Fast (Trapless) Kernel Probes Everywhere

Jinghao Jia, Michael V. Le, Salman Ahmed,
Dan Williams, Hani Jamjoom, Tianyin Xu

Kernel probes as an observability primitive

- Dynamic instrumentation on any kernel instructions
  - Custom handler functions
  - No re-build / reboot required
  - Widely used in tracing and debugging

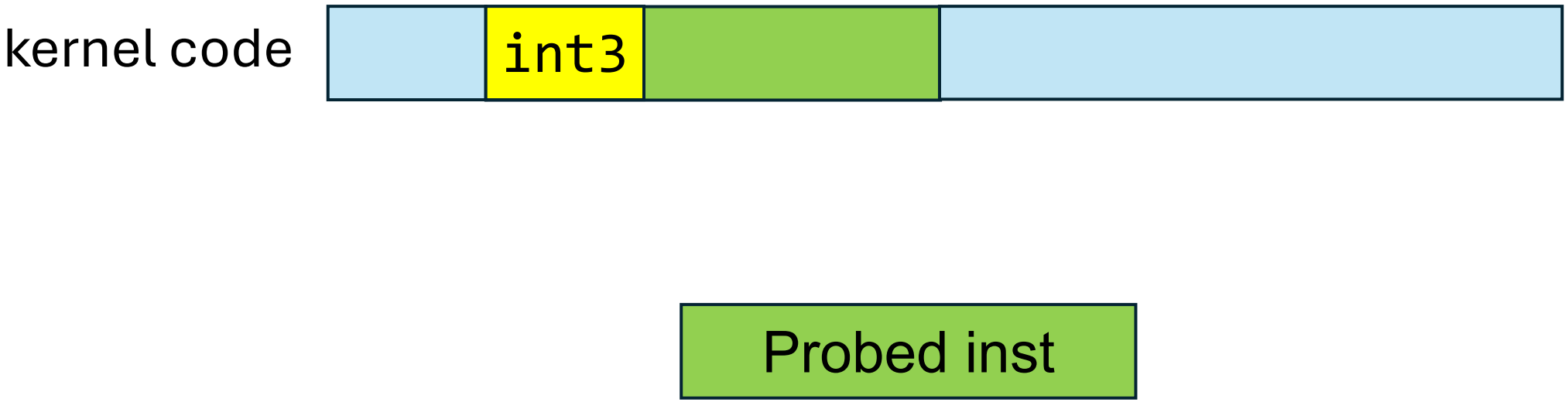
Kernel probes as an observability primitive

- Dynamic instrumentation on any kernel instructions
  - Custom handler functions
  - No re-build / reboot required
  - Widely used in tracing and debugging

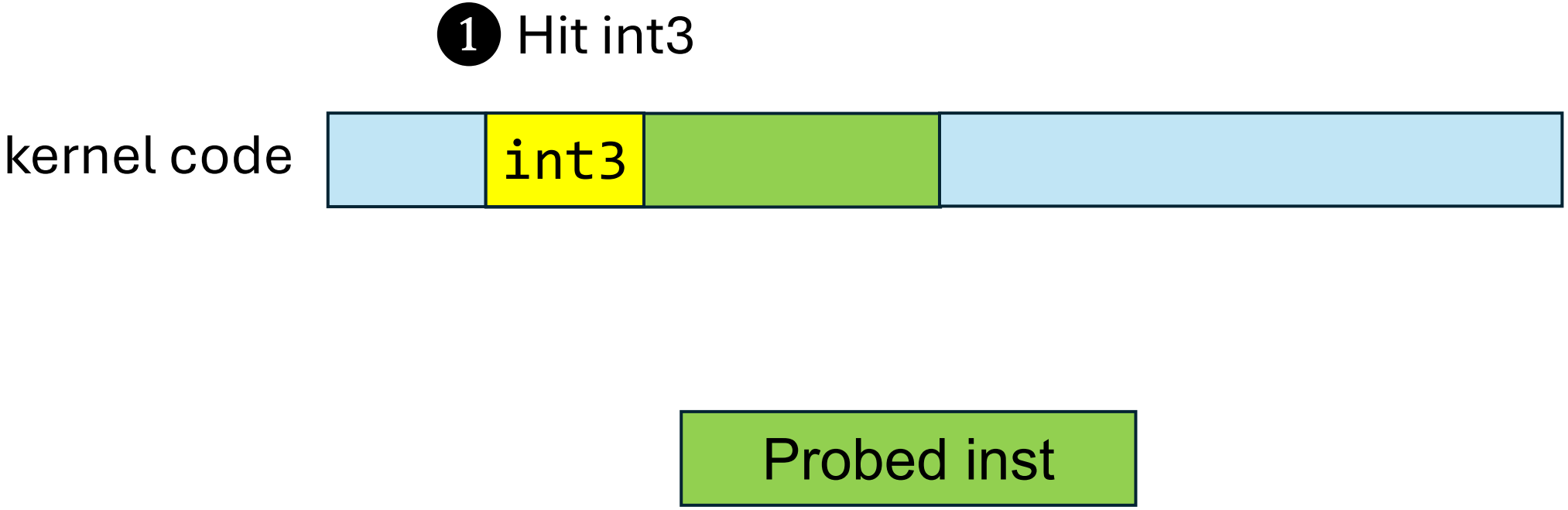
- Performance is important!
  - Our use case: **Kernel Control Flow Integrity (KCFI)**
  - Validating indirect control flow transfers
  - One kprobe on each indirect call instruction

