ISOVALENT
now part of CISCO

eBPF Hookpoint Gotchas:
Why Your Program Fires (or Fails) in Unexpected Ways

Chris Tarazi, Sr. Staff Software Engineer
Donia Chaiehloudj, Software Engineer and Community

Overview

- Tracing hookpoint overview
- Generic gotchas:
  - Kernel versions
  - Different architectures
  - Performance
- uprobe dynamic links gotchas
- kprobe/uprobe inlining gotchas
- Nested executions gotchas

Tracing hookpoint overview

Hookpoint	Scope	Stability	Kernel version	Mechanism	Use case
kprobe	*Any kernel function entry / exit, offsets	No guarantees	2.6.9	Various (interrupt-based, optimization exists)	Dynamic debugging, kernel internals
uprobe	Userspace function entry / exit, offsets	No guarantees	3.5	Same as kprobe (userspace traps into kernel)	Dynamic debugging, application internals
fprobe	Function entry / exit	Same as kprobe/uprobe	5.5	ftrace (NOP patching + trampoline)	More efficient kernel function tracing
tracepoint	Kernel static points	Designed to not break (exceptions possible)	2.6.28	NOP patching	Production tracing and performance accounting

∗Inlined functions, blacklisted functions, or functions marked as notrace are excluded from probes.

Hookpoints performance overview

Empty probe attached:

Case	ns/op	Overhead ns/op	Overhead percent
no probe attached	117	0	0%
tracepoint empty	132	15	13%
fentry empty	141	24	21%
kprobe empty	254	137	117%

Probe with a simple map increment attached:

Case	ns/op	Overhead ns/op	Overhead percent
no probe attached	117	0	0%
tracepoint simple	152	35	30%
fentry simple	159	42	36%
kprobe simple	277	160	136%

Probe with a complex map increment attached:

Case	ns/op	Overhead ns/op	Overhead percent
no probe attached	117	0	0%
tracepoint complex	213	96	82%
fentry complex	220	103	88%
kprobe complex	346	229	196%

Hookpoint ranking

- 🥇 tracepoints
- 🥈 fprobes
- 🥉 kprobes (🦥 slow due to interrupt-based implementation)

- note:
  - overhead depends on whether function is in the hot path or not
  - overhead may not be direct translation to system slowdown

source: https://mastodon.ivan.computer/@mastodon/110737250286611183

ISOVALENT now part of cisco

Gotcha #1 Kernel versions

Gotcha #1: Kernel versions

kprobes, fprobes have no stability guarantee:

Gotcha #1: Kernel versions

kprobes, fprobes have no stability guarantee:

- **kernel internal functions can change with any new release:**
  - function is renamed. removed, inlined: Failed to attach because the symbol not found
  - function arguments changed, struct field changed: Reads the wrong argument → garbage data

Gotcha #1: Kernel versions

kprobes, fprobes have no stability guarantee:

- **kernel internal functions can change with any new release:**
  - function is renamed. removed, inlined: Failed to attach because the symbol not found
  - function arguments changed, struct field changed: Reads the wrong argument → garbage data
- **Alternatives**
  - **Use tracepoints which are more reliable if possible**
    - Statically defined instrumentation points added by kernel developers
    - Guaranteed stable ABI: arguments are documented and maintained
    - `# cat /sys/kernel/debug/tracing/available_events`

Gotcha #1: Kernel versions

kprobes, fprobes have no stability guarantee:

- **kernel internal functions can change with any new release:**
  - function is renamed: “Failed to attach: symbol not found”
  - function is removed: “Attached: 0 events”
  - function inlined: “Failed to attach: symbol not found”
  - function arguments changed: “Reads wrong argument: garbage data”
- **Alternatives**
  - **Use tracepoints which are more reliable if possible**
    - Statically defined instrumentation points added by kernel developers
    - Guaranteed stable ABI: arguments are documented and maintained
    - `# cat /sys/kernel/debug/tracing/available_events`

Q: What if there is no tracepoint for the function you want to cover, what is your option?

A. Hardcode the function name and hope it doesn't change
B. Write separate BPF programs for each kernel version
C. Use CO-RE with BTF for automatic adaptation
D. Abandon eBPF and use kernel modules instead

Gotcha #1: Kernel versions

kprobes, fprobes have no stability guarantee:

- **kernel internal functions can change with any new release:**
  - function is renamed: “Failed to attach: symbol not found”
  - function is removed: “Attached: 0 events”
  - function inlined: “Failed to attach: symbol not found”
  - function arguments changed: “Reads wrong argument: garbage data”
- **Alternatives**
  - **Use tracepoints which are more reliable if possible**
    - Statically defined instrumentation points added by kernel developers
    - Guaranteed stable ABI: arguments are documented and maintained
    - `# cat /sys/kernel/debug/tracing/available_events`

Q: What if there is no tracepoint for the function you want to cover, what is your option?

A. Hardcode the function name and hope it doesn't change
B. Write separate BPF programs for each kernel version
C. Use CO-RE with BTF for automatic adaptation
D. Abandon eBPF and use kernel modules instead

ISOVALENT now part of CISCO

Gotcha #1: Kernel versions

kprobes, fprobes have no stability guarantee:

- **kernel internal functions can change with any new release:**
  - function is renamed: “Failed to attach: symbol not found”
  - function is removed: “Attached: 0 events”
  - function inlined: “Failed to attach: symbol not found”
  - function arguments changed: “Reads wrong argument: garbage data”
- **Alternatives**
  - **Use tracepoints which are more reliable if possible**
    - Statically defined instrumentation points added by kernel developers
    - Guaranteed stable ABI: arguments are documented and maintained
    - `# cat /sys/kernel/debug/tracing/available_events`
  - **CO-RE (Compile Once Run Everywhere) with BTF**
    - CO-RE = BPF programs adapt to kernel at load time
    - BTF (BPF Type Format) = Type information embedded in kernel
      `e.g: bpftool btf dump file /sys/kernel/btf/vmlinux > vmlinux.h`)
    - Relocations = libbpf adjusts offsets automatically

Gotcha #2
Architectures

Gotcha #2: Different architectures

kprobes have an unfriendly interface: the context passed to a kprobe program is struct pt_regs which is a struct representing the registers. It stores the function parameters.

