Linux Plumbers Conference | Richmond, VA | Nov. 13-15, 2023

Advancing Kernel Control Flow Integrity with eBPF

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

University of Illinois Urbana-Champaign | IBM | Virginia Tech

Control-flow security in OS kernels

- Kernel code makes extensive use of **function pointers**
- Corrupted function pointers allows **control-flow hijack attacks**

Control-flow security in OS kernels

- Kernel code makes extensive use of **function pointers**
- Corrupted function pointers allows **control-flow hijack attacks**
- **CVE-2016-0728**: privilege escalation by corrupting function pointer with UAF
    - Overwrite target to invoke kernel credential functions

Control-flow security in OS kernels

- Kernel code makes extensive use of **function pointers**
- Corrupted function pointers allows **control-flow hijack attacks**
- **CVE-2016-0728**: privilege escalation by corrupting function pointer with UAF
  - Overwrite target to invoke kernel credential functions

```
void key_revoke(struct key *key)
{
  ...
  if(key->type->revoke)
    key->type->revoke(key);
  ...
}
```

```
void
keyring_revoke(struct key *key)
{
  ...
}
```

Control-flow security in OS kernels

- Kernel code makes extensive use of **function pointers**
- Corrupted function pointers allows **control-flow hijack attacks**
- **CVE-2016-0728**: privilege escalation by corrupting function pointer with UAF
  - Overwrite target to invoke kernel credential functions

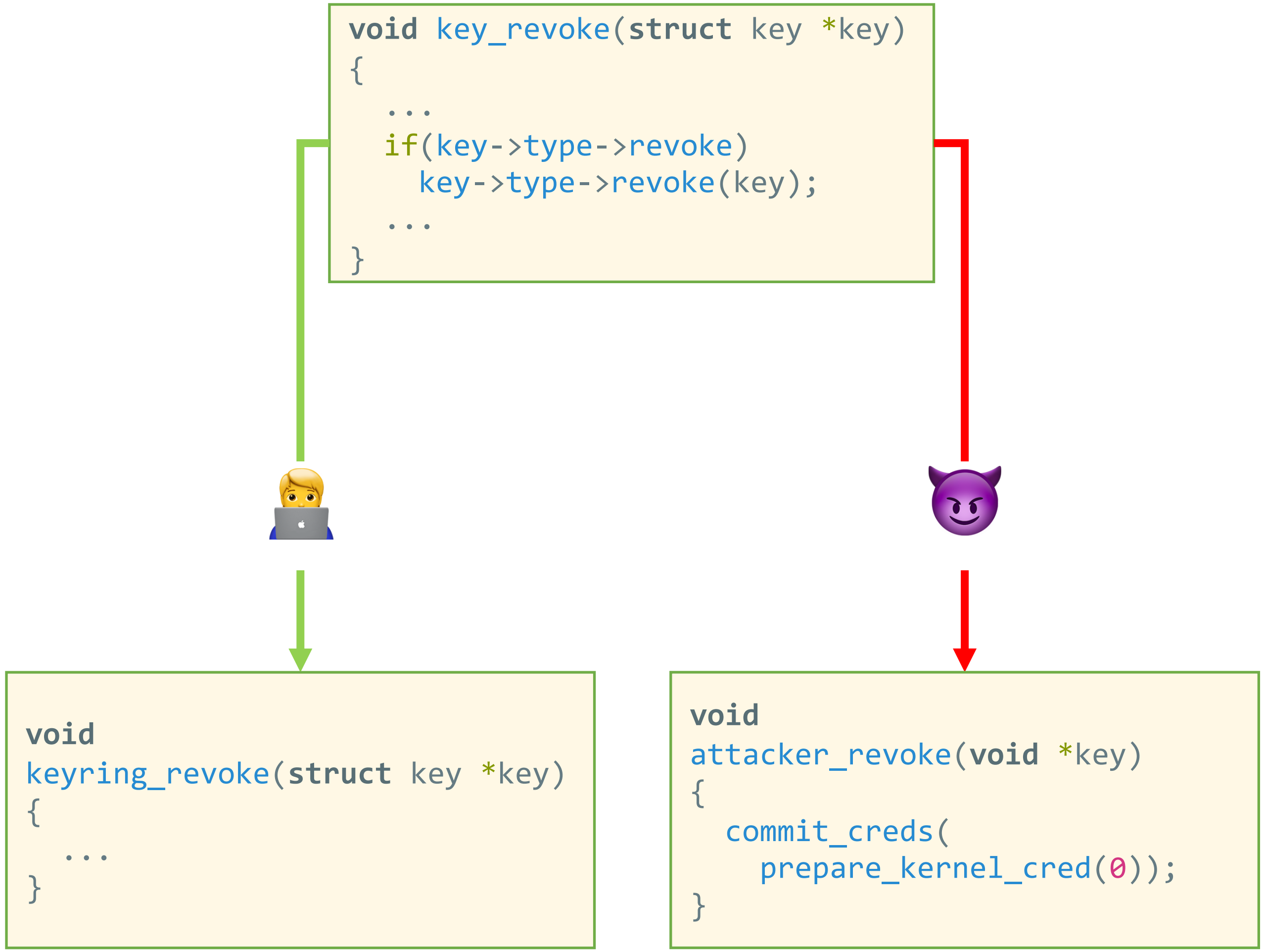
```
void key_revoke(struct key *key)
{
  ...
  if(key->type->revoke)
    key->type->revoke(key);
  ...
}
```

```
void
keyring_revoke(struct key *key)
{
  ...
}
```

```
void
attacker_revoke(void *key)
{
  commit_creds(
    prepare_kernel_cred(0));
}
```

