ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise

(*)**Hongyu Li[1]**, (*)Liwei Guo[2], Yexuan Yang[1], Shangguang Wang[1], Mengwei Xu[1]

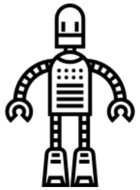
(*) = co-primary

[1]Beijing University of Posts and Telecommunications

[2]University of Electronic Science and Technology of China

Linux suffers from bugs

Software based on Linux



```
[    0.300014] Kernel BUG at 0xffff80001019ed8c [verbose debug info unavailable]
[    0.300034] Internal error: Oops - BUG: 0 [#1] PREEMPT SMP
[    0.300071] Modules linked in:
[    0.300113] CPU: 0 PID: 0 Comm: swapper/0 Not tainted 5.13.0-g7a293f65565c-dirty #616
[    0.300132] Hardware name: linux,dummy-virt (DT)
[    0.300148] IRQ stage: rros
[    0.300164] pstate: 600000c5 (nZCv daIF -PAN -UAO -TCO BTYPE=--)
[    0.300181] pc : 0xffff80001019ed8c
[    0.300197] lr : 0xffff80001019ecf8
[    0.300212] sp : ffff8000105efd10
[    0.300228] x29: ffff8000105efd10 x28: ffff800010488040 x27: 0000000000000000
[    0.300295] x26: ffff8000105f0000 x25: ffff0000c0671580 x24: ffff8000105efdc0
[    0.300359] x23: 0000000000000027 x22: ffff800010503340 x21: ffff0000ffa8ea90
[    0.300421] x20: ffff0000ffa8eab0 x19: ffff0000ffa8eab0 x18: 0000000000000198
[    0.300484] x17: 000000000000b67e x16: 00000000000000a4 x15: ffff8000101e7950
[    0.300546] x14: ffff80001033d4c0 x13: 000000000000004b x12: 000000000000002b
[    0.300609] x11: 0000000000000000 x10: ffff80001031fb88 x9 : ffff800010321ed0
[    0.300673] x8 : ffff80001031fb88 x7 : 0000000000000000 x6 : ffff0000ffa81699
```

Projects	Type	Mem bugs percentage
Chromium	User program	69%
Mozilla	User program	74%
Ubuntu	Kernel program	65%
Microsoft	Kernel Program	70%
Android	Kernel program	65%-90%
IOS/macOS	Kernel program	66.3%/71.5%

Efforts to counter Memory/Thread bugs

- Static analysis[1]
  - Gcc –Wanalyze*
  - Clang
  - cppcheck
  - Codechecker
- Runtime detection[2]
  - Kernel Memory Sanitizer (KMSAN)
  - Kernel Concurrency Sanitizer (KCSAN)
  - Undefined Behavior Sanitizer (UBSAN)
- Kernel testing[2]
  - KUnit/Kselftest/LTP/Kernel CI/Fuzz

[1] https://events.linuxfoundation.org/wp-content/uploads/2021/01/Static-Analysis-JSMoeller-LF-Live-Mentor-Series.pdf
[2] https://www.kernel.org/doc/html/next/dev-tools/index.html

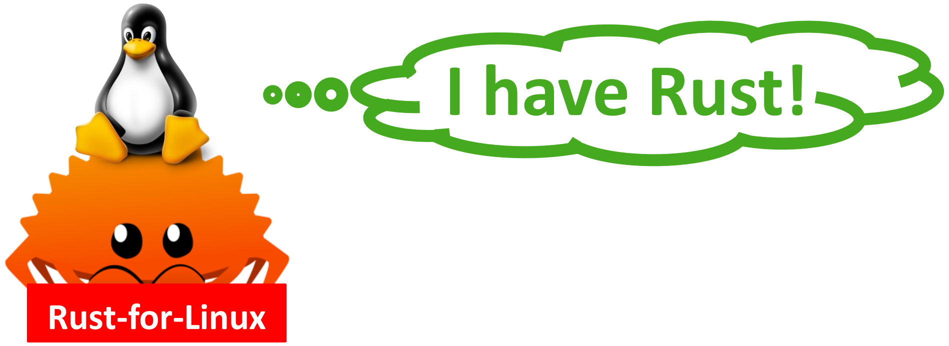
But Memory/Thread bugs still exist[1]

