arm

Evolving ftrace on arm64

Kernel Recipes

Mark Rutland <mark.rutland@arm.com>

2023-09-27

ftrace

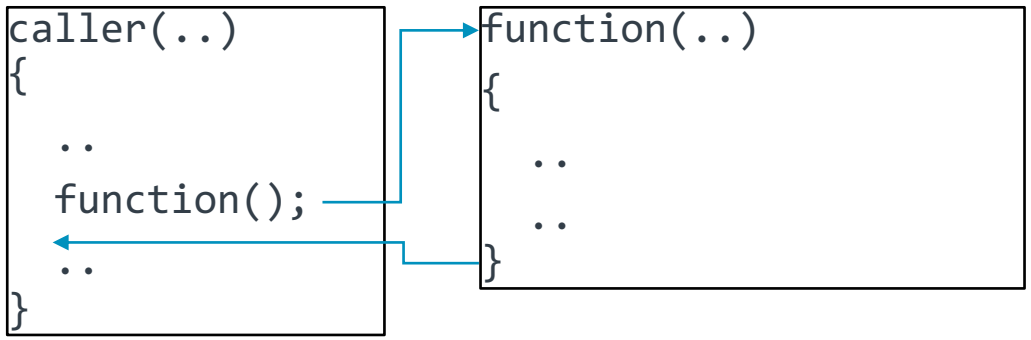
What is it?

- A framework for attaching ***tracers*** to kernel functions
  - … dynamically at runtime
  - … without explicit calls in source code
- Tracers aren’t just for tracing!
  - Used for fault-injection, live-patching, etc
- Used in production environments
  - Must be safe and robust
  - Must have minimal overhead
- Requires some architecture-specific code

```
# mount -t tracefs none /sys/kernel/tracing/
# echo function_graph > /sys/kernel/tracing/current_tracer
# cat /sys/kernel/tracing/trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)               |  do_el0_svc() {
 0)               |    el0_svc_common.constprop.0() {
 0)               |      invoke_syscall() {
 0)               |        __arm64_sys_fcntl() {
 0)               |          __fdget_raw() {
 0)   0.250 us    |            __fget_light();
 0)   0.750 us    |          }
 0)   0.208 us    |          security_file_fcntl();
 0)               |          do_fcntl() {
 0)   0.208 us    |            _raw_spin_lock();
 0)   0.250 us    |            _raw_spin_unlock();
 0)   1.125 us    |          }
 0)   2.917 us    |        }
 0)   3.375 us    |      }
 0)   3.875 us    |    }
 0)   4.334 us    |  }
```