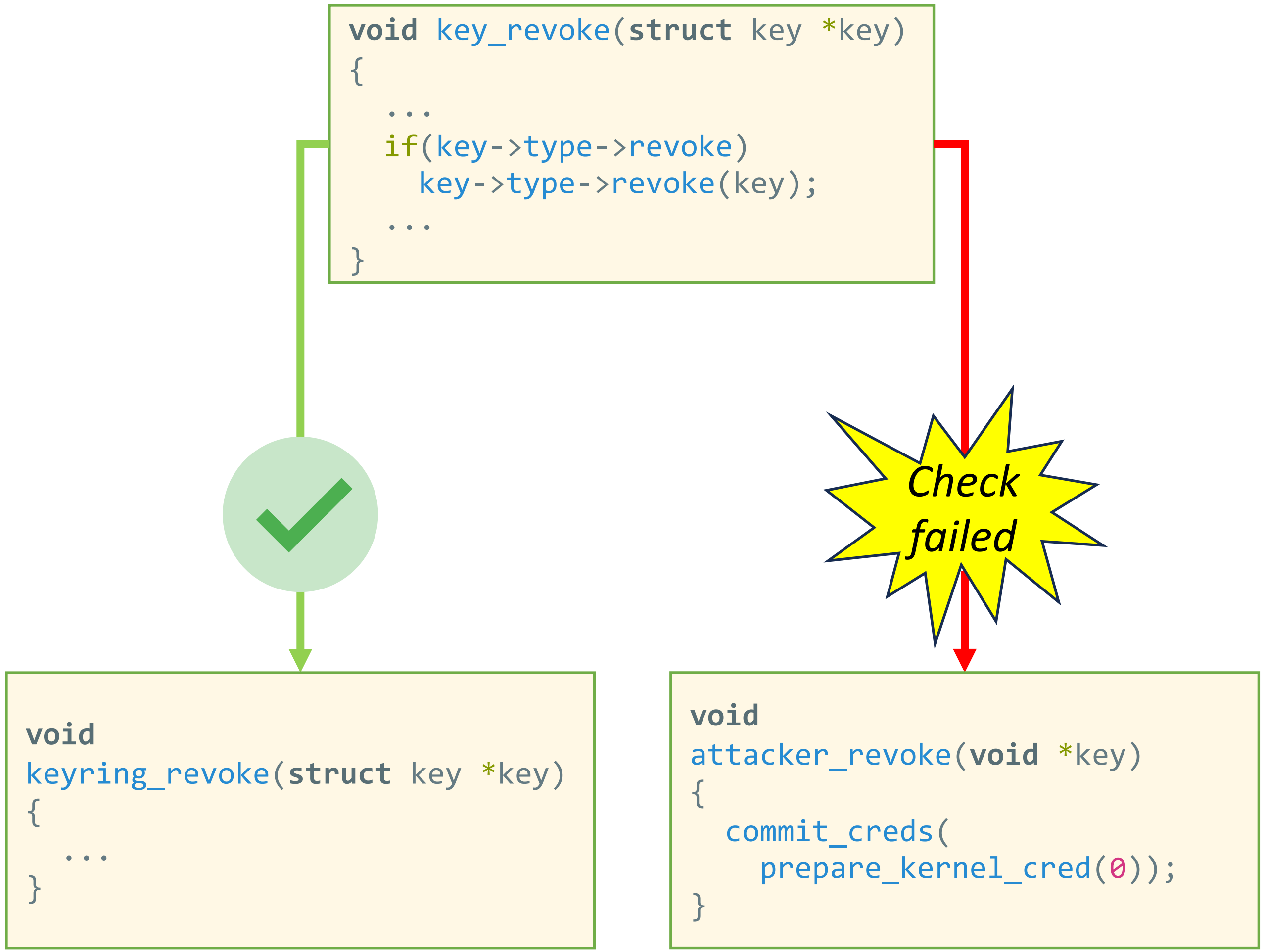
Control-flow integrity

- Restricting program execution to its control-flow graph (CFG)
- Verifies validity of **indirect** control flow transfers
  - Indirect calls
  - Returns
- CFG can be generated via either *static or dynamic* analysis



Control-flow integrity

- Restricting program execution to its control-flow graph (CFG)
- Verifies validity of **indirect** control flow transfers
  - Indirect calls
  - Returns
- CFG can be generated via either *static or dynamic* analysis



Inflexibility of existing KCFI approaches

- State-of-the-Practice: LLVM-based KCFI in Linux
  - Static policy based on function prototypes
  - Enabling/disabling KCFI requires rebuild the kernel

Inflexibility of existing KCFI approaches

- State-of-the-Practice: LLVM-based KCFI in Linux
  - Static policy based on function prototypes
  - Enabling/disabling KCFI requires rebuild the kernel

- KCFI policies are *statically* defined
  - Hard to catch the moving target of state-of-the-art CFI techniques
  - Policy change requires kernel rebuild and reboot
    - Service disruption
    - Increased mitigation time
  - Difficult to make use of runtime context

eBPF can be a powerful tool for KCFI

- **Easy to deploy**
  - KCFI policies can be enabled/disabled/switched at runtime
  - No kernel rebuilding/rebooting
- **Expressiveness and observability**
  - Support for dynamic policies that leverage context information
  - Observability superpower
- **Flexibility and fine granularity**
  - Selectively attaching eBPF checks to different indirect call sites