- **Same BPF code behaves differently on x86_64 vs arm64:**
    - kprobe
        - different registers (RDI vs X0)

Gotcha #2: Different architectures

kprobes have an unfriendly interface: the context passed to a kprobe program is struct pt_regs which is a struct representing the registers. It stores the function parameters.

- **Same BPF code behaves differently on x86_64 vs arm64:**
  - kprobe
    - different registers (RDI vs X0)
- **Alternatives**
  - Use other hook points: fprobes, tracepoints (no raw_tracepoints)
  - libbpf provides helpers for access (tools/lib/bpf/bpf_tracing.h)

Gotcha #3
Dynamic Links

Gotcha #3: uprobe dynamic links

```c
// libmath.c
int add(int a, int b) {
    return a + b;
}

// main.c: loads, attaches and runs add():
int result = add(2, 3);
printf("Result: %d\n", result);
```


ISOVALENT now part of CISCO

Gotcha #3: uprobe dynamic links

```c
// libmath.c
int add(int a, int b) {
    return a + b;
}

// main.c: loads, attaches and runs add():
int result = add(2, 3);
printf("Result: %d\n", result);

// Attach uprobe to add()
# sudo bpftrace -e 'uprobe:./libmath.so:add {
printf("uprobe: add(%d, %d)\n", arg0, arg1); }'
```

Symbol table:

```
┌──────────────────────────┐
│ add()  →  offset 0x1140  │  ← Probe attaches HERE
└──────────────────────────┘
```


ISOVALENT now part of CISCO

Gotcha #3: uprobe dynamic links

```
// libmath.c
int add(int a, int b) {
    return a + b;
}

// main.c: loads, attaches and runs add():
int result = add(2, 3);
printf("Result: %d\n", result);

// Attach uprobe to add()
# sudo bpftrace -e 'uprobe:./libmath.so:add {
printf("uprobe: add(%d, %d)\n", arg0, arg1); }'

Symbol table:

┌──────────────────────────┐
│ add()  →  offset 0x1140  │  ← Probe attaches HERE
└──────────────────────────┘
```

```
=== Running main program ===
LD_LIBRARY_PATH=. ./main
add called: 2 + 3
Result: 5
```

Gotcha #3: uprobe dynamic links

```c
// libmath.c
int add(int a, int b) {
    return a + b;
}
```

```c
// main.c: loads, attaches and runs add():
int result = add(2, 3);
printf("Result: %d\n", result);
```

```
// Attach uprobe to add()
# sudo bpftrace -e 'uprobe:./libmath.so:add {
printf("uprobe: add(%d, %d)\n", arg0, arg1); }'
```

Symbol table:

```
┌──────────────────────────┐
│ add()  →  offset 0x1140  │  ← Probe attaches HERE
└──────────────────────────┘
```

```
=== Running main program ===
LD_LIBRARY_PATH=. ./main
add called: 2 + 3
Result: 5
```

```
// bpftrace output
sudo bpftrace -e 'uprobe:./libmath.so:add {
printf("uprobe: add(%d, %d)\n", arg0, arg1); }'
Attaching 1 probe...
uprobe: add(2, 3)
```


ISOVALENT now part of CISCO

Gotcha #3: uprobe dynamic links

Update library

```c
// libmath.c
// Insert new functions
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
//  add function is now at the bottom
int add(int a, int b) { return a + b; }
```

Gotcha #3: uprobe dynamic links

Update library

```c
// libmath.c
// Insert new functions
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
//  add function is now at the bottom
int add(int a, int b) { return a + b; }
```

```
=== Running main program ===
LD_LIBRARY_PATH=. ./main
add called: 2 + 3
Result: 5
```

```
// bpftrace output
sudo bpftrace -e 'uprobe:./libmath.so:add {
printf("uprobe: add(%d, %d)\n", arg0, arg1); }'
Attaching 1 probe...
uprobe: add(2, 3)
```

Gotcha #3: uprobe dynamic links

Update library

```c
// libmath.c
// Insert new functions
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
//  add function is now at the bottom
int add(int a, int b) { return a + b; }
```

```
=== Running main program ===
LD_LIBRARY_PATH=. ./main
add called: 2 + 3
Result: 5
```

```
// bpftrace output
sudo bpftrace -e 'uprobe:./libmath.so:add { printf("uprobe: add(%d, %d)\n", arg0, arg1); }'
Attaching 1 probe...
uprobe: add(2, 3)
```

Q: Your uprobe is tracing add(). A colleague updates libmath.c and adds new functions. You run your program again. What do you see?

A. add(2, 3) - everything works fine
B. subtract(2, 3) - probe fires on wrong function
C. Nothing - probe silently stops working
D. None of the above

Gotcha #3: uprobe broken dynamic links 🔗

Update library

```
// libmath.c
// Insert new functions
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
//  add function is now at the bottom
int add(int a, int b) { return a + b; }
```

```
=== Running main program ===
LD_LIBRARY_PATH=. ./main
add called: 2 + 3
Result: 5
```

```
// bpftrace output
sudo bpftrace -e 'uprobe:./libmath.so:add {
printf("uprobe: add(%d, %d)\n", arg0, arg1); }'
Attaching 1 probe...
uprobe: add(2, 3)
```

Q: Your uprobe is tracing add(). A colleague updates libmath.c and adds new functions. You run your program again. What do you see?

