Safe Kernel Extensions

Guest Lecture for CS 423

Jinghao Jia

Tenured Grad Student

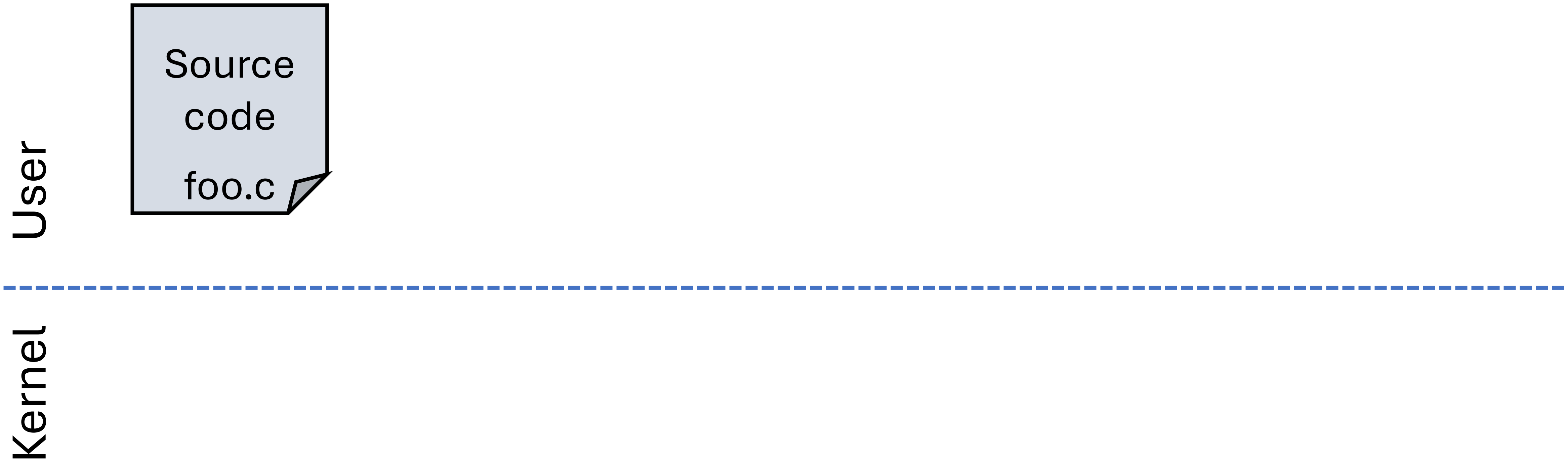
About me

- I was a Ph.D. student at UIUC working on safe kernel extensions
  - Advised by Professor Tianyin Xu from UIUC and Professor Dan Williams from Virginia Tech
  - Graduated this spring
- Have been a TA for CS423 for 3 semesters

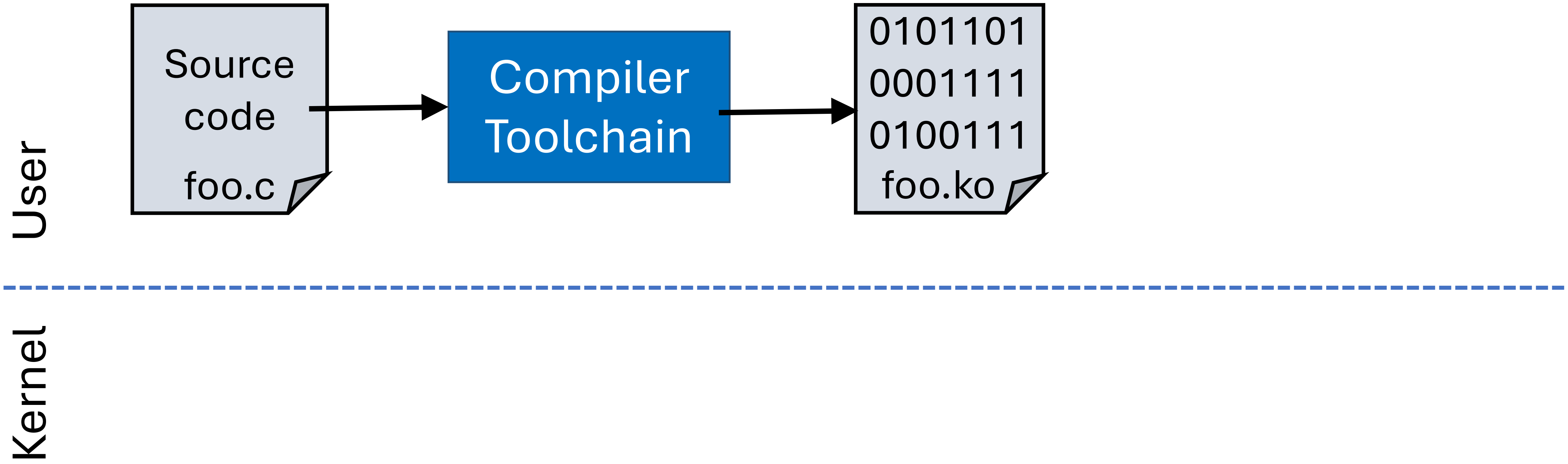
Kernel extensibility is an essential OS capability

- Customizing kernel functionality for **diverse needs**
  - Supporting specialized OS features and policies
- No **complexity** added to core kernel code
  - Modularization produces maintainable code
- No need to perform **disruptive** kernel reboots
  - No effect on service availability

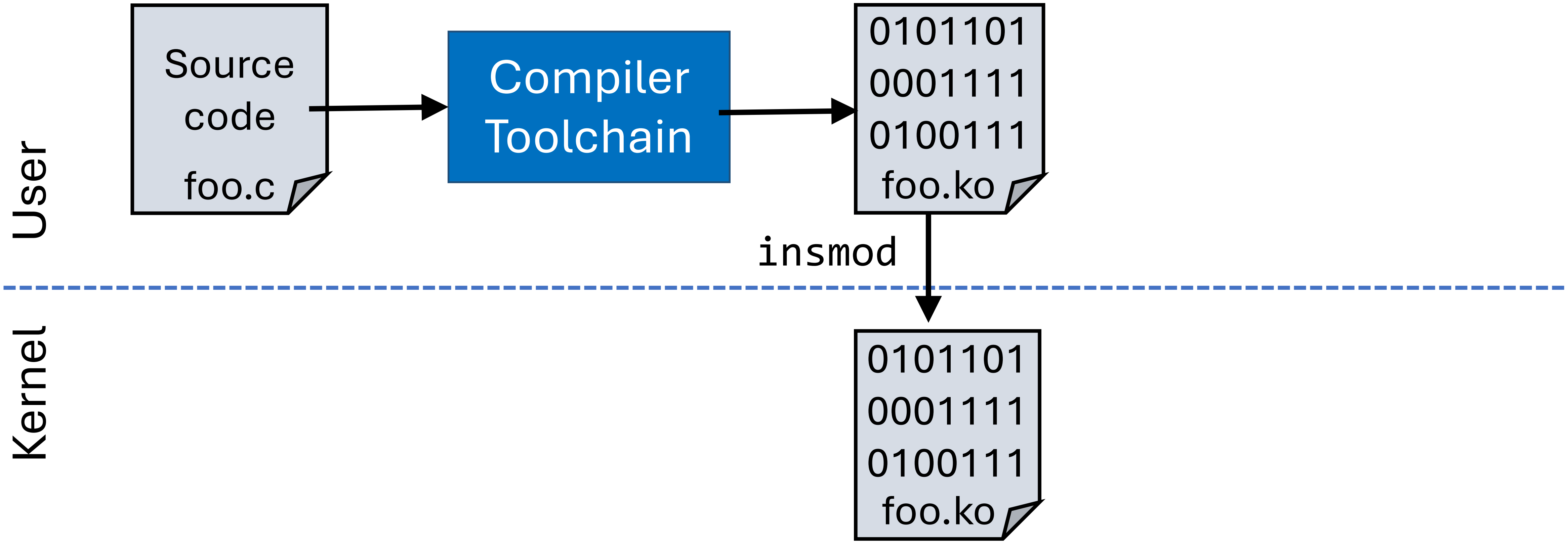
Traditional kernel extensibility: loadable modules



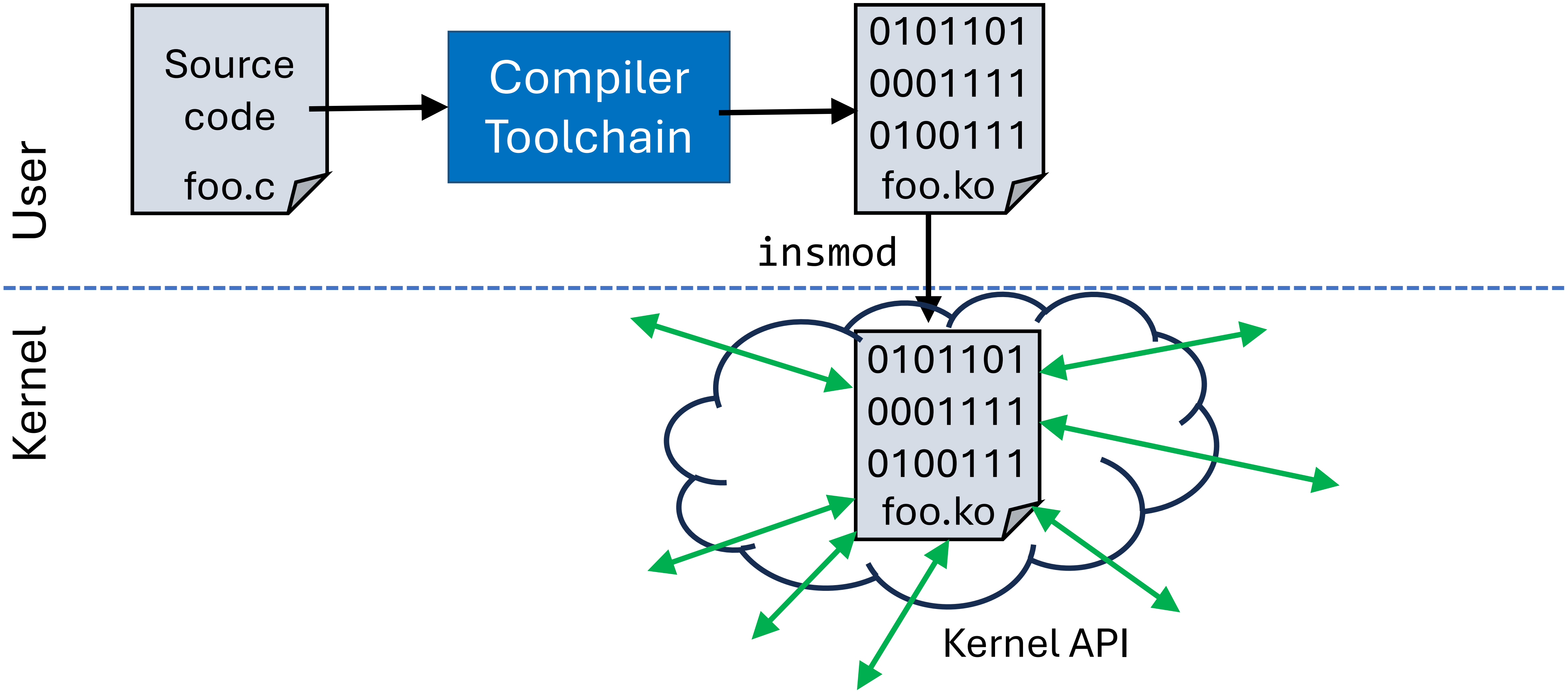
Traditional kernel extensibility: loadable modules



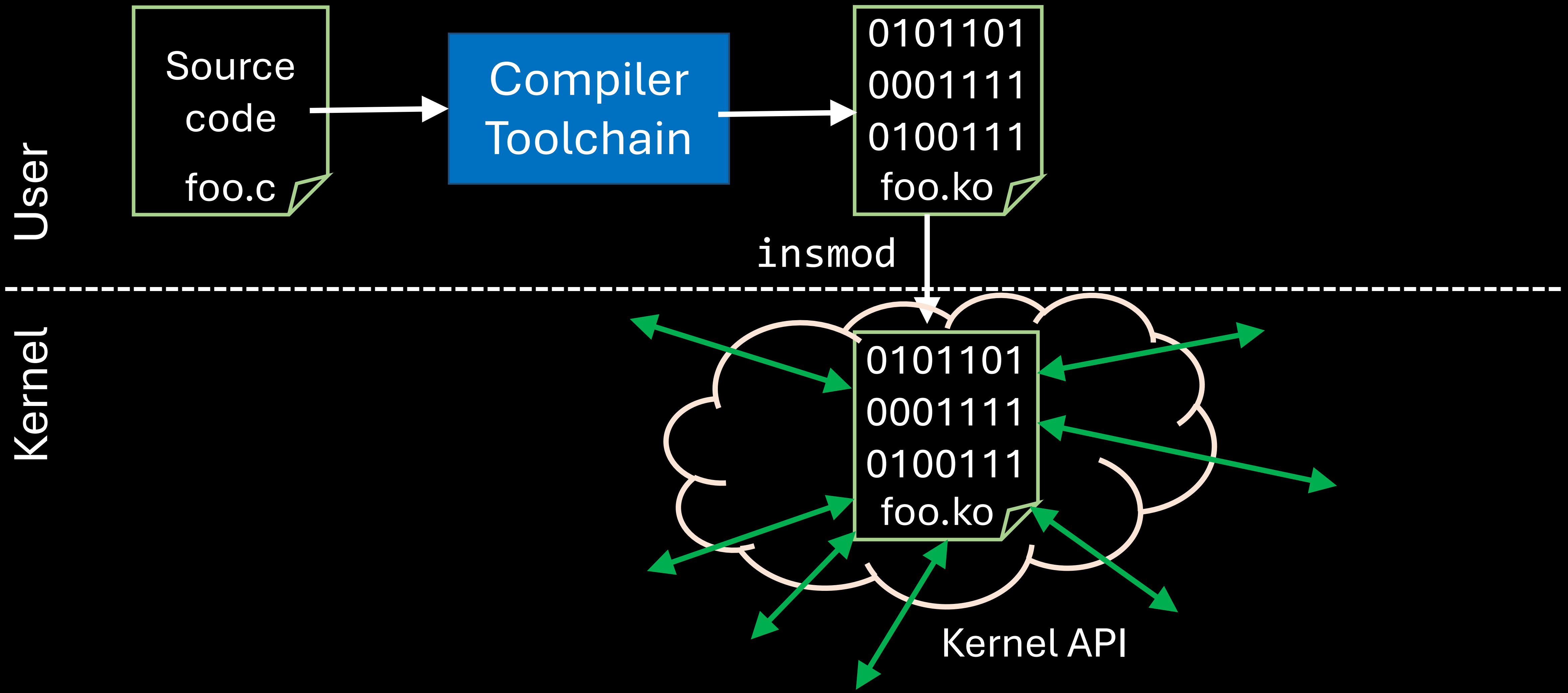
Traditional kernel extensibility: loadable modules



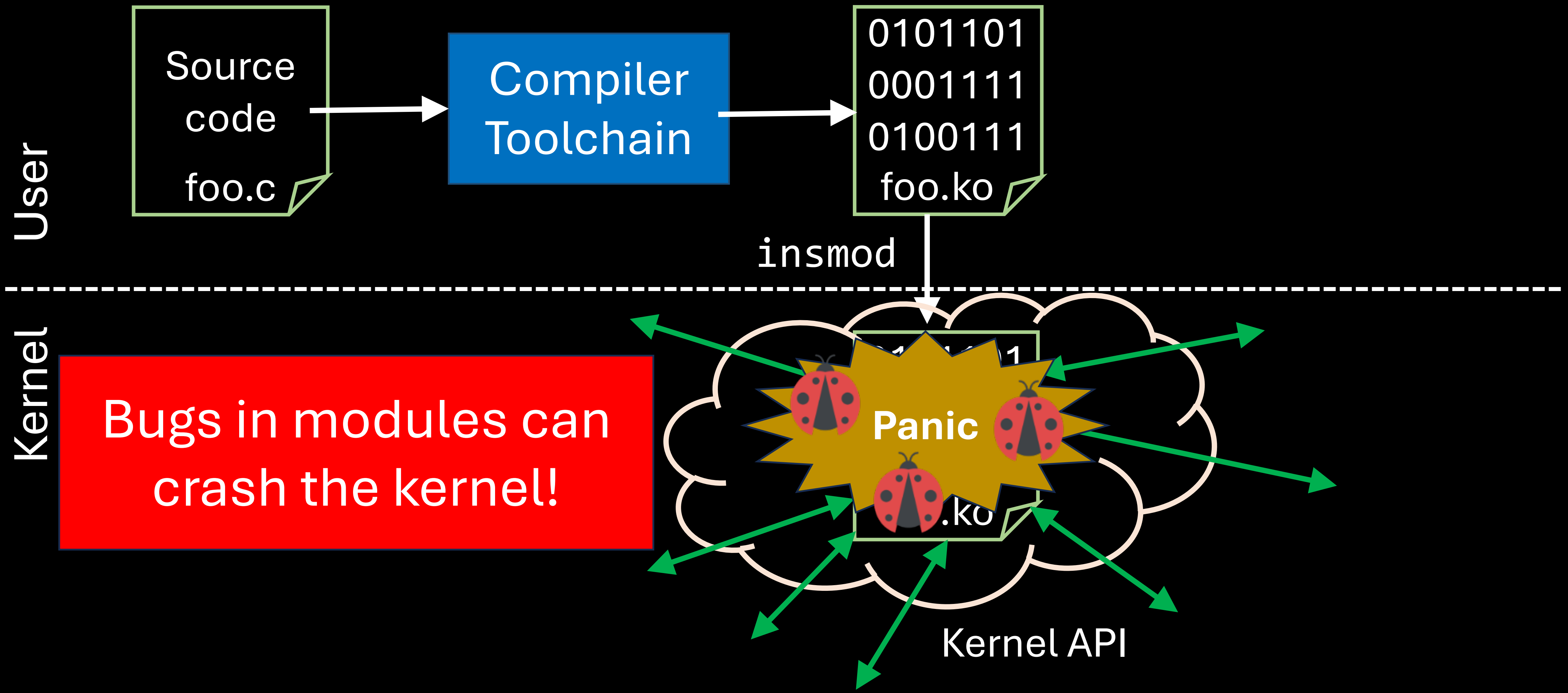
Traditional kernel extensibility: loadable modules