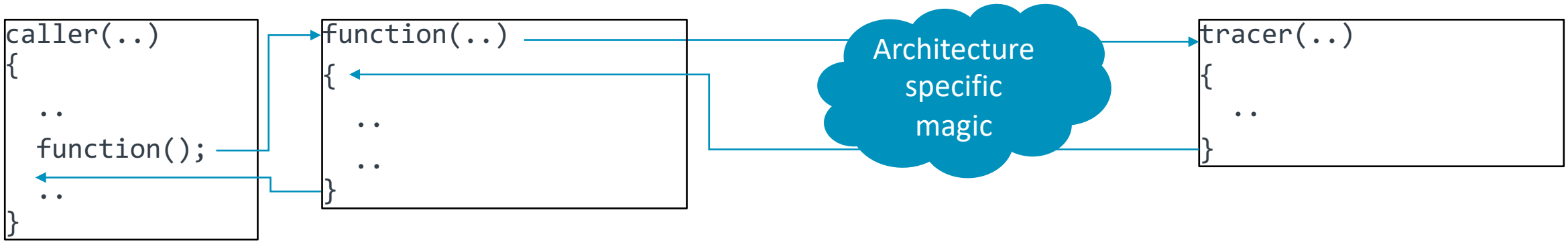
ftrace

In abstract



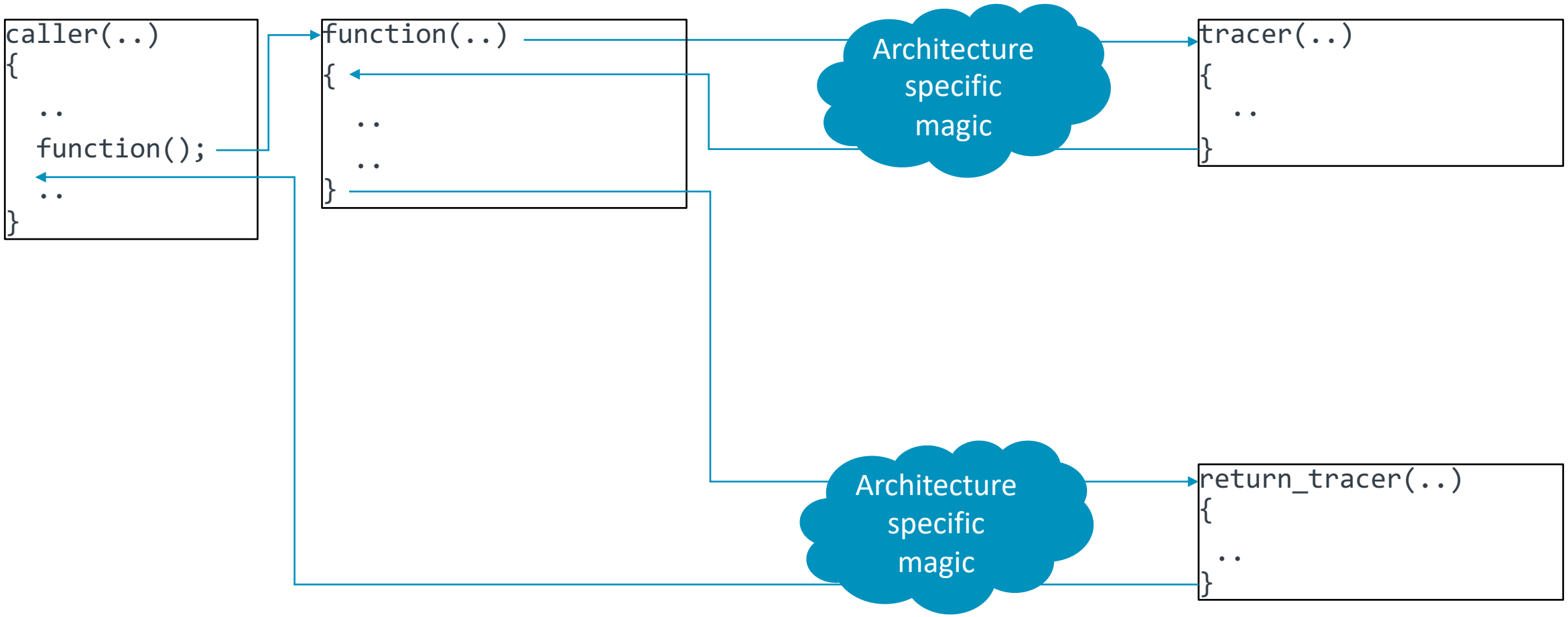
ftrace

In abstract: tracing function entry



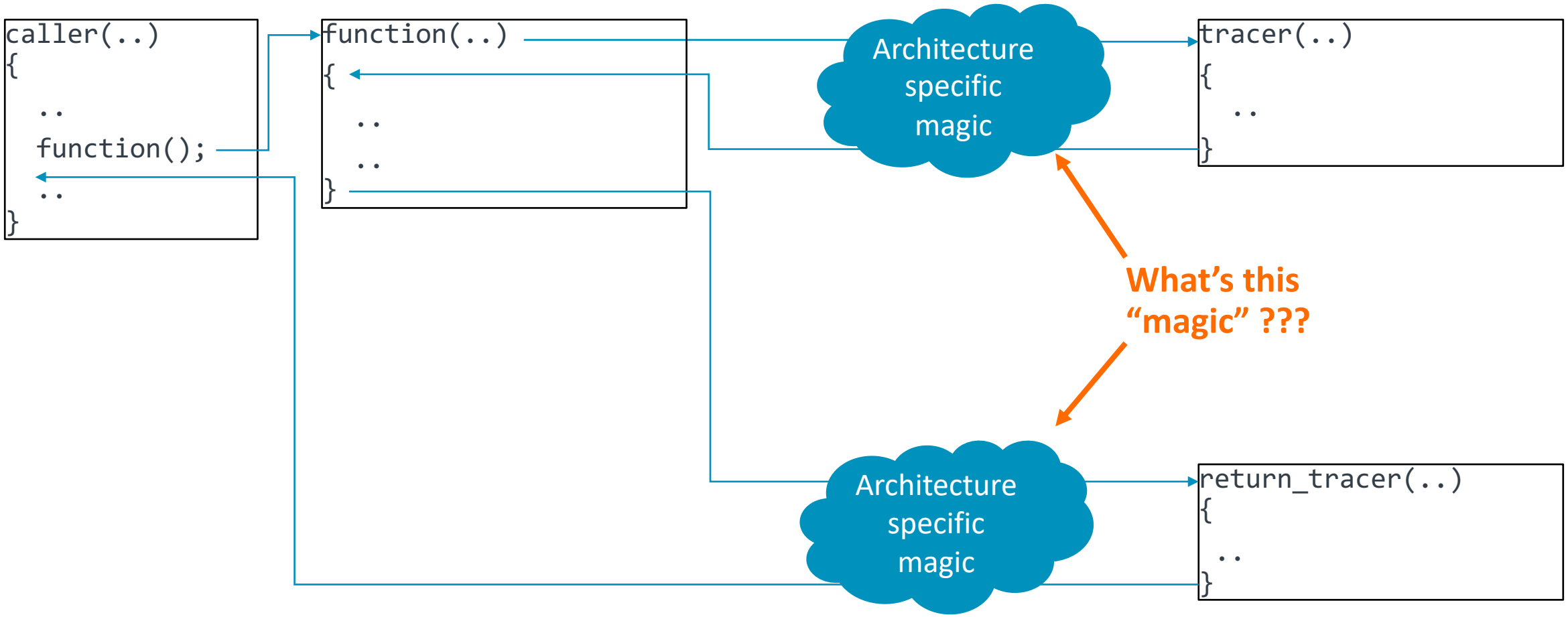
ftrace

In abstract: tracing function entry **and return**