A. add(2, 3) - everything works fine
B. subtract(2, 3) - probe fires on wrong function
C. Nothing - probe silently stops working
D. None of the above

Gotcha #3: uprobe broken dynamic links 🔗

Update library

```
// libmath.so
// Insert new functions
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
//  add function is now at the bottom
int add(int a, int b) { return a + b; }
```

```
=== Running main program ===
LD_LIBRARY_PATH=. ./main
add called: 2 + 3
Result: 5
```

```
// bpftrace output
sudo bpftrace -e 'uprobe:./libmath.so:add { printf("uprobe: add(%d, %d)\n", arg0, arg1); }'
Attaching 1 probe...
uprobe: add(2, 3)
```

Symbol table AFTER update:

```
┌─────────────────────────────┐
│ subtract → offset 0x1140 │   ← Probe still points here! ❌
│ multiply → offset 0x1180 │
│ add      → offset 0x11c0 │   ← add() is now HERE
└─────────────────────────────┘
```

Gotcha #3: uprobe broken dynamic links

Conclusion

- **Problem**
  - Library recompiled/updated, uprobe are attached to the old offset
- **In production:**
  - Library updates are rare... but they happen (security patches!)
  - your monitoring may silently break without warning
- **Solutions:**
  - Monitor for file changes (inotifywait)
  - Use a supervisor program that monitors updates to the relevant shared libraries to re-attach your uprobes after updates

Gotcha #4: uprobe/kprob e inlining

What is inlining?

What is inlining?

> ***Inlining is an optimization where a function call is replaced with the actual code of the function itself at the point of call.***

Gotcha #4: kprobe/uprobe inlining

```c
// target.c

#include <stdio.h>
#include <stdlib.h>

int add(int a, int b) {
   return a + b;
}

int main(int argc, char *argv[]) {
   int x = atoi(argv[1]);

   if (x < 5) {
       printf("Usage: %d <number>\n", x);
       return 0;
   }

   int result = add(x, 10);
   printf("add(%d, 10) = %d\n", x, result);

   return 0;
}
```

Gotcha #4: kprobe/uprobe inlining

```c
// target.c

#include <stdio.h>
#include <stdlib.h>

int add(int a, int b) {
   return a + b;
}

int main(int argc, char *argv[]) {
   int x = atoi(argv[1]);

   if (x < 5) {
       printf("Usage: %d <number>\n", x);
       return 0;
   }

   int result = add(x, 10);
   printf("add(%d, 10) = %d\n", x, result);

   return 0;
}
```

Compilation without any optimisation:

```
# gcc -g -O0 -o target_O0 target.c
```

Disassembled program, symbol of the function is present:

```
# nm target_O0 | grep add
00000000000007d8 T add
```


ISOVALENT now part of CISCO

Gotcha #4: kprobe/uprobe inlining

```c
// target.c

#include <stdio.h>
#include <stdlib.h>

int add(int a, int b) {
   return a + b;
}

int main(int argc, char *argv[]) {
   int x = atoi(argv[1]);

   if (x < 5) {
       printf("Usage: %d <number>\n", x);
       return 0;
   }

   int result = add(x, 10);
   printf("add(%d, 10) = %d\n", x, result);

   return 0;
}
```

Compilation without any optimisation:

```
# gcc -g -O0 -o target_O0 target.c
```

Disassembled program, symbol of the function is present:

```
# nm target_O0 | grep add
00000000000007d8 T add
```

Compilation with a level of optimisation O2:

```
# gcc -g -O2 -o target_O2 target.c
```

Gotcha #4: kprobe/uprobe inlining

```c
// target.c

#include <stdio.h>
#include <stdlib.h>

int add(int a, int b) {
   return a + b;
}

int main(int argc, char *argv[]) {
   int x = atoi(argv[1]);

   if (x < 5) {
       printf("Usage: %d <number>\n", x);
       return 0;
   }

   int result = add(x, 10);
   printf("add(%d, 10) = %d\n", x, result);

   return 0;
}
```

Compilation without any optimisation:

```
# gcc -g -O0 -o target_O0 target.c
```

Disassembled program, symbol of the function is present:

```
# nm target_O0 | grep add
00000000000007d8 T add
```

Compilation with a level of optimisation O2::

```
# gcc -g -O2 -o target_O2 target.c
```

Disassembled program, symbol of the function is present:

```
# nm target_O2 | grep add
0000000000000860 T add
```

Demo selective inlining

I should be able to attach to add() function in both cases: without and with optimisation?!

Gotcha #4: kprobe/uprobe selective inlining

```
# 1. Build non-optimised and optimised versions
gcc -g -O0 -o target_O0 target.c
gcc -g -O2 -o target_O2 target.c
```

```
# 2. Check symbols - addf exists in BOTH binaries!
nm target_O0 | grep addf
# 0000000000000760 T addf

nm target_O2 | grep addf
# 0000000000000760 T addf    ← Symbol exists!
```

```
# 3. Trace O0 (no optimization).
# Terminal 1:
sudo bpftrace -e 'uprobe:./target_O0:addf {
printf("addf called!\n"); }'

# Terminal 2:
./target_O0 3     # x < 5: early return
./target_O0 10    # x >= 5: calls addf

# Result: "addf called!" ✅
```

```
# 4. Trace O2 (optimized) - SAME probe, SAME
program
# Terminal 1:
sudo bpftrace -e 'uprobe:./target_O2:addf {
printf("addf called!\n"); }'

# Terminal 2:
./target_O2 3     # nothing
./target_O2 10    # nothing!
# Result: SILENCE ❌ (function inlined)
```