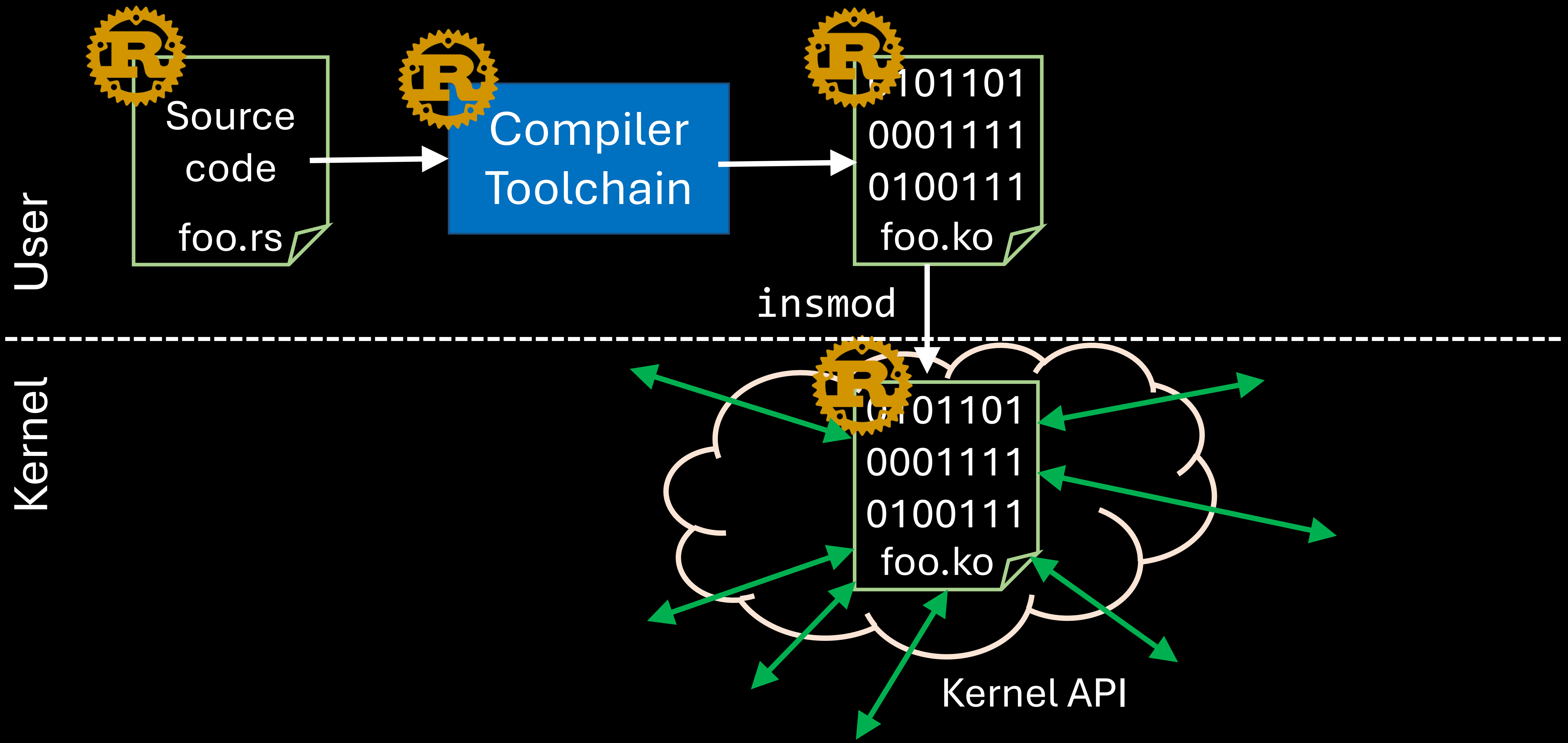
Loadable kernel modules are inherently unsafe!



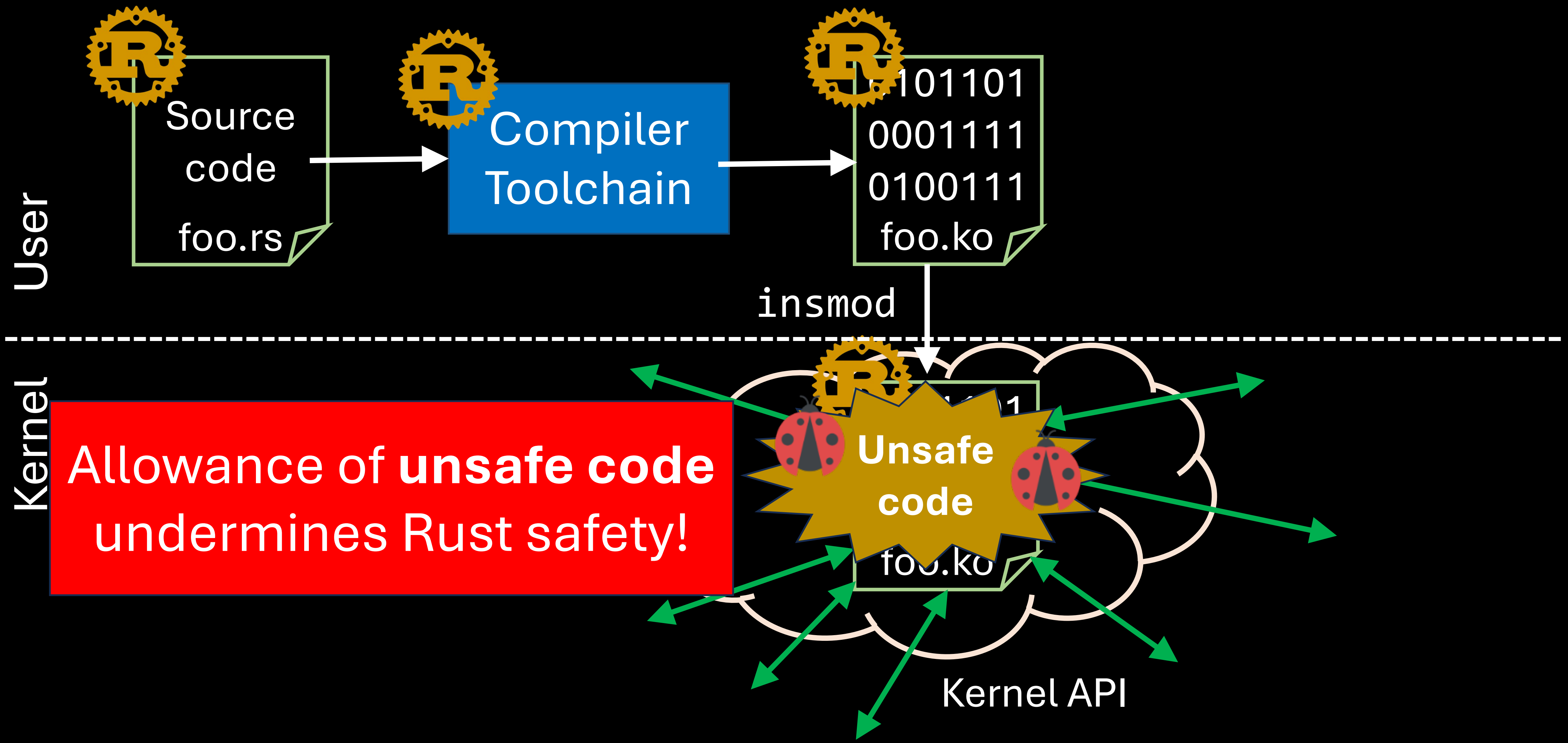
Loadable kernel modules are inherently unsafe!



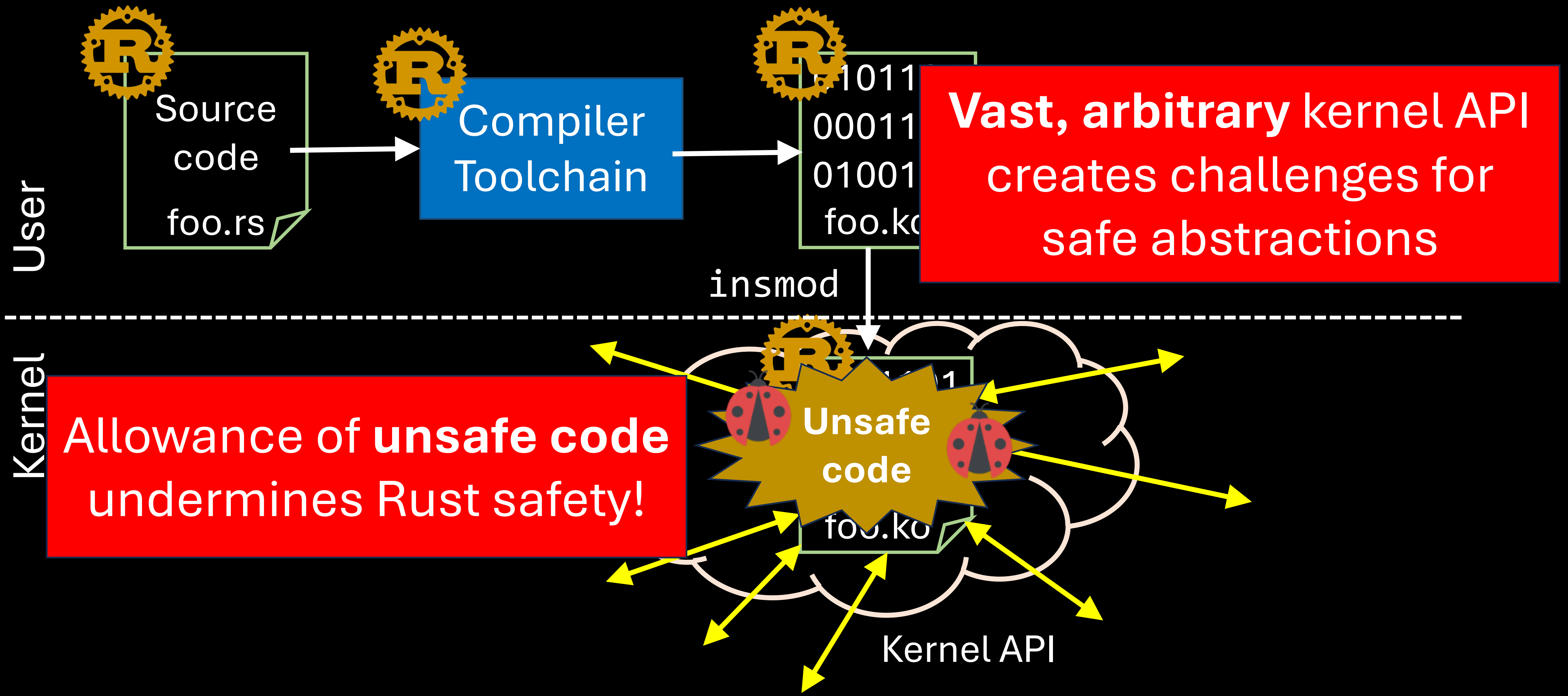
Rust-based kernel modules does not help



Rust-based kernel modules does not help



Rust-based kernel modules does not help



eBPF extensions are taking over the OS kernel

eBPF

cilium

Katran

BPF COMPILER COLLECTION

bpftrace

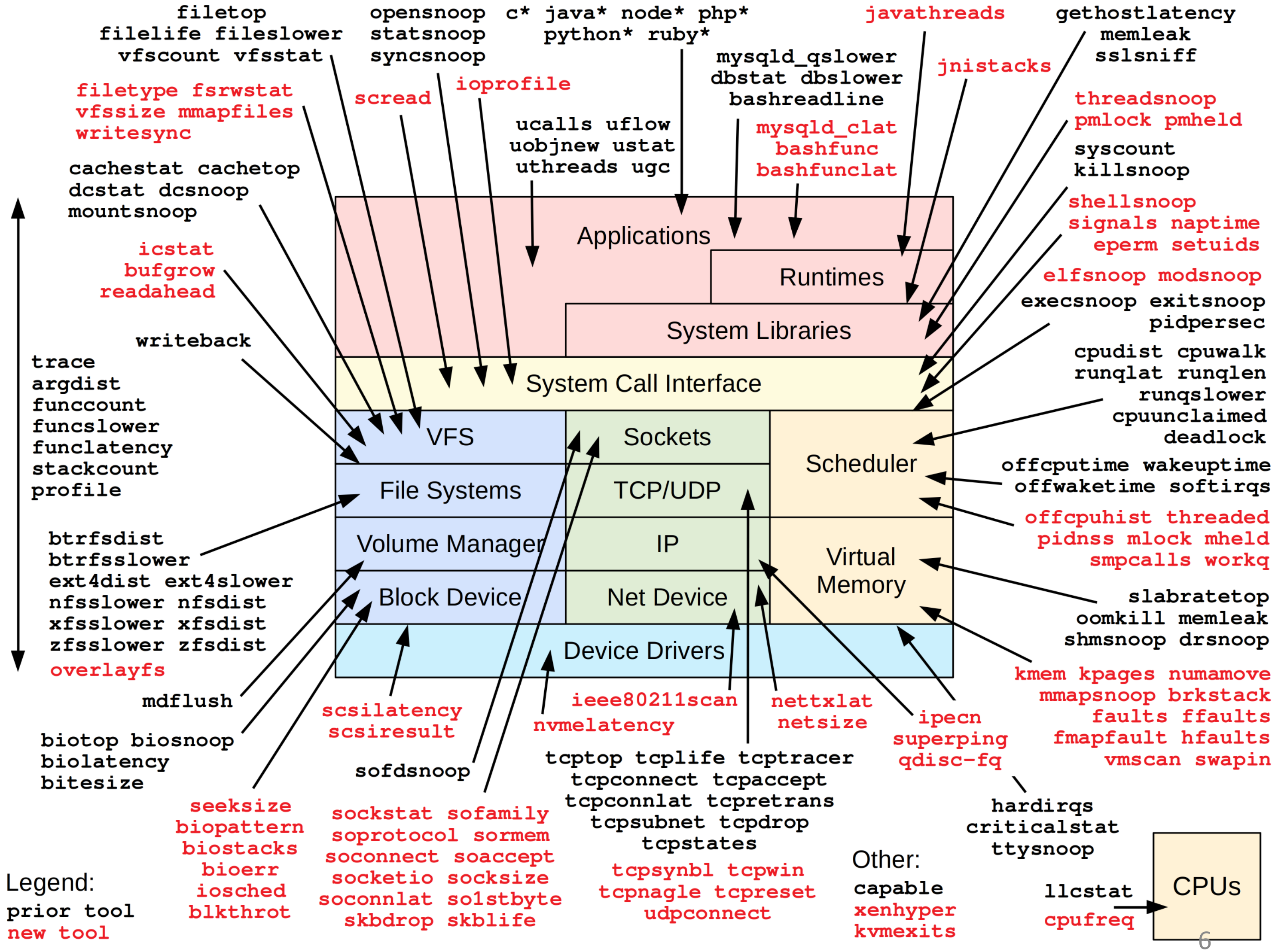


Figure credit: Brendan Gregg

eBPF extensions are taking over the OS kernel

BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing

Yoann Ghigoff, *Orange Labs, Sorbonne Université, Inria, LIP6;* Julien Sopena, *Sorbonne Université, LIP6;* Kahina Lazri, *Orange Labs;* Antoine Blin, *Gandi;* Gilles Muller, *Inria*

XRP: In-Kernel Storage Functions with eBPF

Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, and Junfeng Yang, *Columbia University;* Amy Tai, *Google;* Ryan Stutsman, *University of Utah;* Asaf Cidon, *Columbia University*