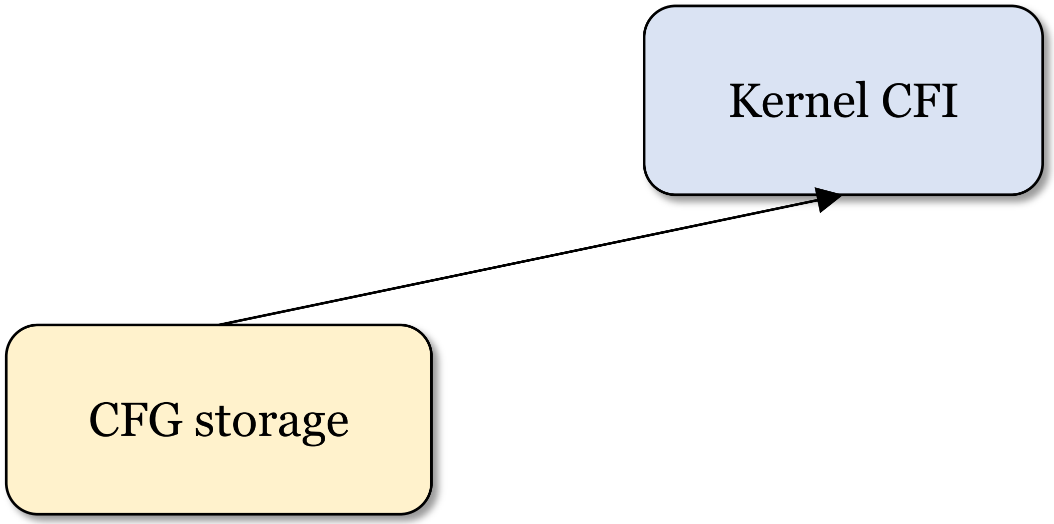
Sketching eBPF-based KCFI

- A simplest form of KCFI: **check against a static CFG**

Kernel CFI

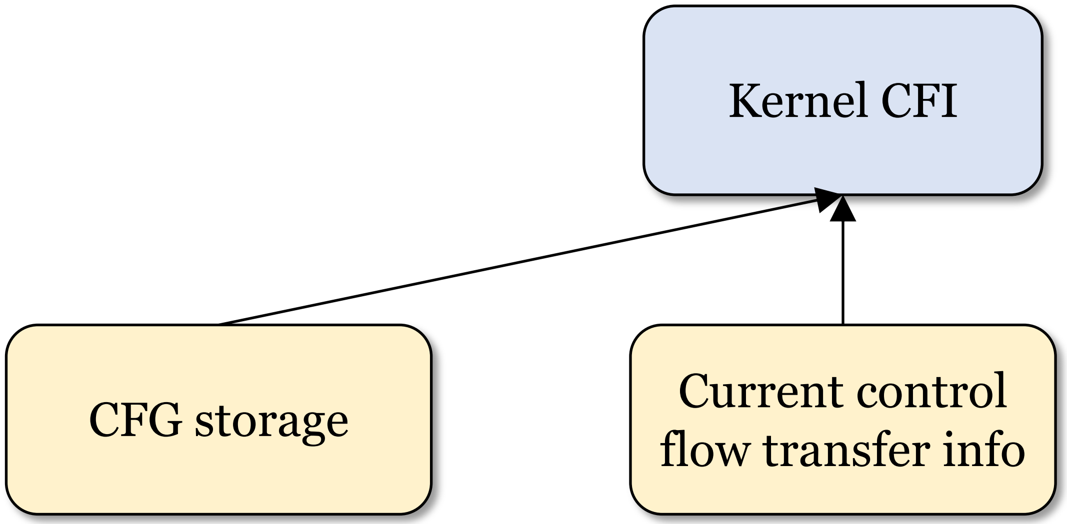
Sketching eBPF-based KCFI

- A simplest form of KCFI: **check against a static CFG**



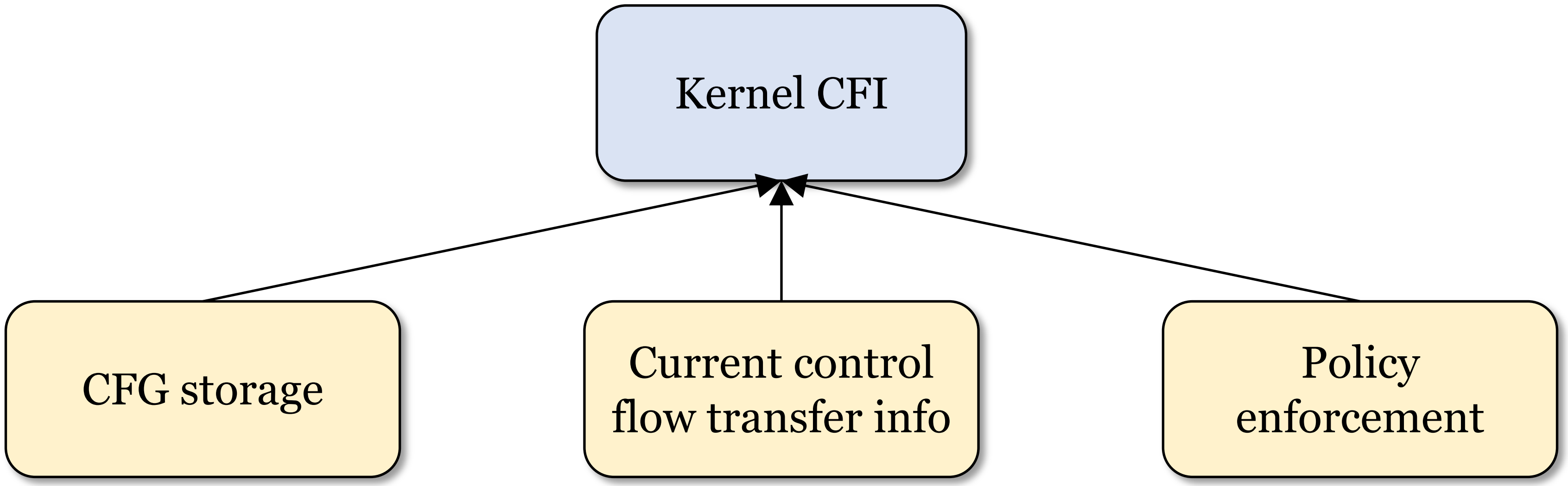
Sketching eBPF-based KCFI

- A simplest form of KCFI: **check against a static CFG**



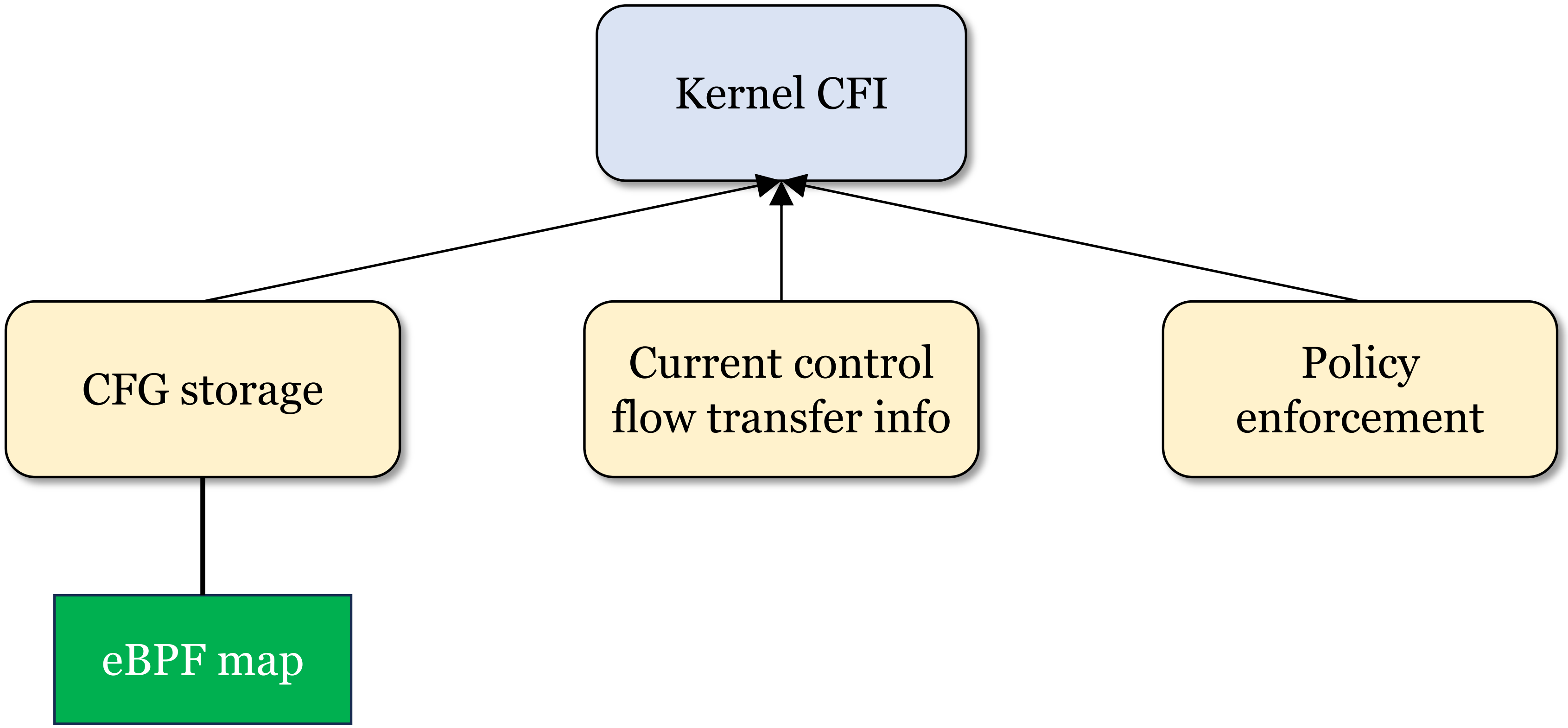
Sketching eBPF-based KCFI

- A simplest form of KCFI: **check against a static CFG**



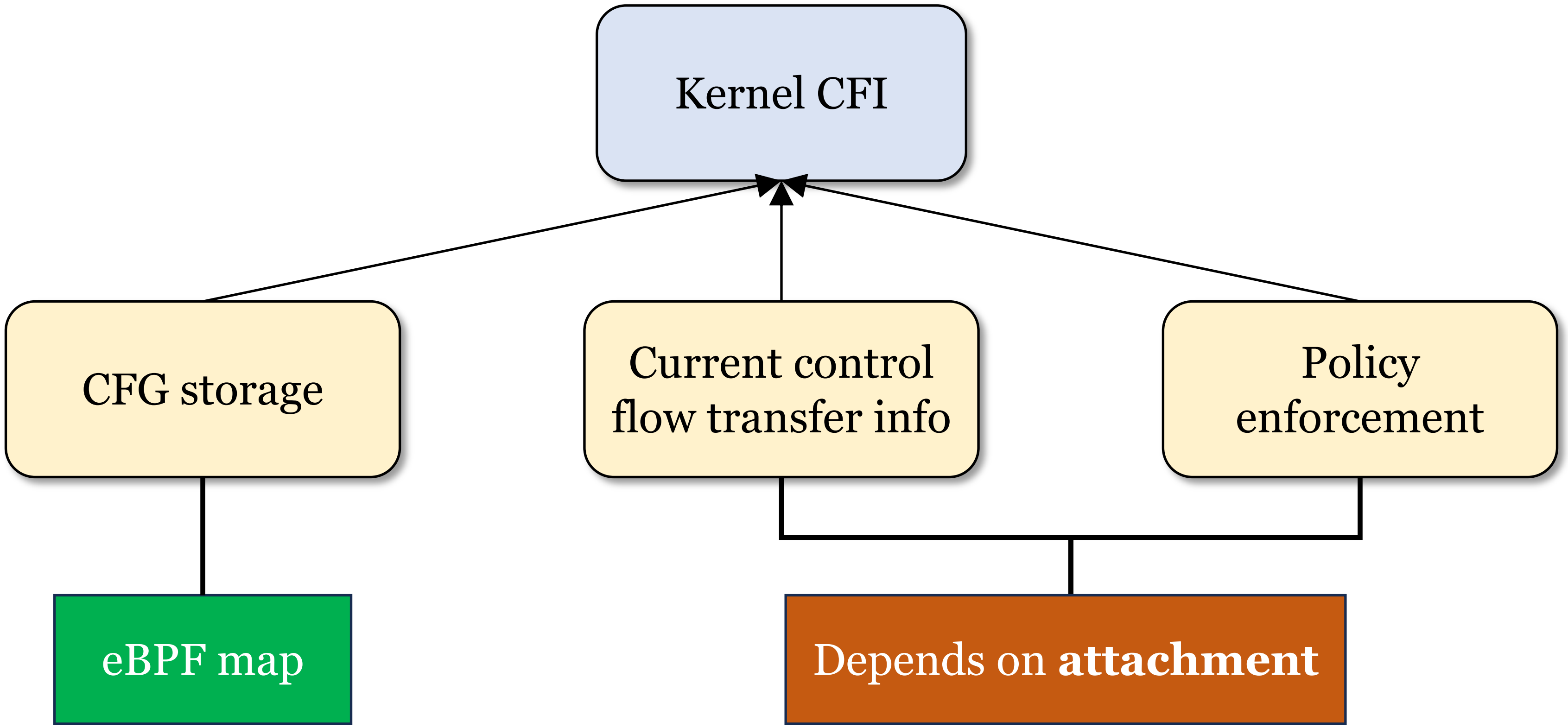
Sketching eBPF-based KCFI

- A simplest form of KCFI: **check against a static CFG**



Sketching eBPF-based KCFI

- A simplest form of KCFI: **check against a static CFG**



Scope and Threat Model

- The kernel is benign, but may contain vulnerabilities
- The attacker attacks the kernel by issuing system calls or by sending network packets
- The eBPF-based KCFI infrastructure is trusted
- Our current focus is on indirect function calls

A kprobe-based Approach

- Attach to indirect calls
  - `kprobe` attaches to most kernel text address

```
...
48 89 44 24 08 mov  %rax,0x8(%rsp)
31 ff          xor  %edi,%edi
31 f6          xor  %esi,%esi
ff d3          call *%rbx # indirect call
...
```

A kprobe-based Approach

- Attach to indirect calls
  - `kprobe` attaches to most kernel text address

```
...
48 89 44 24 08 mov  %rax,0x8(%rsp)
31 ff          xor  %edi,%edi
31 f6          xor  %esi,%esi
ff d3          call *%rbx # indirect call
...
```

```
SEC("kprobe")
int kcfi_prog(struct *pt_regs ctx)
{




  return 0;
}
```

A kprobe-based Approach

- Attach to indirect calls
  - `kprobe` attaches to most kernel text address
- Obtain source and target from registers

```
...
48 89 44 24 08 mov   %rax,0x8(%rsp)
31 ff          xor   %edi,%edi
31 f6          xor   %esi,%esi
ff d3          call *%rbx # indirect call
...
```

```
SEC("kprobe")
int kcfi_prog(struct *pt_regs ctx)
{
  u64 caller = ctx->rip;
  u64 callee = ctx->rbx;



  return 0;
}
```

A kprobe-based Approach

- Attach to indirect calls
  - kprobe attaches to most kernel text address
- Obtain source and target from registers
- Use `bpf_send_signal` to terminate offending task

```
...
48 89 44 24 08 mov   %rax,0x8(%rsp)
31 ff          xor   %edi,%edi
31 f6          xor   %esi,%esi
ff d3          call  *%rbx # indirect call
...
```

```
SEC("kprobe")
int kcfi_prog(struct *pt_regs ctx)
{
  u64 caller = ctx->rip;
  u64 callee = ctx->rbx;

  if (!call_allowed(caller, callee))
    bpf_send_signal(SIGKILL);

  return 0;
}
```

A kprobe-based Approach

- Attach to indirect calls
  - `kprobe` attaches to most kernel text address
- Obtain source and target from registers
- Use `bpf_send_signal` to terminate offending task
- **Problem:** `kprobe` uses interrupt by default
  - Significant context switch overhead
  - **~26x** on QEMU for a single indirect call