Basic kprobes are implemented by traps

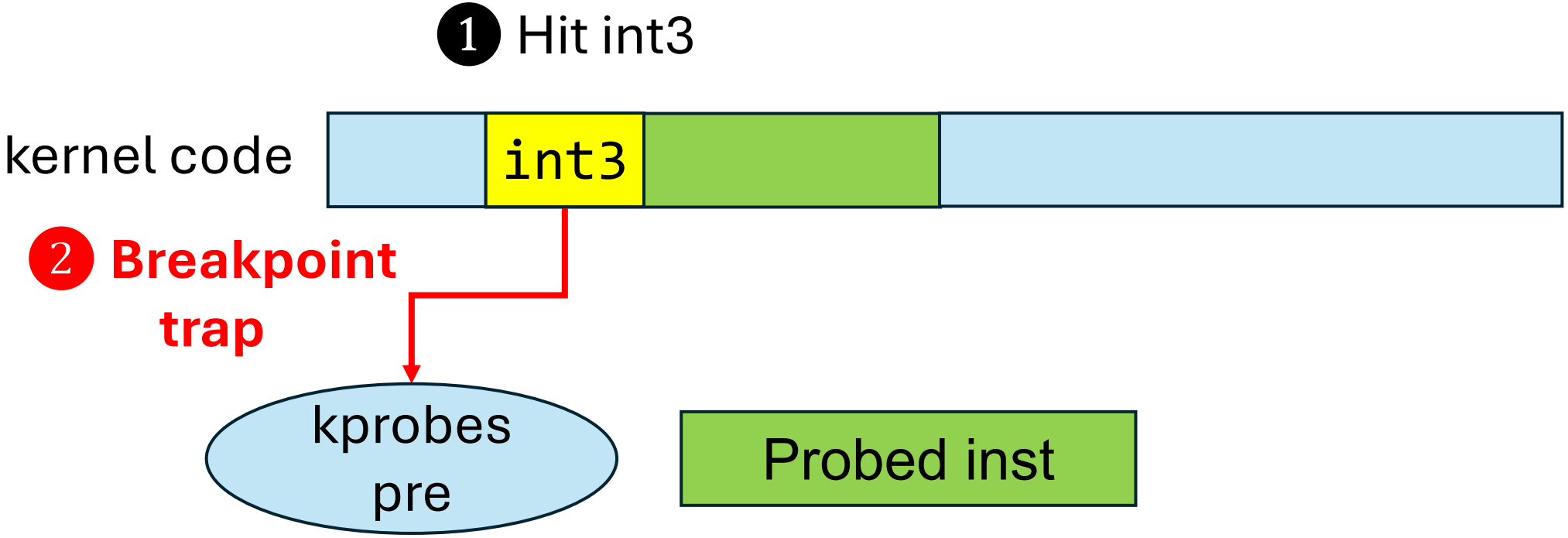
Basic kprobes are implemented by traps



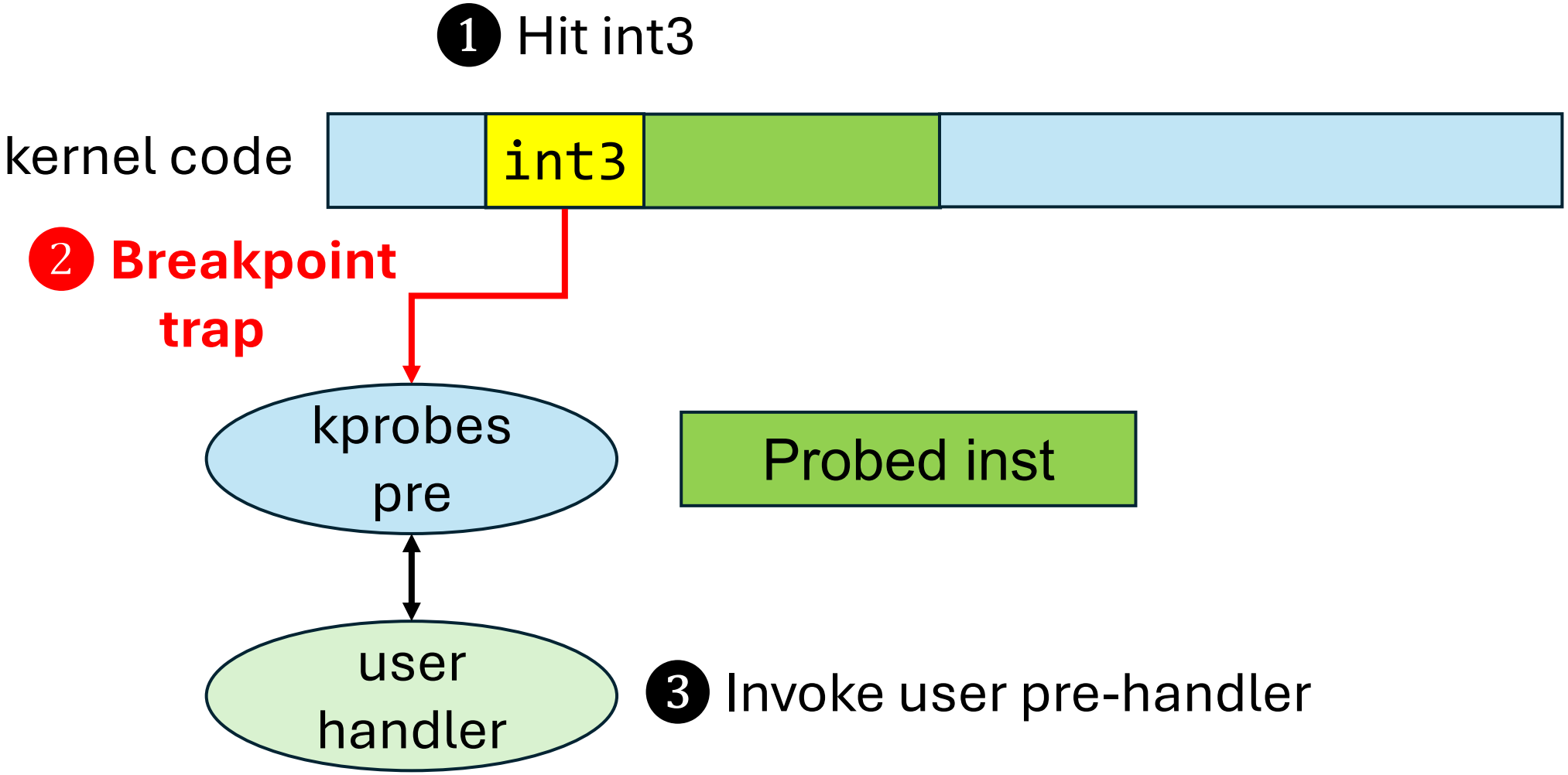
Basic kprobes are implemented by traps



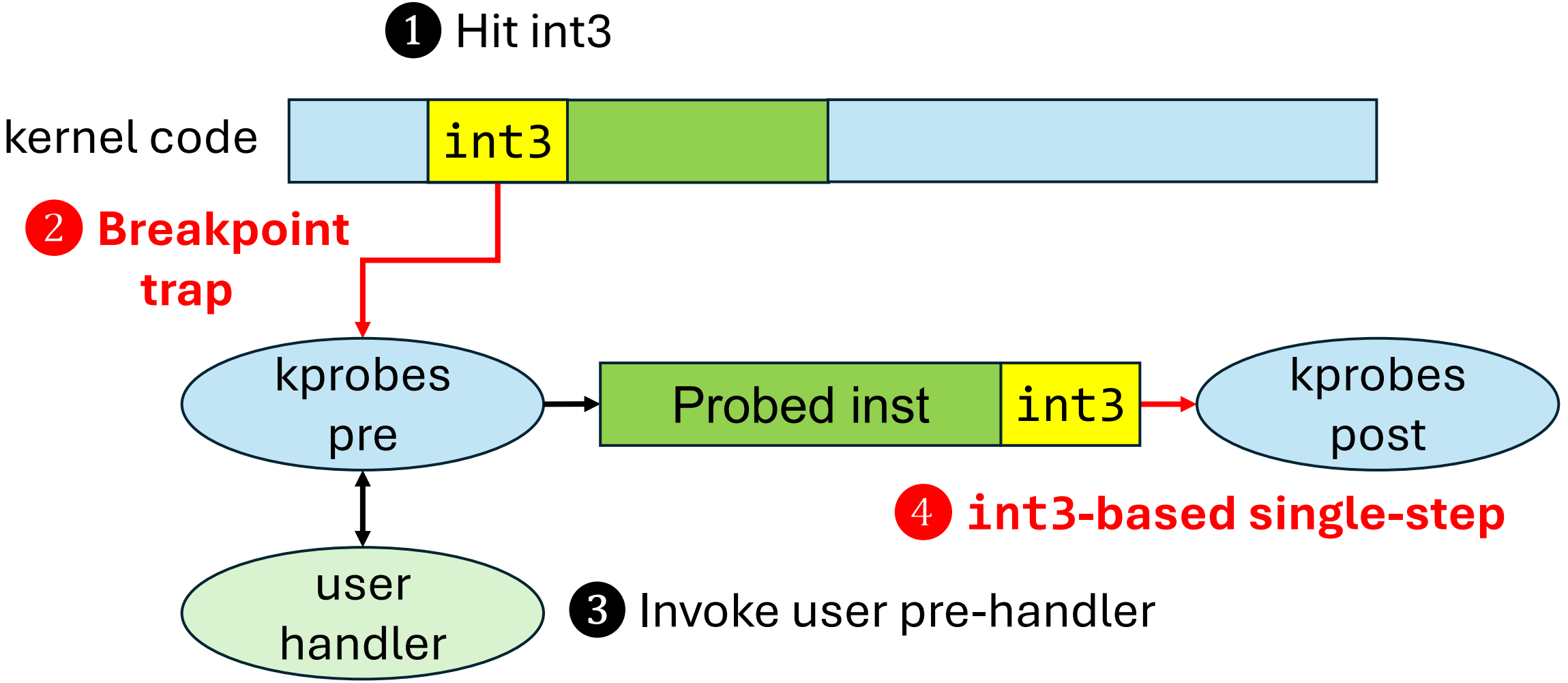
Basic kprobes are implemented by traps



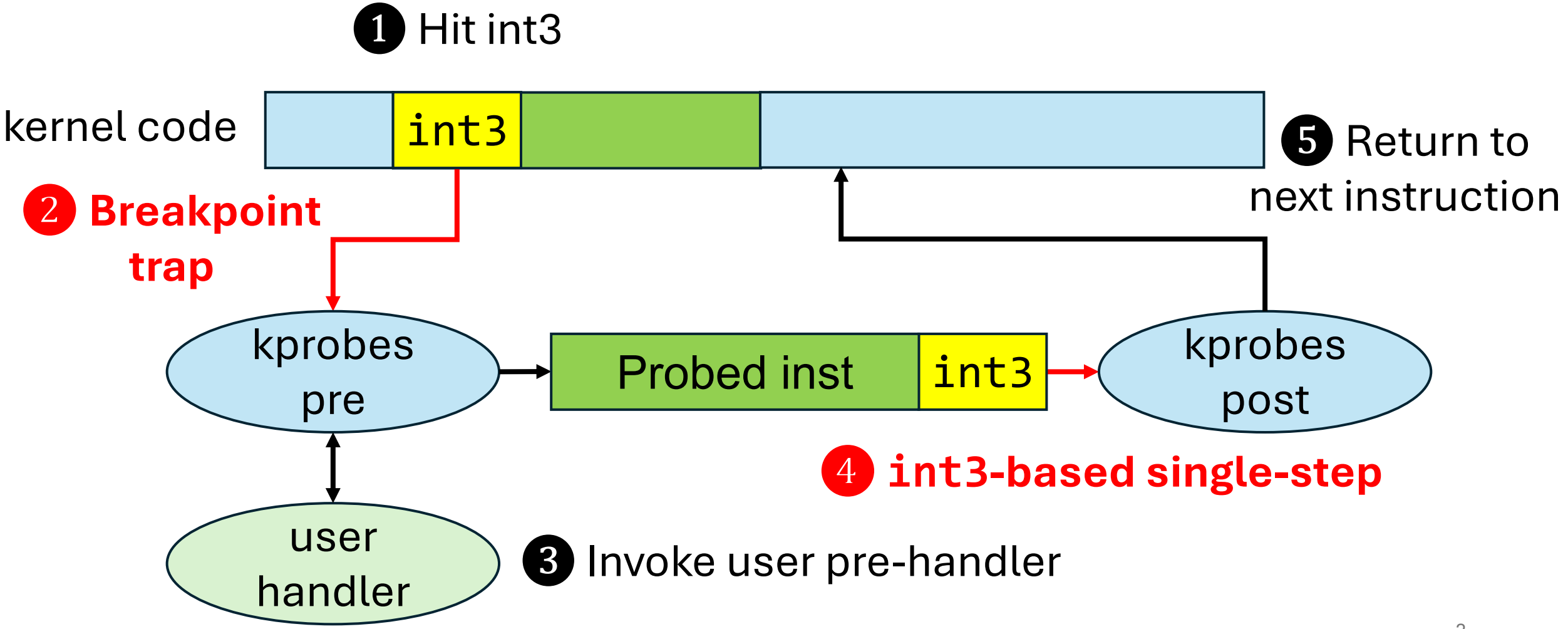
Basic kprobes are implemented by traps



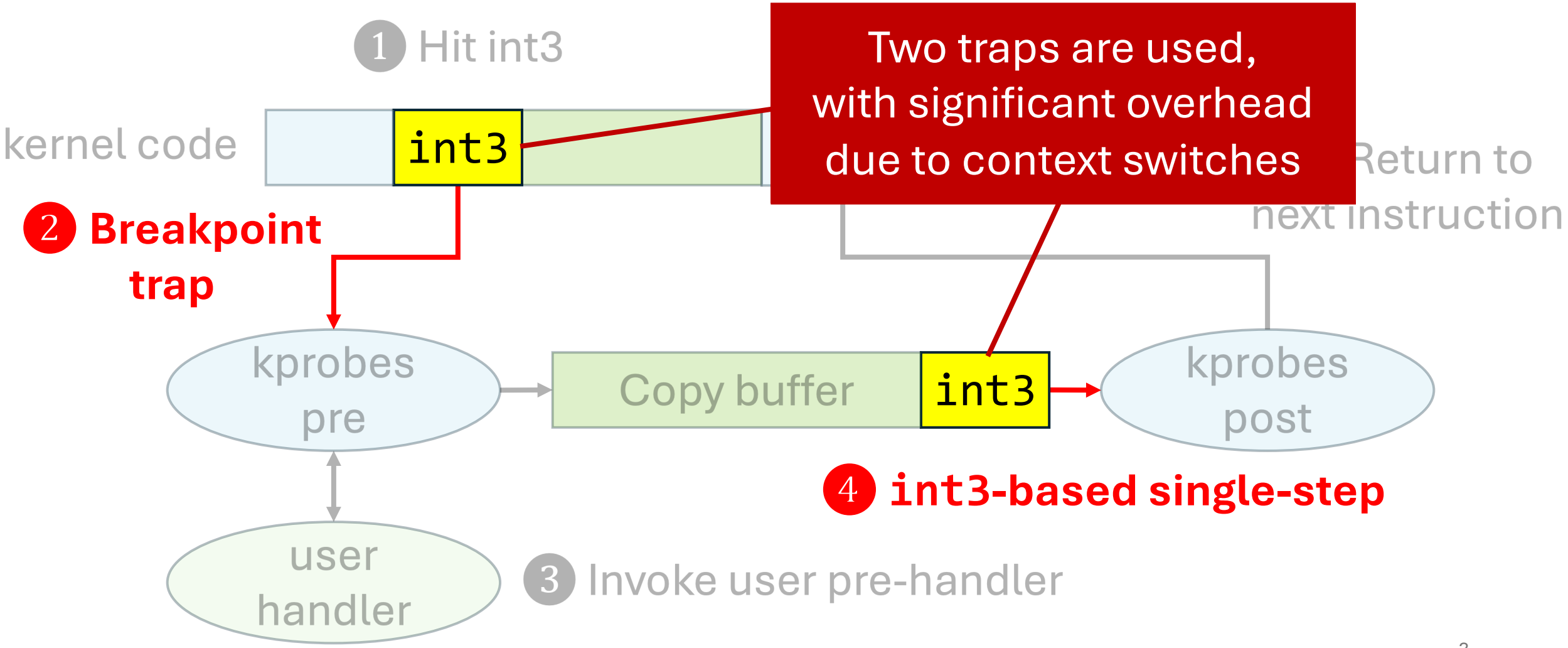
Basic kprobes are implemented by traps



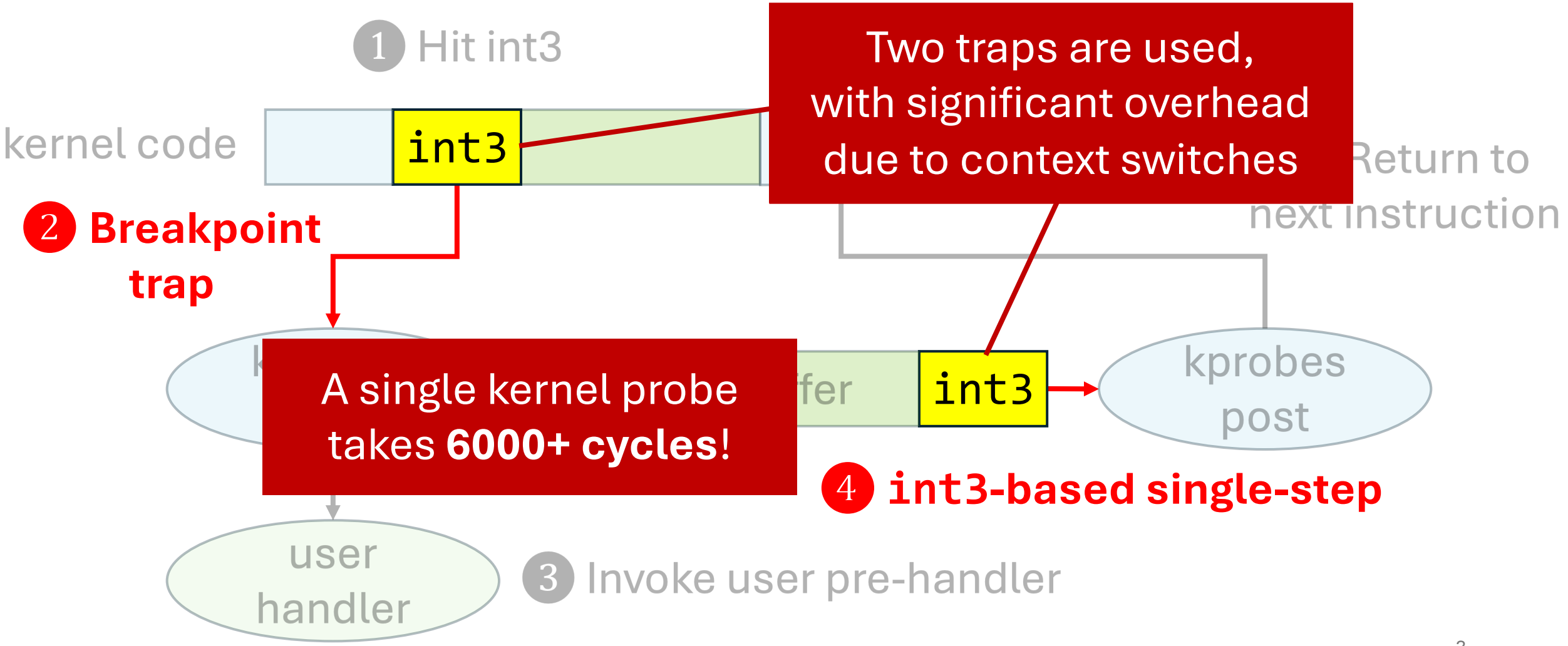
Basic kprobes are implemented by traps



Trap-based kprobes are too slow



Trap-based kprobes are too slow



How to build a fast kernel probe mechanism (with no trap)?

Contributions

- Uno-kprobe: a **fast, universally trapless** kernel probe mechanism
  - Single-probe performance increased by a factor of 10x
  - 3x for kprobe-based KCFI enforcements

- An implementation of Uno-kprobe on top of x86 Linux-kprobe
  - Address fundamental limitations of Linux kprobe optimizations
  - Code available: github.com/hardos-ebpf-fuzzing/atc24-uno-kprobe

Design principle of *trapless* kprobes

- Replace traps with direct control flow transfer instructions
  - E.g., `call` and jmp