DINT: Fast In-Kernel Distributed Transactions with eBPF

Yang Zhou, *Harvard University;* Xingyu Xiang, *Peking University;* Matthew Kiley, *Harvard University;* Sowmya Dharanipragada, *Cornell University;* Minlan Yu, *Harvard University*

Electrode: Accelerating Distributed Protocols with eBPF

Yang Zhou, *Harvard University;* Zezhou Wang, *Peking University;* Sowmya Dharanipragada, *Cornell University;* Minlan Yu, *Harvard University*

eBPF extensions are taking over the OS kernel

BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing

Yoann Ghigoff, *Orange Labs, Sorbonne Université, Inria, LIP6;* Julien Sopena, *Sorbonne Université, LIP6;* Kahina Lazri, *Orange Labs;* Antoine Blin, *Gandi;* Gilles Muller, *Inria*

XRP: In-Kernel Storage Functions with eBPF

Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, and Junfeng Yang, *Columbia University;* Amy Tai, *Google;* Ryan Stutsman, *University of Utah;* Asaf Cidon, *Columbia University*

DINT: Fast In-Kernel Distributed Transactions with eBPF

Yang Zhou, *Harvard University;* Xingyu Xiang, *Peking University;* Matthew Kiley, *Harvard University;* Sowmya Dharanipragada, *Cornell University;* Minlan Yu, *Harvard University*

Electrode: Accelerating Distributed Protocols with eBPF

Yang Zhou, *Harvard University;* Zezhou Wang, *Peking University;* Sowmya Dharanipragada, *Cornell University;* Minlan Yu, *Harvard University*

LINUX PLUMBERS CONFERENCE September 20-24, 2021

eBPF in CPU Scheduler

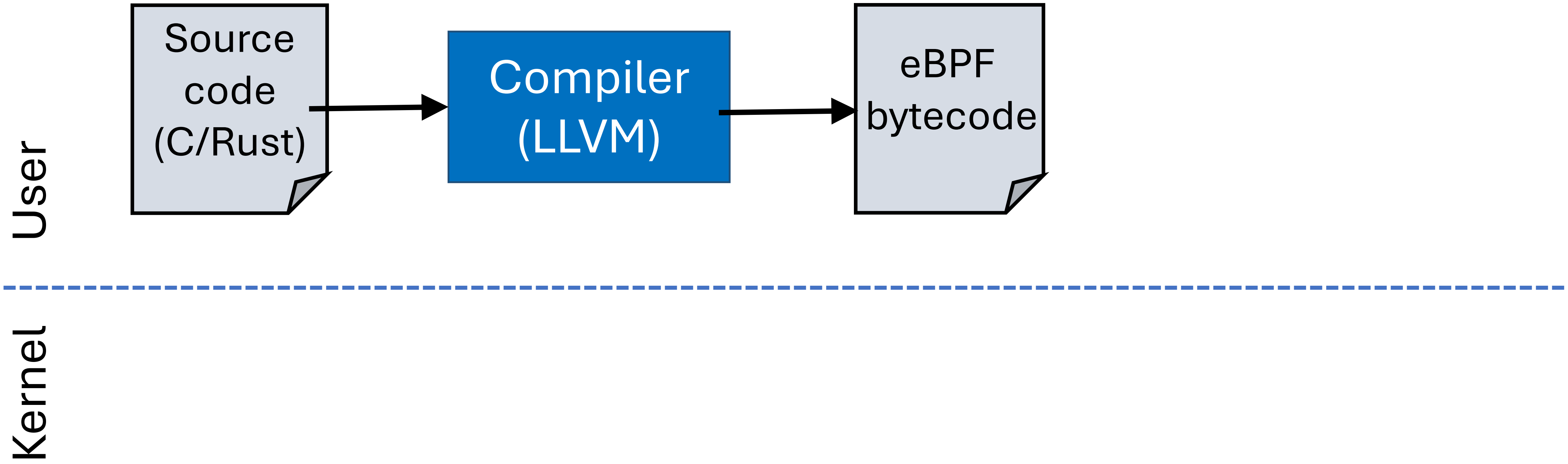
Hao Luo <haoluo@google.com>
Barret Rhoden <brho@google.com>

Towards Programmable Memory Management with eBPF

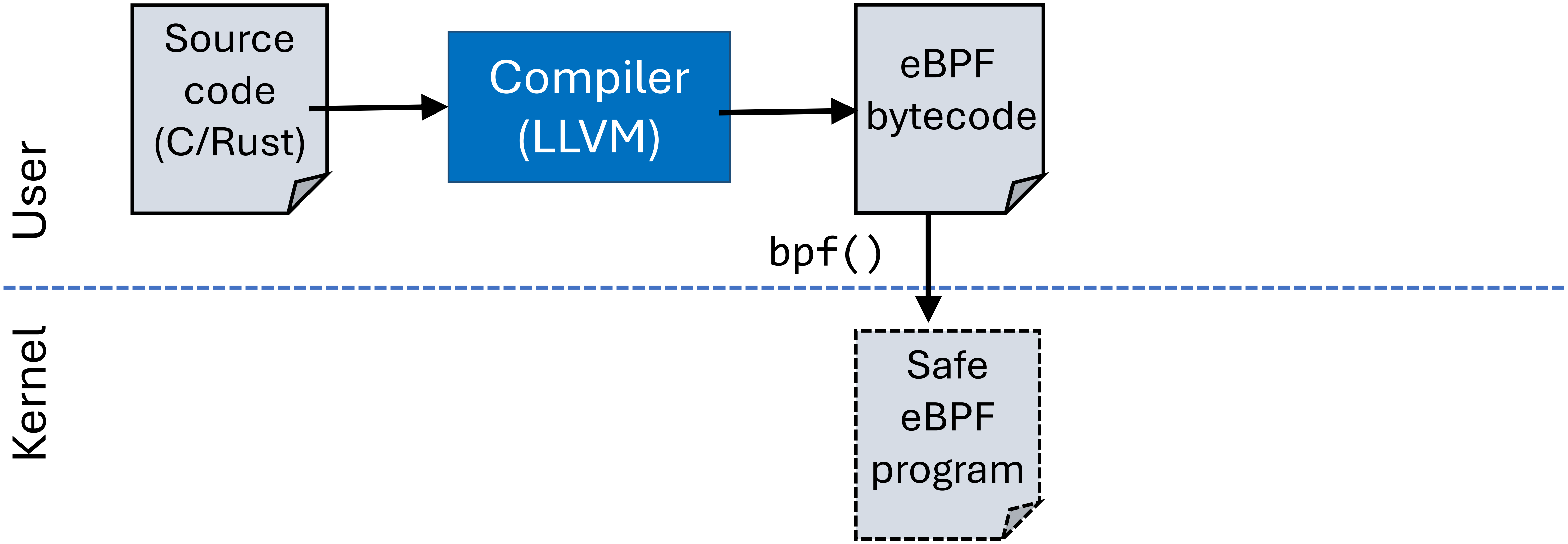
Presented by Kaiyang Zhao <kaiyang2@cs.cmu.edu>

Basic principle of eBPF: Safety verification

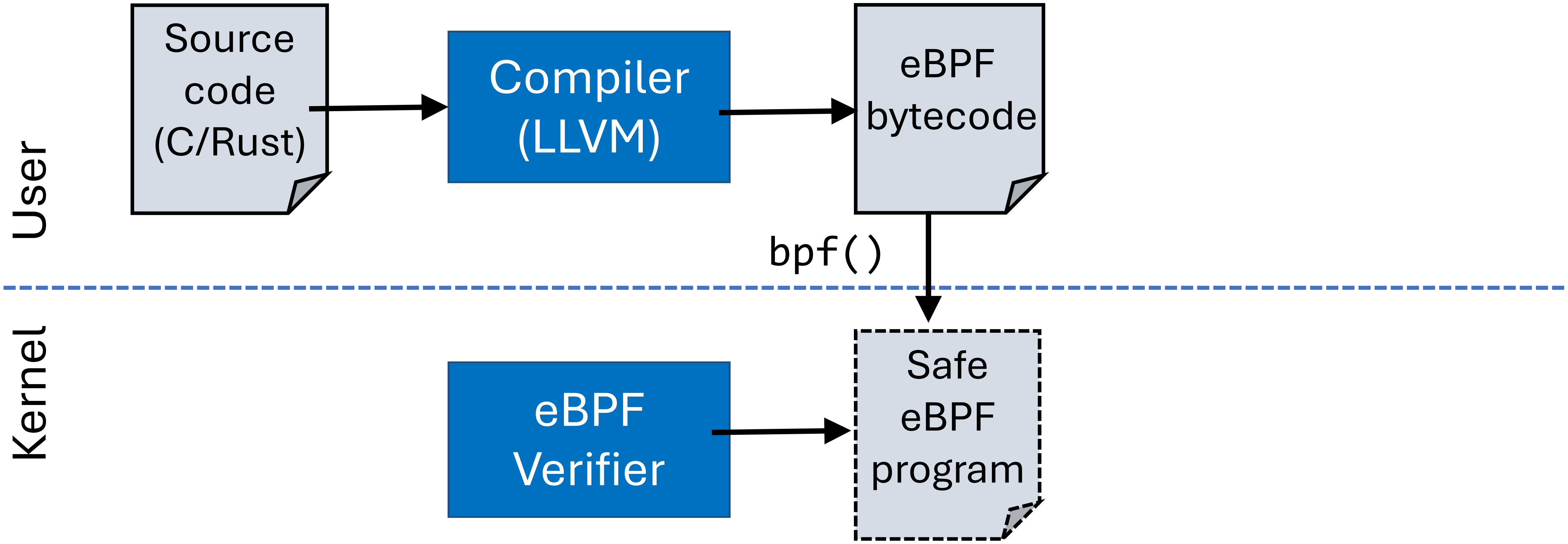
Basic principle of eBPF: Safety verification



Basic principle of eBPF: Safety verification

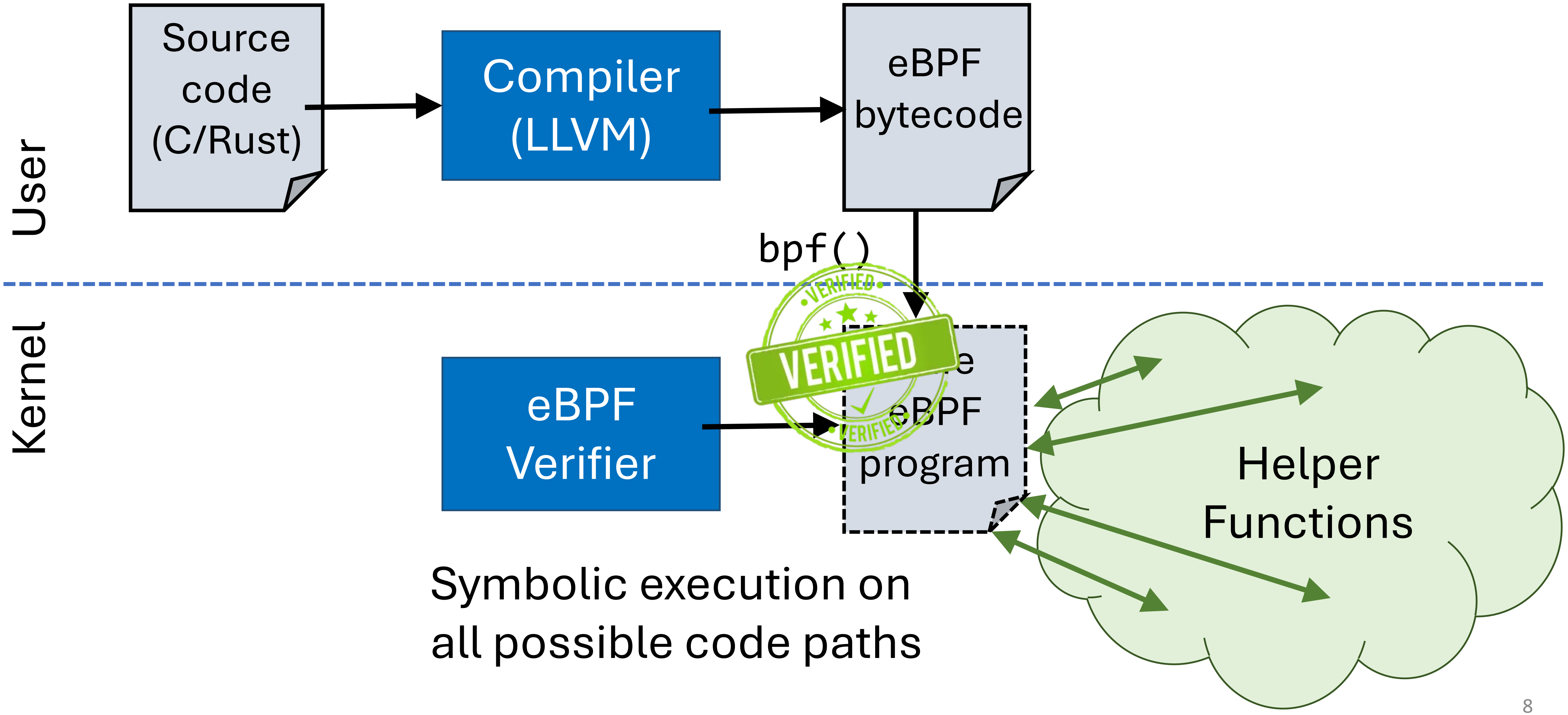


Basic principle of eBPF: Safety verification



Symbolic execution on all possible code paths

Basic principle of eBPF: Safety verification

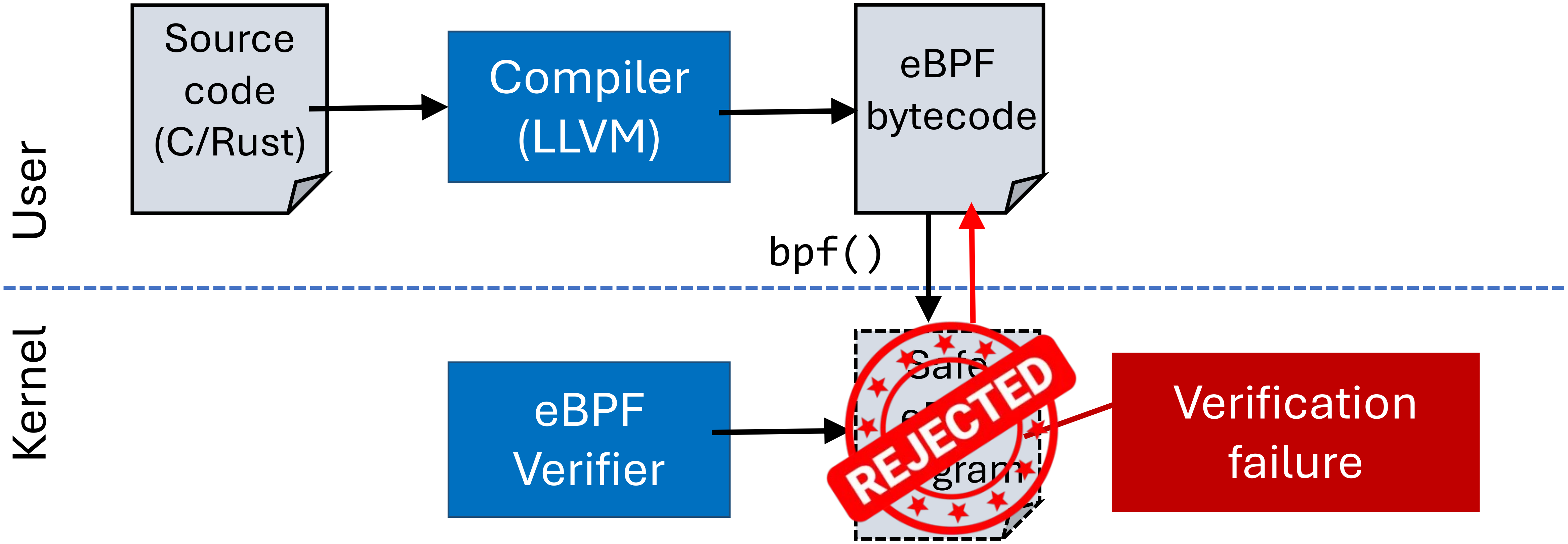


Symbolic execution on all possible code paths

Basic principle of eBPF: Safety verification

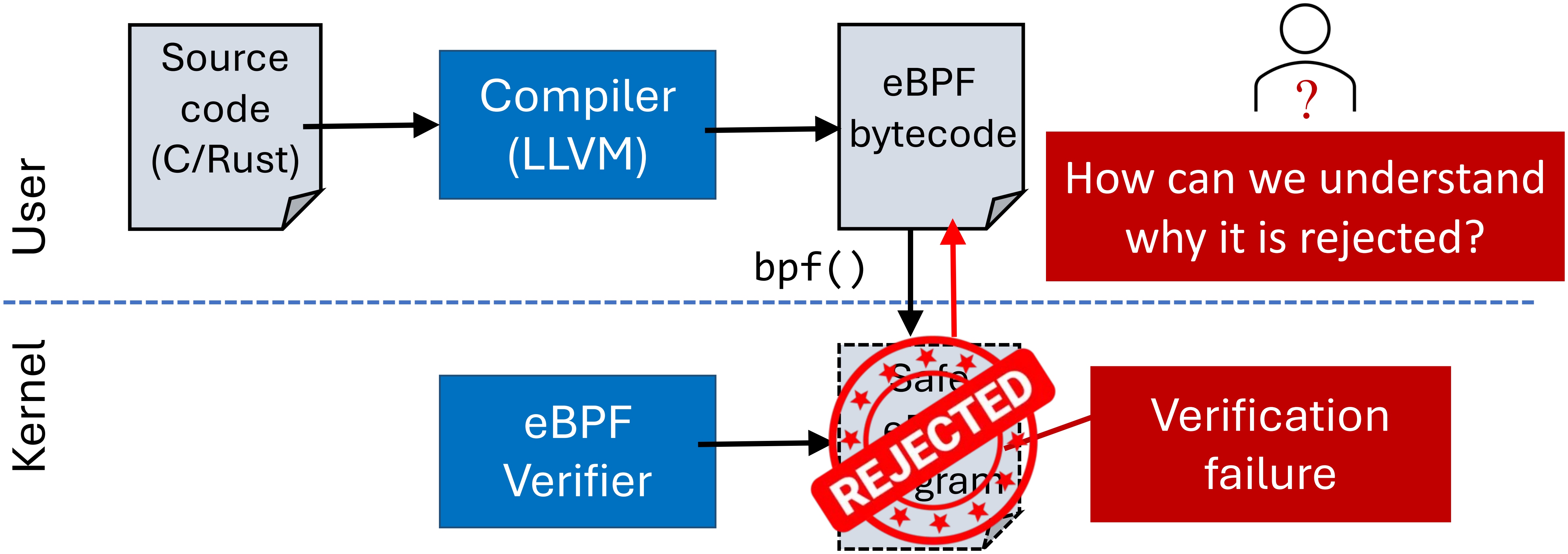
- ✅ Memory safety
- ✅ Type safety
- ✅ Resource safety
- ✅ Runtime safety (e.g., termination)
- ✅ No undefined behavior
- ✅ …