```
...
48 89 44 24 08 mov   %rax,0x8(%rsp)
31 ff                xor   %edi,%edi
31 f6                xor   %esi,%esi
ff d3                call  *%rbx # indirect call
...
```

```
SEC("kprobe")
int kcfi_prog(struct *pt_regs ctx)
{
  u64 caller = ctx->rip;
  u64 callee = ctx->rbx;

  if (!call_allowed(caller, callee))
    bpf_send_signal(SIGKILL);

  return 0;
}
```

What about jump optimization?

- Optimizes `kprobe` instrumentation into a synchronous jump

```
...
48 89 44 24 08    mov  %rax,0x8(%rsp)
31 ff             xor  %edi,%edi
31 f6             xor  %esi,%esi
ff d3             call *%rbx
...
```

What about jump optimization?

- Optimizes `kprobe` instrumentation into a synchronous jump

- Attaching to `call` cannot be optimized
  - `call` instructions are not boost-able

```
...
48 89 44 24 08      mov  %rax,0x8(%rsp)
31 ff               xor  %edi,%edi
31 f6               xor  %esi,%esi
ff d3               call *%rbx
...
```

What about jump optimization?

- Optimizes `kprobe` instrumentation into a synchronous jump
- Attaching to `call` cannot be optimized
  - `call` instructions are not boost-able
- Attaching to LLVM-KCFI instructions?

```
...
48 89 44 24 08    mov  %rax,0x8(%rsp)
31 ff             xor  %edi,%edi
31 f6             xor  %esi,%esi
41 ba 5b 4a 1a a9 mov  $0xa91a4a5b,%r10d
44 03 53 fc       add  -0x4(%rbx),%r10d
74 02             je   ffffffff8106b991
0f 0b             ud2
ff d3             call *%rbx
...
```

What about jump optimization?

- Optimizes `kprobe` instrumentation into a synchronous jump
- Attaching to `call` cannot be optimized
  - `call` instructions are not boost-able
- Attaching to LLVM-KCFI instructions?
  - LLVM-KCFI instrumentations are special :(
  - KCFI failure handler decodes these instructions
  - Overwriting the instruction with `kprobe` breaks the handler

```
...
48 89 44 24 08     mov  %rax,0x8(%rsp)
31 ff              xor  %edi,%edi
31 f6              xor  %esi,%esi
41 ba 5b 4a 1a a9  mov  $0xa91a4a5b,%r10d
44 03 53 fc        add  -0x4(%rbx),%r10d
74 02              je   ffffffff8106b991
0f 0b              ud2
ff d3              call *%rbx
...
```

What about jump optimization?

- Optimizes `kprobe` instrumentation into a synchronous jump
- Attaching to `call` cannot be optimized
  - `call` instructions are not boost-able
- Attaching to LLVM-KCFI instructions?
  - LLVM-KCFI instrumentations are special :(
  - KCFI failure handler decodes these instructions
  - Overwriting the instruction with `kprobe` breaks the handler

```
...
48 89 44 24 08     mov   %rax,0x8(%rsp)
31 ff              xor   %edi,%edi
31 f6              xor   %esi,%esi
41 ba 5b 4a 1a a9  mov   $0xa91a4a5b,%r10d
44 03 53 fc        add   -0x4(%rbx),%r10d
74 02              je    ffffffff8106b991
0f 0b              ud2
ff d3              call  *%rbx
...
```

Is there a more efficient solution?