Design principle of *trapless* kprobes

- Replace traps with direct control flow transfer instructions
  - E.g., `call` and jmp
- Strawman approach: use nops to allocate extra space
  - Insert a 5-byte nop before every kernel instruction

Design principle of *trapless* kprobes

- Replace traps with direct control flow transfer instructions
  - E.g., `call` and jmp

- Strawman approach: use nops to allocate extra space
  - Insert a 5-byte nop before every kernel instruction
  - Rewrite nop into a 5-byte `call` instruction

Design principle of *trapless* kprobes

- Replace traps with direct control flow transfer instructions
  - E.g., `call` and jmp
- Strawman approach: use nops to allocate extra space
  - Insert a 5-byte nop before every kernel instruction
  - Rewrite nop into a 5-byte `call` instruction
- Transfer control flow to a trampoline
  - Trampoline is responsible for setting up a call to the handler

Design principle of *trapless* kprobes

- Replace traps
  - E.g., `call` ar

```
83 ff 02              cmp  $0x2,%edi
75 05                 jne  0xa
ba 00 01 00 00        mov  $0x100,%edx
ff c7                 inc  %edi
```

tructions

- Strawman approach: use nops to allocate extra space
  - Insert a 5-byte nop before every kernel instruction
  - Rewrite nop into a 5-byte `call` instruction

- Transfer control flow to a trampoline
  - Trampoline is responsible for setting up a call to the handler

Design principle of *trapless* kprobes

- Replace traps
  - E.g., call ar
- Strawman ap
  - Insert a 5-by
  - Rewrite nop
- Transfer control flow to a trampoline
  - Trampoline is responsible for setting up a call to the handler

```
0f 1f 44 00 08     nop   0x8(%rax,%rax,1)
83 ff 02           cmp   $0x2,%edi
0f 1f 44 00 08     nop   0x8(%rax,%rax,1)
75 05              jne   0xa
0f 1f 44 00 08     nop   0x8(%rax,%rax,1)
ba 00 01 00 00     mov   $0x100,%edx
0f 1f 44 00 08     nop   0x8(%rax,%rax,1)
ff c7              inc   %edi
```

Design principle of *trapless* kprobes

- Replace traps
  - E.g., call a
- Strawman ap
  - Insert a 5-by
  - Rewrite nop
- Transfer cont
  - Trampoline i

```
0f 1f 44 00 08     nop  0x8(%rax,%rax,1)
83 ff 02           cmp  $0x2,%edi
0f 1f 44 00 08     nop  0x8(%rax,%rax,1)
75 05              jne  0xa
e8 d7 00 00 01     call *0x10000d7(%rip)
ba 00 01 00 00     mov  $0x100,%edx