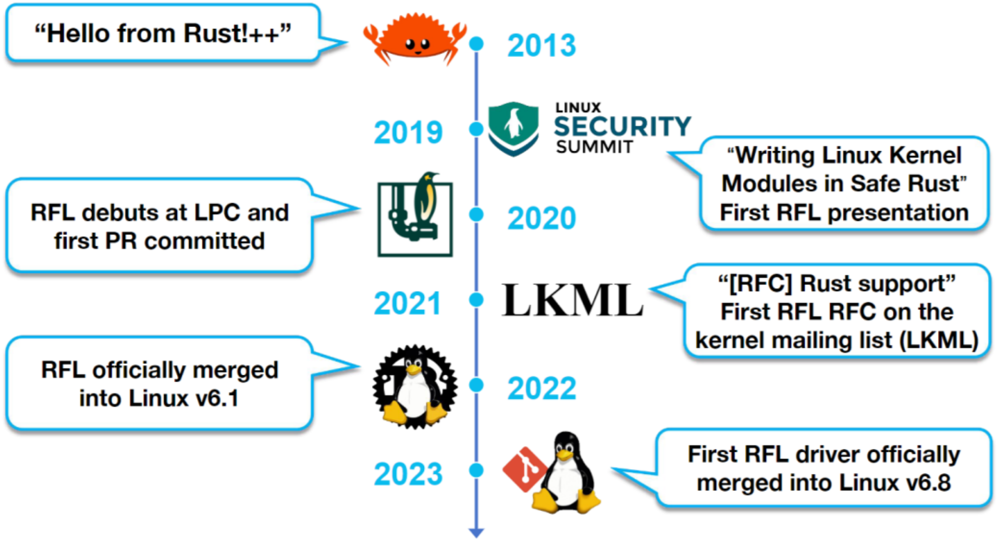
Vulnerabilities by types/categories

Year	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	File Inclusion	CSRF	XXE	SSRF	Open Redirect	Input Validation
2014	20	31	0	0	0	0	0	0	0	0	22
2015	13	17	0	0	0	0	0	0	0	0	5
2016	36	76	0	0	0	0	0	0	0	0	17
2017	64	86	0	0	1	0	0	0	0	0	20
2018	32	70	0	0	0	0	0	0	0	0	11
2019	30	124	0	0	1	0	0	0	0	0	4
2020	10	41	0	0	1	0	0	0	0	0	2
2021	19	54	0	0	2	0	0	0	0	0	7
2022	41	149	0	0	0	[illegible]	0	0	0	0	2
2023	18	172	0	0	0	[illegible]	0	0	0	0	2
2024	30	452	0	0	0	[illegible]	[illegible]	0	0	0	0
Total	313	1272			5	[illegible]					92

[1] https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

Rust can help





Rust-for-Linux (RFL) wants to write drivers with Rust

Before meeting Rust:

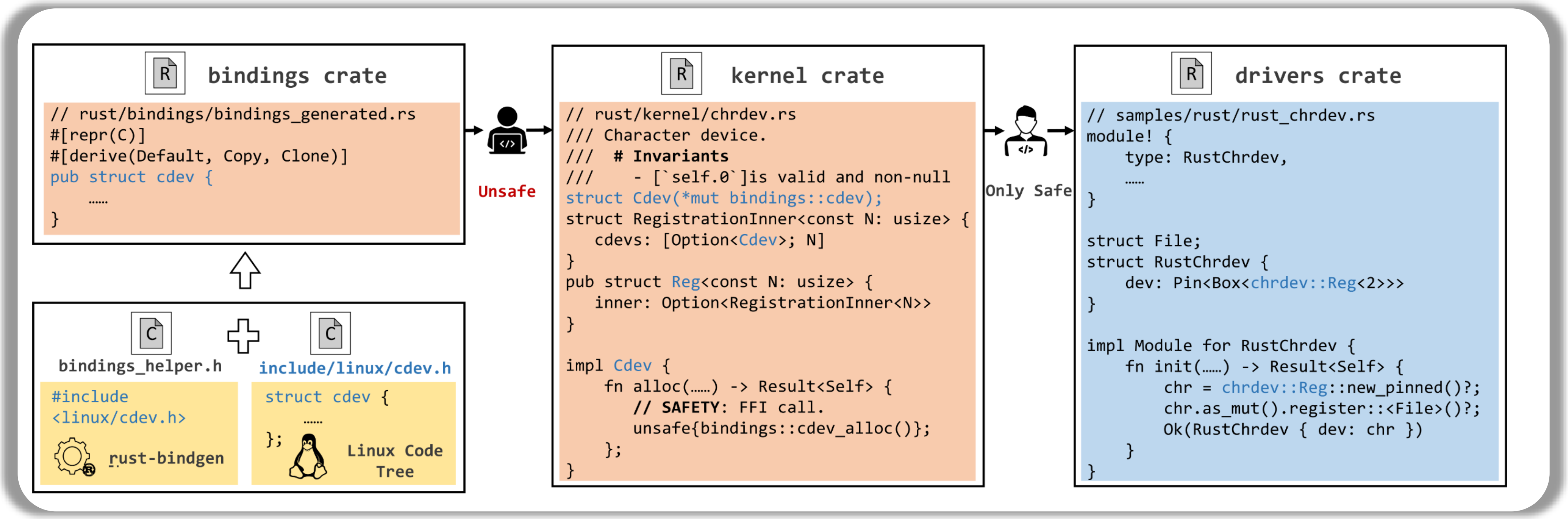
*"I have yet to see a language that comes even close to **C**."*

