



Rex: safe and usable kernel extensions in Rust

Jinghao Jia

University of Illinois Urbana-Champaign

About us

- I was a Ph.D. student at UIUC working on safe kernel extensions
- Other involved parties in the project
 - Ruowen Qin from Red Hat
 - ROSA lab from Virginia Tech led by Professor Dan Williams
 - xlab from UIUC led by Professor Tianyin Xu
 - IBM Research

eBPF extensions are taking over the OS kernel

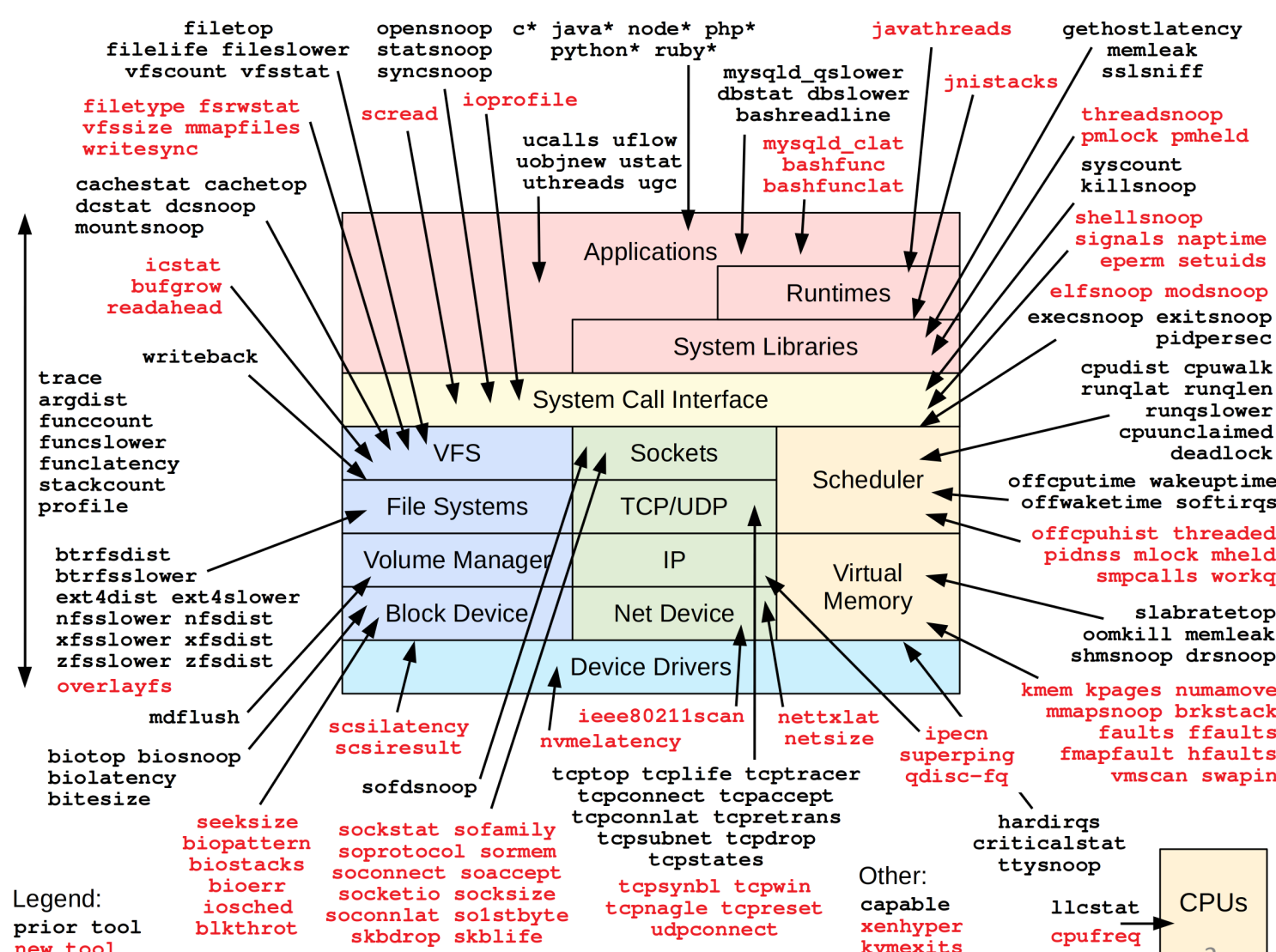
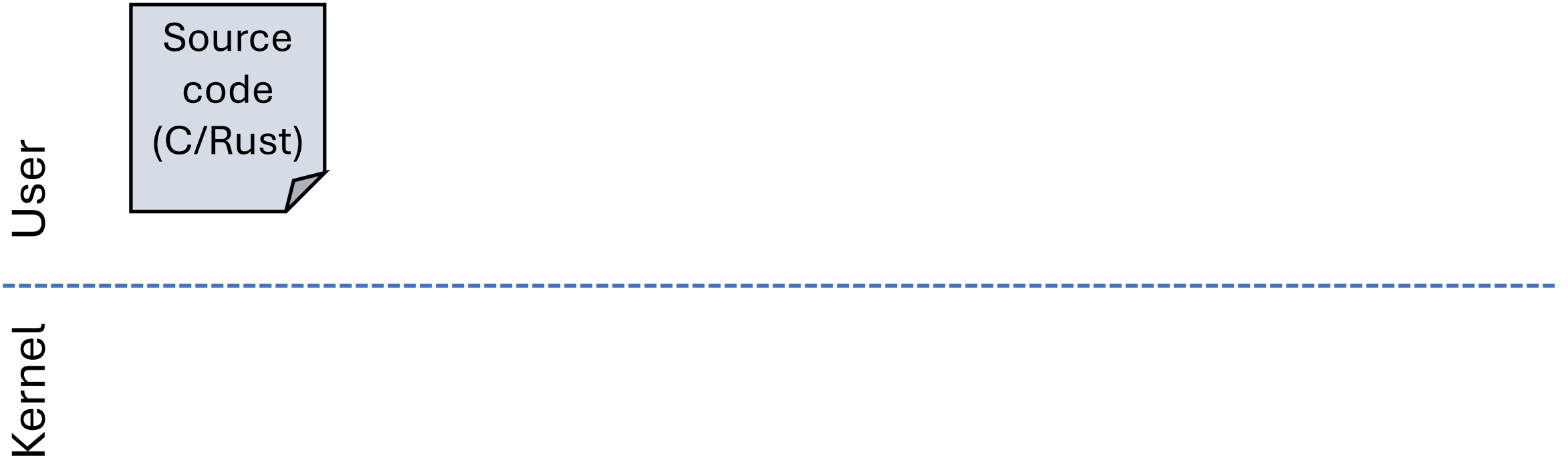
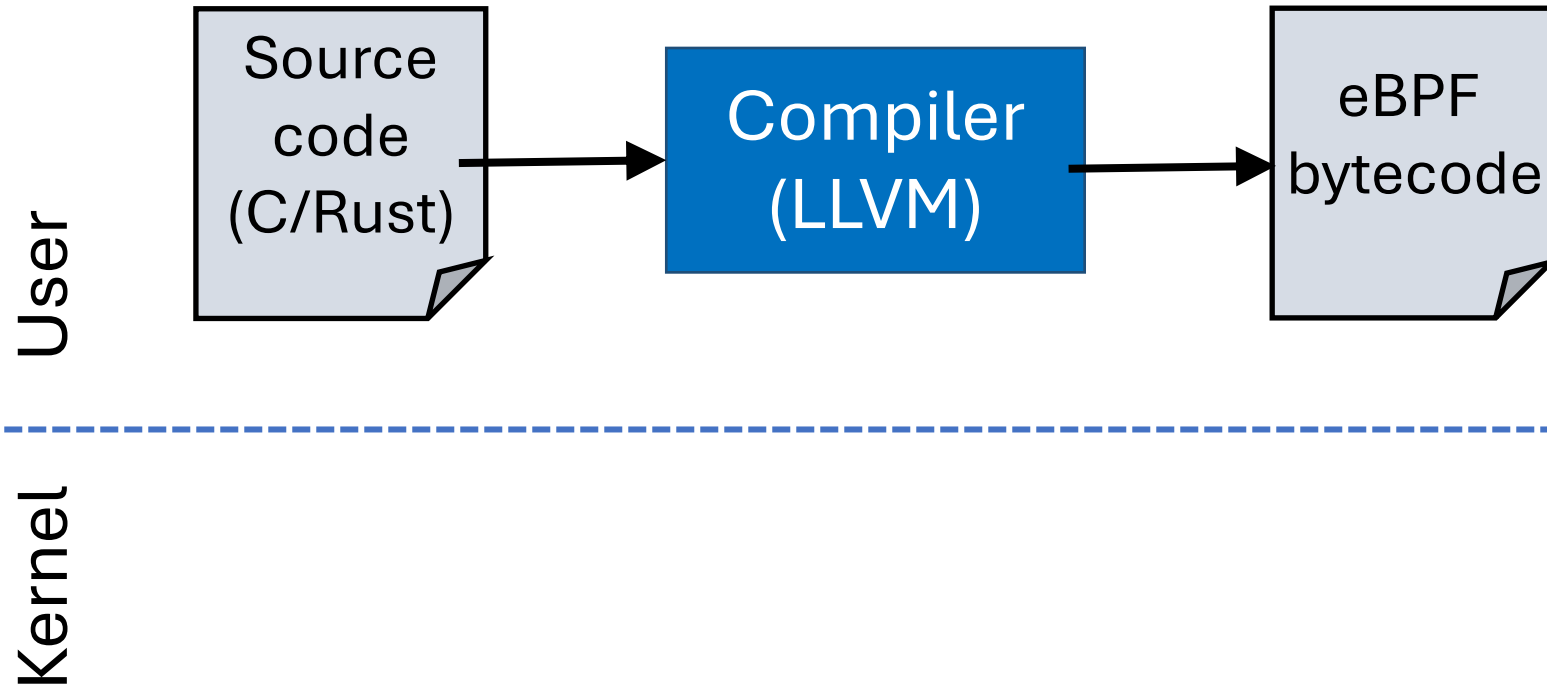