0f 1f 44 00 08     nop  0x8(%rax,%rax,1)
ff c7              inc  %edi
```

```
kprobe_trampoline:
... # save registers
call kprobe_handler
... # restore registers
ret
```

Design principle of *trapless* kprobes

- Replace traps
  - E.g., call a
- Strawman ap
  - Insert a 5-by
  - Rewrite nop
- Tr

```
0f 1f 44 00 08     nop  0x8(%rax,%rax,1)
83 ff 02           cmp  $0x2,%edi
0f 1f 44 00 08     nop  0x8(%rax,%rax,1)
75 05              jne  0xa
e8 d7 00 00 01     call *0x10000d7(%rip)
ba 00 01 00 00     mov  $0x100,%edx
0f 1f 44 00 08     nop  0x8(%rax,%rax,1)
ff c7              inc  %edi
```

```
kprobe_trampoline:
    # save registers
```

Too many nops lead to performance issues!
~30% overhead measured on LEBench

Majority of nops are unnecessary

- In many cases, the target instruction can be rewritten into a `jmp` instruction without needing extra space from nops

Majority of nops are unnecessary

- In many cases, the target instruction can be rewritten into a `jmp` instruction without needing extra space from nops

```
83 ff 02              cmp   $0x2,%edi
75 05                 jne   0xa
ba 00 01 00 00        mov   $0x100,%edx
ff c7                 inc   %edi
```