After meeting Rust:

*"While I won't make any promises, I'd like to see **Rust** merging into the Linux kernel with the next release."*

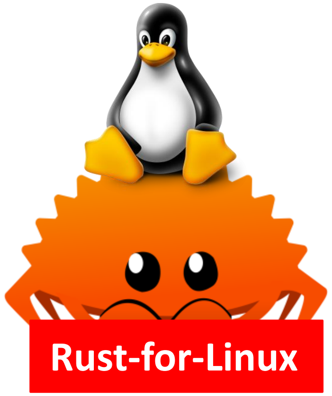
Background

1. Rust's safety rules cause limited expressiveness (Double linked list)
2. Code in the *Unsafe* block can break the safety rules
   - ➢ Calling function from foreign function interface (FFI) needs *unsafe* blocks
3. It's proven possible to wrap *unsafe* blocks under safe APIs

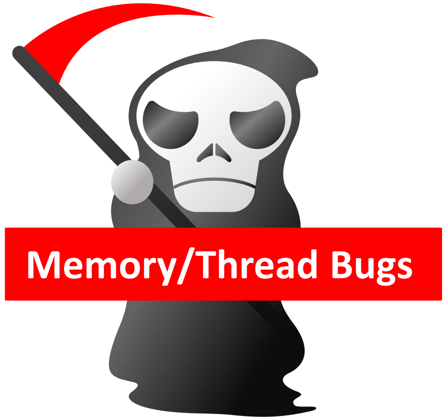


Motivation

While we know about Rust, RFL is rarely studied



RQ1: what is the status quo of RFL?
RQ2: does RFL live up to the hype?
RQ3: what are the lessons learned from RFL?