Figure credit: Brendan Gregg

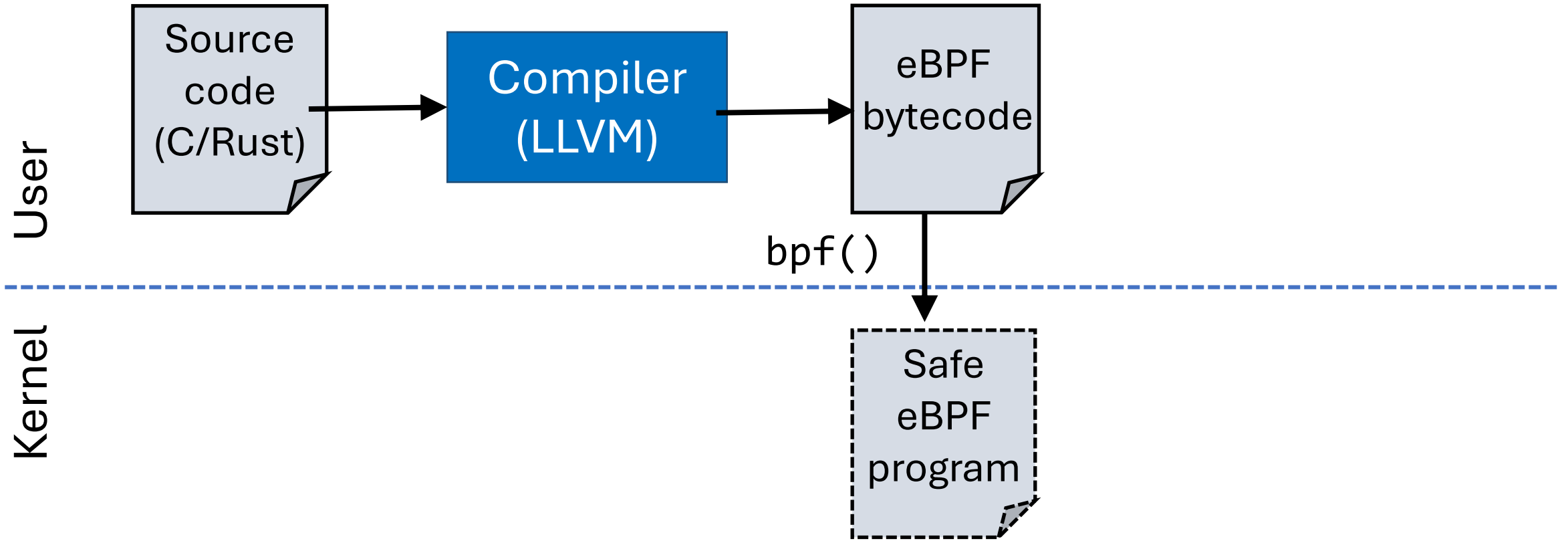
Basic principle of eBPF: Safety verification



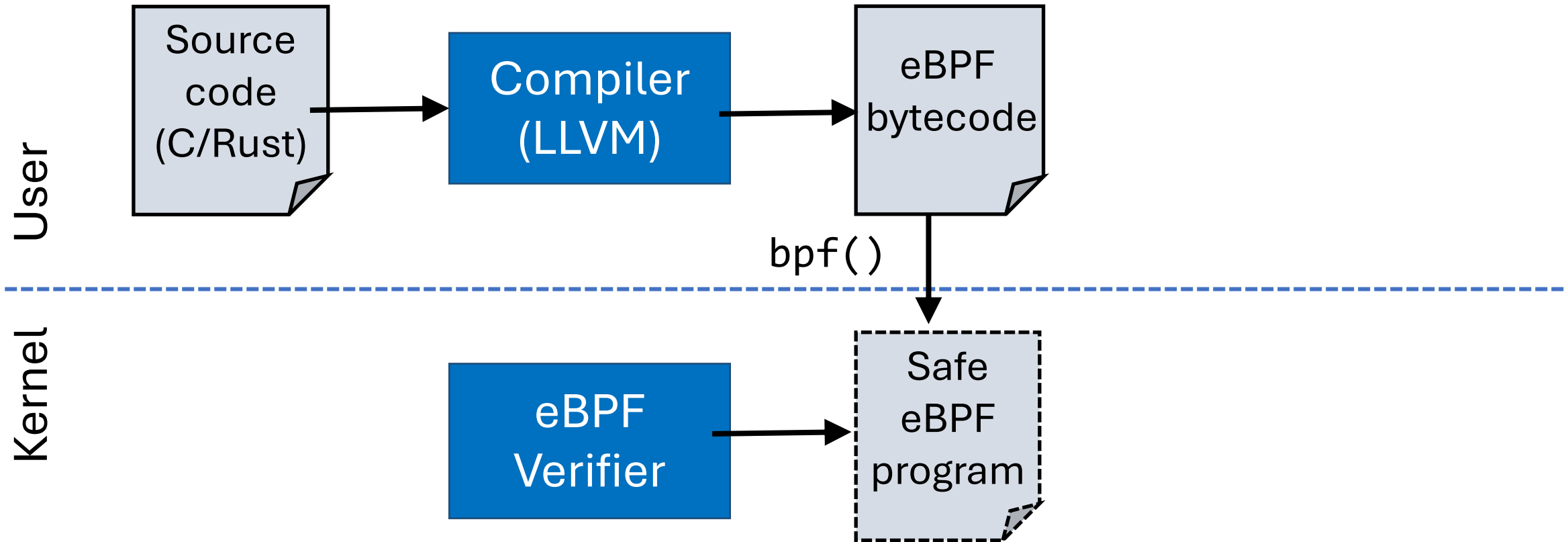
Basic principle of eBPF: Safety verification