An `fprobe`-based approach

- Derived from Daniel Borkmann's suggestion on using `fentry`.
- `BPF_TRACE_KPROBE_MULTI` allows attaching to functions via `fprobe`
  - program is executed under the same context when the function is called
  - More efficient than interrupts :)

```
...
ff d3                call *%rbx # indirect call
...
```

An `fprobe`-based approach

- Derived from Daniel Borkmann's suggestion on using `fentry`.
- `BPF_TRACE_KPROBE_MULTI` allows attaching to functions via `fprobe`
  - program is executed under the same context when the function is called
  - More efficient than interrupts :)

```
...
ff d3          call *%rbx # indirect call
...
```

```
<foo>:
0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)
8d 04 37       lea  (%rdi,%rsi,1),%eax
...
```

An `fprobe`-based approach

- Derived from Daniel Borkmann's suggestion on using `fentry`.

- `BPF_TRACE_KPROBE_MULTI` allows attaching to functions via `fprobe`
  - program is executed under the same context when the function is called
  - More efficient than interrupts :)

```
...
ff d3           call *%rbx # indirect call
...
```

```
<foo>:
e8 d7 00 00 00 call *0xd7(%rip)
8d 04 37       lea  (%rdi,%rsi,1),%eax
...
```

```
SEC("kprobe.multi")
int kcfi_prog(struct *pt_regs ctx)
{




   return 0;
}
```

An fprobe-based approach

- Derived from Daniel Borkmann's suggestion on using `fentry`.

- `BPF_TRACE_KPROBE_MULTI` allows attaching to functions via `fprobe`
  - program is executed under the same context when the function is called
  - More efficient than interrupts :)

- Obtain caller/callee from stack traces
  - callee is the currently probed function
  - use `bpf_get_stack` to get caller address

```
...
ff d3          call *%rbx # indirect call
...
```

```
<foo>:
e8 d7 00 00 00 call *0xd7(%rip)
8d 04 37       lea  (%rdi,%rsi,1),%eax
...
```

```
SEC("kprobe.multi")
int kcfi_prog(struct *pt_regs ctx)
{
  u64 st[2] = { 0 };
  bpf_get_stack(st, sizeof(st), 0);



  return 0;
}
```

An `fprobe`-based approach

- Derived from Daniel Borkmann's suggestion on using `fentry`.
- `BPF_TRACE_KPROBE_MULTI` allows attaching to functions via `fprobe`
  - program is executed under the same context when the function is called
  - More efficient than interrupts :)
- Obtain caller/callee from stack traces
  - callee is the currently probed function
  - use `bpf_get_stack` to get caller address
- Enforcement is similar to `kprobe`

```
...
ff d3            call *%rbx # indirect call
...
```

```
<foo>:
e8 d7 00 00 00   call *0xd7(%rip)
8d 04 37         lea  (%rdi,%rsi,1),%eax
...
```

```
SEC("kprobe.multi")
int kcfi_prog(struct *pt_regs ctx)
{
  u64 st[2] = { 0 };
  bpf_get_stack(st, sizeof(st), 0);

  if (!call_allowed(st[1], st[0]))
    bpf_send_signal(SIGKILL);

  return 0;
}
```

An fprobe-based approach

- Derived from Daniel Borkmann's suggestion on using `fentry`.
- `BPF_TRACE_KPROBE_MULTI` allows attaching to functions via `fprobe`
  - program is executed under the same context when the function is called
  - More efficient than interrupts :)
- Obtain caller/callee from stack traces
  - callee is the currently probed function
  - use `bpf_get_stack` to get caller address
- Enforcement is similar to `kprobe`
- Requires using LLVM-KCFI

```
...
ff d3             call *%rbx # indirect call
...
```

```
<foo>:
e8 d7 00 00 00    call *0xd7(%rip)
8d 04 37          lea  (%rdi,%rsi,1),%eax
...
```

```
SEC("kprobe.multi")
int kcfi_prog(struct *pt_regs ctx)
{
  u64 st[2] = { 0 };
  bpf_get_stack(st, sizeof(st), 0);

  if (!call_allowed(st[1], st[0]))
    bpf_send_signal(SIGKILL);

  return 0;
}
```

Limitations of using `fprobe`

- Less coverage than LLVM-KCFI
  - `noinstr`/`notrace` functions
  - Tracing subsystem and library functions are compiled without `fprobe` support
  - **~10K** (out of 59K) functions cannot be attached
- `fprobe` doesn't distinguish between direct and indirect calls
  - The program always executes when the function is invoked
  - **258K** direct calls vs. **15K** indirect calls
  - **7x** slowdown for LEBench on QEMU

Existing eBPF attachment is limited

Existing eBPF attachment is limited

Mechanism
`kprobe`
`fprobe`

Existing eBPF attachment is limited

Mechanism	Hook point
`kprobe`	Indirect call
`fprobe`	Function entry

Existing eBPF attachment is limited

Mechanism	Hook point	eBPF invocation
`kprobe`	Indirect call	Interrupt
`fprobe`	Function entry	Synchronous call

Existing eBPF attachment is limited

Mechanism	Hook point	eBPF invocation	Overhead
`kprobe`	Indirect call	Interrupt	Context switch
`fprobe`	Function entry	Synchronous call	Function call

Existing eBPF attachment is limited

Mechanism	Hook point	eBPF invocation	Overhead	KCFI coverage
`kprobe`	Indirect call	Interrupt	Context switch	Same as LLVM-KCFI*
`fprobe`	Function entry	Synchronous call	Function call	17% less than LLVM-KCFI

* kprobe cannot attach to indirect calls in its own infrastructure

Existing eBPF attachment is limited

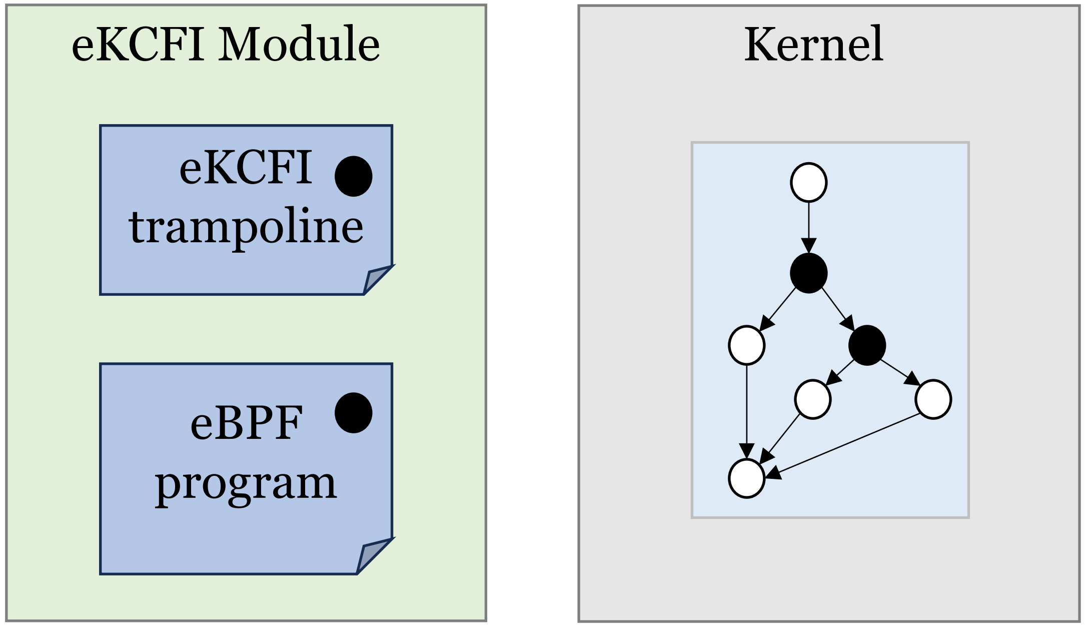
Mechanism	Hook point	eBPF invocation	Overhead	KCFI coverage
`kprobe`	Indirect call	Interrupt	Context switch	Same as LLVM-KCFI*
`fprobe`	Function entry	Synchronous call	Function call	17% less than LLVM-KCFI