Basic principle of eBPF: Safety verification



Symbolic execution on
all possible code paths

Basic principle of eBPF: Safety verification



Symbolic execution on all possible code paths

Safety at the cost of usability

cilium

```
/* cilium/bpf/bpf_sock.c */
int __sock4_post_bind(struct bpf_sock *ctx)
{
  struct lb4_service *svc;
  struct lb4_key key = ...;
  ...
  svc = __lb4_lookup_service(&key);
  if (!svc) {
    key.dport = bpf_htons(ctx->src_port);
    svc = sock4_nodeport_wildcard_lookup(&key, ...);
  }
  ...
}
```

Safety at the cost of usability

cilium

```
/* cilium/bpf/bpf_sock.c */
int __sock4_post_bind(struct bpf_sock *ctx)
{
  struct lb4_service *svc;
  struct lb4_key key = ...;
  ...
  svc = __lb4_lookup_service(&key);
  if (!svc) {
    key.dport = bpf_htons(ctx->src_port);
    svc = sock4_nodeport_wildcard_lookup(&key, ...);
  }
  ...
}
```

Look up the service from a map using key

Safety at the cost of usability

cilium

```
/* cilium/bpf/bpf_sock.c */
int __sock4_post_bind(struct bpf_sock *ctx)
{
  struct lb4_service *svc;
  struct lb4_key key = ...;
  ...
  svc = __lb4_lookup_service(&key);
  if (!svc) {
    key.dport = bpf_htons(ctx->src_port);
    svc = sock4_nodeport_wildcard_lookup(&key,
  }
  ...
}
```

Look up the service from a map using key

Redo a wildcard lookup if not found in the last round

Safety at the cost of usability


```
/* cilium/bpf/bpf_sock.c */
int __sock4_post_bind(struct bpf_sock *ctx)
{
  struct lb4_service *svc;
  struct lb4_key key = ...;
```

This simple code does not pass the eBPF verifier!

```
    key.dport = bpf_htons(ctx->src_port);
    svc = sock4_nodeport_wildcard_lookup(&key, ...);
  }
  ...
}
```

Safety at the cost of usability

```
32: (85) call bpf_map_lookup_elem#1
33: (15) if r0 == 0x0 goto pc+2"
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0)
; R6=ctx(id=0,off=0,imm=0)
34: (69) r1 = *(u16 *)(r0 +4)"
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0)
; R6=ctx(id=0,off=0,imm=0)
35: (55) if r1 != 0x0 goto pc+51
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0) R1=inv0
; R6=ctx(id=0,off=0,imm=0)
36: (69) r2 = *(u16 *)(r6 +44)
invalid bpf_context access off=44 size=2
```

Verifier log

Safety at the cost of usability

cilium

```
32: (85) call bpf_map_lookup_elem#1
33: (15) if r0 == 0x0 goto pc+2"
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0)
; R6=ctx(id=0,off=0,imm=0)
34: (69) r1 = *(u16 *)(r0 +4)"
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0)
; R6=ctx(id=0,off=0,imm=0)
35: (55) if r1 != 0x0 goto pc+51
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0) R1=inv0
; R6=ctx(id=0,off=0,imm=0)
36: (69) r2 = *(u16 *)(r6 +44)
invalid bpf_context access off=44 size=2
```

Which source-code line do these instructions map to?

Verifier log

Safety at the cost of usability

cilium

```
32: (85) call bpf_map_lookup_elem#1
33: (15) if r0 == 0x0 goto pc+2"
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0)
; R6=ctx(id=0,off=0,imm=0)
34: (69) r1 = *(u16 *)(r0 +4)"
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0)
; R6=ctx(id=0,off=0,imm=0)
35: (55) if r1 != 0x0 goto pc+51
; R0=map_value(id=0,off=0,ks=12,vs=12,imm=0) R1=inv0
; R6=ctx(id=0,off=0,imm=0)
36: (69) r2 = *(u16 *)(r6 +44)
invalid bpf_context access off=44 size=2
```

Which source-code line do these instructions map to?

Why is it an invalid access?

Verifier log

Root cause: verifier does not understand compiler

- The program uses `bpf_htons()` to convert the endianness of the `src_port` field in the context.
  - `src_port` is defined as a 32-bit int, while `bpf_htons()` only performs operations on the upper 16 bits

- The compiler optimizes the code to only load the upper 16 bits

- The verifier checks context field accesses based on its size
  - Expect a 32-bit load on `src_port`, but only sees a 16-bit load
  - Reject the extension program with size mismatch error

Workarounds

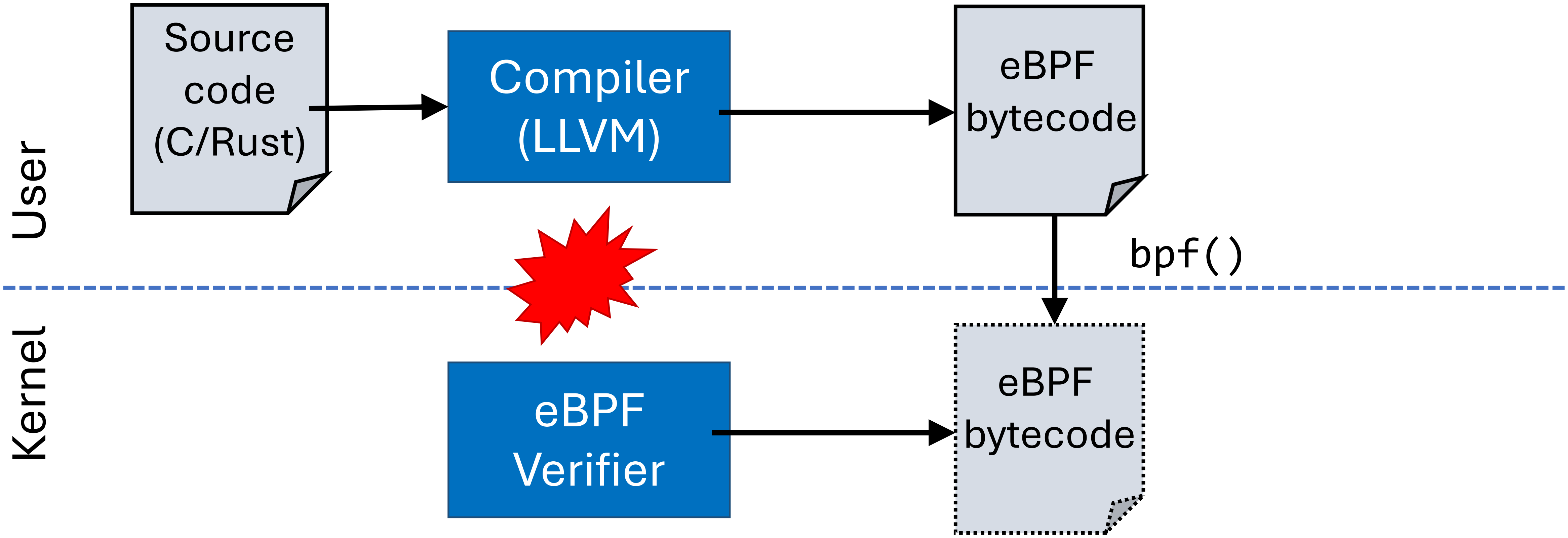
```
+static __always_inline __maybe_unused __be16
+ctx_src_port(struct bpf_sock *ctx)
+{
+. volatile __u32 sport = ctx->src_port;
+  return (__be16)bpf_htons(sport);
+}
+
...
   if (!svc) {
-      key.dport = bpf_htons(ctx->src_port);
+      key.dport = ctx_src_port(ctx);
       svc = sock4_nodeport_wildcard_lookup(&key, ...);
}
```

Workarounds

```
+static __always_inline __maybe_unused __be16
+ctx_src_port(struct bpf_sock *ctx)
+{
+.  volatile __u32 sport = ctx->src_port;
+   return (__be16)bpf_htons(sport);
+}
+
...
   if (!svc) {
-     key.dport = bpf_htons(ctx->src_port);
+     key.dport = ctx_src_port(ctx);
      svc = sock4_nodeport_wildcard_lookup(&key, ...);
}
```

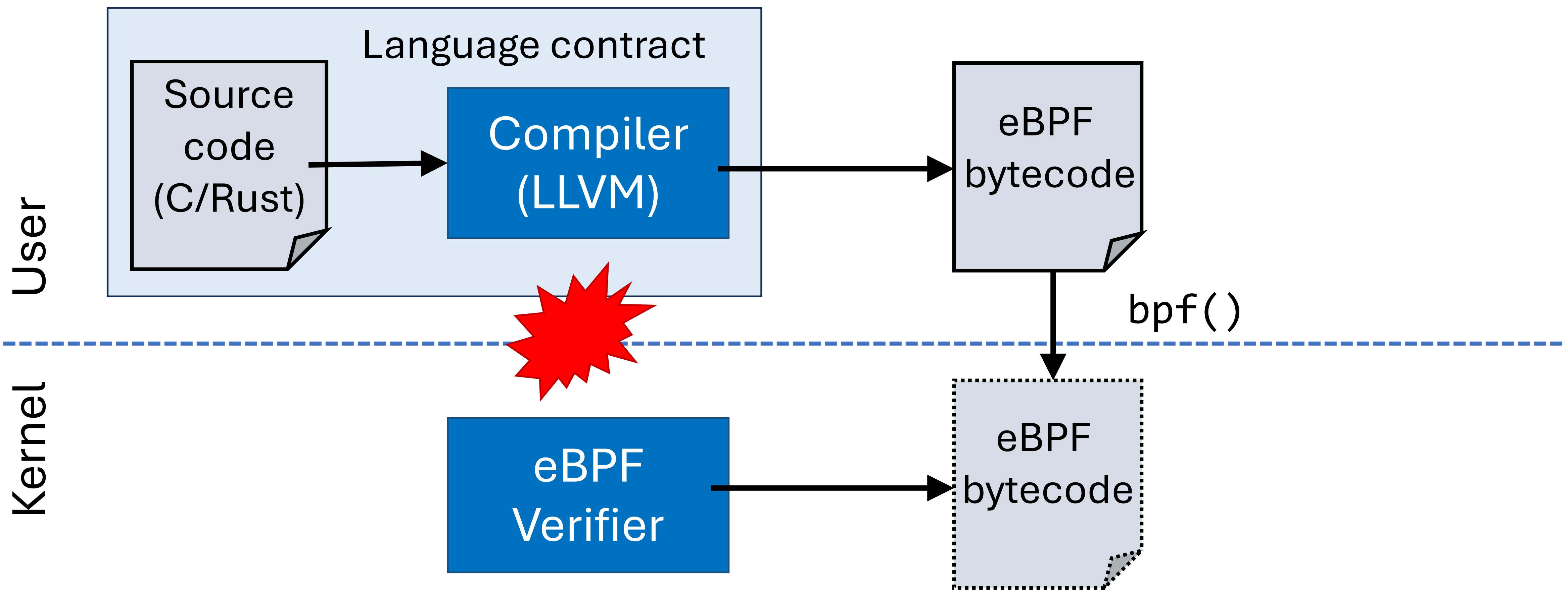
Used `volatile` to force a 32-bit load from the compiler

The language-verifier gap



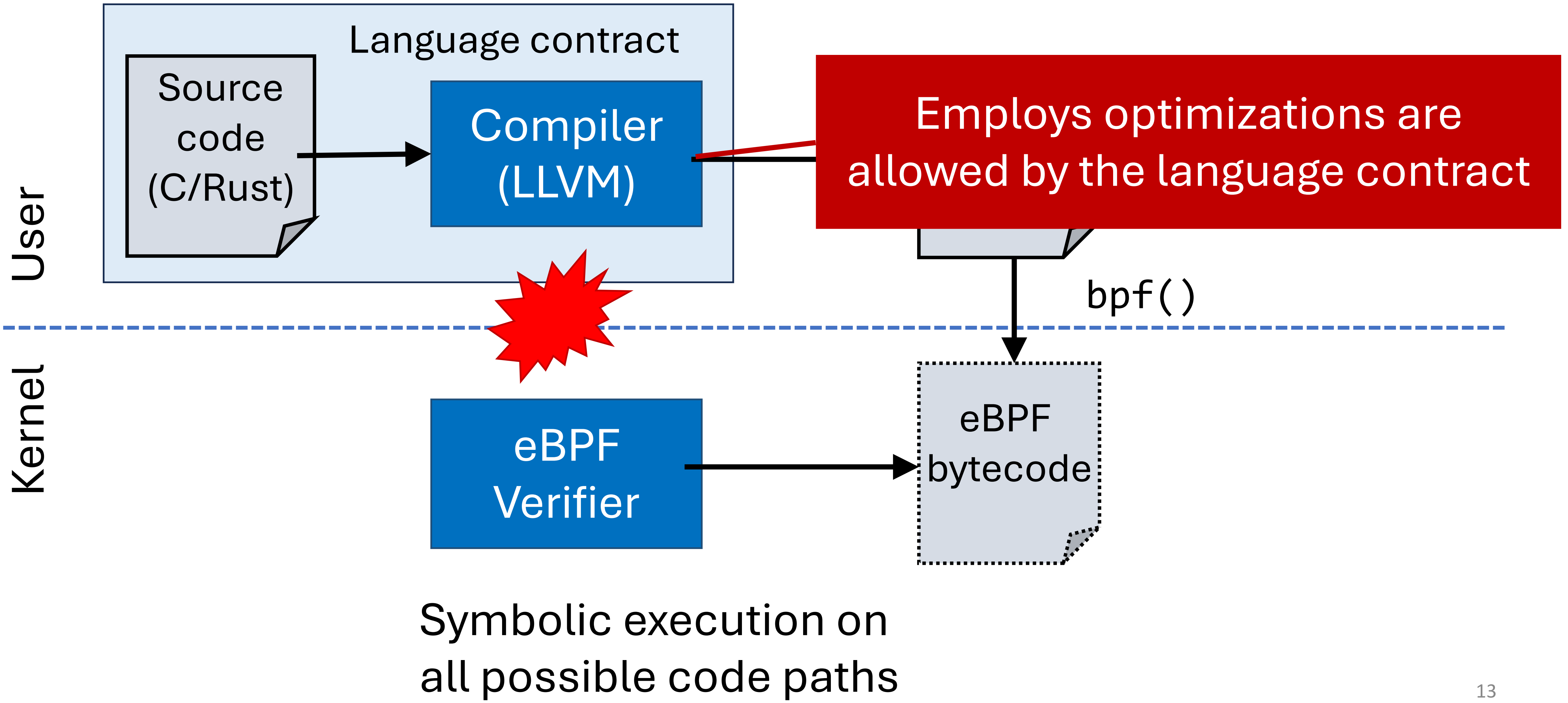
Symbolic execution on all possible code paths