Basic principle of eBPF: Safety verification

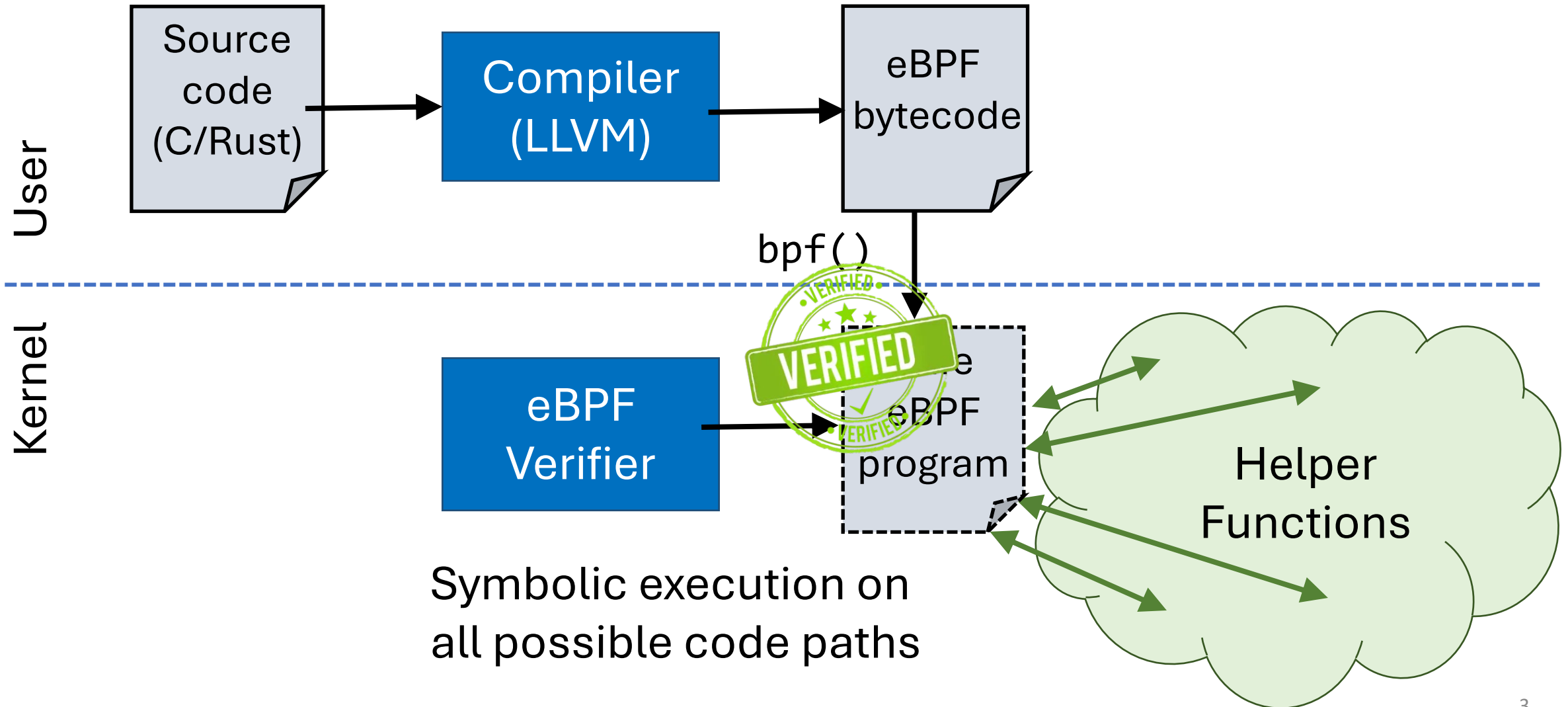


Basic principle of eBPF: Safety verification

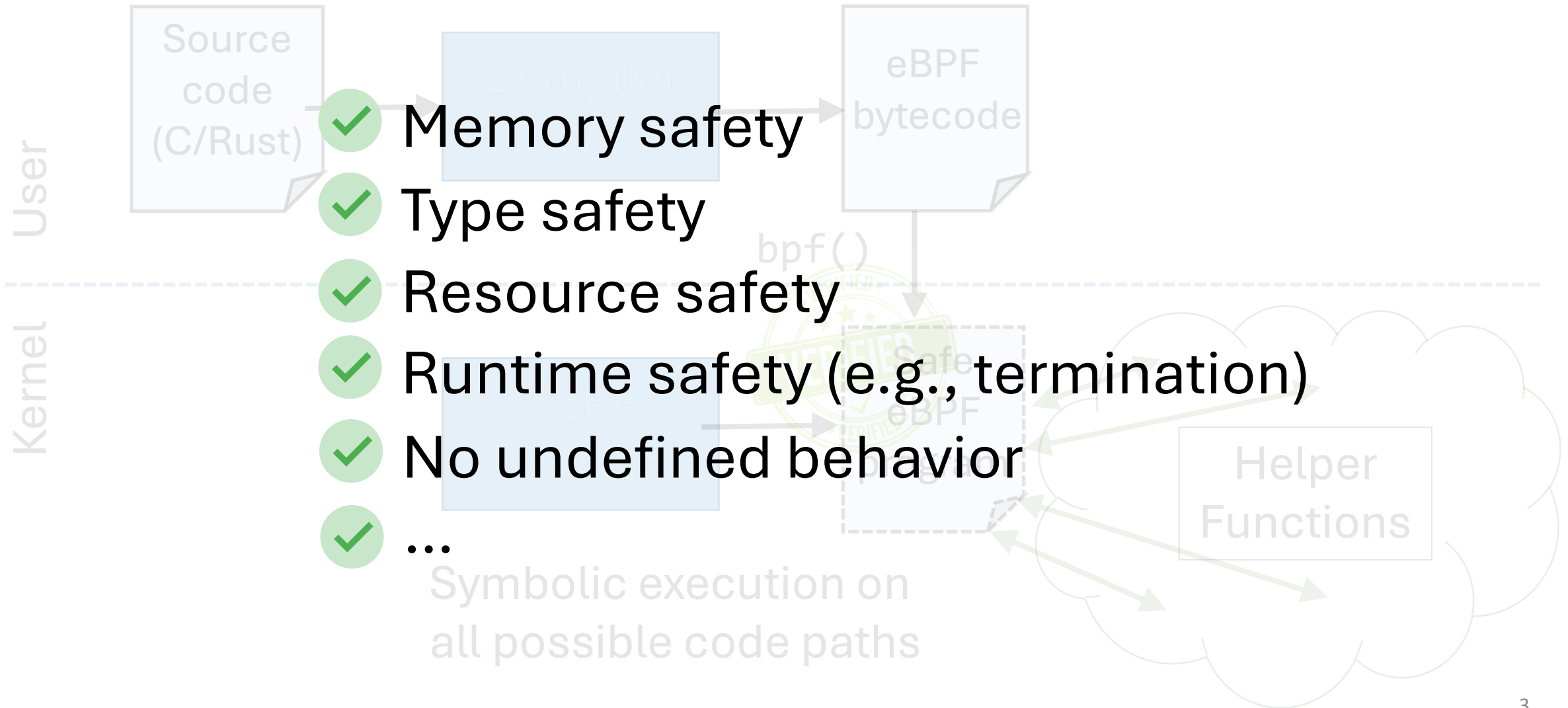


Symbolic execution on
all possible code paths

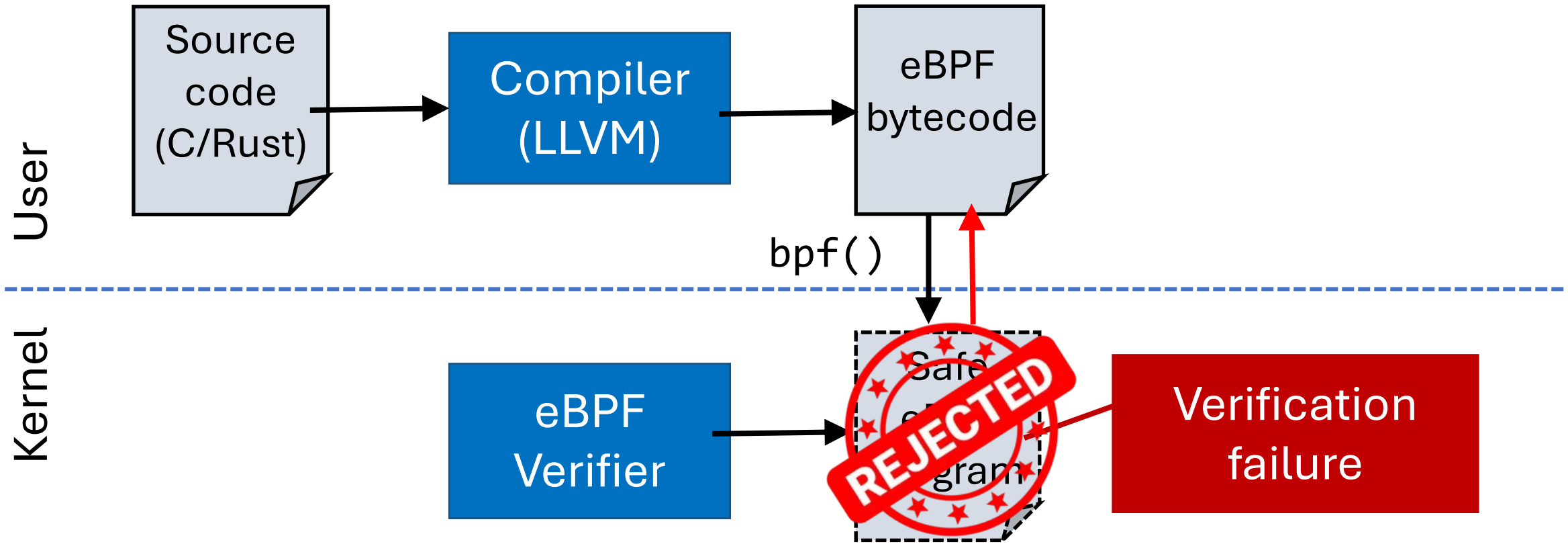
Basic principle of eBPF: Safety verification



Basic principle of eBPF: Safety verification

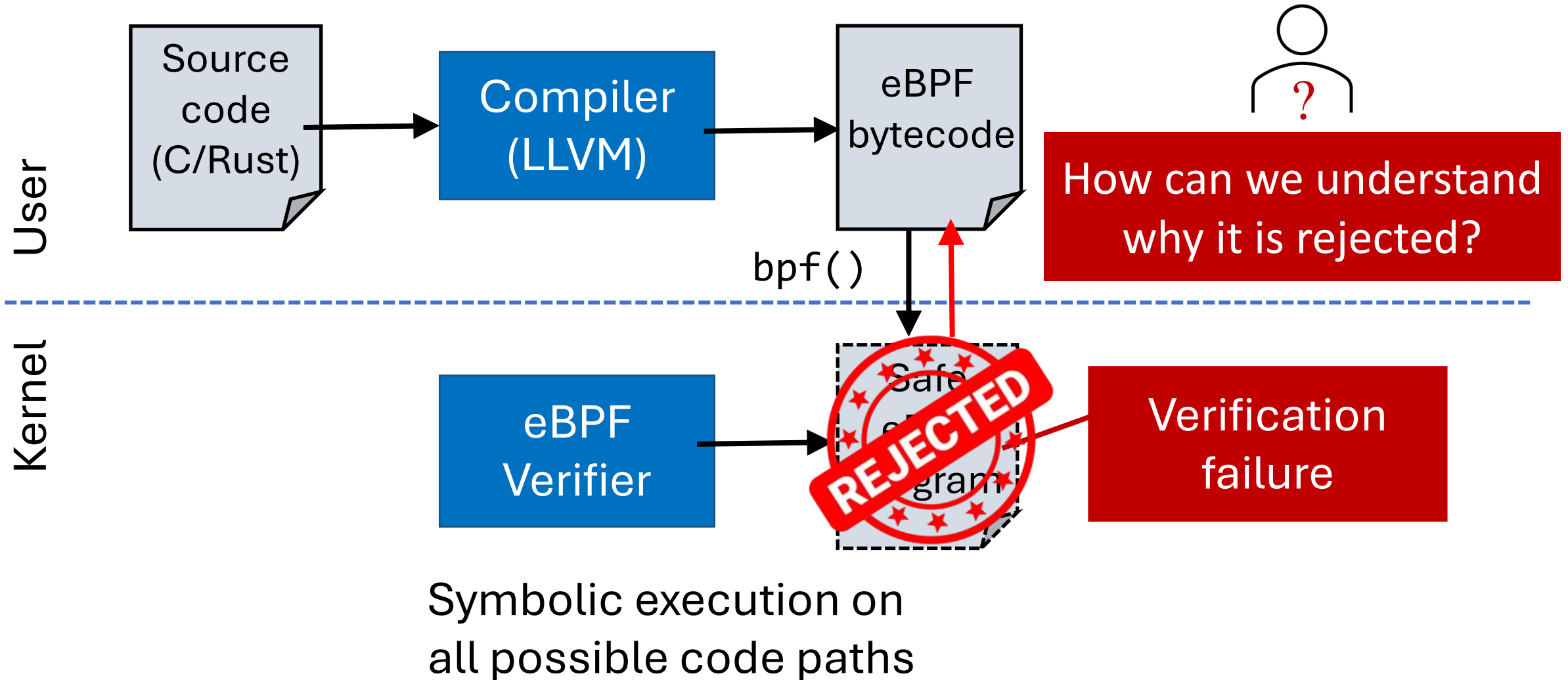


Basic principle of eBPF: Safety verification



Symbolic execution on
all possible code paths

Basic principle of eBPF: Safety verification



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 = ...;
    ...
    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/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/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



```
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