ftrace

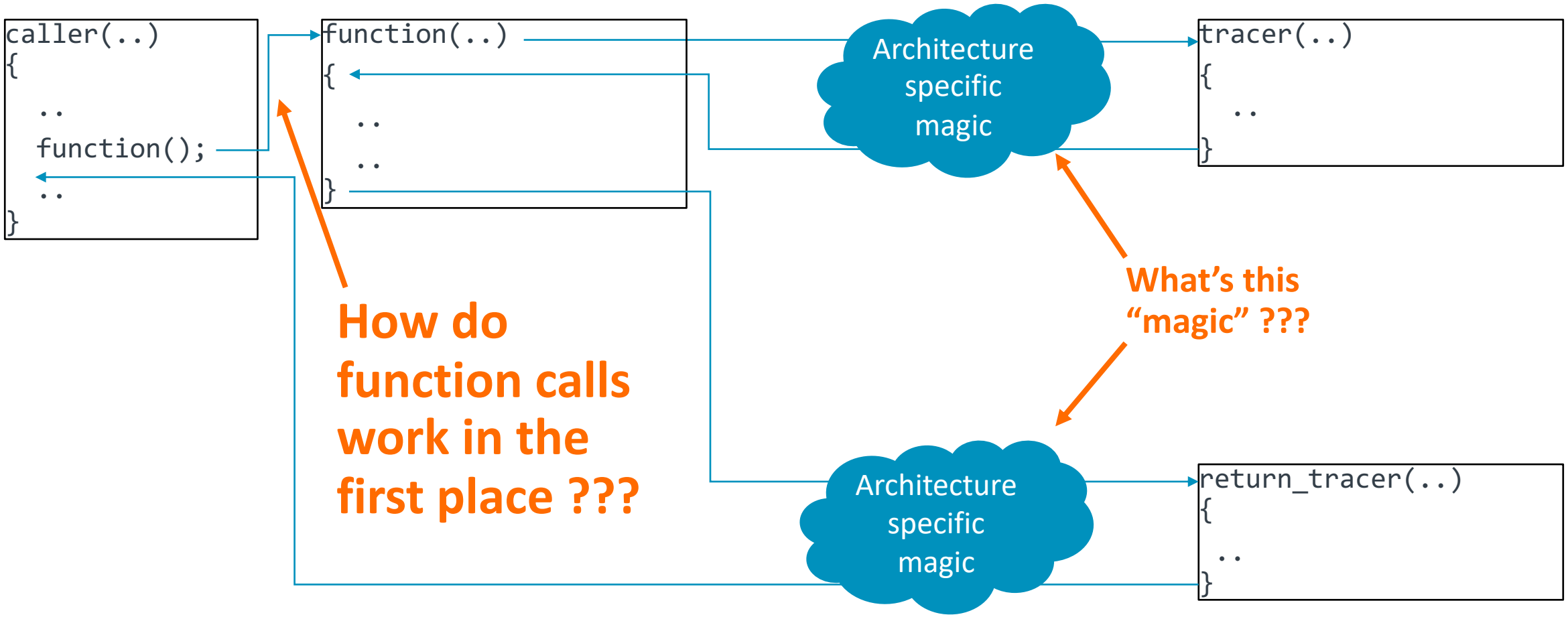
In abstract: tracing function entry **and return**



arm

ftrace

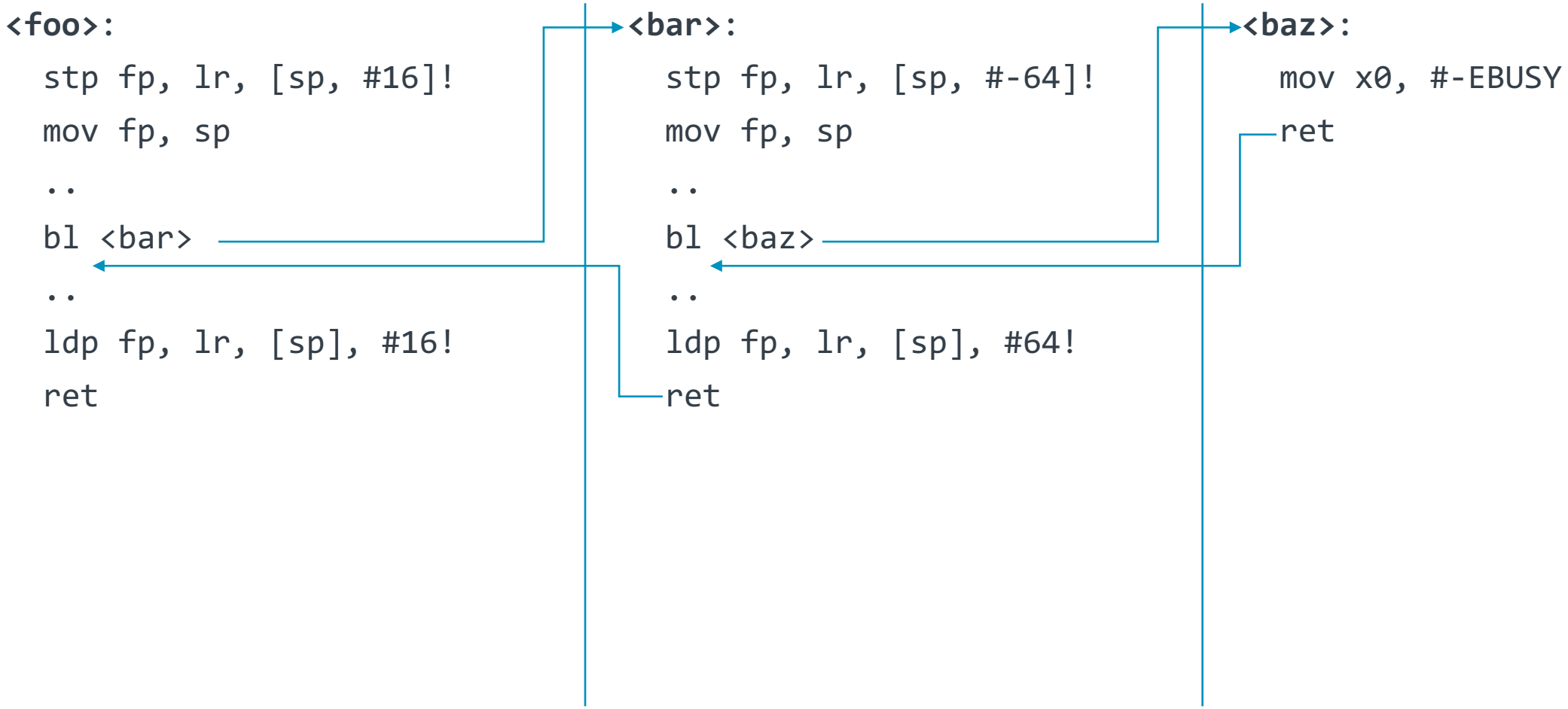
In abstract: tracing function entry **and return**