The language-verifier gap

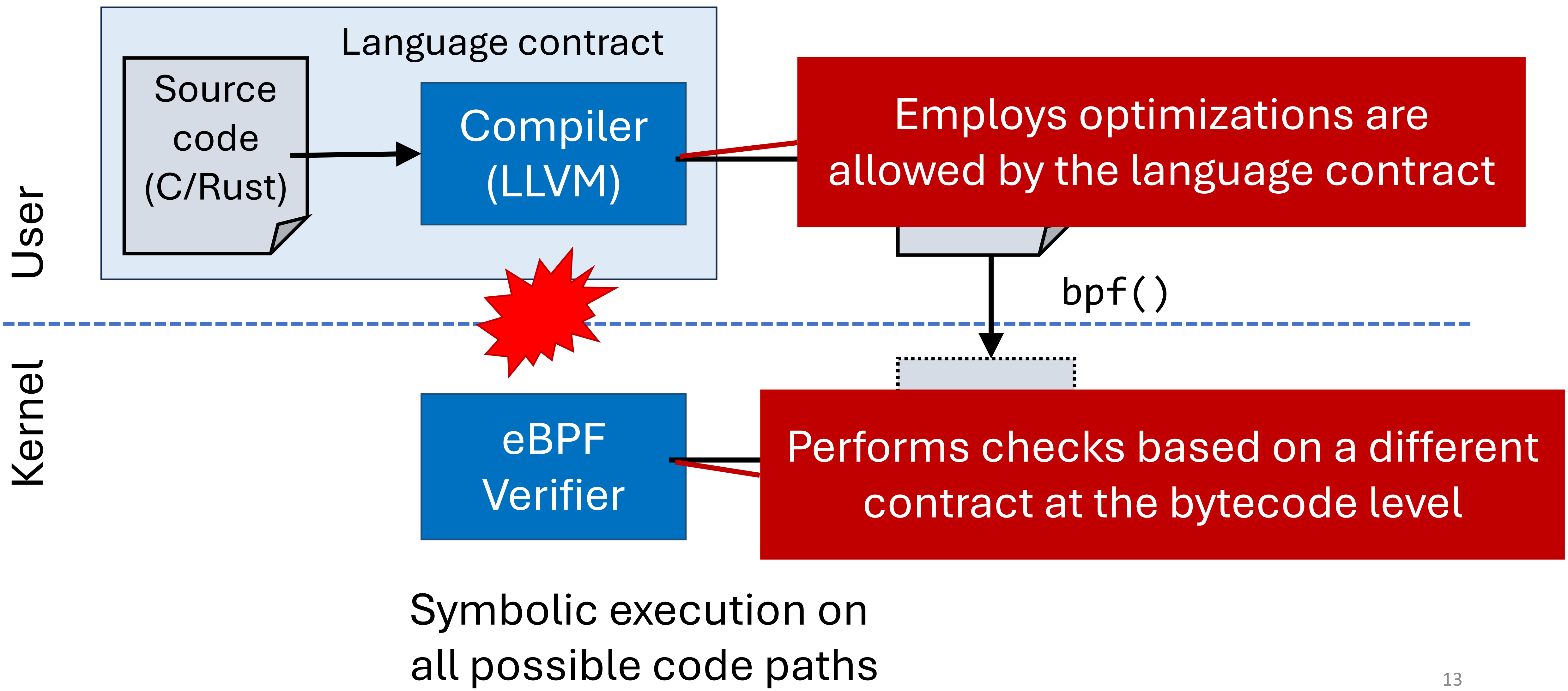


Symbolic execution on all possible code paths

The language-verifier gap



The language-verifier gap



The language-verifier gap causes many problems

Workaround	Count
Refactoring extension code into smaller ones	27
Hinting compilers to generate verifier-friendly code	22
Tweaking code to assist verification	15
Dealing with verifier bugs	9
Reinventing the wheels	1

Closing the language-verifier gap

- Running kernel extensions *safely* without a verifier
  - Key challenge: how to ensure safety?

Closing the language-verifier gap

- Running kernel extensions *safely* without a verifier
  - Key challenge: how to ensure safety?
- Insight: Language-based safety + runtime mechanism
  - Rust as the safe language (safe Rust only)
  - Runtime safety checks for other safety properties
    - e.g., termination and stack safety

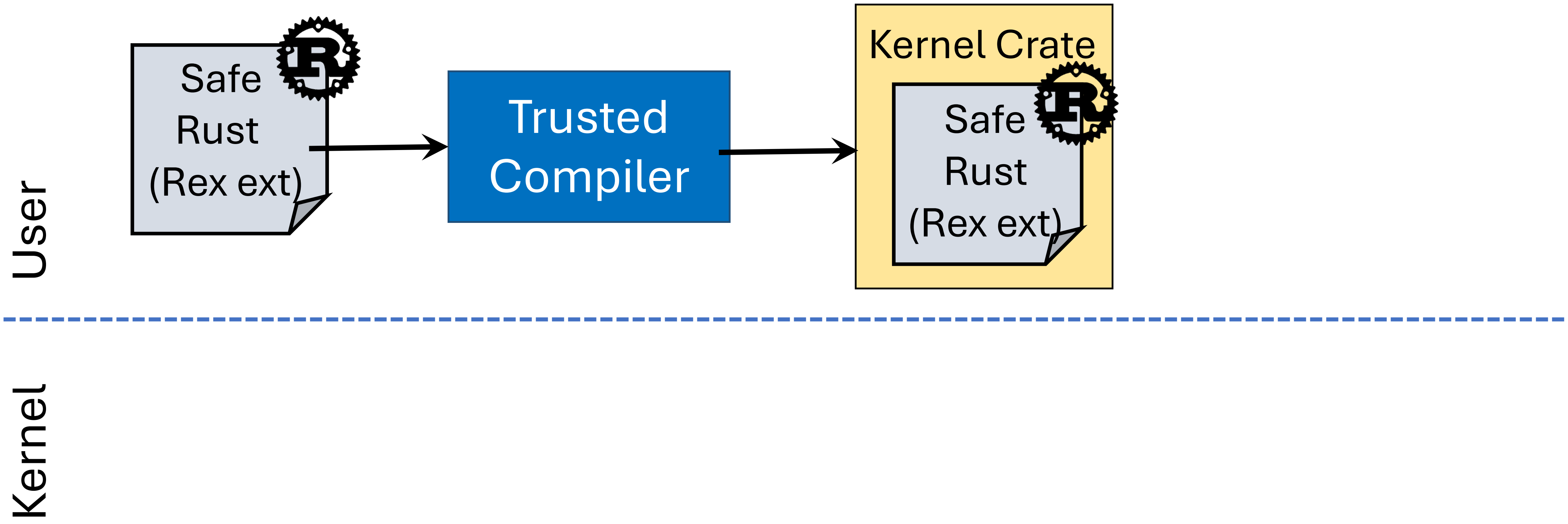
Rex: Safe, usable Rust kernel extensions

User

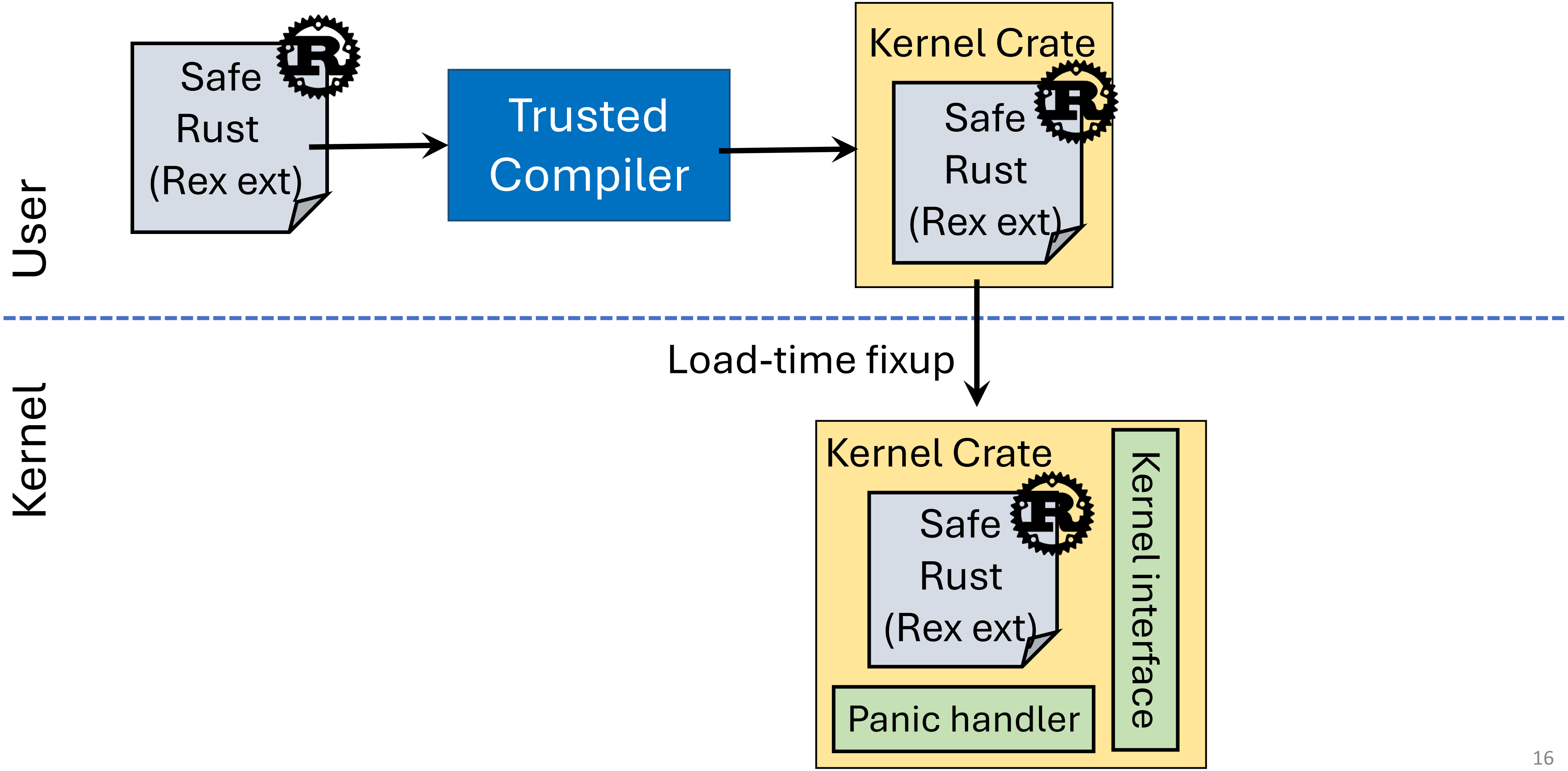
Kernel

Rex: Safe, usable Rust kernel extensions

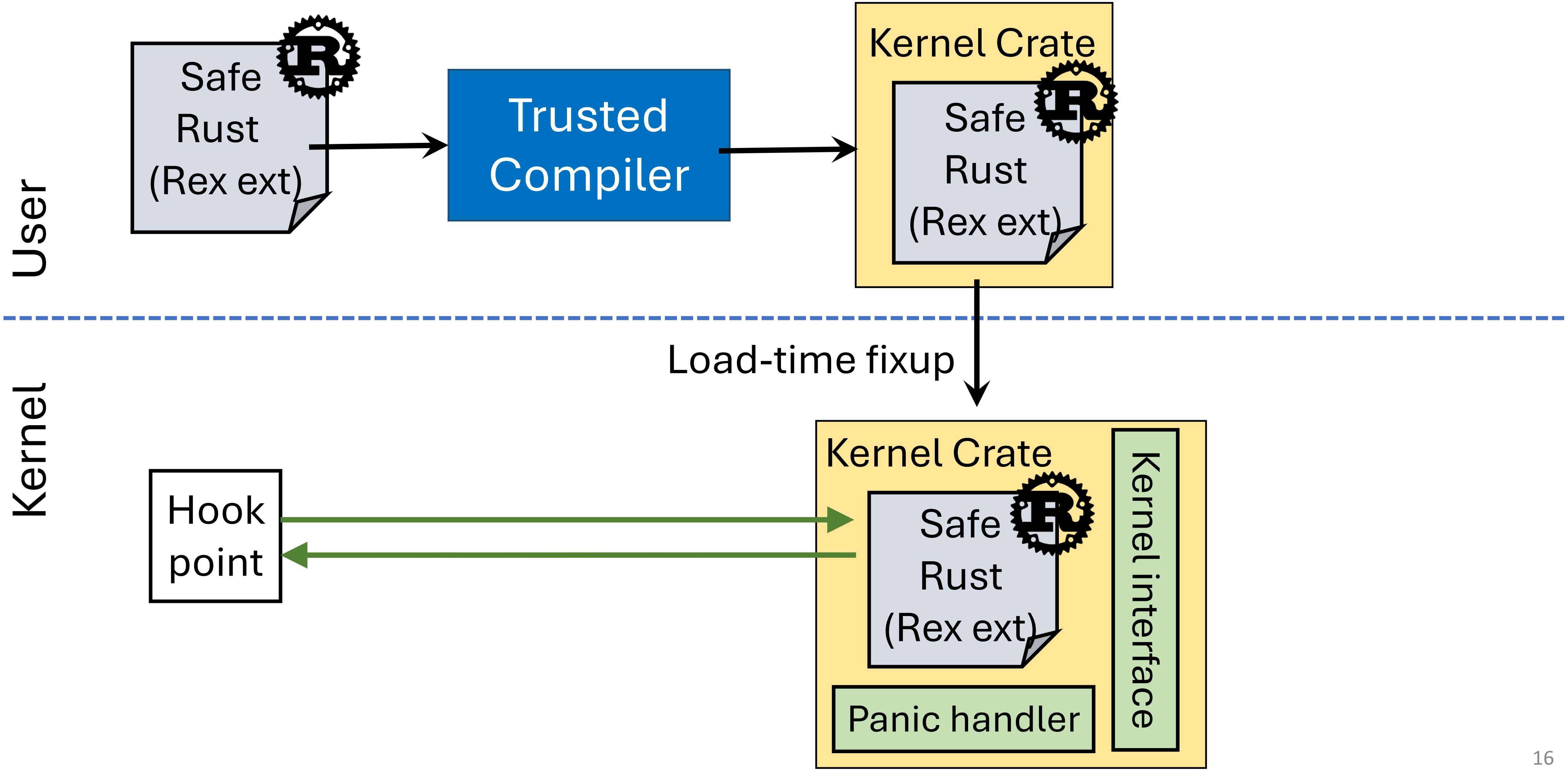
Rex: Safe, usable Rust kernel extensions



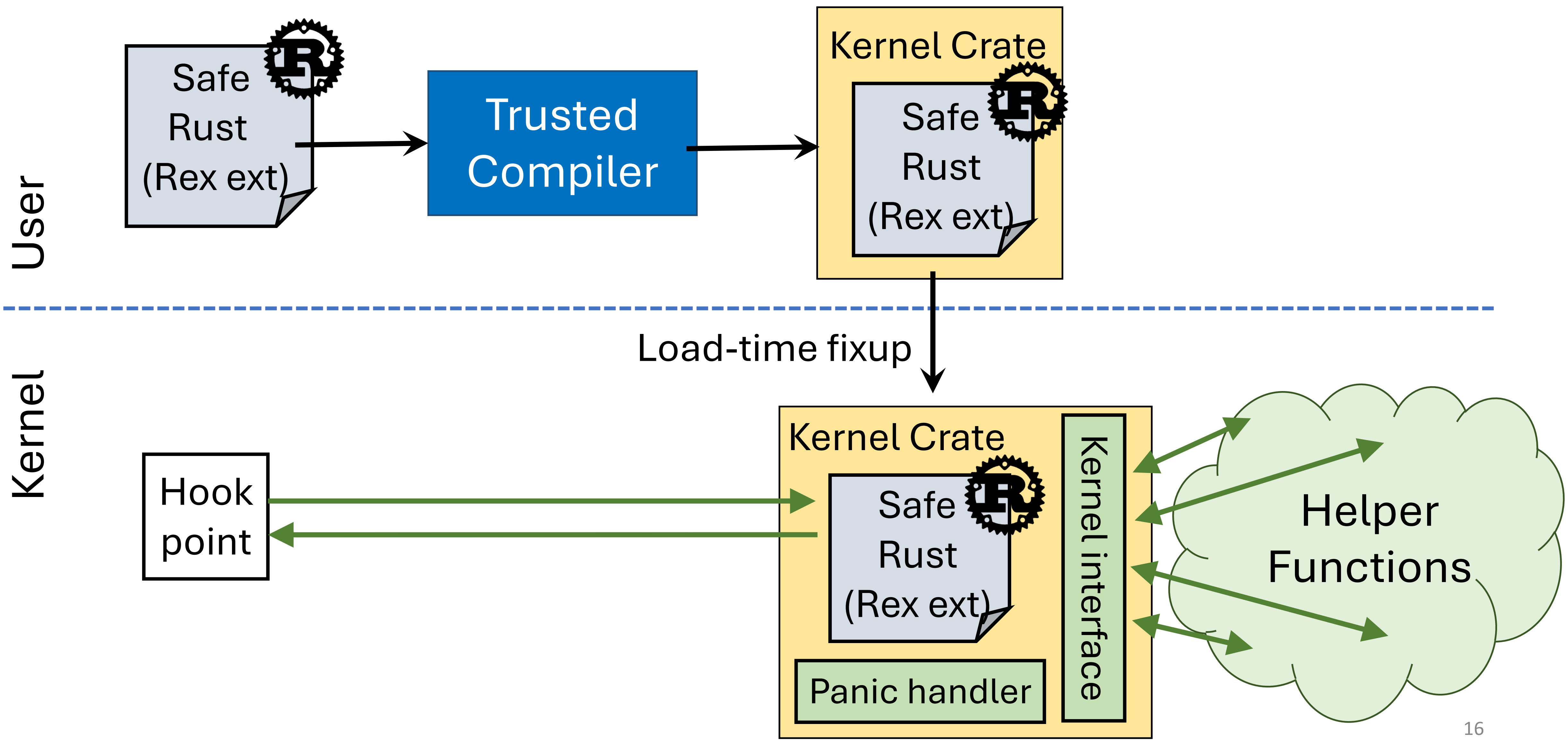
Rex: Safe, usable Rust kernel extensions