Gotcha #4: kprobe/uprobe selective inlining

```
# 5. Compare disassembly - WHY?
# O0: main CALLS addf
objdump -d target_O0 | grep -A 15 "<main>:"
#   ...
#   bl <addf>      ← BRANCH to addf function

# O2: main has addf INLINED
objdump -d target_O2 | grep -A 15 "<main>:"
#   ...
#   add w0, w0, #0xa   ← This IS addf(x,10), no branch!
```

```
# 6. Solution: Attach at offset (uprobe/kprobe)
# Find where addf is inlined:
objdump -d target_O2
# main at 0x6c0, "add w0,w0,#0xa" at 0x6e8
# offset = 0x6e8-0x6c0 = 0x28

sudo bpftrace -e 'uprobe:./target_O2:main+0x28 {
printf("Hit inlined addf!\n"); }'

# Now ./target_O2 10 triggers: "Hit inlined addf!" ✅
```

Gotcha #4: kprobe/uprobe selective inlining

Disassembled binary

No optimisation

```
# objdump -d -S --disassemble=main ./target_O0

00000000000007f8 <main>:
// ...
int main(int argc, char *argv[]) {
 7f8:   a9bd7bfd        stp     x29, x30, [sp, #-48]!
 7fc:   910003fd        mov     x29, sp
 800:   b9001fe0        str     w0, [sp, #28]
 804:   f9000be1        str     x1, [sp, #16]
    int x = atoi(argv[1]);
 808:   f9400be0        ldr     x0, [sp, #16]
 80c:   91002000        add     x0, x0, #0x8
 810:   f9400000        ldr     x0, [x0]
 814:   97ffff8f        bl      650 <atoi@plt>
 818:   b9002be0        str     w0, [sp, #40]

    if (x < 5) {
 81c:   b9402be0        ldr     w0, [sp, #40]
// ...
    }

    int result = addf(x, 10);
 840:   52800141        mov     w1, #0xa
 844:   b9402be0        ldr     w0, [sp, #40]
 848:   97ffffe4        bl      7d8 <add>
 84c:   b9002fe0        str     w0, [sp, #44]
// ...
```


ISOVALENT
now part of cisco

Gotcha #4: kprobe/uprobe selective inlining

Disassembled binary

No optimisation

```
# objdump -d -S --disassemble=main ./target_O0

00000000000007f8 <main>:
// ...
int main(int argc, char *argv[]) {
 7f8:   a9bd7bfd        stp     x29, x30, [sp, #-48]!
 7fc:   910003fd        mov     x29, sp
 800:   b9001fe0        str     w0, [sp, #28]
 804:   f9000be1        str     x1, [sp, #16]
    int x = atoi(argv[1]);
 808:   f9400be0        ldr     x0, [sp, #16]
 80c:   91002000        add     x0, x0, #0x8
 810:   f9400000        ldr     x0, [x0]
 814:   97ffff8f        bl      650 <atoi@plt>
 818:   b9002be0        str     w0, [sp, #40]

    if (x < 5) {
 81c:   b9402be0        ldr     w0, [sp, #40]
// ...
    }

    int result = addf(x, 10);
 840:   52800141        mov     w1, #0xa
 844:   b9402be0        ldr     w0, [sp, #40]
 848:   97fffe4         bl      7d8 <add>
 84c:   b9002fe0        str     w0, [sp, #44]
// ...
```

Optimised

```
# objdump -d -S --disassemble=main ./target_O2

00000000000006c0 <main>:
// ...
int main(int argc, char *argv[]) {
 6c0:   aa0103e0        mov     x0, x1
 6c4:   a9bf7bfd        stp     x29, x30, [sp, #-16]!
// ...
    int x = atoi(argv[1]);

    if (x < 5) {
 6e0:   7100101f        cmp     w0, #0x4
 6e4:   5400012d        b.le    708 <main+0x48>
}

// ...

 6e8:   11002803        add     w3, w0, #0xa
 6ec:   90000001        adrp    x1, 0 <__abi_tag-0x278>
 6f0:   52800040        mov     w0, #0x2                         // #2
 6f4:   91228021        add     x1, x1, #0x8a0
 6f8:   97ffffd6        bl      650 <__printf_chk@plt>

    int result = addf(x, 10);
    printf("addf(%d, 10) = %d\n", x, result);
// ...
```

Gotcha #4: kprobe/uprobe selective inlining

Conclusion

The Problem

- Symbol exists in binary (visible with `nm`)
- But compiler inlined the function into the caller
- uprobe is attached on symbol but it never fires,
  → executed instructions inlined in the caller

How to Detect Selective Inlining?

1. **Show disassembly code**

   ```
   objdump -d s
   llvm-dwarfdump <your_binary>
   ```

2. **Check if function is inlined**

   not inlined: `bl <function>` → real call

   inlined: `<instruction>` → No `bl`, just the instructions directly

Gotcha #4: kprobe/uprobe selective inlining

Conclusion

The Problem

- Symbol exists in binary (visible with `nm`)
- But compiler inlined the function into the caller
- uprobe is attached on symbol but it never fires,
  → executed instructions inlined in the caller

How to Detect Selective Inlining?

1. **Show disassembly code**

   `objdump -d s`

   `llvm-dwarfdump <your_binary>`

2. **Check if function is inlined**

   not inlined: `bl <function>` → real call

   inlined: `<instruction>` → No `bl`, just the instructions directly

Solution: Use uprobe or kprobe + offset

1. Look inside the caller function's disassembly
2. Find the instruction that does the work (e.g., add for addition)
3. Calculate offset: `instr_address - fnc_start_address`
4. Probe to the offset:
   `uprobe:binary:<caller_function>+<offset>`

   → Fires when execution reaches that exact instruction

Caveats

- !! Multiple probes for all callsites of your function
- Offsets change with every recompilation
- Different compiler versions = different offsets
- Not practical for production, useful for debugging

Gotcha #5: kprobe/uprob e inlining

Gotcha #5: kprobe/uprobe inlining

```c
// target.c
int allocate_resource(int size) {
    if (size <= 0 || size >= 1024) return -1;

    int resource_id = 0;
    for (int i = 0; i < size; i++) { ... }  // Loop
    snprintf(log_buffer, ...);               // String formatting
    printf("%s\n", log_buffer);              // I/O
    return resource_id;
}
```

Gotcha #5: kprobe/uprobe inlining

```c
// target.c
int allocate_resource(int size) {
    if (size <= 0 || size >= 1024) return -1;

    int resource_id = 0;
    for (int i = 0; i < size; i++) { ... }  // Loop
    snprintf(log_buffer, ...);              // String formatting
    printf("%s\n", log_buffer);             // I/O
    return resource_id;
}
```

Q: How many symbols does this generate when compiling with the following command: gcc -O2 -o target_gcc target.c ?