arm

Function calls on arm64

The Link Register (LR), Frame Pointer (FP), and Frame Records



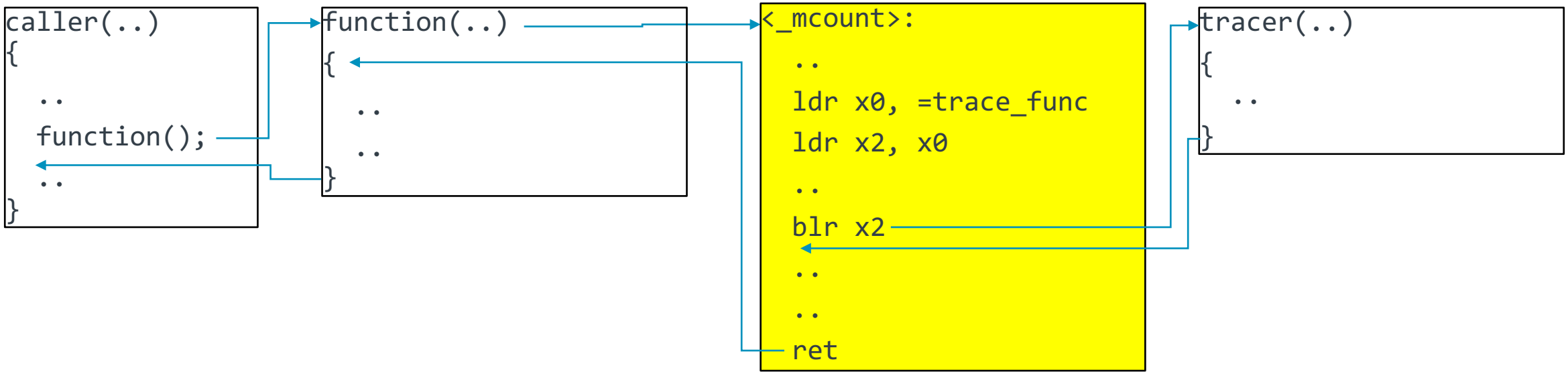
Magic v1: mcount

- GCC and LLVM support the gprof profiler with the **-pg** compiler option
  - Inserts calls to **mcount** at function entry
  - Compiler saves/restores registers, etc
- Compiler doesn't provide mcount itself
  - Usually provided by gprof
  - We can write our own!
- We can write our own mcount which calls tracers!

```
<function>:
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp
    mov     x0, lr
    bl      _mcount
    ..
    .. // function body here
    ..
    ldp     fp, lr, [sp], #16
    ret
```