```
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)
```

Which source-code line do these instructions map to?

invalid bpf_context access off=44 size=2

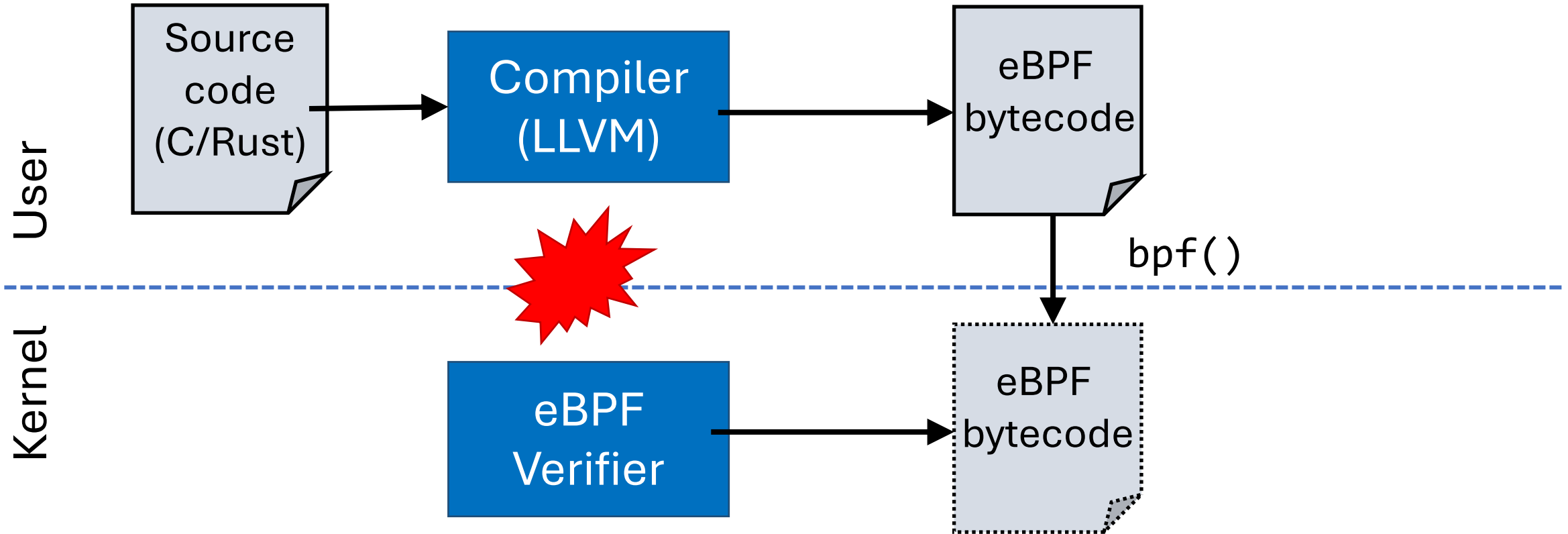
Why is it an invalid access?

Verifier log

Root cause: verifier does not understand compiler

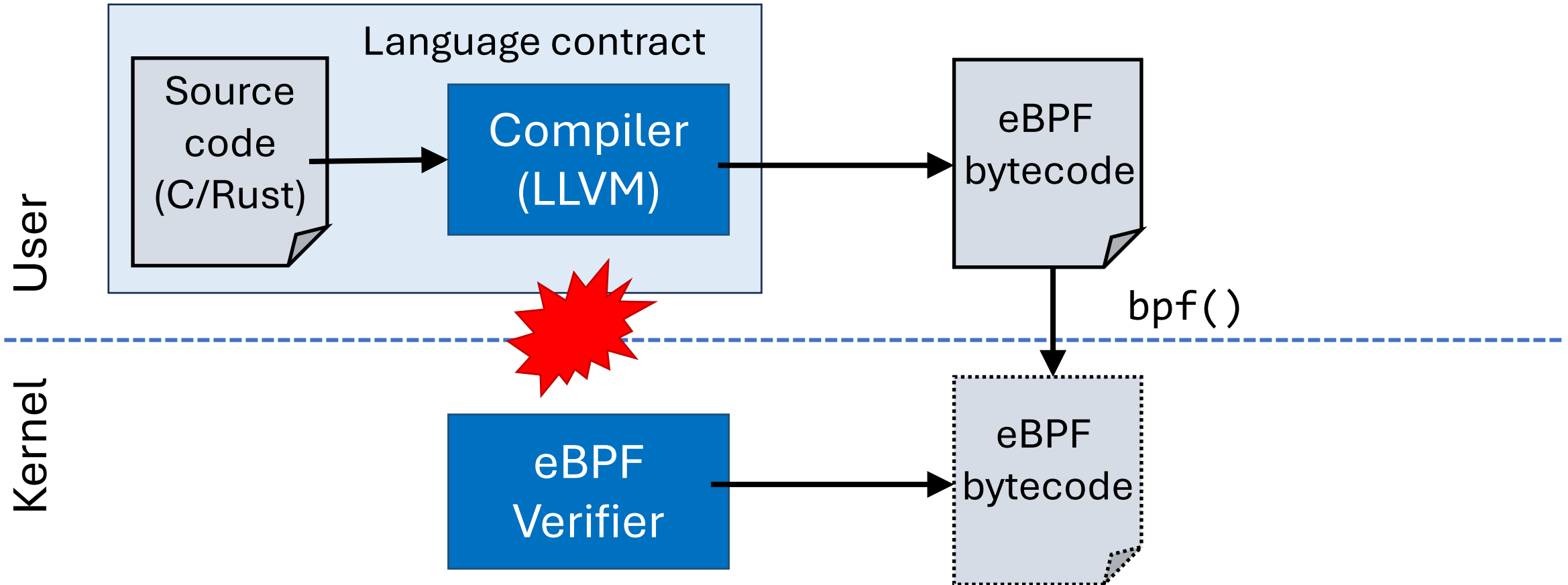
- The program uses `bpf_htons()` to convert the endianness of the `src_port` field in the context.
 - `src_port` is defined as a 32-bit int, while `bpf_htons()` only performs operations on the upper 16 bits
- The compiler optimizes the code to only load the upper 16 bits
- The verifier checks context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

The language-verifier gap



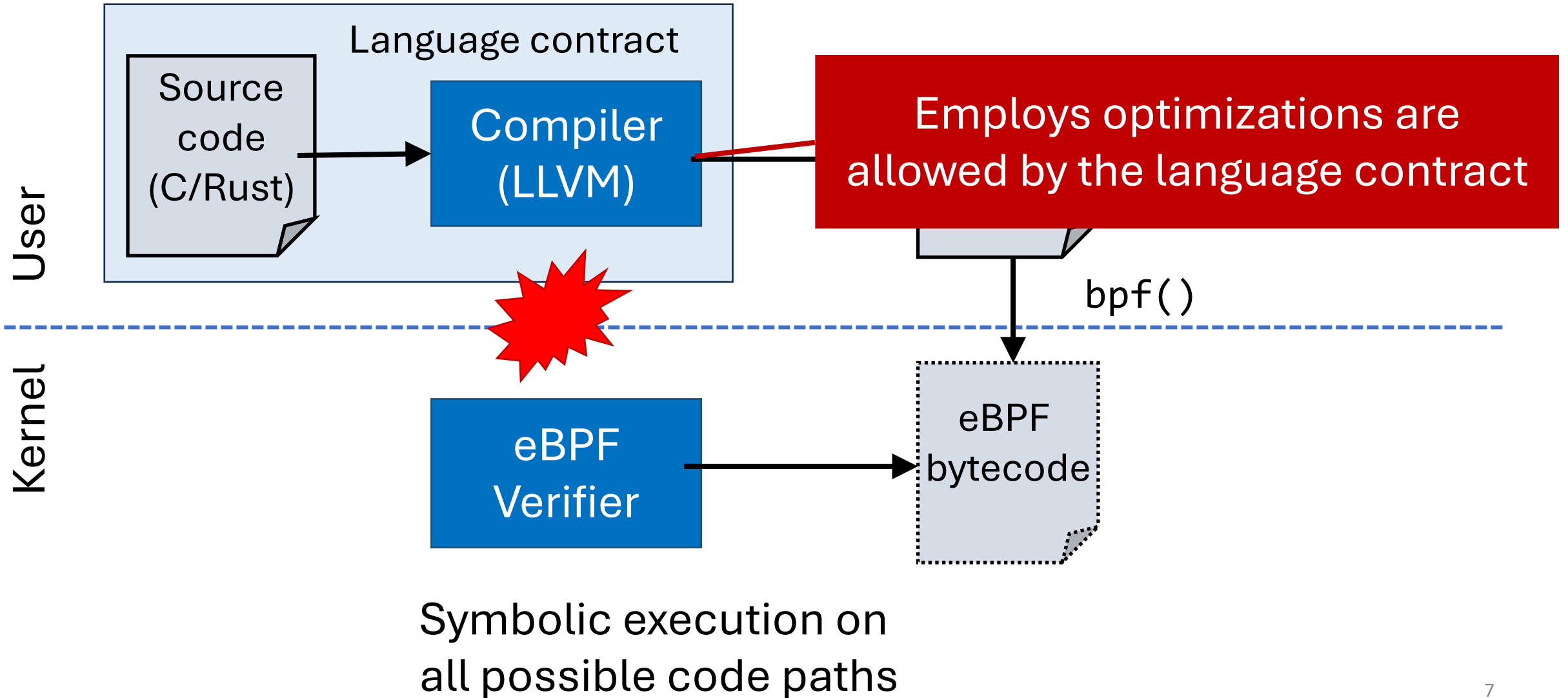
Symbolic execution on
all possible code paths

The language-verifier gap

