (23 patches: depends on LTO)
    - Hopefully uncontroversial and should land quickly after LTO, given how many prototype fixes are landed

## Do it yourself!

https://outflux.net/blog/archives/2019/11/20/experimenting-with-clang-cfi-on-upstream-linux/

https://github.com/samitolvanen/linux/tree/clang-cfi

```
$ make defconfig CC=clang LD=ld.lld
$ scripts/config \
   -e CONFIG_LTO -e CONFIG_THINLTO -d CONFIG_LTO_NONE -e CONFIG_LTO_CLANG \
   -e CONFIG_CFI_CLANG -e CONFIG_CFI_CLANG_SHADOW \
   -e CONFIG_CFI_PERMISSIVE -e CONFIG_SHADOW_CALL_STACK
$ make -j$(getconf _NPROCESSORS_ONLN) CC=clang LD=ld.lld
```

# 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_int$53641d38e2dc4a151b75cbe816cbb86b.cfi_jt+0x0/0x10):
WARNING: CPU: 3 PID: 2806 at kernel/cfi.c:29 __cfi_check_fail+0x38/0x40
...
Call Trace:
...
```

# Thoughts?

Kees ("Case") Cook
keescook@chromium.org
keescook@google.com
kees@outflux.net
@kees_cook

https://outflux.net/slides/2020/lss/cfi.pdf

http://www.openwall.com/lists/kernel-hardening/
http://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project