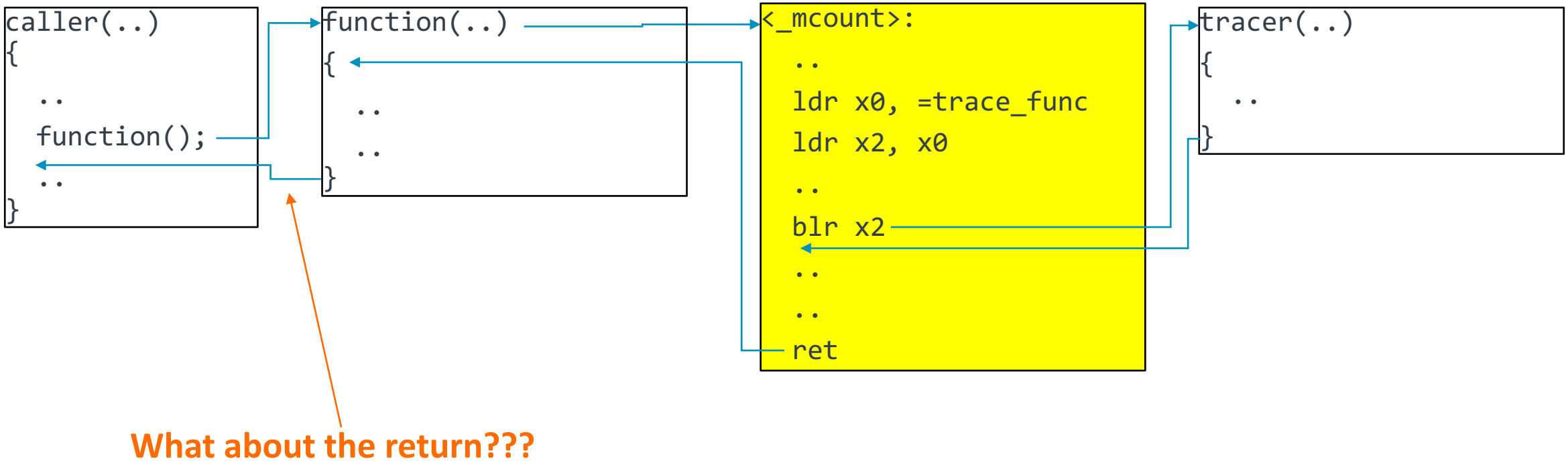
Magic v1: mcount

Using mcount as a trampoline



Magic v1: mcount

Using mcount as a trampoline



Magic v1.1: mcount + return

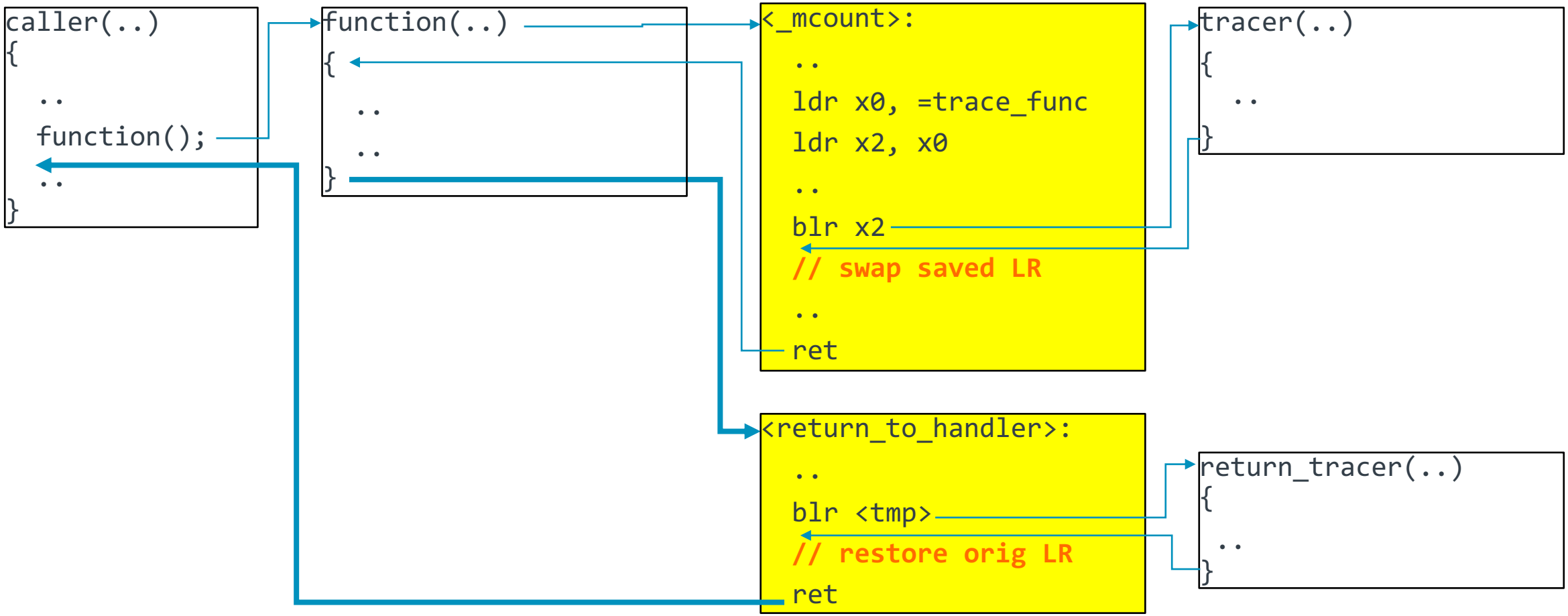
Modifying the frame record

- The compiler stores the return address to the stack **before** calling mcount
  - Placed in a **frame record** pointed to by the **FP**
- When mcount is called, the FP points to the traced function's frame record
  - So mcount can read/write the traced function's **LR**
- We can make the traced function return to a tracer by modifying the saved LR!

```
<function>:
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp
    mov     x0, lr
    bl      _mcount
    ..
    .. // function body here
    ..
    ldp     fp, lr, [sp], #16
    ret
```