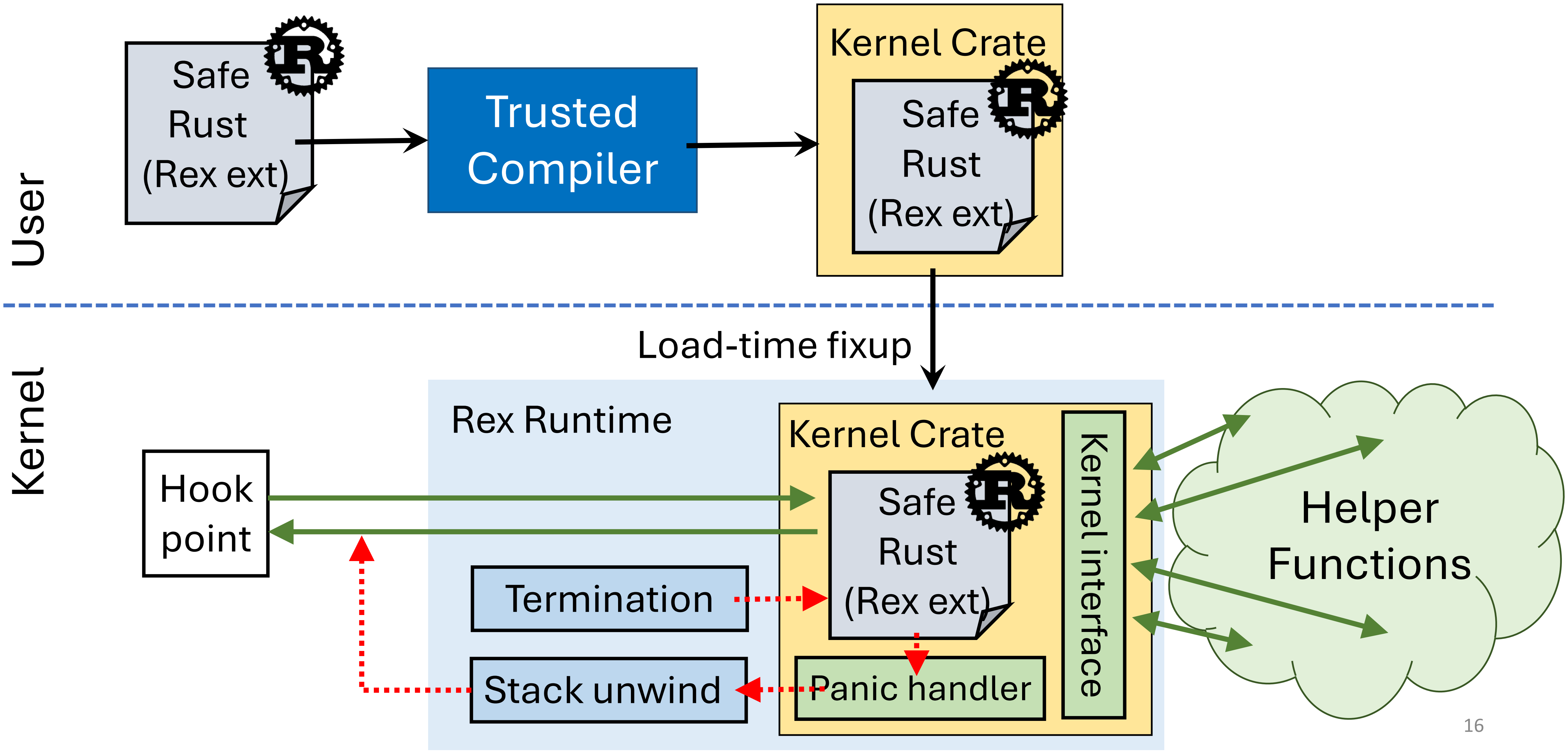
Rex: Safe, usable Rust kernel extensions



Rex: Safe, usable Rust kernel extensions



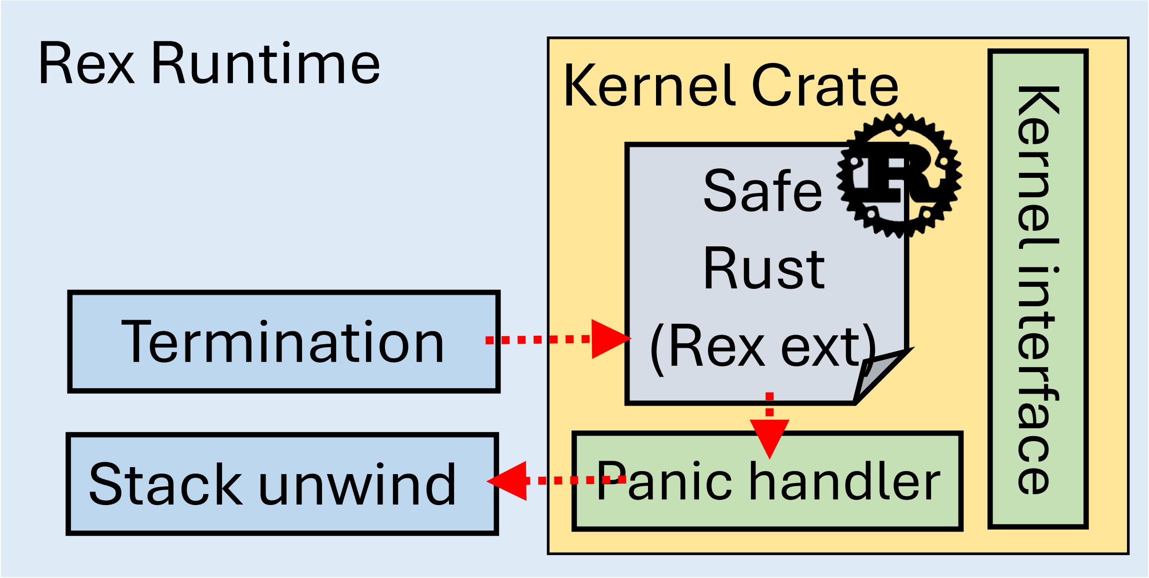
Rex: Safe, usable Rust kernel extensions



Roadmap

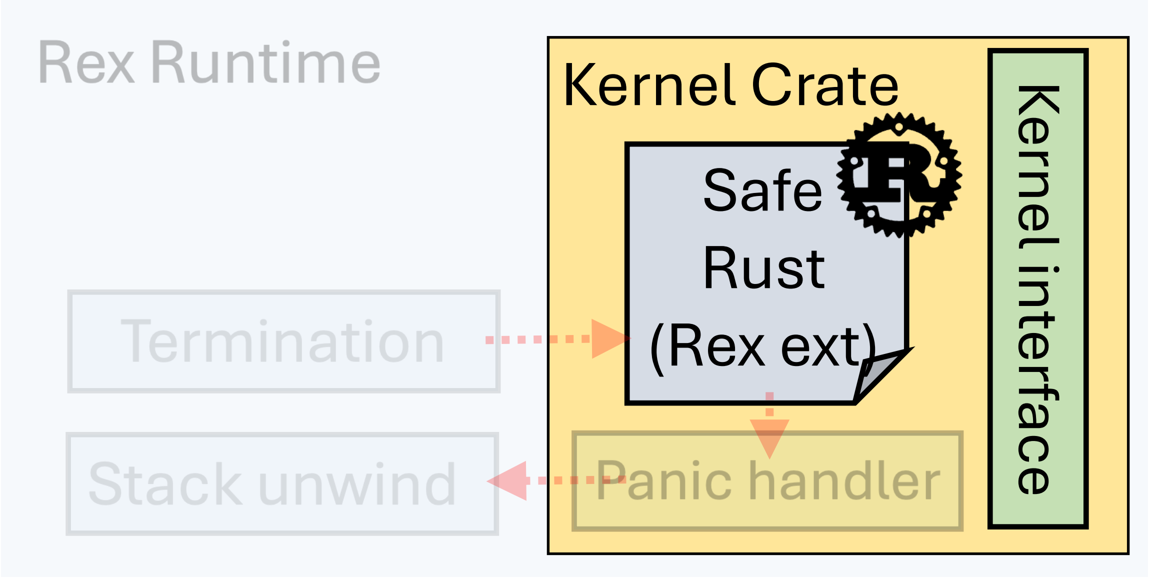
- Rex kernel crate: Compile-time safety
  - Memory safety
  - Extended type safety
  - Safe resource management

- Rex Runtime safety
  - Safe exception handling
  - Kernel stack safety
  - Program Termination



Roadmap

- Rex kernel crate: Compile-time safety
  - Memory safety
  - Extended type safety
  - Safe resource management
- Rex Runtime safety
  - Safe exception handling
  - Kernel stack safety
  - Program Termination



Memory safety

- Rex enforces extensions to access kernel memory safely
  - Memory owned by the Rex extension
    - Safe Rust already ensures type safety (no unsafe memory access)
    - Safe kernel interface provided by the Rex kernel crate
  - Memory owned by the kernel
    - Kernel objects with static size are handled by Rust type system
    - Rust slices for dynamic pointers (runtime checks)
      - In principle similar to `dynptrs` in eBPF

Extended type safety

- There is the desire of transforming a byte stream to typed data
  - Common in networking use cases
  - **Inevitably requires unsafe code** under the Rust type system

Extended type safety

- There is the desire of transforming a byte stream to typed data
  - Common in networking use cases
  - **Inevitably requires unsafe code** under the Rust type system

```
void *payload = ctx->data;
/* extract ip header */
struct iphdr *ip = payload;
__u8 protocol = ip->protocol;
```

Extended type safety

- There is the desire of transforming a byte stream to typed data
  - Common in networking use cases
  - **Inevitably requires unsafe code** under the Rust type system

```
let payload: &[u8] = ctx.data;
// extract ip header
let ip = unsafe { &*(payload.as_ptr() as *const iphdr) };
let protocol = ip.protocol;
```

Extended type safety

- There is the desire of transforming a byte stream to typed data
  - Common in networking use cases
  - **Inevitably requires unsafe code** under the Rust type system

- eBPF considers the cast safe under two rules
  - #1 Not making a pointer by casting a scalar value
  - #2 New value fitting in the memory boundary
- Rex follows the same rules and extends the Rust type safety

Extended type safety

- Idea: the compiler has full knowledge of the types

Extended type safety

- Idea: the compiler has full knowledge of the types
- Create an **auto trait**: `rex::NoRef`
  - Default implemented on a type if all fields are `rex::NoRef`
  - Negative implementation on reference and pointer types

```
pub unsafe auto trait NoRef {}
impl<T: ?Sized> !NoRef for &T {}
impl<T: ?Sized> !NoRef for &mut T {}
impl<T: ?Sized> !NoRef for *const T {}
impl<T: ?Sized> !NoRef for *mut T {}
```

Extended type safety

- Idea: the compiler has full knowledge of the types
- Create an **auto trait**: `rex::NoRef`
  - Default implemented on a type if all fields are `rex::NoRef`
  - Negative implementation on reference and pointer types
  - Compiler auto-implements only on types w/o **any** refs/pointers

Extended type safety

- Idea: the compiler has full knowledge of the types
- Create an **auto trait**: `rex::NoRef`
  - Default implemented on a type if all fields are `rex::NoRef`
  - Negative implementation on reference and pointer types
  - Compiler auto-implements only on types w/o **any** refs/pointers

```
pub struct my_hdr {
  request_id: u16,
  seq_num: u16,
  num_dgram: u16,
  other: NonNull<u64>,
}
```

Extended type safety

- Idea: the compiler has full knowledge of the types
- Create an **auto trait**: `rex::NoRef`
  - Default implemented on a type if all fields are `rex::NoRef`
  - Negative implementation on reference and pointer types
  - Compiler auto-implements only on types w/o **any** refs/pointers

```rust
pub struct my_hdr {
  request_id: u16,
  seq_num: u16,
  num_dgram: u16,
  other: NonNull<u64>,
}
```

NonNull contains a `*const u64`

Extended type safety

- Idea: the compiler has full knowledge of the types
- Create an **auto trait**: `rex::NoRef`
  - Default implemented on a type if all fields are `rex::NoRef`
  - Negative implementation on reference and pointer types
  - Compiler auto-implements only on types w/o **any** refs/pointers

```
pub struct my_hdr {
   request_id: u16,
   seq_num: u16,
   num_dgram: u16,
   other: NonNull<u64>,
}
```

`NonNull` contains a `*const u64`

`Rex::NoRef` is not implemented for `NonNull` and `my_hdr`

Extended type safety

- Idea: the compiler has full knowledge of the types
- Create an **auto trait**: `rex::NoRef`
  - Default implemented on a type if all fields are `rex::NoRef`
  - Negative implementation on reference and pointer types
  - Compiler auto-implements only on types w/o **any** refs/pointers
- Require type to be `rex::NoRef` to convert from bytes (#1)
- Check memory bounds at runtime (#2)

```
fn from_bytes<T: NoRef>(bytes: &[u8]) -> &T {
  assert!(data.len() >= mem::size_of::<T>());
  ...
}
```

Safe resource management

- Rex leverages Resource-Acquisition-Is-Initialization (RAII)
  - Ties lifetime of acquirable resources with Rust wrapper types
  - **Acquire** resource in object constructor
    - e.g., calling `spin_lock()` constructs a `SpinlockGuard` object
  - **Release** resource in the destructor (via `Drop` trait in Rust)
    - e.g. `SpinlockGuard` implements the `Drop` trait and release the lock

- Compiler inserts a `drop` call when objects go out of scope
  - No need to **explicitly** manage resources

Safe resource management

- Certain APIs from the core library interferes with RAII

Safe resource management

- Certain APIs from the core library interferes with RAII
  - `core::mem::forget`
  - `core::mem::ManuallyDrop`
  - `core::intrinsics::abort`

Safe resource management

- Certain APIs from the core library interferes with RAII
  - `core::mem::forget`
  - `core::mem::ManuallyDrop`
  - `core::intrinsics::abort`

Prevents execution of destructors

Safe resource management

- Certain APIs from the core library interferes with RAII
  - `core::mem::forget`
  - `core::mem::ManuallyDrop`
  - `core::intrinsics::abort`

Prevents execution of destructors

↳ Additionally **crashes** the kernel with an illegal instruction

Safe resource management

- Certain APIs from the core library interferes with RAII
  - `core::mem::forget`
  - `core::mem::ManuallyDrop`
  - `core::intrinsics::abort`

  Prevents execution of destructors

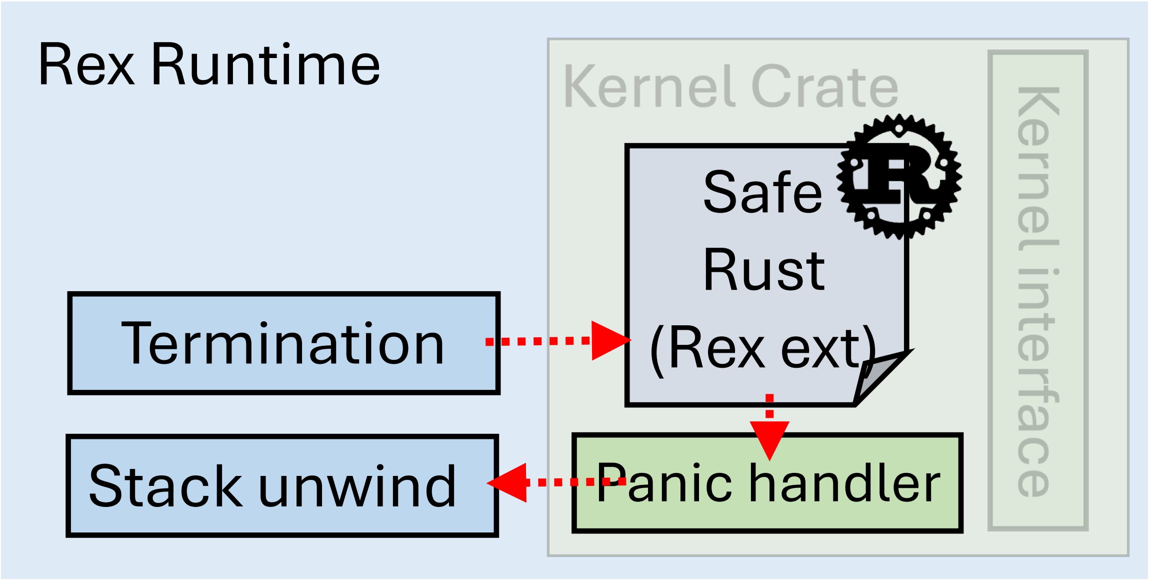
  ↳ Additionally **crashes** the kernel with an illegal instruction