A. 1
B. 2
C. 10
D. wth are symbols?

ISOVALENT now part of CISCO

Gotcha #5: kprobe/uprobe inlining

A typical scenario...

```c
// target.c
int allocate_resource(int size) {
    if (size <= 0 || size >= 1024) return -1;

    int resource_id = 0;
    for (int i = 0; i < size; i++) { ... }  // Loop
    snprintf(log_buffer, ...);               // String formatting
    printf("%s\n", log_buffer);              // I/O
    return resource_id;
}
```

Q: How many symbols does this generate when compiling with the following command: gcc `-O2 -o target_gcc target.c` ?

A. 1
B. 2
C. 10
D. wth are symbols?

```
$ nm target | grep -E "allocate_resource"
0000000000000c40 T allocate_resource
0000000000000ae0 t allocate_resource.part.0
```


ISOVALENT now part of CISCO

Gotcha #5: kprobe/uprobe partial inlining

GCC Optimisation

gcc's `-fpartial-inlining (enabled by -O2)` splits functions based on execution patterns:

Fast path example:

- quick validation
- error path

Slow path example, extracted to .part:

- complex error handling
- complex processes

Before:

```
allocate_resource()

 if (size⩽0||size⩾1024)
   return -1

 for (i=0; i<size; i++)
   resource_id += ...
 snprintf(log_buffer,...)
 printf("%s", log_buffer)
 return resource_id
```

gcc →

After:

```
allocate_resource()                0xc20

 if (size⩽0||size⩾1024)
   return -1
 jump to .part.0
```

```
allocate_resource.part.0           0xb90

 for (i=0; i<size; i++)
   resource_id += ...
 snprintf(log_buffer,...)
 printf("%s", log_buffer)
 return resource_id
```

Gotcha #5: kprobe/uprobe partial inlining

GCC Optimisation

Why does GCC do this?

- Smaller hot/fast code → fits better in instruction cache
- Better branch prediction → CPU expects the fast path
- Reduced register pressure → cold/slow path variables don't pollute hot path

Gotcha #5: kprobe/uprobe partial inlining

Demo Summary

```
# 1. Build with partial inlining
make target_partial

# 2. Check symbol table - see the split!
nm target_partial | grep allocate_resource
# 0xc20 T allocate_resource
# 0xb90 t allocate_resource.part.0

# 3. Compare sizes
readelf -s target_partial | grep allocate_resource

# 4. Probe the wrapper (catches valid calls)
sudo bpftrace -e 'uprobe:./target_partial:allocate_resource {
    printf("allocate_resource called\n");
}'

# 5. Probe the error path
sudo bpftrace -e 'uprobe:./target_partial:allocate_resource.part.0 {
    printf("error path called!\n");
}'

# 6. Run the target and observe
./target_partial
```


ISOVALENT now part of CISCO

Demo partial inlining

So what about kprobe/uprobe attachments and symbols?

Gotcha #5: kprobe/uprobe partial inlining

Why probe misses events?

Program output:

```
./target

// Hot path
allocate_resource(-5) = -1

allocate_resource(9999) = -1
Allocated resource #2424 (size=100 bytes)
Total allocated so far: 100 bytes

// Cold path
allocate_resource(100) = 2424
Allocated resource #3024 (size=256 bytes)
Total allocated so far: 356 bytes

allocate_resource(256) = 3024
```

PROBE: allocate_resource

`$ sudo bpftrace -e 'uprobe:...:allocate_resource {...}'`
 	
`Attaching 1 probe...`
`(nothing)                    ← SILENT! No events!`

PROBE: allocate_resource.part.0

`$ sudo bpftrace -e 'uprobe:...:allocate_resource.part.0 {...}'`
 	
`Attaching 1 probe...`
`allocate_resource called with size=100`
`allocate_resource called with size=256`
`                              ↑ ACTUAL ALLOCATIONS!`

Gotcha #5: kprobe/uprobe partial inlining

Alternatives

- **generally:** look for alternative functions that has defined tracepoints (not 1:1 translation)
- **uprobe:** gcc -O2 -fnopartial-inlining -o <output> <your_program>
- **kprobe:**
  - kernel recompilation in production is not common → identify symbols using tools

```
1.  Check for suffixed symbols
nm <your_binary> | grep -E "\.(part|cold|isra|constprop)"

2.  Verify with DWARF
llvm-dwarfdump <your binary>| grep -A5 "<your_function>"

3.  See source mapping
objdump -S <your_binary> | grep -A30 "<your_function>:"
```


ISOVALENT now part of cisco

Gotcha #6: missed executions

Gotcha #6: missed executions

```
# bpftool p | grep misses

5845:  tracing     name fentry_do_sys_openat2              tag xxx gpl recursion_misses 9482
5847:  kprobe      name kprobe_do_sys_openat2              tag xxx gpl recursion_misses 245
11066: tracepoint  name tracepoint_lock_contention_begin_1  tag xxx gpl recursion_misses 25
```

Gotcha #6: missed executions

```
# bpftool p | grep misses

5845:  tracing    name fentry_do_sys_openat2            tag xxx gpl recursion_misses 9482
5847:  kprobe     name kprobe_do_sys_openat2            tag xxx gpl recursion_misses 245
11066: tracepoint name tracepoint_lock_contention_begin_1 tag xxx gpl recursion_misses 25
```

Question: What does a "recursion_miss" counter represent?