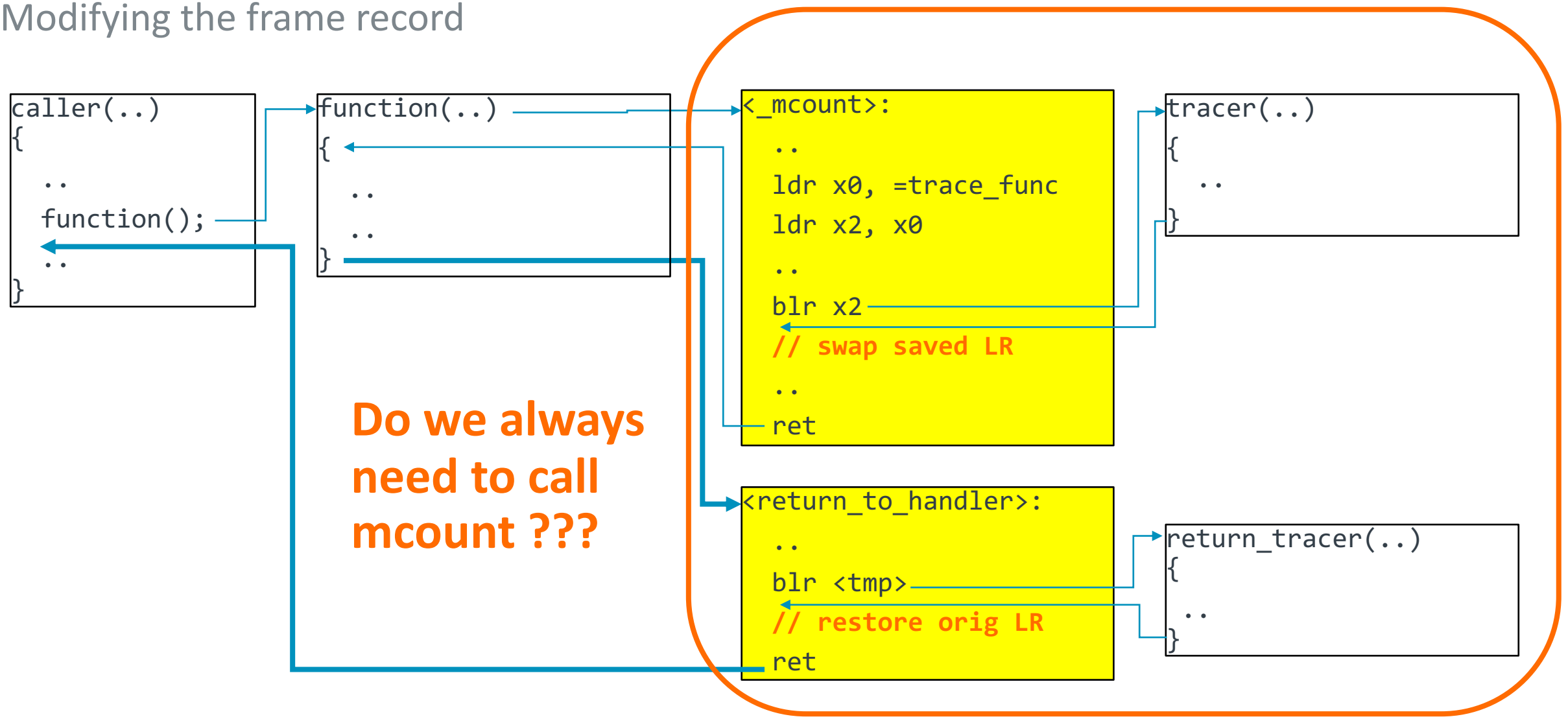
Magic v1.1: mcount + return

Modifying the frame record



Magic v1.1: mcount + return

Modifying the frame record



Magic v1.2: disabling mcount

- Functions aren't traced all the time
  - Tracing usage is generally bursty
  - Few functions are live-patched
- Traceable functions **always** call mcount
  - Buy one function call, get one free…
- We can patch the function call to remove the overhead:
  - Enabled → `BL _mcount`
  - Disabled → `NOP`
- Bonus: we can patch the tracer call in the trampoline, too

```
<function>:
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp
    mov     x0, lr
    bl      _mcount  ⇔  nop
    ..
    .. // function body here
    ..
    ldp     fp, lr, [sp], #16
    ret
```


arm

Magic v1.2: mcount + return + disabling

What do we have so far?

- A mechanism to hook function entry
  - Compiler instrumentation with mcount
  - An entry trampoline (the mcount function)
- A mechanism to hook function returns
  - Only requires the entry hook to modify the return address
  - A return trampoline (the return_to_handler function)
- A mechanism to dynamically enable/disable hooks
  - Runtime instruction patching

Pointer authentication

A new challenge

- Security feature in ARMv8.3-A (~2017)
  - Protects against ROP (and JOP) attacks
- Compiler inserts two new hint instructions
  - **PACIASP** signs LR against SP
  - **AUTIASP** authenticates LR against same SP
  - Authentication failure is **fatal**

```
<function>:
    paciasp
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp
    ..
    ..
    ..
    ..
    ldp     fp, lr, [sp], #16
    autiasp
    ret
```

Pointer authentication

A new challenge

- Security feature in ARMv8.3-A (~2017)
  - Protects against ROP (and JOP) attacks
  - i.e. prevents modification of saved LR
- Compiler inserts two new hint instructions
  - **PACIASP** signs LR against SP
  - **AUTIASP** authenticates LR against same SP
  - Authentication failure is **fatal**
- **The SP changes between function entry and call to mcount!**
  - In mcount we don’t know the offset
  - mcount cannot safely change the saved **LR**
  - Incompatible with return tracing!

```
<function>:
    paciasp
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp
    ..
    mov     x0, lr
    bl      _mcount
    ..
    ldp     fp, lr, [sp], #16
    autiasp
    ret
```

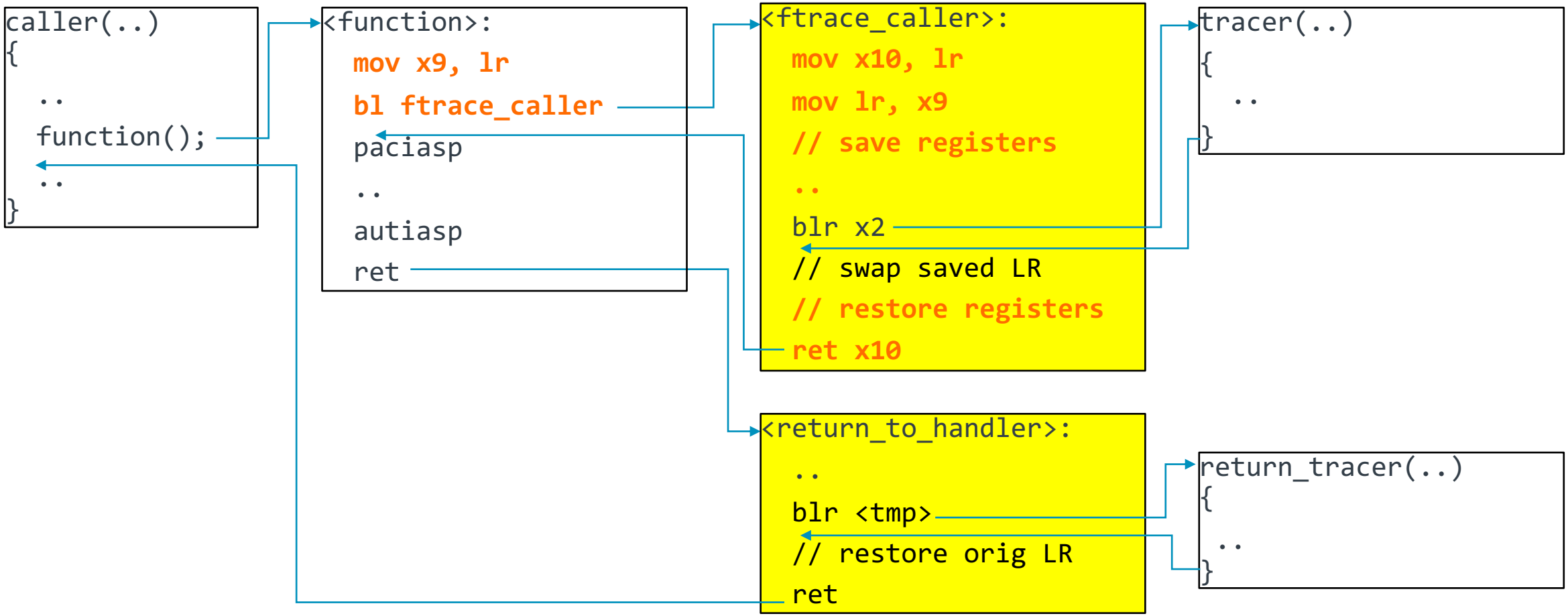
Magic v2: patchable-function-entry

- GCC 8+ supports **-fpatchable-function-entry=N**
  - Inserts N NOPs at function entry
- Compiler inserts NOPs **early** in the function
  - **Before LR is signed**
  - Before any registers are saved to the stack
- We can write our own trampoline call!
  - Save the LR into a register
  - Call the entry trampoline
  - Trampoline restores the saved LR
  - Trampoline saves registers to stack

```
<function>:
    nop  ⇒  mov x9, lr
    nop  ⇒  bl ftrace_caller
    paciasp
    stp fp, lr, [sp, #-16]
    ..
    ..
    ..
    ldp fp, lr, [sp, #16]
    autiasp
    ret
```

Magic v2: patchable-function-entry

It’s practically the same!