- Rex forbids the usage of such language items
  - Reject via the disallowed methods/types linter configuration

Roadmap

- Rex kernel crate: Compile-time safety
  - Memory safety
  - Extended type safety
  - Safe resource management

- Rex Runtime safety
  - Safe exception handling
  - Kernel stack safety
  - Program Termination



Safe exception handling

- Certain safety properties of Rust are checked at runtime
  - Violations (e.g., out-of-bound access) trigger Rust panics
- Rust performs Itanium-ABI exception handling in user space
  - Unwind each stack frame and executes cleanup code
  - Not suitable in context of safe kernel extensions
    - Stack unwinding **cannot** fail (causing kernel crash or resource leaks)
    - Executing user-defined destructors is **not safe**
- Rex supports safe exception handling with two components
  - Graceful exit
  - Resource cleanup

Safe exception handling (graceful exit)

```
// In-kernel dispatcher function
rex_dispatcher_func:
// save old stack pointer
mov %rsp, PER_CPU_VAR(rex_old_sp)
...

// invoke the REX program
call *%rdx

rex_exit:
// reset to old stack pointer
mov PER_CPU_VAR(rex_old_sp), %rsp
...
ret
```

Safe exception handling (graceful exit)

```
// In-kernel dispatcher function
rex_dispatcher_func:
// save old stack pointer
mov %rsp, PER_CPU_VAR(rex_old_sp)
...

// invoke the REX program
call *%rdx

rex_exit:
// reset to old stack pointer
mov PER_CPU_VAR(rex_old_sp), %rsp
...
ret
```

Save the old stack pointer before program invocation

Safe exception handling (graceful exit)

```
// In-kernel dispatcher function
rex_dispatcher_func:
// save old stack pointer
mov %rsp, PER_CPU_VAR(rex_old_sp)
...

// invoke the REX program
call *%rdx

rex_exit:
// reset to old stack pointer
mov PER_CPU_VAR(rex_old_sp), %rsp
...
ret
```

```
// Rex program
rex_prog1:
...
```

Safe exception handling (graceful exit)

```
// In-kernel dispatcher function
rex_dispatcher_func:
// save old stack pointer
mov %rsp, PER_CPU_VAR(rex_old_sp)
...

// invoke the REX program
call *%rdx

rex_exit:
// reset to old stack pointer
mov PER_CPU_VAR(rex_old_sp), %rsp
...
ret
```

```
// Rex program
rex_prog1:
...
```

Panic

Safe exception handling (graceful exit)

```
// In-kernel dispatcher function
rex_dispatcher_func:
// save old stack pointer
mov %rsp, PER_CPU_VAR(rex_old_sp)
...

// invoke the REX program
call *%rdx

rex_exit:
// reset to old stack pointer
mov PER_CPU_VAR(rex_old_sp), %rsp
...
ret
```

```
// Rex program
rex_prog1:
...
```

Panic

```
// Rex panic handler
rust_begin_unwind:
// Cleanup resources
...
call rex_landingpad
```

Safe exception handling (graceful exit)

```
// In-kernel dispatcher function
rex_dispatcher_func:
// save old stack pointer
mov %rsp, PER_CPU_VAR(rex_old_sp)
...

// invoke the REX program
call *%rdx

rex_exit:
// reset to old stack pointer
mov PER_CPU_VAR(rex_old_sp), %rsp
...
ret
```

```
// Rex program
rex_prog1:
...
```

Panic

```
// Rex panic handler
rust_begin_unwind:
// Cleanup resources
...
call rex_landingpad
```

```
// In-kernel landingpad
rex_landingpad:
// Report error
...
jmp rex_exit
```

Safe exception handling (graceful exit)

```
// In-kernel dispatcher function
rex_dispatcher_func:
// save old stack pointer
mov %rsp, PER_CPU_VAR(rex_old_sp)
...

// invoke the REX program
call *%rdx

rex_exit:
// reset to old stack pointer
mov PER_CPU_VAR(rex_old_sp), %rsp
...
ret
```

```
// Rex program
rex_prog1:
...
```

Panic

```
// Rex panic handler
rust_begin_unwind:
// Cleanup resources
...
call rex_landingpad
```

```
// In-kernel landingpad
rex_landingpad:
// Report error
...
jmp rex_exit
```

Safe exception handling (graceful exit)

```
// In-kernel dispatcher function
rex_dispatcher_func:
// save old stack pointer
mov %rsp, PER_CPU_VAR(rex_old_sp)
...

// invoke the REX program
call *%rdx

rex_exit:
// reset to old stack pointer
mov PER_CPU_VAR(rex_old_sp), %rsp
...
ret
```

```
// Rex program
rex_prog1:
...
```

Panic

```
// Rex panic handler
rust_begin_unwind:
// Cleanup resources
...
call rex_landingpad
```

```
// In-kernel landingpad
...
jmp rex_exit
```

Restore the old stack pointer to unwind stack

Safe exception handling (graceful exit)

- Integrate with kernel stack trace for better user experience
  - Dump stack trace of Rex panics to the kernel console
  - De-mangle Rex symbols and register them into `kallsyms` on load
  - Enable frame-pointers for the kernel stack unwinder

Safe exception handling (graceful exit)

- Integrate with kernel stack trace for better user experience
  - Dump stack trace of Rex panics to the kernel console
  - De-mangle Rex symbols and register them into `kallsyms` on load
  - Enable frame-pointers for the kernel stack unwinder

```
Panic from Rex prog: called `Option::unwrap()` on a `None` value
Call Trace:
  rex_landingpad+0x64/0xb0
  rex_prog_4168211f00000000::rust_begin_unwind+0x15a/0x1a0
  rex_prog_4168211f00000000::core::panicking::panic_fmt+0x9/0x10
  rex_prog_4168211f00000000::core::panicking::panic+0x53/0x60
  rex_prog_4168211f00000000::core::option::unwrap_failed+0x9/0x10
  rex_prog_4168211f00000000::err_injector+0x8f/0xa0
  rex_dispatcher_func+0x32/0x32
```

Safe exception handling (resource cleanup)

- Safe exception handling **must clean up** acquired resources
  - Itanium EH uses DWARF to identify acquired resources

- Insight: extensions can only acquire resources via helpers
  - Only these resources need to be released
  - Record resources allocated via helper functions in a per-CPU buffer
  - Upon panic, iterate the resources and perform cleanup

Kernel stack safety

- Safe kernel extension should **never overflow the stack**
  - Unlike user space, kernel stack does not grow
  - Stack check from the eBPF verifier can be tricked via BPF tail calls [1]
- Rex divides stack protection into two cases
  - Extension programs that have no indirect/recursive calls
    - Static call graph is a DAG
    - Compute total stack usage statically based on the call graph
  - Extension programs that have indirect/recursive calls
    - Check stack usage dynamically before each function call
    - A failed check triggers a Rust panic and will be gracefully handled

[1] Overflowing the kernel stack with BPF, Linux Plumbers Conference, 2023

Safe termination

- A non-terminating extension could hold CPU for a long time
  - Prevents other tasks/interrupts from being executed
- Rex uses a dynamic timeout mechanism for termination
  - Set up an hrtimer per CPU that triggers periodically
  - Timer interrupt (hardirq) can preempt programs in softirq or task
  - If timeout, set program instruction pointer to the timeout handler
- **Defer** termination if the program is in a helper call
  - Preventing inconsistent states (e.g., reference count, lock)

Implementation

- Rex is supported on the latest stable Linux kernel and Rust
  - Linux-6.17 + Rust-1.91.0

- Rex Kernel crate provides the kernel interface
  - Kernel symbol bindings are implemented manually
    - Emitted as relocations in the compiled PIE binary
  - Kernel data-type bindings are automated by bindgen
    - Structs, constants, and type aliases
  - Program-type-specific code
    - Currently supports kprobe, perf-event, tracepoint, xdp, and tc

Implementation

- Kernel support
  - Program loading and symbol resolution
    - Map the program binary into the kernel address space
    - Patch GOT relocations of referenced kernel symbols
  - In-kernel Rex runtime
    - Stack unwinder and termination timer handler
- Compiler support
  - Rex pass in LLVM backend of rustc
    - Perform Rex-specific instrumentations (e.g., stack checks)
    - Generate `"extern C"` program entry points (actively moving to proc-macros)

Usability evaluation

- Heuristic evaluation
  - **No language-verifier gap anymore**
- Dogfooding
  - Used Rex to implement the BPF Memcached Cache (BMC)
  - **Much cleaner, simpler code**
    - 326 lines of Rust code vs. 513 lines of C code

Case study: BMC cache invalidation

Original eBPF-BMC

```
// Searches for SET command in payload
for (unsigned int off = 0;
     off < BMC_MAX_PACKET_LENGTH &&
     payload + off + 1 <= data_end;
     off++) {
    if (set_found == 0 &&
        payload[off] == 's' &&
        payload + off + 3 <= data_end &&
        payload[off + 1] == 'e' &&
        payload[off + 2] == 't') {
            off += 3;
            set_found = 1;
    }
    ...
}
```

Case study: BMC cache invalidation

Original eBPF-BMC

```
// Searches for SET command in payload
for (unsigned int off = 0;
     off < BMC_MAX_PACKET_LENGTH &&
     payload + off + 1 <= data_end;
     off++) {
    if (set_found == 0 &&
        payload[off] == 's' &&
        payload + off + 3 <= data_end &&
        payload[off + 1] == 'e' &&
        payload[off + 2] == 't') {
            off += 3;
            set_found = 1;
    }
    ...
}
```

Additional limit to fit in verifier's complexity limit

Case study: BMC cache invalidation

Original eBPF-BMC

```
// Searches for SET command in payload
for (unsigned int off = 0;
     off < BMC_MAX_PACKET_LENGTH &&
     payload + off + 1 <= data_end;
     off++) {
    if (set_found == 0 &&
        payload[off] == 's' &&
        payload + off + 3 <= data_end &&
        payload[off + 1] == 'e' &&
        payload[off + 2] == 't') {
            off += 3;
            set_found = 1;
    }
    ...
}
```

Additional limit to fit in verifier's complexity limit

Boilerplate to explicitly check end of payload

Case study: BMC cache invalidation

Original eBPF-BMC

```
// Searches for SET command in payload
for (unsigned int off = 0;
     off < BMC_MAX_PACKET_LENGTH &&
     payload + off + 1 <= data_end;
     off++) {
    if (set_found == 0 &&
        payload[off] == 's' &&
        payload + off + 3 <= data_end &&
        payload[off + 1] == 'e' &&
        payload[off + 2] == 't') {
            off += 3;
            set_found = 1;
    }
    ...
}
```

Additional limit to fit in verifier's complexity limit

Boilerplate to explicitly check end of payload

Verbose logic to match SET command in payload

Case study: BMC cache invalidation

Original eBPF-BMC

```
// Searches for SET command in payload
for (unsigned int off = 0;
     off < BMC_MAX_PACKET_LENGTH &&
     payload + off + 1 <= data_end;
     off++) {
    if (set_found == 0 &&
        payload[off] == 's' &&
        payload + off + 3 <= data_end &&
        payload[off + 1] == 'e' &&
        payload[off + 2] == 't') {
            off += 3;
            set_found = 1;
    }
    ...
}
```

Rex-BMC

```
// Searches for SET command in payload
let set_iter = payload
    .windows(4)
    .enumerate()
    .filter_map(|(i, v)|
        if v == b"set " {
            Some(i)
        } else {
            None
        }
);
...
```

Case study: BMC cache invalidation

Original eBPF-BMC

No extra code to handle complexity limit from verifier

```
// S                                      d
for
     payload + off + 1 <= data_end;
     off++) {
    if (set_found == 0 &&
        payload[off] == 's' &&
        payload + off + 3 <= data_end &&
        payload[off + 1] == 'e' &&
        payload[off + 2] == 't') {
            off += 3;
            set_found = 1;
    }
    ...
}
```

Rex-BMC

```
// Searches for SET command in payload
let set_iter = payload
    .windows(4)
    .enumerate()
    .filter_map(|(i, v)|
        if v == b"set " {
            Some(i)
        } else {
            None
        }
);
...
```

Case study: BMC cache invalidation

Original eBPF-BMC

```
// S                                                d
for
    payload + off + 1 <= data_end;
       && 
        payload[off + 1] == 'e' &&
        payload[off + 2] == 't') {
            off += 3;
            set_found = 1;
    }
    ...
}
```

No extra code to handle complexity limit from verifier

Rust slices already provides bound checks in its methods

Rex-BMC

```
// Searches for SET command in payload
let set_iter = payload
    .windows(4)
    .enumerate()
    .filter_map(|(i, v)|
        if v == b"set " {
            Some(i)
        } else {
            None
        }
);
...
```

Case study: BMC cache invalidation

Original eBPF-BMC

```
// S...                                              d
for
    payload + off + 1 <= data_end;
        payload + off + 3 < data_end &&
        payload[off + 1] == 'e' &&
    }
    ...
}
```

No extra code to handle complexity limit from verifier

Rust slices already provides bound checks in its methods

Easy match of SET command using Rust byte slice

Rex-BMC

```
// Searches for SET command in payload
let set_iter = payload
     .windows(4)
     .enumerate()
     .filter_map(|(i, v)|
          if v == b"set " {
             Some(i)
          } else {
             None
          }
);
...
```

Case study: BMC cache invalidation

Original eBPF-BMC

No extra code to handle complexity limit from verifier

Rust slices already provides bound checks in its methods

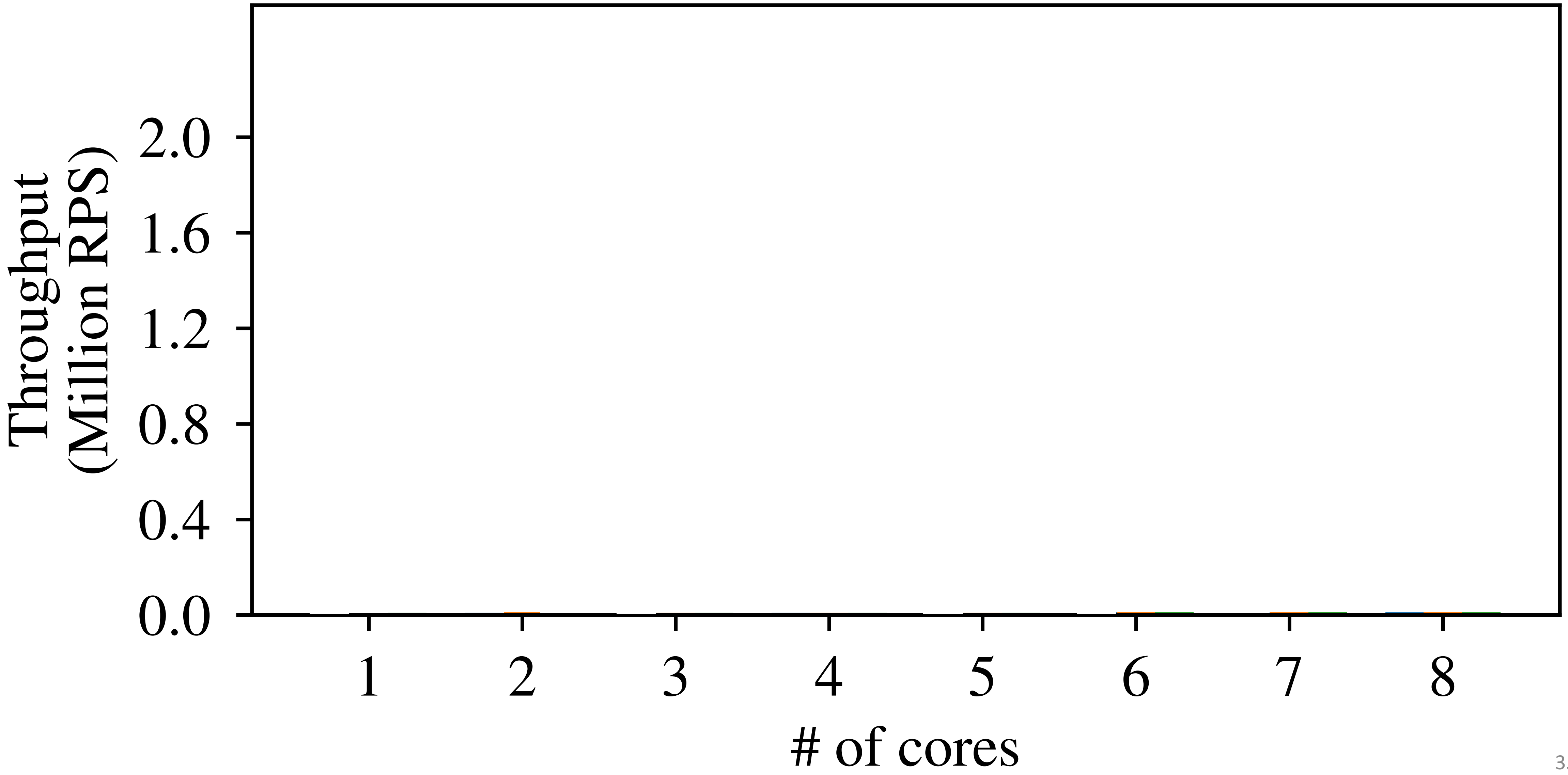
Easy match of SET command using Rust byte slice