A) The number of times the BPF program failed to execute due to missing kernel functions
B) How many times recursion protection prevented a program from running again while already executing
C) A count of recursive function calls that the BPF verifier rejected
D) The number of missed hardware events due to recursion in the perf subsystem

ISOVALENT now part of CISCO

Gotcha #6: missed executions

```
# bpftool p | grep misses

5845:  tracing    name fentry_do_sys_openat2             tag xxx gpl recursion_misses 9482
5847:  kprobe     name kprobe_do_sys_openat2             tag xxx gpl recursion_misses 245
11066: tracepoint name tracepoint_lock_contention_begin_1 tag xxx gpl recursion_misses 25
```

Question: What does a "recursion_miss" counter represent?

A) The number of times the BPF program failed to execute due to missing kernel functions
B) How many times recursion prevented the BPF program from running again while already executing
C) A count of recursive function calls that the BPF verifier rejected
D) The number of missed hardware events due to recursion in the perf subsystem

Gotcha #6: missed executions

- be aware! your tracing programs may not always execute
- must assume that you will not always see all events in the kernel when using tracing programs (kprobes, fprobes, tracepoints)
- why you should care?
  - security monitoring
  - debugging
  - etc
- we will discuss
  - kprobes
  - fprobes
  - tracepoints

Gotcha #6: missed executions

- probes can miss executions in two different ways (**only if on the same CPU**)
  a. recursion of the program
  b. nested executions of any other eBPF programs

- how does a probe recurse?
  a. a probe on a function which calls another function with another probe attached – don't do this!
  b. within a probe, interrupt fires in which the interrupt handler calls the function that has a probe attached – random misses!

Example 1:

```
do_sys_openat2() [kprobe attached, in progress]
└ bpf_trace_printk()
    └ spin_lock(trace_printk_lock) [contention occurs]
        └ contention_begin [tracepoint BPF prog fires]
            └ bpf_trace_printk()
                └ spin_lock(trace_printk_lock)
                    └ contention_begin [tracepoint executes again]
                        └ SKIPPED (nmissed++)
```

Example 2:

```
do_sys_openat2() [probed, in progress]
    └ IRQ/NMI fires
        └ IRQ handler calls do_sys_openat2()
            └ SKIPPED (nmissed++)
```

kprobes execution misses

- kprobe misses are handled at two different layers
  - handler / attach layer
    - kprobe-specific recursion check
  - BPF program execution (JIT)
    - per CPU recursion check

ISOVALENT now part of CISCO

kprobe: handler / attach layer

- **3 handlers**
  - **int3 (breakpoint),** fallback default behavior with:
    - `CONFIG_KPROBES_ON_FTRACE=n`
      and no optimization

kprobe: handler / attach layer

- **3 handlers**
  - **int3 (breakpoint),** fallback default behavior with:
    - `CONFIG_KPROBES_ON_FTRACE=n` and no optimization
  - **kprobe** ftrace handler relies on:
    - `CONFIG_KPROBES_ON_FTRACE=y`

kprobe: handler / attach layer

- **3 handlers**
  - **int3 (breakpoint),** fallback default behavior with:
    - `CONFIG_KPROBES_ON_FTRACE=n`
      and no optimization
  - **kprobe** ftrace handler relies on:
    - `CONFIG_KPROBES_ON_FTRACE=y`
  - **“opt”**
    - `CONFIG_KPROBES_ON_FTRACE=n`
    - `CONFIG_OPTPROBES=y` (automatically enabled x86/x86-64 & non-preemptive kernel)
    - `"debug.kprobes_optimization" sysctl == 1`

Handler	**Optimized**
int3 (breakpoint)	No
kprobe ftrace, opt	Yes

kprobe: handler / attach layer

- **3 handlers**
  - **int3 (breakpoint),** fallback default behavior with:
    - `CONFIG_KPROBES_ON_FTRACE=n` and no optimization
  - **kprobe** ftrace handler relies on:
    - `CONFIG_KPROBES_ON_FTRACE=y`
  - **"opt"**
    - `CONFIG_KPROBES_ON_FTRACE=n`
    - `CONFIG_OPTPROBES=y` (automatically enabled x86/x86-64 & non-preemptive kernel)
    - `"debug.kprobes_optimization" sysctl == 1`

Handler	Optimized
int3 (breakpoint)	No
kprobe ftrace, opt	Yes

```
int handler(...) {
    …
    if (kprobe_running()) {
        kprobes_inc_nmissed_count(...);
    }
    …
}

static inline struct kprobe *kprobe_running(void)
{
    return __this_cpu_read(current_kprobe);
}

DECLARE_PER_CPU(struct kprobe *, current_kprobe);
```


ISOVALENT now part of CISCO

kprobe: BPF program execution layer

- once the handlers proceed to invoke the BPF program, additional checks are made
- `trace_call_bpf()`
  - called for kprobes programs
  - checks if there is any BPF program running on the same CPU as the kprobe program

```
if (unlikely(__this_cpu_inc_return(bpf_prog_active) ≠ 1)) {
    /*
     * since some bpf program is already running on this cpu,
     * don't call into another bpf program (same or different)
     * and don't send kprobe event into ring-buffer,
     * so return zero here
     */
    …
    bpf_prog_inc_misses_counters(...);
    …
}
```