Magic v2: patchable-function-entry

Much better code generation!

plain

```
<get_cpu_ops>:
    adrp    x1, cpu_ops
    add     x1, x1, #:lo12:cpu_ops
    ldr     x0, [x1, w0, sxtw #3]
    ret
```

mcount

```
<get_cpu_ops>:
    stp     fp, lr, [sp, #-32]!
    mov     fp, sp
    str     x19, [sp, #16]
    mov     w19, w0
    mov     x0, x30
    bl      _mcount
    adrp    x1,  cpu_ops
    add     x1, x1, #:l012:cpu_ops
    ldr     x0, [x1, w19, sxtw #3]
    ldr     x19, [sp, #16]
    ldp     fp, lr, [sp], #32
    ret
```

patchable-function-entry

```
<get_cpu_ops>:
    nop     ⇒  mov x9, lr
    nop     ⇒  bl ftrace_caller
    adrp    x1, cpu_ops
    add     x1, x1, #:lo12:cpu_ops
    ldr     x0, [x1, w0, sxtw #3]
    ret
```

Magic v2.1: patchable-function-entry + ???

- Increasingly likely that different functions have different tracers attached:
  - Some functions might have live-patches applied
  - … and some other functions might be hooked by BPF
  - … and all functions might be traced by the graph tracer
- Our ftrace_caller trampoline can only call a single tracer
  - A special tracer (ftrace_ops_list_func) handles multiplexing tracers
  - … which iterates over all registered tracers
  - … which ends up being much more expensive than a single call
- Some architectures JIT trampolines at runtime
  - Using distinct trampolines for different functions
  - … **but this isn't feasible on arm64**

Magic v2.1: patchable-function-entry + ???

Why not JIT trampolines at runtime?

- Kernels and modules are **big**:
  - Kernels are regularly ~60MiB, debug kernels ~200+ MiB, allyesconfig kernels ~900+ MiB
  - Some people are regularly using 128+ MiB modules (seriously)
  - VA space reserved for modules is 2GiB

- **B** and **BL** have limited range: +/-128MiB
  - Need PLTs / veneers with **BR** to branch further
  - PLTs need to be generated at (dynamic) link time

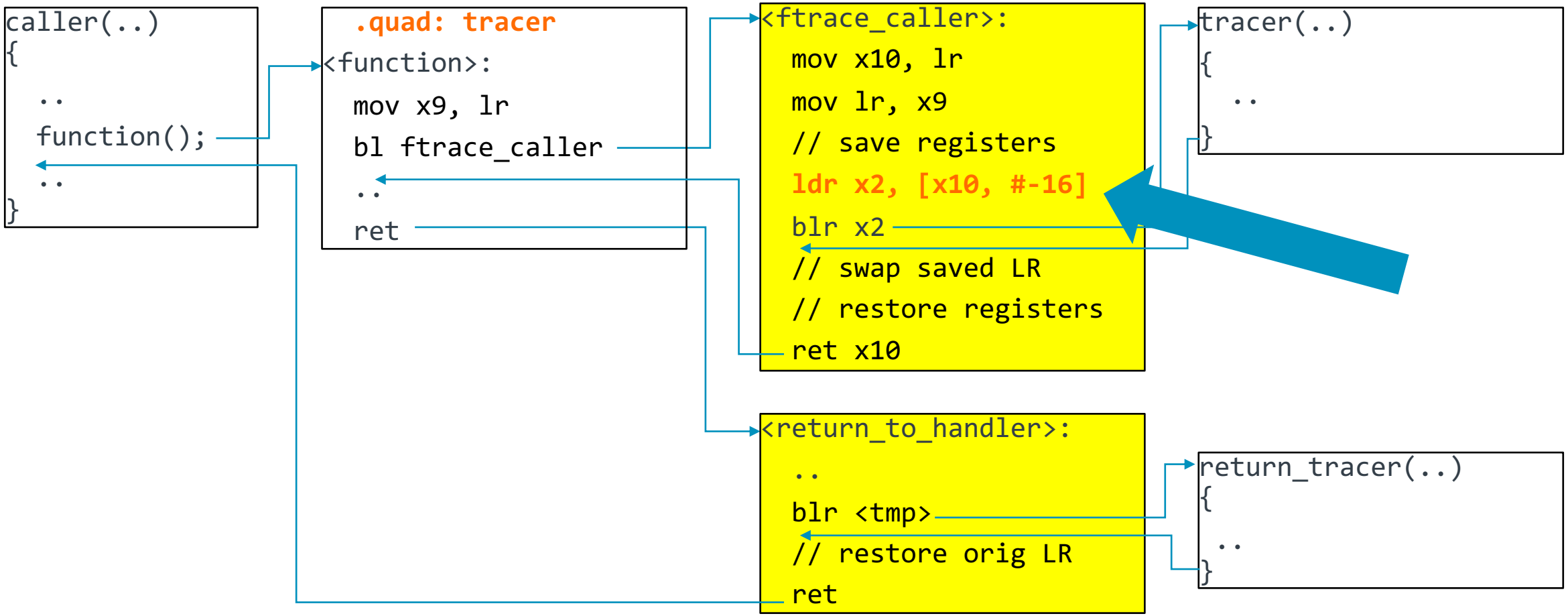
- Can’t use indirect branches to reach trampolines
  - CMODX + memory ordering + races prevent patching multiple instructions atomically
  - Live-patching multi-instruction sequences is **expensive** and **very painful**
  - … we need this to be **robust**