- A new attachment mechanism is desired:
  - Synchronous invocation
  - Instrument precisely indirect call sites covered by LLVM-KCFI

* kprobe cannot attach to indirect calls in its own infrastructure

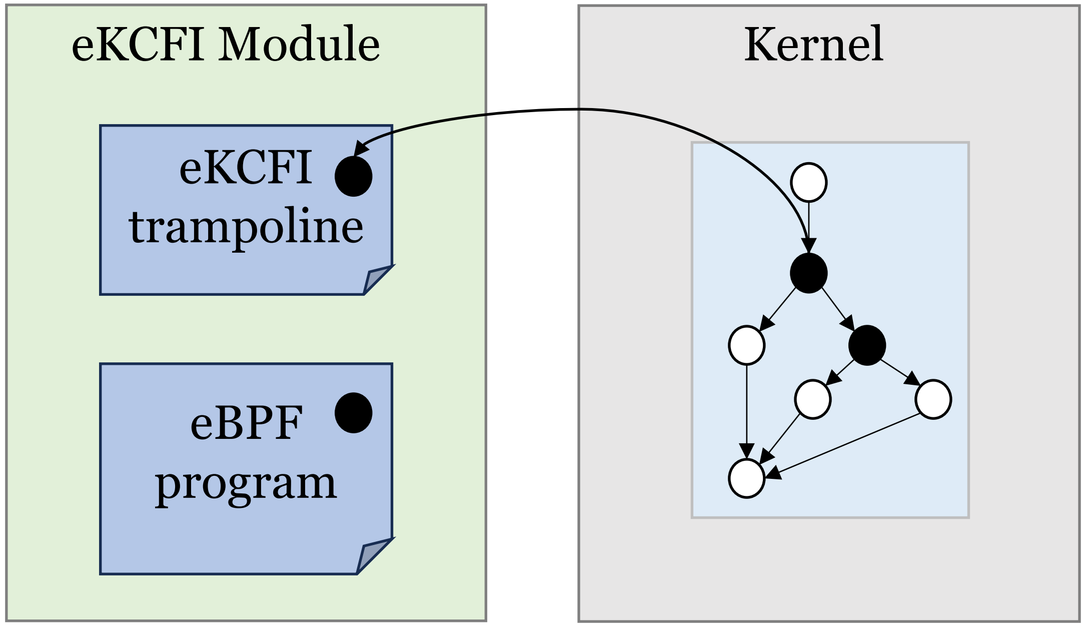
eKCFI Overview

- A new way to hook eBPF programs to indirect call sites
  - Instrument kernel code to create hooking point at indirect calls
  - Allows synchronous invocation of eBPF programs
- The policy program decides whether to allow the control-flow transfer
  - Continue execution
  - Kernel panic



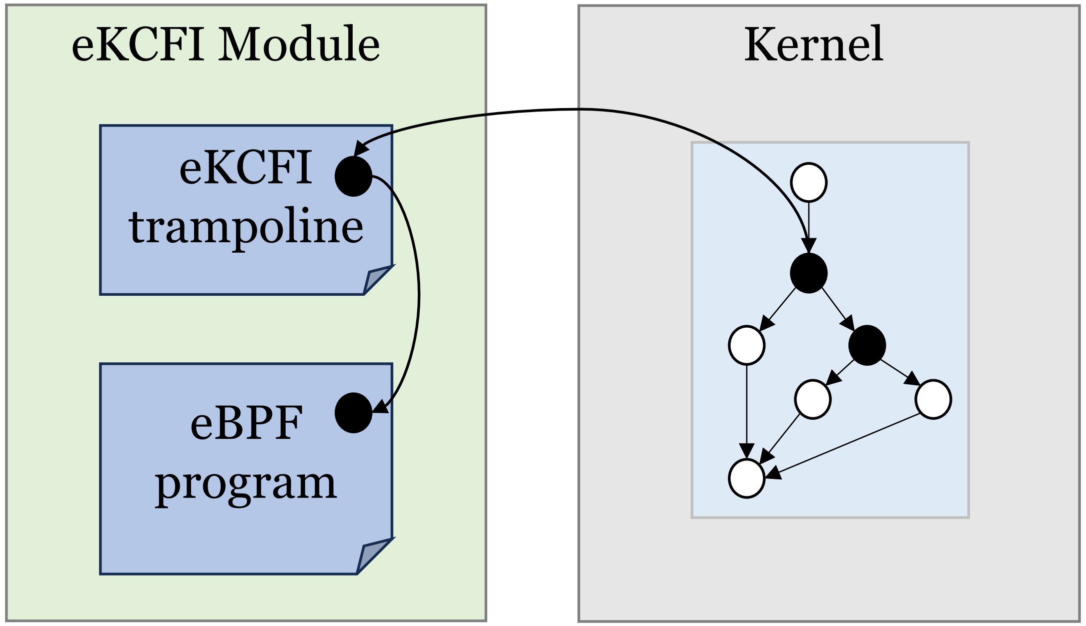
eKCFI Overview

- A new way to hook eBPF programs to indirect call sites
  - Instrument kernel code to create hooking point at indirect calls
  - Allows synchronous invocation of eBPF programs
- The policy program decides whether to allow the control-flow transfer
  - Continue execution
  - Kernel panic



eKCFI Overview

- A new way to hook eBPF programs to indirect call sites
  - Instrument kernel code to create hooking point at indirect calls
  - Allows synchronous invocation of eBPF programs
- The policy program decides whether to allow the control-flow transfer
  - Continue execution
  - Kernel panic



Instrumenting kernel code

- Leveraging the kernel text patching mechanism used by `fprobe`

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
ff d3          call *%rbx # indirect call
...
```

Instrumenting kernel code

- Leveraging the kernel text patching mechanism used by `fprobe`
- Using LLVM to generate instructions at indirect control-flow transfers
  - a `mov` to store call target in `rax`
  - a 5-byte nop for dynamic rewriting

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
0f 1f 44 00 08 nopl 0x8(%rax,%rax,1)
ff d3          call *%rbx # indirect call
...
```

Instrumenting kernel code

- Leveraging the kernel text patching mechanism used by `fprobe`
- Using LLVM to generate instructions at indirect control-flow transfers
  - a `mov` to store call target in `rax`
  - a 5-byte nop for dynamic rewriting
- Dynamically rewriting the `nop` into a call to the `eKCFI` trampoline

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

Instrumenting kernel code

- Leveraging the kernel text patching mechanism used by `fprobe`
- Using LLVM to generate instructions at indirect control-flow transfers
  - a `mov` to store call target in `rax`
  - a 5-byte nop for dynamic rewriting
- Dynamically rewriting the nop into a call to the `eKCFI` trampoline

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

✅ Saves registers

Instrumenting kernel code

- Leveraging the kernel text patching mechanism used by `fprobe`
- Using LLVM to generate instructions at indirect control-flow transfers
  - a `mov` to store call target in `rax`
  - a 5-byte nop for dynamic rewriting
- Dynamically rewriting the `nop` into a call to the `eKCFI` trampoline