Majority of nops are unnecessary

- In many cases, the target instruction can be rewritten into a `jmp` instruction without needing extra space from nops

```
83 ff 02              cmp   $0x2,%edi
75 05                 jne   0xa
e9 d7 00 00 01        jmp   *0x10000d7(%rip)
ff c7                 inc   %edi
```

```
per_kprobe_trampoline:
... # save registers
call kprobe_handler
... # restore registers
mov  $0x100,%edx
jmp  ...
```

❶ Jump to trampoline

Majority of nops are unnecessary

- In many cases, the target instruction can be rewritten into a `jmp` instruction without needing extra space from nops

```
83 ff 02              cmp   $0x2,%edi
75 05                 jne   0xa
e9 d7 00 00 01        jmp   *0x10000d7(%rip)
ff c7                 inc   %edi
```

```
per_kprobe_trampoline:
... # save registers
call kprobe_handler
... # restore registers
mov  $0x100,%edx
jmp  ...
```

1. Jump to trampoline
2. Invoke user handler

Majority of nops are unnecessary

- In many cases, the target instruction can be rewritten into a `jmp` instruction without needing extra space from nops

```
83 ff 02              cmp   $0x2,%edi
75 05                 jne   0xa
e9 d7 00 00 01        jmp   *0x10000d7(%rip)
ff c7                 inc   %edi
```

```
per_kprobe_trampoline:
... # save registers
call kprobe_handler
... # restore registers
mov  $0x100,%edx
jmp  ...
```

1. Jump to trampoline
2. Invoke user handler
3. Executed copied probe target

Majority of nops are unnecessary

- In many cases, the target instruction can be rewritten into a `jmp` instruction without needing extra space from nops

```
83 ff 02              cmp   $0x2,%edi
75 05                 jne   0xa
e9 d7 00 00 01        jmp   *0x10000d7(%rip)
ff c7                 inc   %edi
```

```
per_kprobe_trampoline:
... # save registers
call kprobe_handler
... # restore registers
mov  $0x100,%edx
jmp  ...
```

1. Jump to trampoline
2. Invoke user handler
3. Executed copied probe target
4. Jump back to next instruction

Such rewriting does not always work

1. No enough space to avoid overwriting other branch targets
   - Inserted `jmp` spans **basic block (BB) boundaries**
   - Breaking instruction decoding

Such rewriting does not always work

1. No enoug anch targets
   - Inserted j
   - Breaking i

```
0: 83 ff 02          cmp  $0x2,%edi
3: 75 04             jne  0x9
5: 41 80 c0 01       add  $0x1,%r8b
9: ff c7             inc  %edi
```

Such rewriting does not always work

1. No enoug anch targets
   - Inserted j
   - Breaking i

```
0: 83 ff 02          cmp  $0x2,%edi
3: 75 04             jne  0x9
5: 41 80 c0 01       add  $0x1,%r8b
9: ff c7             inc  %edi
```

jne jumps here if taken

Such rewriting does not always work

1. No enoug anch targets
   - Inserted j
   - Breaking i

```
0: 83 ff 02          cmp  $0x2,%edi
3: 75 04             jne  0x9
5: 41 80 c0 01       add  $0x1,%r8b
9: ff c7             inc  %edi
```

```
0: 83 ff 02          cmp  $0x2,%edi
3: 75 04             jne  0x9
5: e9 d7 00 00 01    jmp  *0x10000d7(%rip)
a: c7 ...            ???
```

Such rewriting does not always work

1. No enoug anch targets
   - Inserted j
   - Breaking i

```
0: 83 ff 02          cmp  $0x2,%edi
3: 75 04             jne  0x9
5: 41 80 c0 01       add  $0x1,%r8b
9: ff c7             inc  %edi
```

```
0: 83 ff 02          cmp  $0x2,%edi
3: 75 04             jne  0x9
5: e9 d7 00 00 01    jmp  *0x10000d7(%rip)
a: c7 ...            ???
```

jne lands on 01 byte, not an instruction boundary!

Such rewriting does not always work

1. No enough space to avoid overwriting other branch targets
   - Inserted `jmp` spans **basic block (BB) boundaries**
   - Breaking instruction decoding

2. Address-dependent instructions
   - Text addresses of these instructions matter
   - RIP-related instructions often use current address to calculate offset
   - Kernel uses text address to handle page-fault triggering instructions

Uno-kprobe Overview

- *Selectively* insert nops at places that cannot be directly rewritten
  - No enough space to avoid overwriting other branch targets
  - Address-dependent instructions

Uno-kprobe Overview

- *Selectively* insert nops at places that cannot be directly rewritten
  - No enough space to avoid overwriting other branch targets
  - Address-dependent instructions

- Perform nop insertions using an LLVM Machine IR (MIR) pass
  - Basic blocks are explicit
  - Closely models native code -- more information and control over code generation

Uno-kprobe Overview

- *Selectively* insert nops at places that cannot be directly rewritten
  - No enough space to avoid overwriting other branch targets
  - Address-dependent instructions
- Perform nop insertions using an LLVM Machine IR (MIR) pass
  - Basic blocks are explicit
  - Closely models native code -- more information and control over code generation
- Integrate with current x86 Linux-kprobe and its optimizations

Uno-kprobe Overview

- *Selectively* insert nops at places that cannot be directly rewritten
  - No enough space to avoid overwriting other branch targets
  - Address-dependent instructions
- Perform nop insertions using an LLVM Machine IR (MIR) pass
  - Basic blocks are explicit
  - Closely models native code -- more information and control over code generation
- Integrate with current x86 Linux-kprobe and its optimizations
- Implement a more space-efficient kprobe trampoline

Selectively inserting nop using LLVM

```
bb.0.entry:
    FENTRY_CALL
    $eax = MOV32rr $esi
    CMP32ri8 killed renamable $edi, 2, ...
    JCC_1 %bb.2, 5, implicit $eflags

bb.1.if.then:
    renamable $eax = ADD32rr killed ...

bb.2.if.end:
    RET64 $eax
```