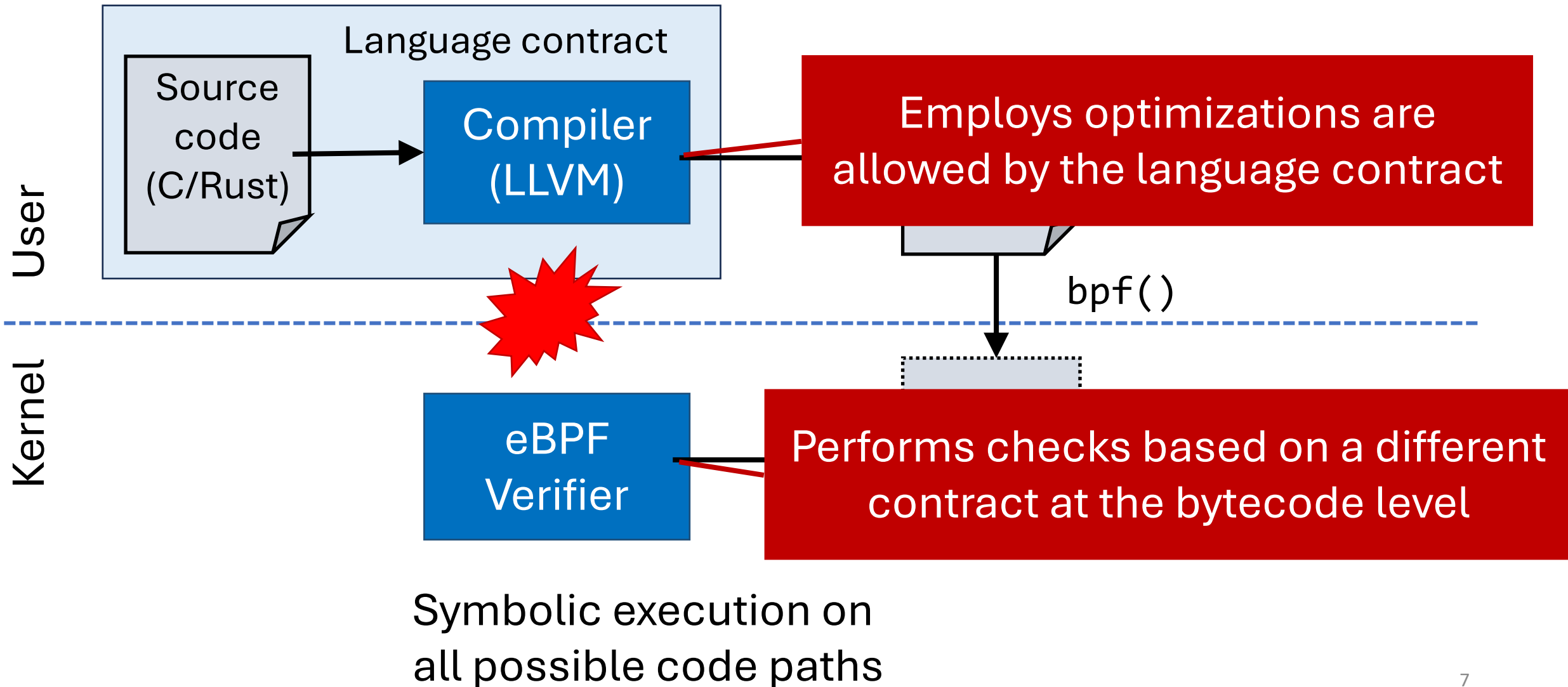


Symbolic execution on
all possible code paths

The language-verifier gap



The language-verifier gap



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

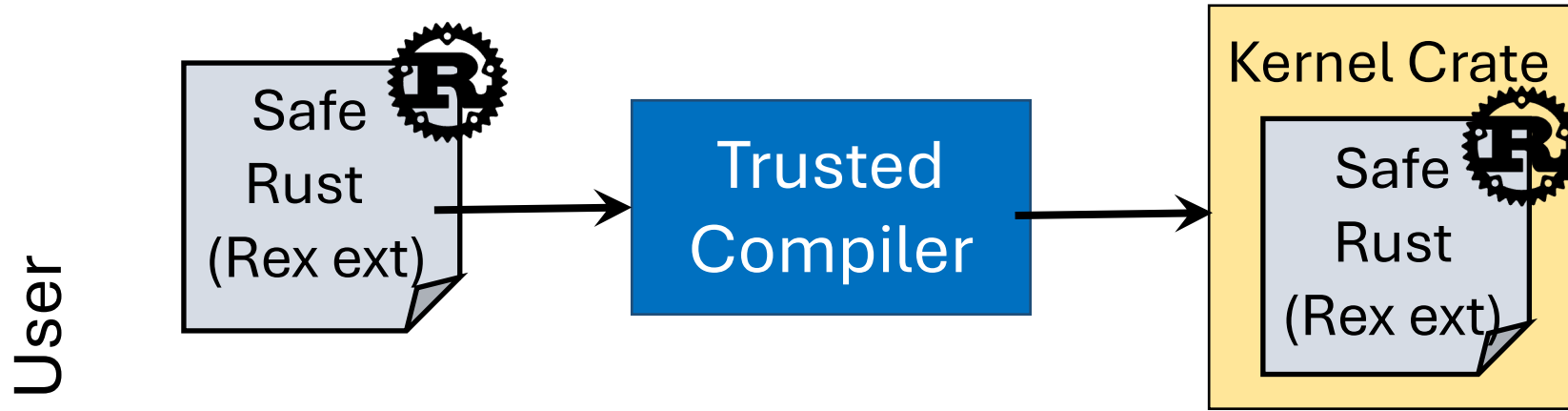
User

Kernel

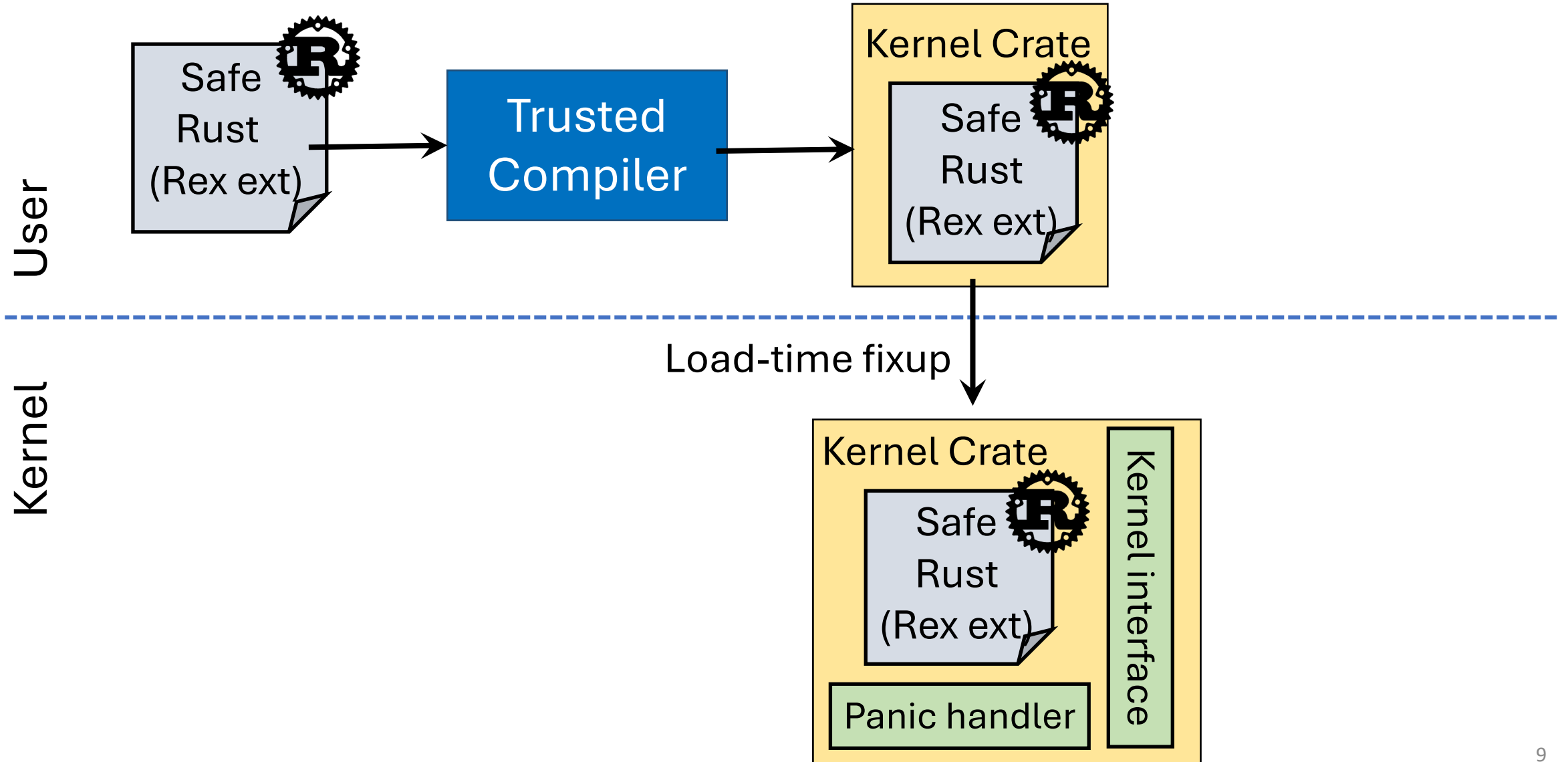
Rex: Safe, usable Rust kernel extensions



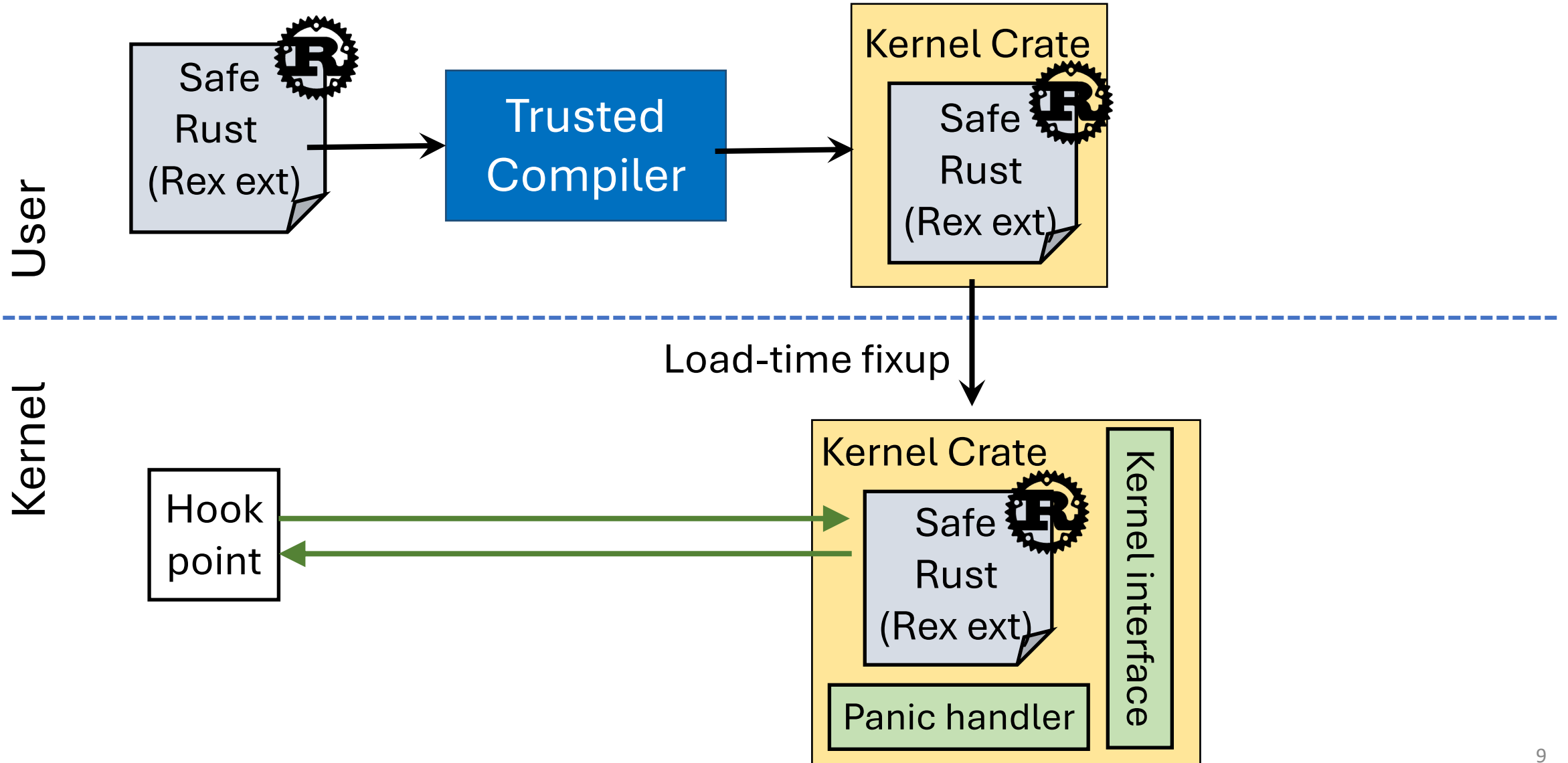
Rex: Safe, usable Rust kernel extensions



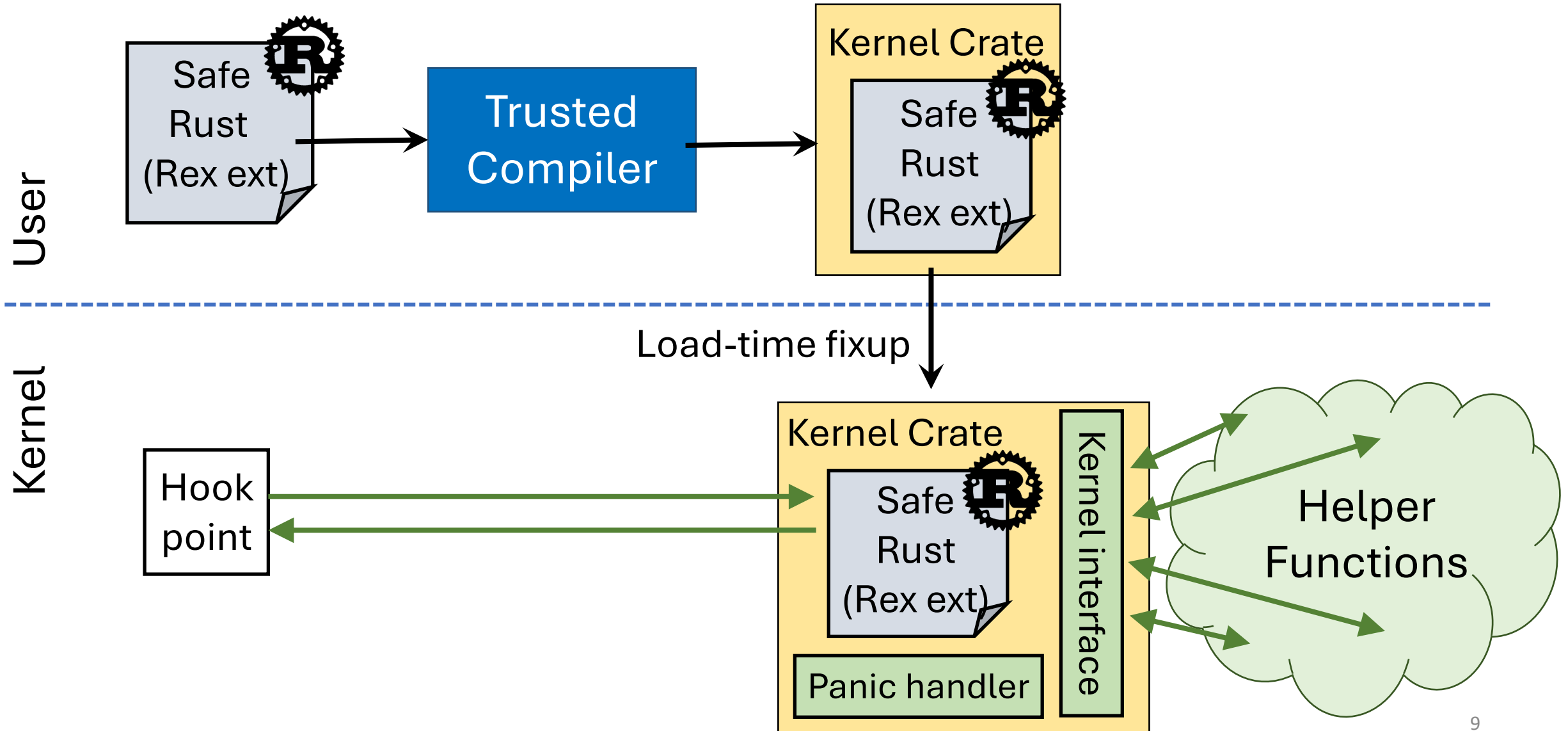
Rex: Safe, usable Rust kernel extensions



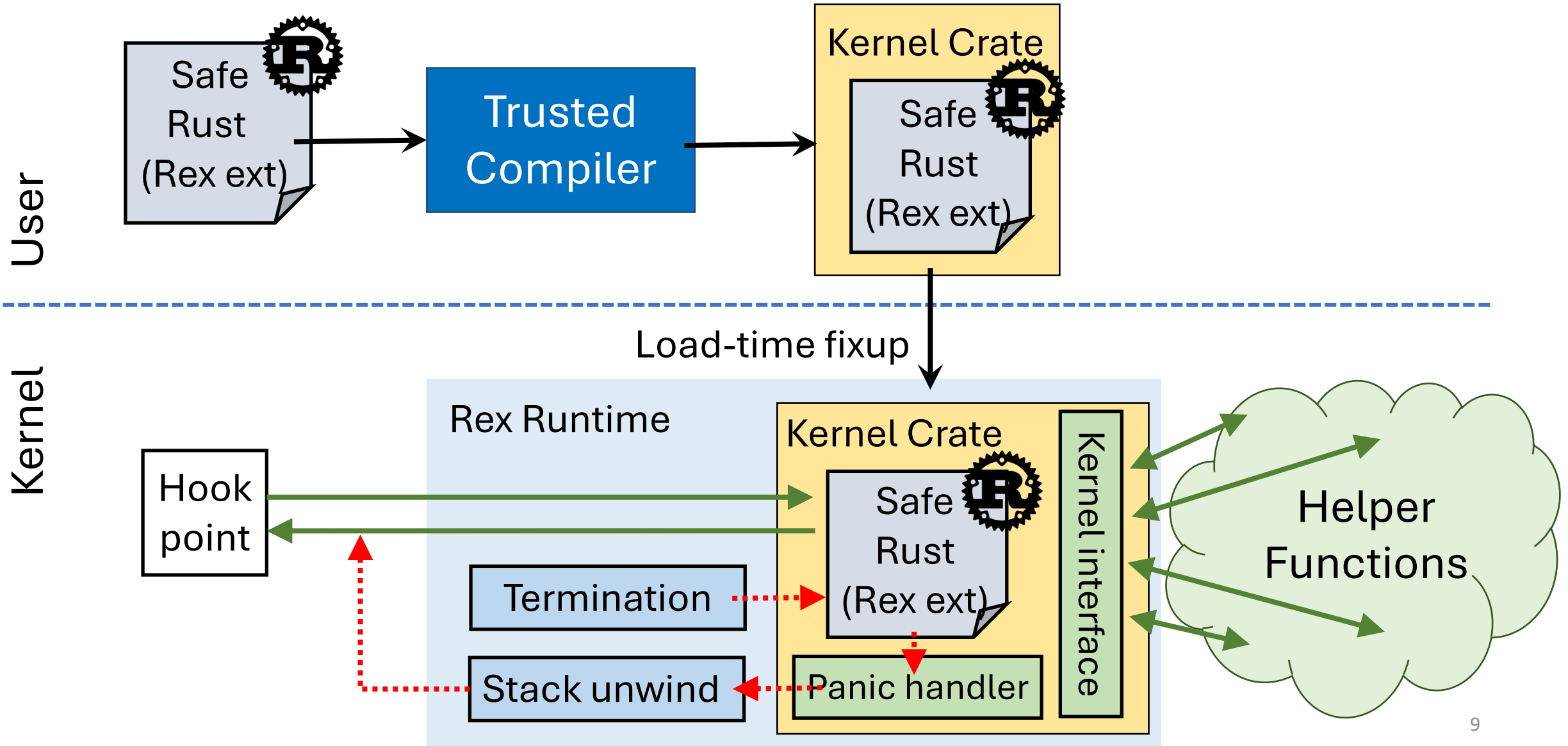
Rex: Safe, usable Rust kernel extensions



Rex: Safe, usable Rust kernel extensions

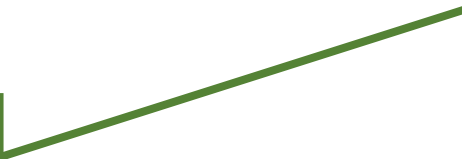


Rex: Safe, usable Rust kernel extensions



Safety in Rex

- 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



Focus of this talk

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 **must not** 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
```

Restore the old stack
pointer to unwind stack

```
jmp rex_exit
```

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 `ksyms` 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 `ka11syms` 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

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
- Classroom experience
 - Designed 4 assignments on kernel tracing and networking
 - Our undergraduate students can write correct Rex extensions!

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

```
// ...  
for (off = 0; off < 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;  
    }  
    ...  
}
```

No extra code to handle
complexity limit from verifier

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

```
// ...  
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

```
// ...  
for  
    payload + off + 1 <= 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

```
// ...  
for  
    payload + off + 1 <= 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  
        }  
    );  
...
```

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

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>



Discussion points

- How does the Rust-for-Linux community think about the idea of a new kernel extension mechanism that sits at the middle ground of Rust kernel modules and eBPF?
- Is there any aspect of Rex that is also useful for Rust-for-Linux and how can we contribute?
- Does the trust we put on the Rust toolchain make sense and how can we potentially make it more trustworthy?