fprobe: handler layer

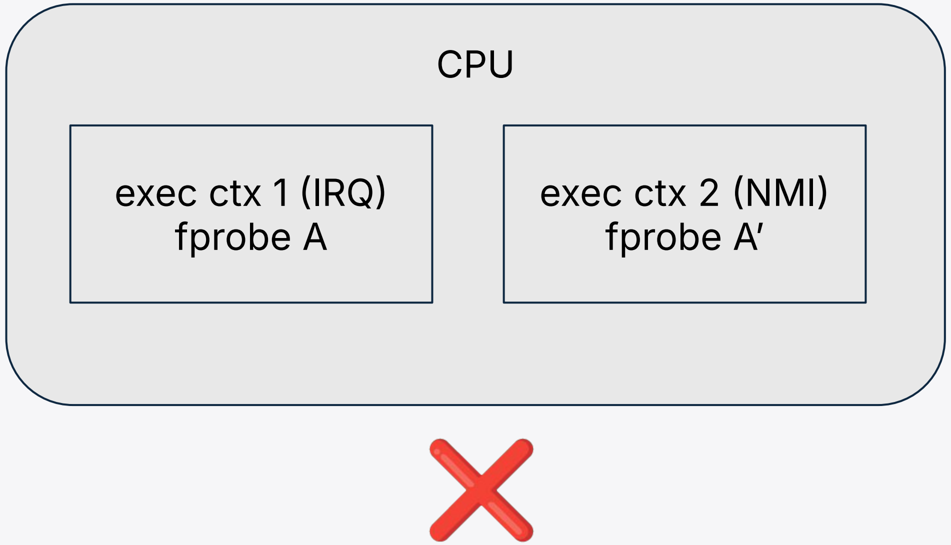
- handler uses `ftrace_test_recursion_trylock()`
  - allows 1 level of nesting per CPU, per execution context

```c
int handler(...) {
    …
    if (ftrace_test_recursion_trylock(...) < 0) {
        return;
    }
    …
}
```

fprobe: handler layer

- handler uses `ftrace_test_recursion_trylock()`
  - allows 1 level of nesting per CPU, per execution context

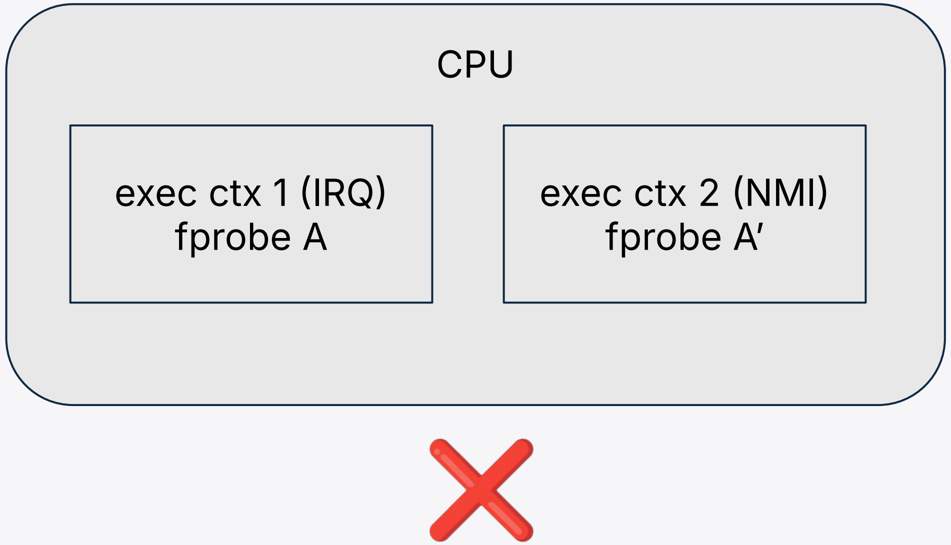
```
int handler(...) {
    …
    if (ftrace_test_recursion_trylock(...) < 0) {
        return;
    }
    …
}
```

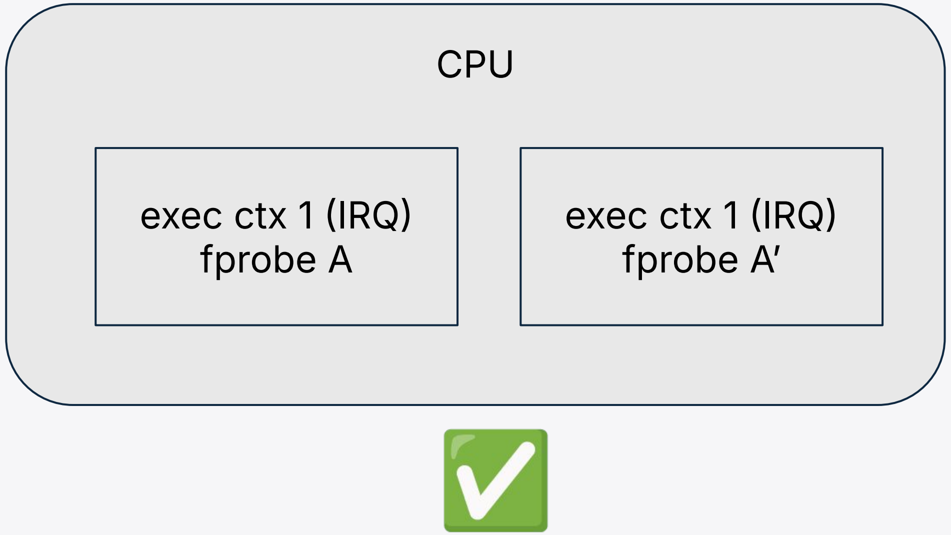


fprobe: handler layer

- handler uses `ftrace_test_recursion_trylock()`
  - allows 1 level of nesting per CPU, per execution context

```
int handler(...) {
    …
    if (ftrace_test_recursion_trylock(...) < 0) {
        return;
    }
    …
}
```