Selectively inserting nop using LLVM

Address-dependent instructions

```
bb.0.entry:
    FENTRY_CALL
    $eax = MOV32rr $esi
    CMP32ri8 killed renamable $edi, 2, ...
    KPROBE_NOP
    JCC_1 %bb.2, 5, implicit $eflags

bb.1.if.then:
    renamable $eax = ADD32rr killed ...

bb.2.if.end:
    KPROBE_NOP
    RET64 $eax
```

Selectively inserting nop using LLVM

Address-dependent instructions

Not enough space to avoid overwriting other branch targets

```
bb.0.entry:
    FENTRY_CALL
    $eax = MOV32rr $esi
    CMP32ri8 killed renamable $edi, 2, ...
    KPROBE_NOP
    JCC_1 %bb.2, 5, implicit $eflags

bb.1.if.then:
    KPROBE_NOP
    renamable $eax = ADD32rr killed ...

bb.2.if.end:
    KPROBE_NOP
    RET64 $eax
```

Integration with Linux's optimizations

- Linux-kprobe implements ad hoc `jmp`-based optimizations
  - **~21%** of the kernel instructions are not eligible for trapless kprobes

- We apply Uno-kprobe design on top of Linux-kprobe
  - Respect existing optimization and focus on unoptimizable cases

Space-efficient Uno-kprobe trampoline

- A trampoline is required for trapless kprobe mechanisms
  - Saves and restores register contexts
  - Prevents handlers from corrupting current context

Space-efficient Uno-kprobe trampoline

- A trampoline is required for trapless kprobe mechanisms
  - Saves and restores register contexts
  - Prevents handlers from corrupting current context

- The trampoline in Linux's `jmp`-based optimization is not scalable

Space-efficient Uno-kprobe trampoline

- A trampoline is required for trapless kprobe mechanisms
  - Sa…ter contexts
  - Pr…orrupting current context

```
jne   0xa
mov   $0x100,%edx
inc   %edi
```

- The trampoline in Linux's `jmp`-based optimization is not scalable

Space-efficient Uno-kprobe trampoline

```
jne   0xa
jmp   *0x10000d7(%rip)
inc   %edi
```

```
per_kprobe_trampoline:
... # save registers
call kprobe_handler
... # restore registers
mov  $0x100,%edx
jmp ... # jump back
```

- Copied probe target implies per-kprobe trampoline
- Memory usage scales linearly

Space-efficient Uno-kprobe trampoline

- A trampoline is required for trapless
  - Sa ter context
  - Pr orrupting c
- The is not scalable

```
jne   0xa
jmp   *0x10000d7(%rip)
inc   %edi
```

```
per_kprobe_trampoline:
... # save registers
call kprobe_handler
... # restore registers
mov  $0x100,%edx
    back
```

- Uno-kprobe executes probe target in place
- Global trampoline taking constant 96 bytes

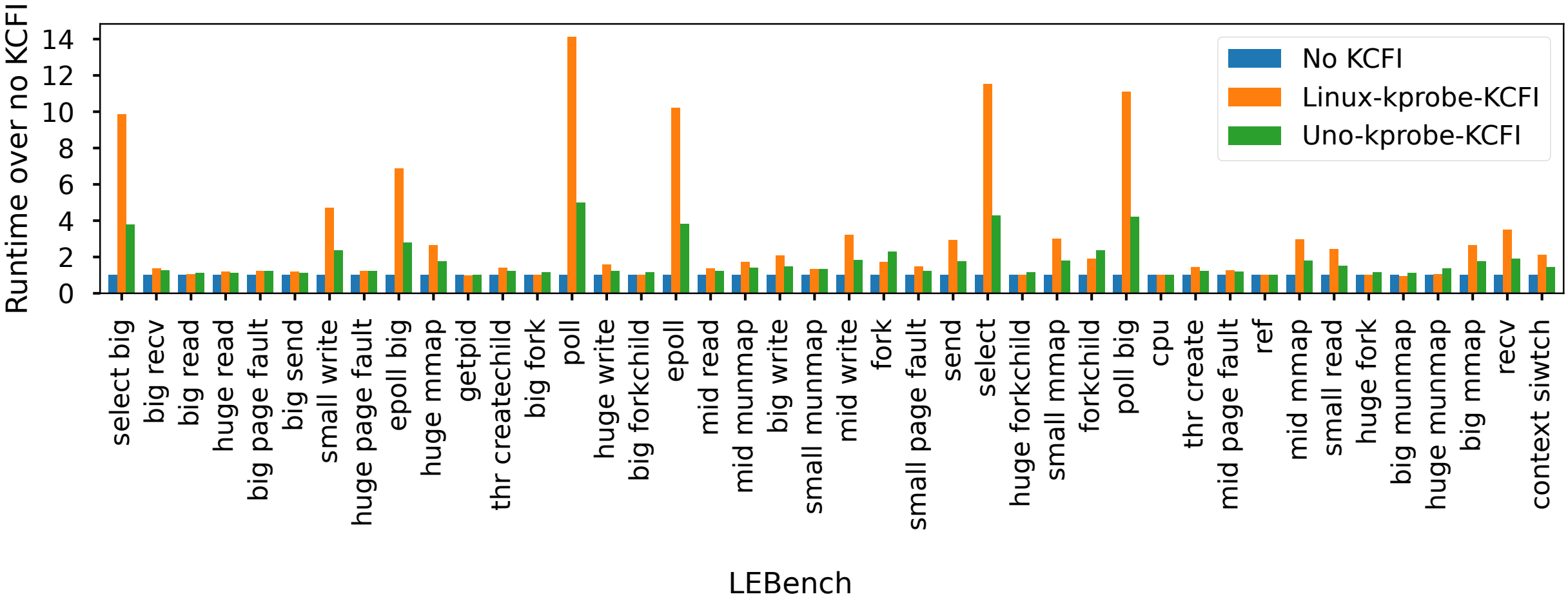
```
jne   0xa
call *0x10000d7(%rip)
mov   $0x100,%edx
inc   %edi
```

```
kprobe_trampoline:
... # save registers
call kprobe_handler
... # restore registers
ret
```