Rex-BMC

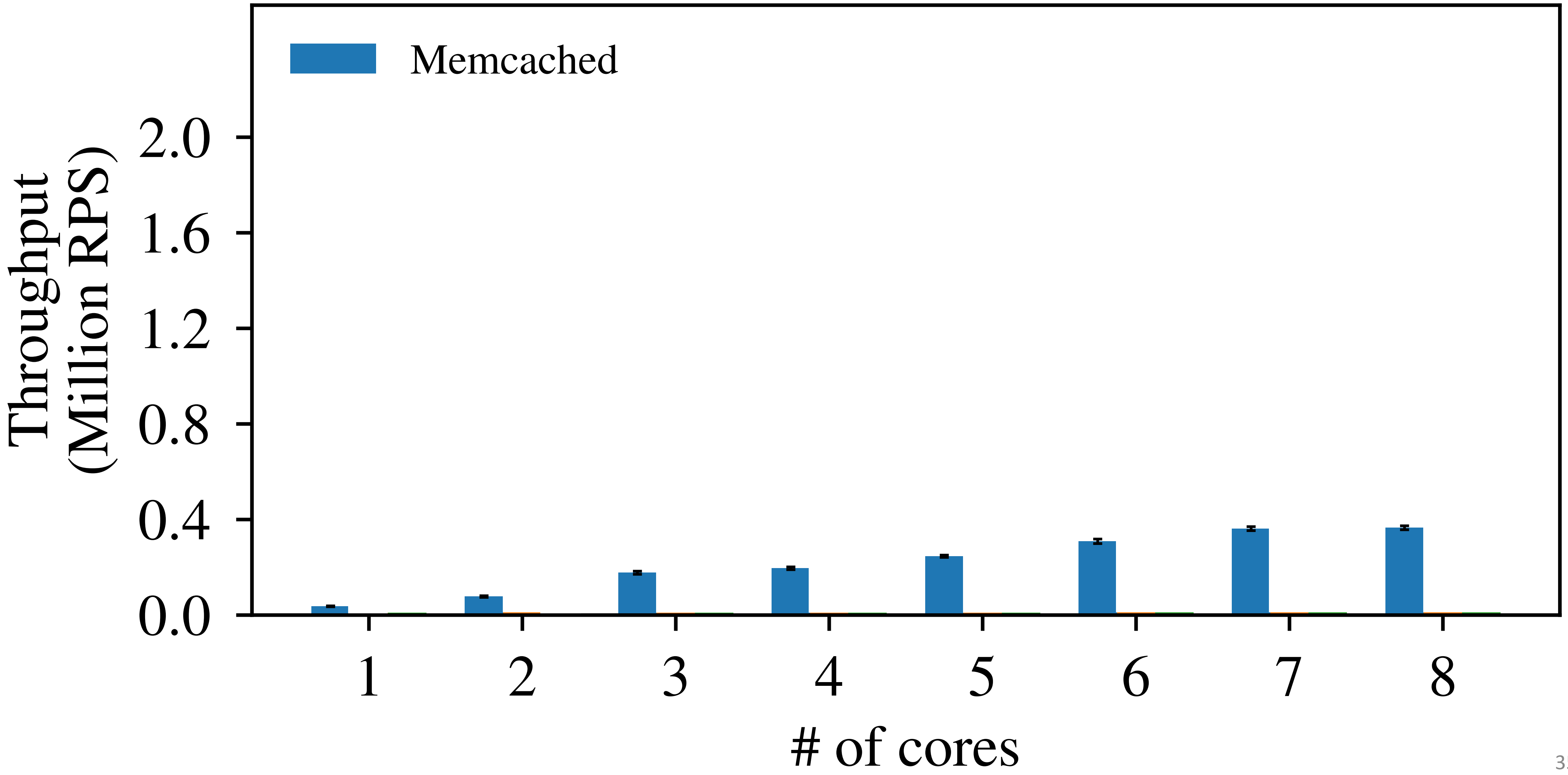
```
// Searches for SET command in payload
let set_iter = payload
    .windows(4)
    .enumerate()
    .filter_map(|(i, v)|
        if v == b"set " {
            Some(i)
        } else {
            None
        }
);
...
```

Rex-BMC is much *cleaner* and *simpler*

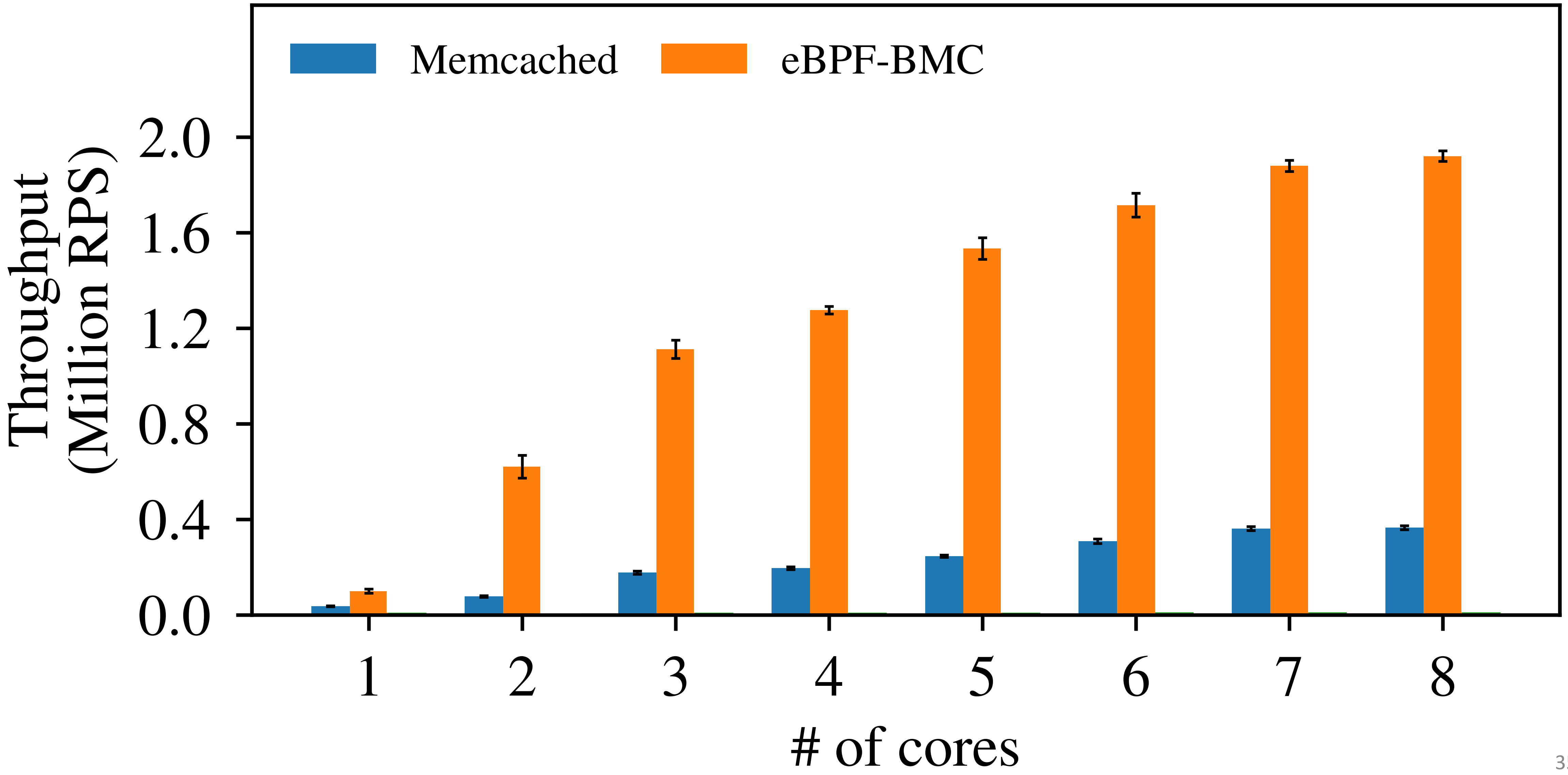
Performance evaluation



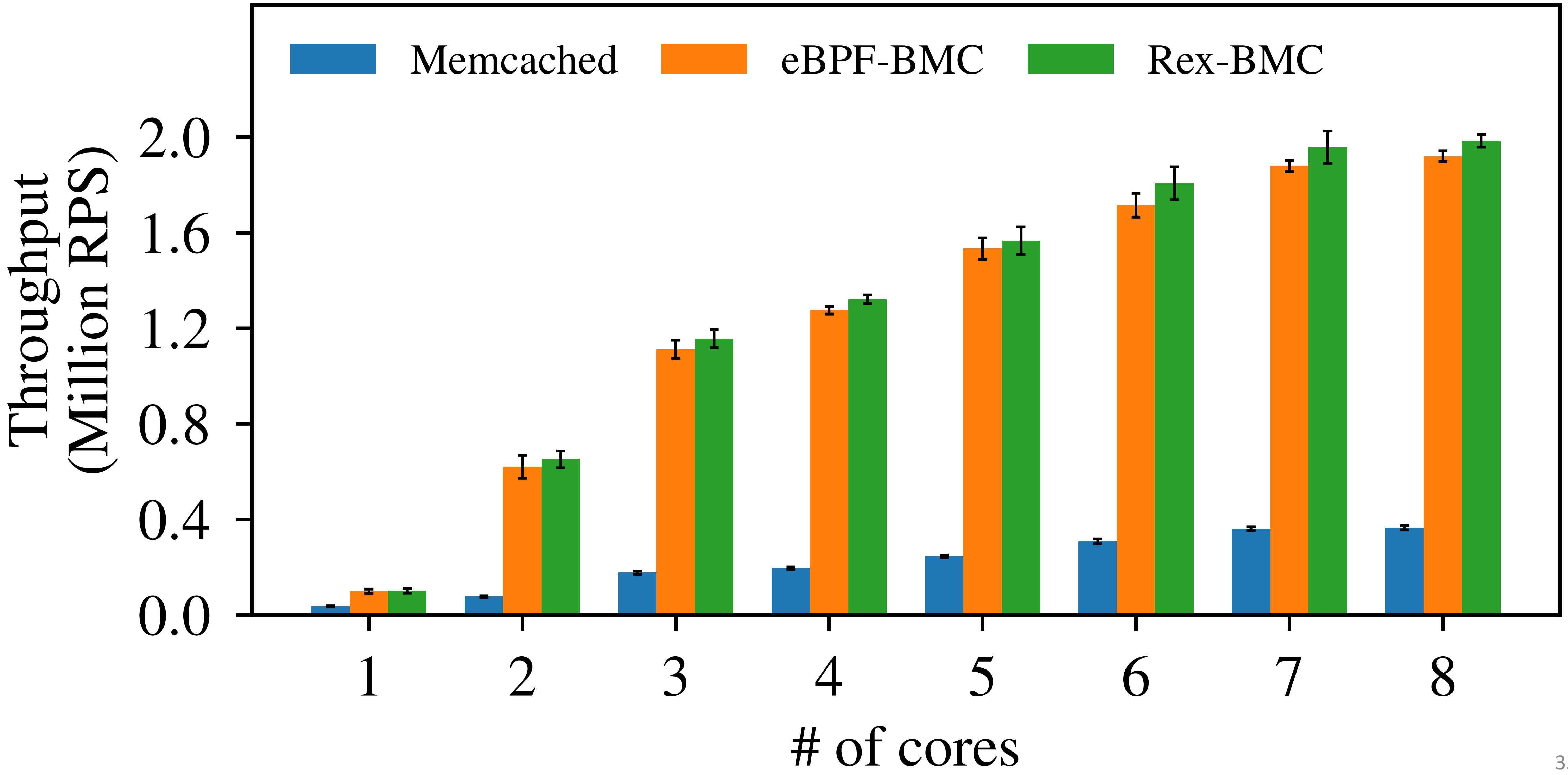
Performance evaluation



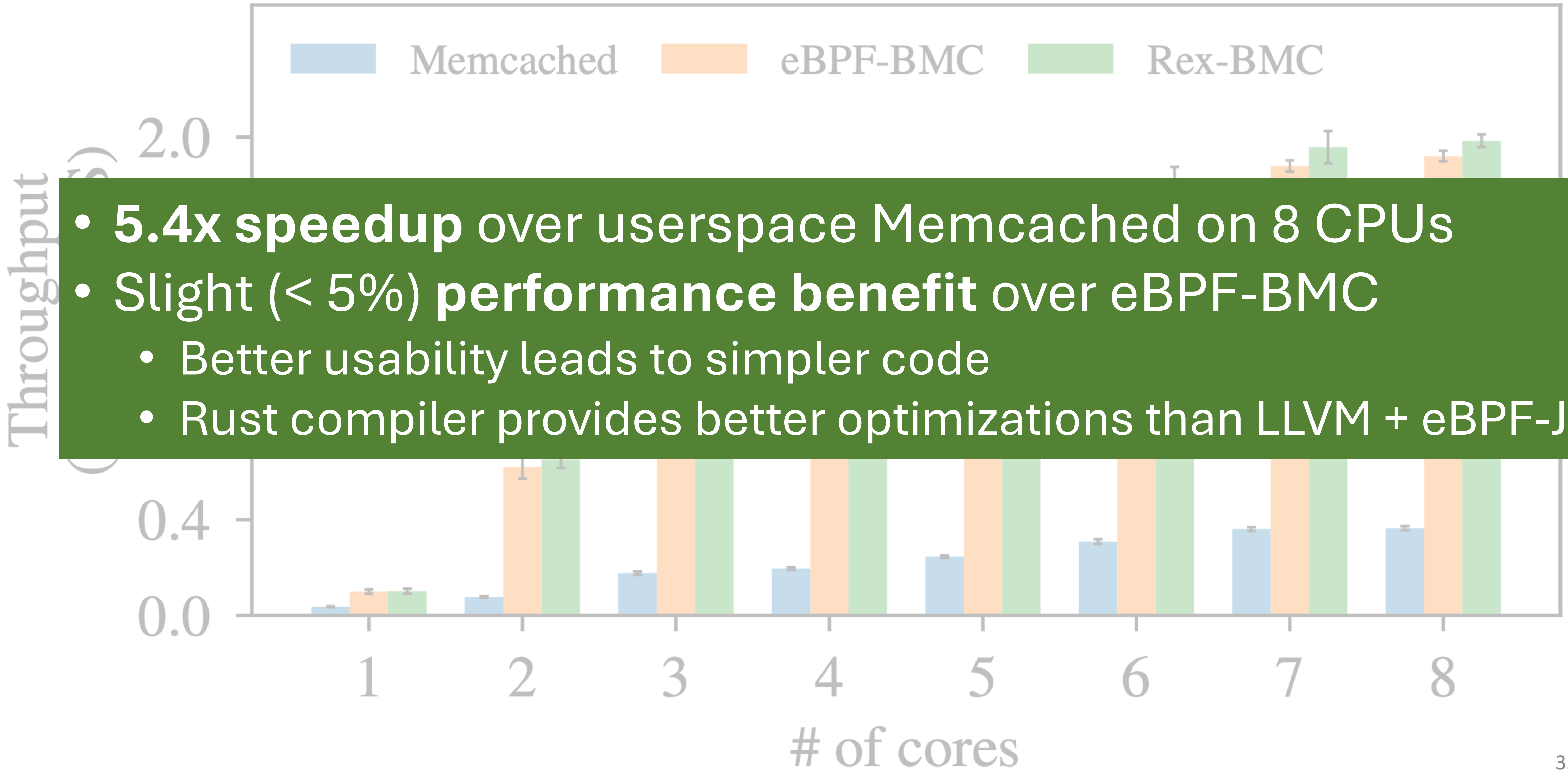
Performance evaluation



Performance evaluation



Performance evaluation



- **5.4x speedup** over userspace Memcached on 8 CPUs
- Slight (< 5%) **performance benefit** over eBPF-BMC
  - Better usability leads to simpler code
  - Rust compiler provides better optimizations than LLVM + eBPF-JIT

Conclusion

- Static verification in kernel extensions leads to poor usability
- Insight of Rex: Language-based safety + runtime mechanism
- Rex delivers usability without losing safety and performance
- Code: https://github.com/rex-rs/rex

GitHub