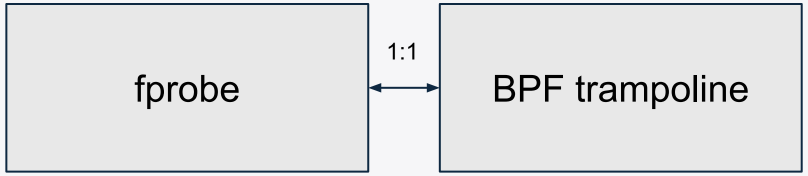




fprobe: BPF program execution layer

- BPF trampoline-based
- two entrypoints for trampoline
  - __bpf_prog_enter_recur()
  - __bpf_prog_enter_sleepable_recur()
- per program check results in execution misses only for self-recursion

```
/* At BPF program execution layer */
if (unlikely(this_cpu_inc_return(*(prog->active)) != 1))
{
    bpf_prog_inc_misses_counter(prog);
}
```



kprobe & fprobe comparison

- notice the difference between the checks:
  - `prog->active` and `bpf_prog_active` (from previous slide)

Feature	Kprobes	Fprobes
Hook type basis	Primarily interrupt-based (int3) or optimized (jump ins.)	BPF trampoline-based
Handler layer check	Misses if any other kprobe is running	Allows 1 level of execution nesting/recursion, misses on further levels
BPF execution layer check	Misses if any other BPF program is running on the same CPU	Misses only on self-recursion (exact same fprobe program)
Nesting/recursion tolerance	No nesting or recursion of any BPF program on the same CPU	Allows nesting of different BPF programs on the same CPU. Only prevents self-recursion

- **tracepoints**: similar to kprobes (per CPU check) except without the kprobe-specific restrictions
- **raw tracepoints**: similar to fprobes (per program check) since they are both trampoline-based except without the fprobe-specific restrictions

ISOVALENT now part of CISCO

BPF LSM hooks

```
SEC("kprobe/do_sys_openat2")
int BPF_KPROBE(kprobe_do_sys_openat2, …) {
    …
}

SEC("fentry/do_sys_openat2")
int BPF_PROG(fentry_do_sys_openat2, …) {
    …
}
```

```
SEC("lsm/file_open")
int BPF_PROG(lsm_file_open, struct file *file, …)
{
    …
}
```


ISOVALENT now part of CISCO

BPF LSM hooks

```
SEC("kprobe/do_sys_openat2")
int BPF_KPROBE(kprobe_do_sys_openat2, …) {
    …
}

SEC("fentry/do_sys_openat2")
int BPF_PROG(fentry_do_sys_openat2, …) {
    …
}
```

```
SEC("lsm/file_open")
int BPF_PROG(lsm_file_open, struct file *file, …)
{
    …
}
```

-> otherwise, use tracepoints as they are the highest performance tracing hook type

hookpoint missed execution summary

Hookpoint	kprobe	fprobe / trampoline	tracepoint / perf event	raw tracepoint
When misses occur	Per CPU: ● Any nested kprobes ● Nested BPF programs	Per program: ● Self nested fprobe	Per CPU: ● Nested BPF programs	Per program: ● Nested BPF programs

Key Takeaways

Generic

Performance

- Use tracepoints where possible

Kernel versions

- Use BTF and CORE

System Architecture

- use fprobe or tracepoint (no raw) instead of kprobe
- libbpf helpers

  → same code, ≠ architectures

kprobe/uprobe

Dynamic Links

- Library updated → offsets change → probe breaks silently
- Re-attach probes after library updates or use tracepoints

Selective Inlining

- Symbol exists but some call sites are inlined
- Solution: Trace the caller function or use offset

Partial Inlining

- Compiler splits function into fast/slow paths, missing probes
- Solution: kprobe: probe the .part suffix function, uprobe: compile with no partial inlining

Missed Executions

kprobes

- Cannot recurse
- Use optimized kprobes if necessary, otherwise prefer fprobes or tracepoints

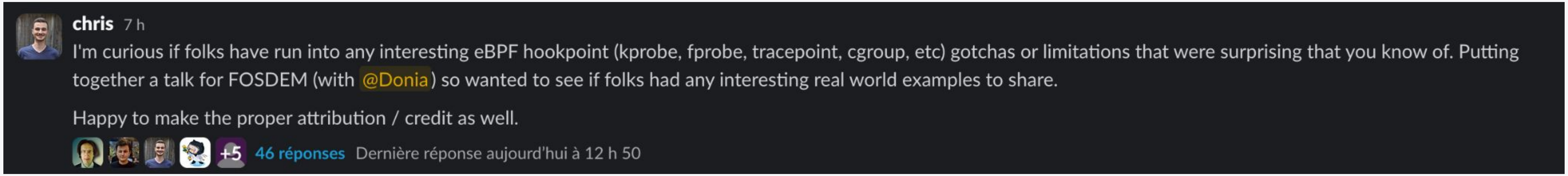
fprobes, tracepoints

- Limited recursion

kprobes, fprobes, tracepoints

- If cannot tolerate any missed executions, consider BPF LSM.

Thanks, Credits & Resources



🙏 Thanks (in order of responses in the Slack thread) Martynas Pumputis, Mahé Tardy, Paul Chaignon, Daniel Borkman, Dylan Reimerink, Jiri Olsa, Kornilios Kourtis, Kev Sheldrake.

🙏 Thanks as well to Masami Hiramatsu.

LPC Talks:

- Kernel func tracing in the face of compiler optimization (https://www.youtube.com/watch?v=kOYEsChbw-0)
- Where have all the kprobes gone (https://www.youtube.com/watch?v=Erqy3rxDp4g)

Articles:

- Bouncing on trampolines to run eBPF programs (https://bootlin.com/blog/bouncing-on-trampolines-to-run-ebpf-programs/)
- eBPF Tracepoints, Kprobes, or Fprobes: Which One Should You Choose? (https://labs.iximiuz.com/tutorials/ebpf-tracing-46a570d1)
- An introduction to KProbes (https://lwn.net/Articles/132196/)
- Linux Tracing Technologies Guide (https://docs.kernel.org/trace/)

LKML:

- kprobe: Support nested kprobes (https://lwn.net/ml/linux-kernel/158894789510.14896.13461271606820304664.stgit@devnote2/)