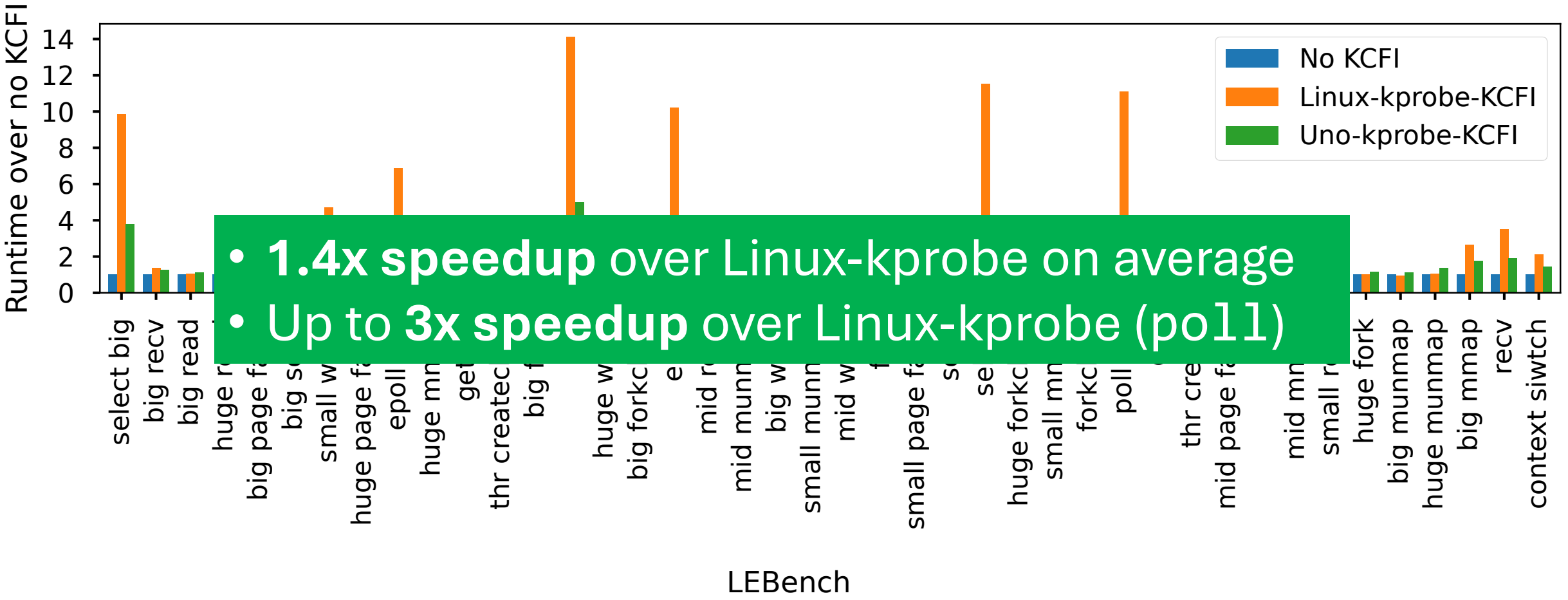
Evaluation

- Coverage: **96%** of kernel code can enjoy trapless kprobe
  - Up from 79% in Linux-kprobe
  - Remaining 4% cannot be processed trivially in LLVM
    - Assembly source code (inline-asm and assembly source files)
    - Page-fault-triggering instructions
- Single-probe performance
  - Non-optimizable instruction in Linux-kprobe: **10x**
- Overhead from nops : **10%** on average on LEBench

Performance Improvement of KCFI



Performance Improvement of KCFI



- **1.4x speedup** over Linux-kprobe on average
- Up to **3x speedup** over Linux-kprobe (poll)

Upstreamed optimizations and bug fixes

- Merged and released in Linux v6.9

x86/kprobes: Refactor can_{probe,boost} return type to bool

```
Both can_probe and can_boost have int return type but are using int as
boolean in their context.

Refactor both functions to make them actually return boolean.

Link: https://lore.kernel.org/all/20240204031300.830475-2-jinghao7@illi

Signed-off-by: Jinghao Jia <jinghao7@illinois.edu>
Acked-by: Masami Hiramatsu (Google) <mhiramat@kernel.org>
```

x86/kprobes: fix incorrect return address calculation in kprobe_emulate

```
kprobe_emulate_call_indirect currently uses int3_emulate_call to emulat
indirect calls. However, int3_emulate_call always assumes the size of
the call to be 5 bytes when calculating the return address. This is
incorrect for register-based indirect calls in x86, which can be either
2 or 3 bytes depending on whether REX prefix is used. At kprobe runtime
the incorrect return address causes control flow to land onto the wrong
place after return -- possibly not a valid instruction boundary. This
can lead to a panic like the following:
```

x86/kprobes: Boost more instructions from grp2/3/4/5

```
With the instruction decoder, we are now able to decode and recognize
instructions with opcode extensions. There are more instructions in
these groups that can be boosted:

Group 2: ROL, ROR, RCL, RCR, SHL/SAL, SHR, SAR
Group 3: TEST, NOT, NEG, MUL, IMUL, DIV, IDIV
Group 4: INC, DEC (byte operation)
Group 5: INC, DEC (word/doubleword/quadword operation)
```

x86/kprobes: Prohibit kprobing on INT and UD

```
Both INT (INT n, INT1, INT3, INTO) and UD (UD0, UD1, UD2) serve special
purposes in the kernel, e.g., INT3 is used by KGDB and UD2 is involved
in LLVM-KCFI instrumentation. At the same time, attaching kprobes on
these instructions (particularly UD) will pollute the stack trace dumped
in the kernel ring buffer, since the exception is triggered in the copy
buffer rather than the original location.

Check for INT and UD in can_probe and reject any kprobes trying to
attach to these instructions.
```

Conclusion

- Uno-kprobe, a universally trapless kernel probe mechanism
- An implementation of Uno-kprobe on top of Linux-kprobe
  - Integrating with the existing optimizations
- Uno-kprobe covers 96% of Linux kernel code
  - A single kprobe runs 10x faster
- Code: github.com/hardos-ebpf-fuzzing/atc24-uno-kprobe