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

- ✓ Saves registers
- ✓ Obtains callee from `rax`, caller from its return address

Instrumenting kernel code

- Leveraging the kernel text patching mechanism used by `fprobe`
- Using LLVM to generate instructions at indirect control-flow transfers
  - a `mov` to store call target in `rax`
  - a 5-byte nop for dynamic rewriting
- Dynamically rewriting the `nop` into a call to the `eKCFI` trampoline

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

- ✓ Saves registers
- ✓ Obtains callee from `rax`, caller from its return address
- ✓ Prevents recursive kCFI instrumentation

Instrumenting kernel code

- Leveraging the kernel text patching mechanism used by `fprobe`
- Using LLVM to generate instructions at indirect control-flow transfers
  - a `mov` to store call target in `rax`
  - a 5-byte nop for dynamic rewriting
- Dynamically rewriting the nop into a call to the `eKCFI` trampoline

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

- ✓ Saves registers
- ✓ Obtains callee from `rax`, caller from its return address
- ✓ Prevents recursive kCFI instrumentation
- ✓ Invokes eBPF program with kCFI context

Instrumenting kernel code

- Leveraging the kernel text patching mechanism used by `fprobe`
- Using LLVM to generate instructions at indirect control-flow transfers
  - a `mov` to store call target in `rax`
  - a 5-byte `nop` for dynamic rewriting
- Dynamically rewriting the `nop` into a call to the `eKCFI` trampoline

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

- ✓ Saves registers
- ✓ Obtains callee from `rax`, caller from its return address
- ✓ Prevents recursive kCFI instrumentation
- ✓ Invokes eBPF program with kCFI context
- ✓ Interprets return value of eBPF program

Instrumenting kernel code

- `eKCFI` trampoline invokes the eBPF policy program

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

```
SEC("ekcfi")
int kcfi_prog(struct *ekcfi_ctx ctx)
{


}
```

Instrumenting kernel code

- `eKCFI` trampoline invokes the eBPF policy program

- The trampoline provides caller and callee information in context

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

```
SEC("ekcfi")
int kcfi_prog(struct *ekcfi_ctx ctx)
{
  u64 caller = ctx->caller;
  u64 callee = ctx->callee;


}
```

Instrumenting kernel code

- eKCFI trampoline invokes the eBPF policy program
- The trampoline provides caller and callee information in context
- Enforcement implemented by program return value
  - interpreted by trampoline

```
...
48 89 44 24 08 mov %rax,0x8(%rsp)
31 ff          xor %edi,%edi
31 f6          xor %esi,%esi
48 89 d8       mov %rbx,%rax
e8 d7 00 00 00 call *0xd7(%rip) #trampoline
ff d3          call *%rbx # indirect call
...
```

```
SEC("ekcfi")
int kcfi_prog(struct *ekcfi_ctx ctx)
{
  u64 caller = ctx->caller;
  u64 callee = ctx->callee;

  if (!call_allowed(caller, callee))
    return EKCFI_RET_PANIC;

  return EKCFI_RET_ALLOW;
}
```

Adding eKCFI to the design space

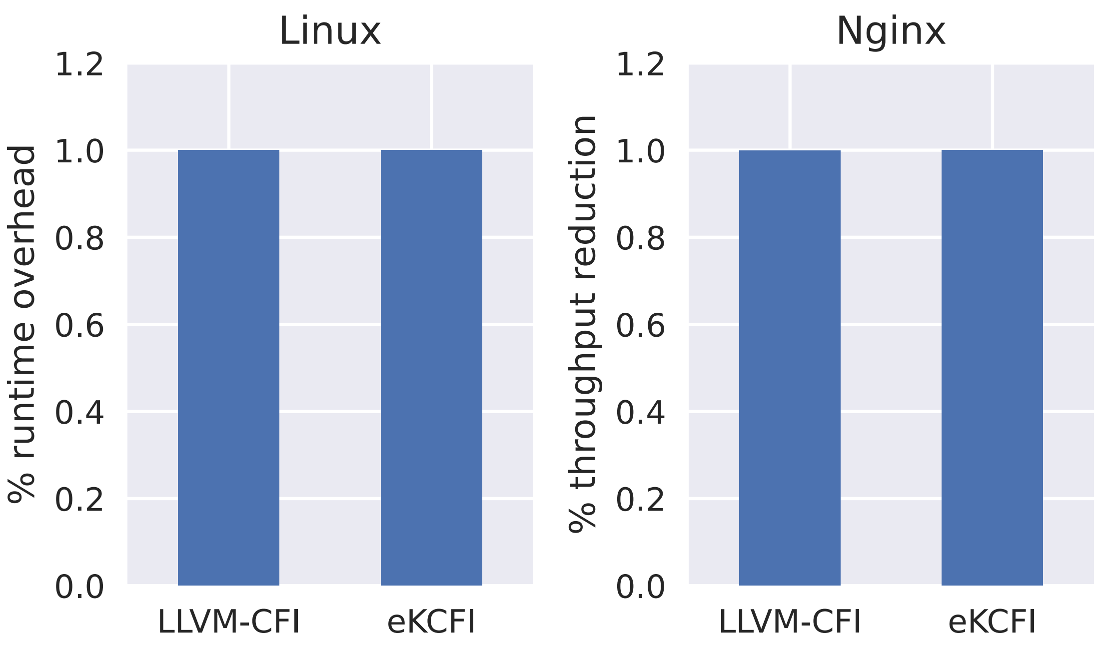
Mechanism	Hook point	eBPF invocation	Overhead	KCFI coverage
`kprobe`	Indirect call	Interrupt	Context switch	Same as LLVM-KCFI
`fprobe`	Function entry	Synchronous call	Function call	17% less than LLVM-KCFI