Magic v2.1: patchable-function-entry + per-callsite ops

- Both GCC and LLVM support **-fpatchable-function-entry=N,M**
  - Inserts M NOPs **before** function entry
  - Inserts M-N NOPs **at** function entry
- GCC and LLVM also support **-falign-functions=N**
  - Aligns function entrypoints to N bytes
- We can reserve 8 bytes **before** the function
  - We can use this for **data**, not instructions
  - Place a pointer to the tracer here
  - Patch the pointer **atomically**
  - … and have the entry trampoline find this based on the **LR**

```
<function - 8>:
    nop  ⇒  lower_32_bits(tracer)
    nop  ⇒  upper_32_bits(tracer)
<function>:
    nop  ⇒  mov x9, lr
    nop  ⇒  bl ftrace_caller
    paciasp
    stp fp, lr, [sp, #-16]
    ..
    ldp fp, lr, [sp, #16]
    autiasp
    ret
```

Magic v2: patchable-function-entry + per-callsite ops



Magic v2: patchable-function-entry + per-callsite ops

What have we managed to do?

- Made per-callsite tracers much **cheaper**
  - Most callsites never need to walk the list of all tracers
- **Avoided arbitrary limitations**
  - No need to allocate executable memory within branch range
  - If a function is traceable, **any** tracer can trace it
- Kept the logic **simple** and **robust**
  - No need to JIT trampolines
  - Only need to patch one branch and one pointer per callsite
  - Changes to entry trampoline are **trivial**
- Made future work possible
  - Simple to extend to support direct calls

Finally

What have I missed?

- A few other changes
  - Benchmarking with the ftrace-ops module
  - Replacing regs with args-only
- Direct calls got implemented!
  - Used by BPF so far
- Thanks to various people
  - **Steve Rostedt** and **Masami HIRAMATSU**: ftrace maintainers
  - **AKASHI Takahiro**: original arm64 ftrace implementation
  - **Torsten Duwe**: patchable-function-entry (GCC & Linux)
  - **Fangrui Song**: patchable-function-entry (LLVM)
  - **Xu Kuohai**: early attempts at arm64 trampolines & direct calls
  - **Florent Revest**: arm64 direct calls

arm

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה

Additional material

arm64

Unconditional branch instructions

Branch

- **`B <label>`**
  Branch to **PC** +/-128MiB)

- **`BR <Xn>`**
  Branch to register **Xn**

Branch-with-link

- **`BL <label>`**
  Place **PC** + 4 into **LR**
  Branch to (**PC** +/-128MiB)

- **`BLR <Xn>`**
  Place **PC** + 4 into **LR**
  Branch to register **Xn**

Return

- **RET**
  Return to register **LR**

- **RET <Xn>**
  Return to register **Xn**

arm64

Registers

General Purpose

- **R0** … **R30**
  - **X0** … **X30** – 64-bit aliases
  - **W0** … **W30** – 32-bit aliases
- **FP** – Frame Pointer (alias of X29)
- **LR** – Link Register (alias of X30)

Fixed Purpose

- **ZR** – Zero register
  - **XZR** – 64-bit alias
  - **WZR** – 32-bit alias
- **SP** – Stack Pointer
- **PC** – Program Counter

Special

- **NZCV** – Condition flags
- **DAIF** – Interrupt masks
- […] – and *many* more

arm64

Registers in AAPCS64

Register	Alias	Purpose	Notes
R0 – R7		Arguments & return values	Caller-saved
R8		Indirect result location	Caller-saved
R9 – R15		Temporary	Caller-saved
R16 / R17	IP0 / IP1	Temporary	Volatile (clobbered by PLTs)
R18		Temporary / Platform (e.g. shadow call stack)	Caller-saved / Fixed
R19 – R28		Temporary	Callee-saved
R29	FP	Frame Pointer	Callee-saved
R30	LR	Link Register	-
SP		Stack Pointer	Callee-saved

arm64

CMODX: Concurrent MODification and eXecution of instructions

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where each of the instruction before modification and the instruction after modification is one of a `B`, `B.cond`, `BL`, `BRK`, `CBNZ`, `CBZ`, `HVC`, `ISB`, `NOP`, `SMC`, `SVC`, `TBNZ` or `TBZ` instruction.

For the `B`, `B.cond`, `BL`, `BRK`, `CBNZ`, `CBZ`, `HVC`, `ISB`, `NOP`, `SMC`, `SVC`, `TBNZ` and `TBZ` instructions, the architecture guarantees that after modification of the instruction, behavior is consistent with execution of either:

- The instruction originally fetched.
- A fetch of the modified instruction.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed.

Arm® Architecture Reference Manual (ARM DDI 0487J.a)
Section B2.2.5

Instrumentation comparison

Kernel Image size

ftrace	Image size	Text size	Data Size	Relocations size
<none>	39,707,136	24,509,194	15,043,038	6,854,656
mcount	47,671,808 (+20%)	29,777,838 (+21%)	17,663,080 (+17%)	9,431,040 (+38%)
PFE	46,885,376 (+18%)	28,988,138 (+18%)	17,663,260 (+17%)	9,431,040 (+38%)
PFE + call-ops	47,475,200 (+19%)	29,626,920 (+21%)	17,664,576 (+17%)	9,431,040 (+38%)

GCC 12.2.0, Linux 6.4, defconfig + CONFIG_ARM64_PTR_AUTH_KERNEL=n