Adding eKCFI to the design space

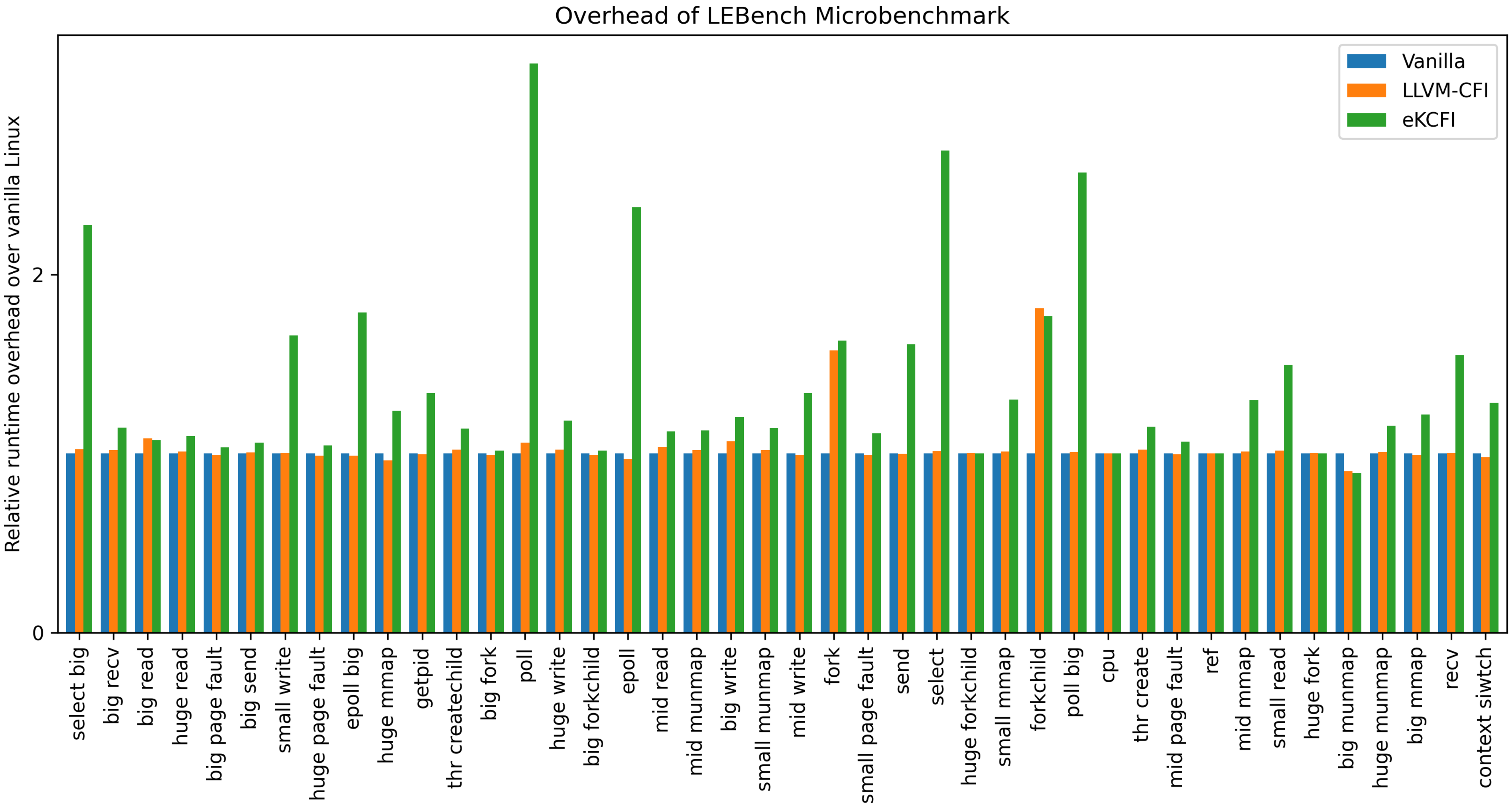
Mechanism	Hook point	eBPF invocation	Overhead	KCFI coverage
`kprobe`	Indirect call	Interrupt	Context switch	Same as LLVM-KCFI
`fprobe`	Function entry	Synchronous call	Function call	17% less than LLVM-KCFI
`eKCFI`	Indirect call	Synchronous call	Function call	Same as LLVM-KCFI

Application Performance

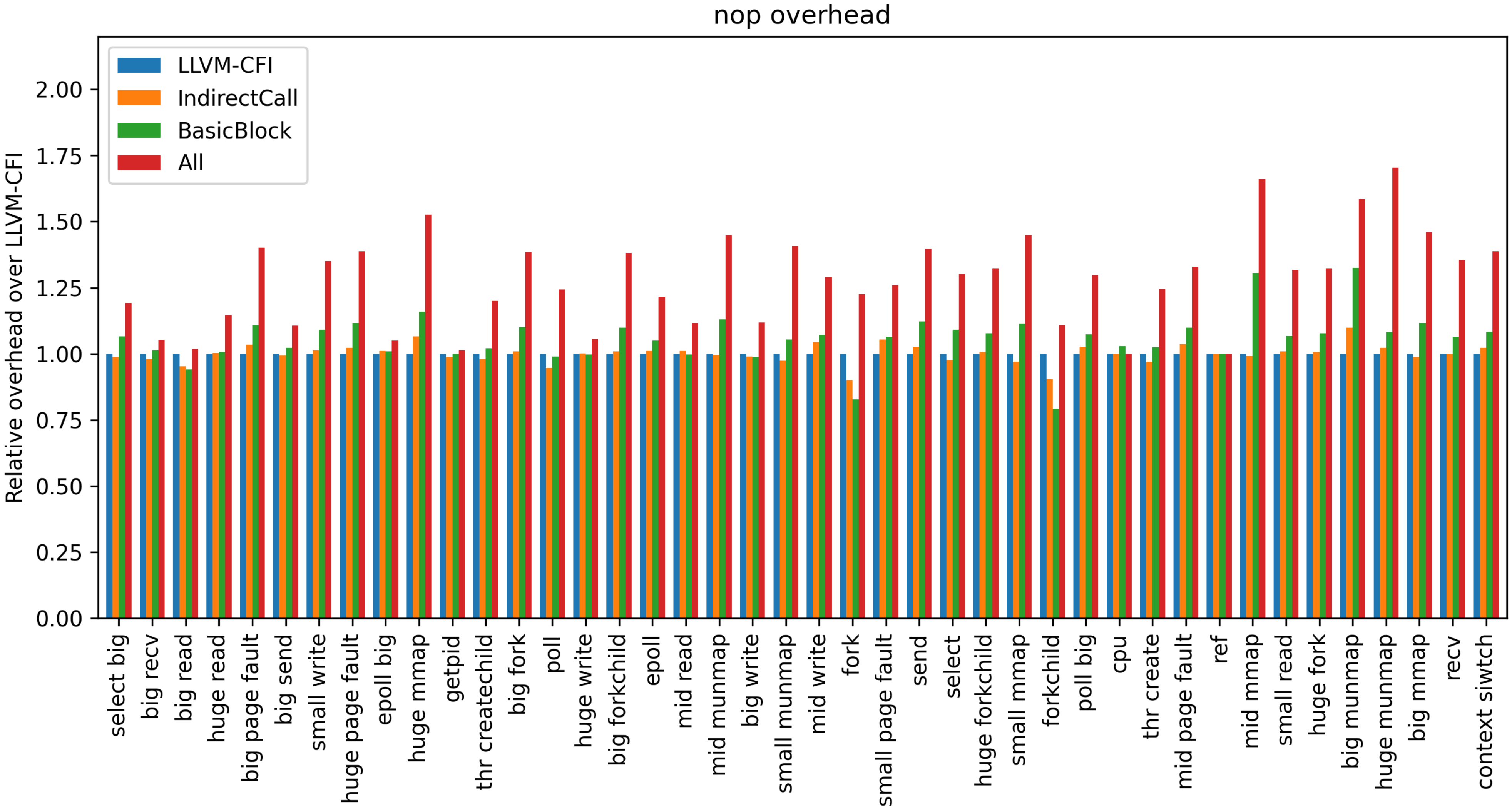
- Evaluate on NGINX and Linux kernel compilation
- Policy: enforce a fine-grained CFG from dynamic traces
- `eKCFI` achieves roughly the same performance comparing to LLVM-KCFI



Microbenchmark Performance



Nops overhead



Discussion and Limitation

- Limitations of `eKCFI` (or eBPF-based KCFI in general)
  - Need to trust the eBPF subsystem
  - Attackers may be able to corrupt memory of helper code or map content
- Protection and Mitigation
  - Hardware-based mechanisms (e.g. MPK) might be useful for maps
  - Protecting helper functions is still hard
    - helpers call deep into core kernel code
- Complements LLVM-KCFI, not necessarily replace

Conclusion

- eBPF can make kernel CFI (KCFI) more flexible and usable.
- Existing eBPF mechanism is insufficient for practical KCFI
  - Performance and hook point limitations
- We develop `eKCFI`, an eBPF-based KCFI framework
  - A new hooking mechanism for efficient indirect call checking

Backup slides

Call site equivalence classes

# of targets	LLVM-KCFI	eKCFI
1	18.5%	70.8%
≤ 5	48.2%	95.6%
≥ 100	10.9%	0.1%

Comparison of equivalent classes for different KCFI techniques considering 742 dynamically traced call sites.