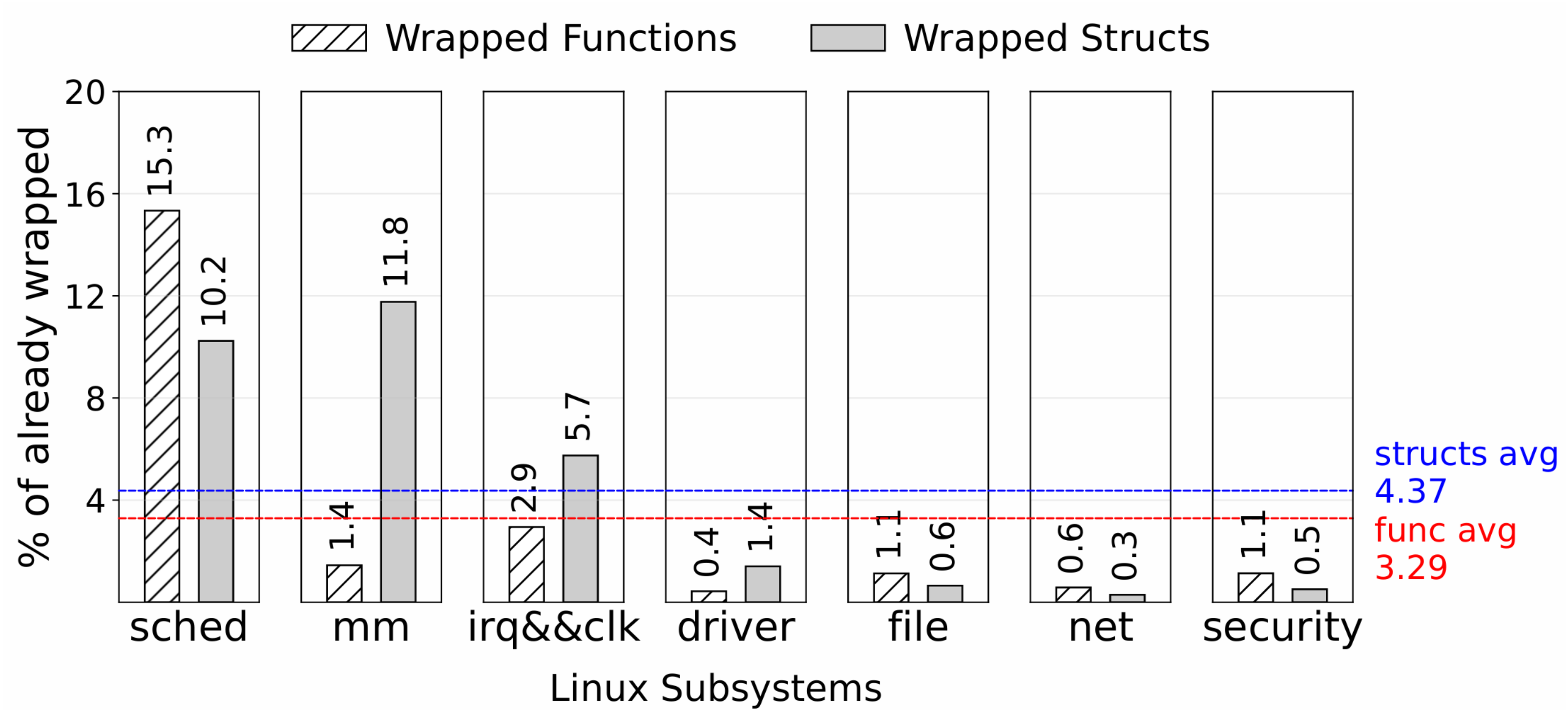


RQ1: what is the status quo of RFL?

Q1: RFL development status
Q2: How to construct safety abstraction
Q3: How to rustify device drivers

Q1: RFL development status (RQ1)

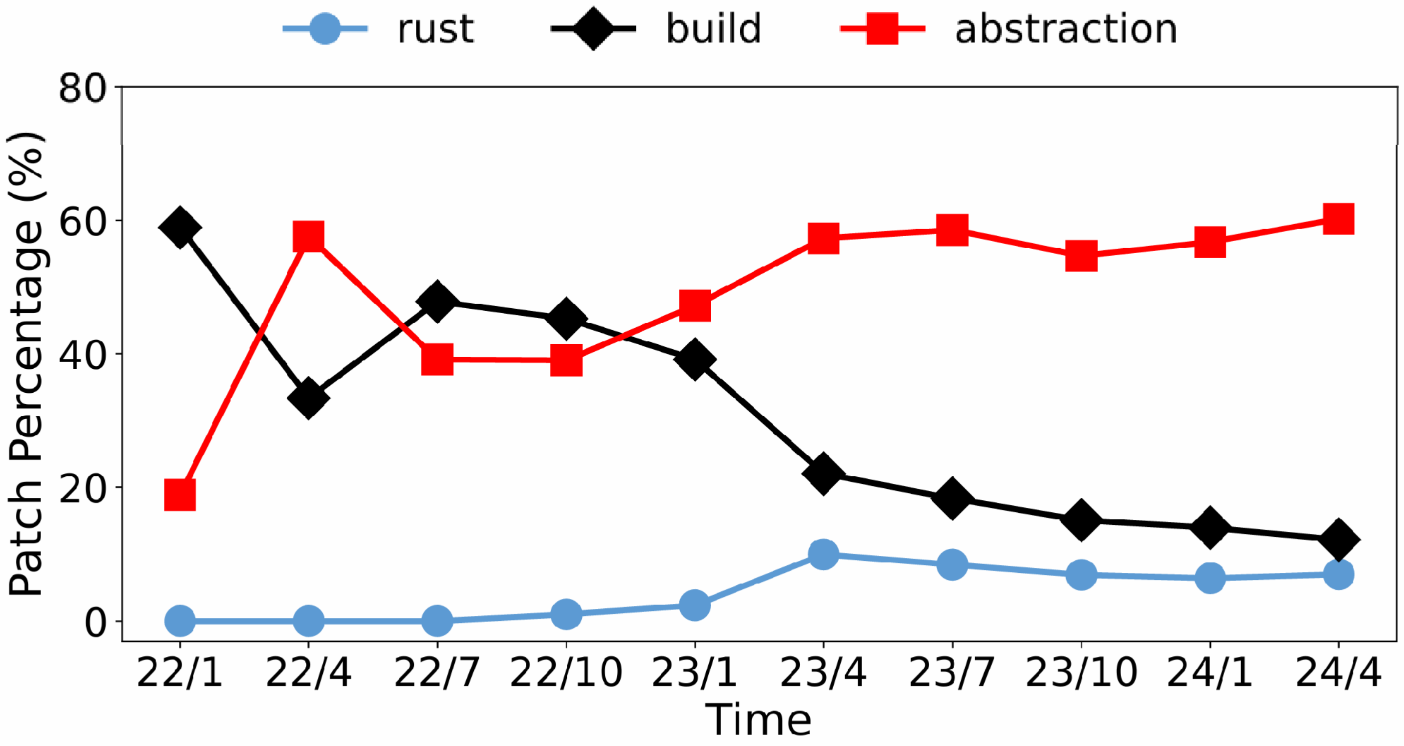
1. Development progress



Insight 1: drivers, netdev, and file systems are the long tail of RFL code.

Q1: RFL development status (RQ1)

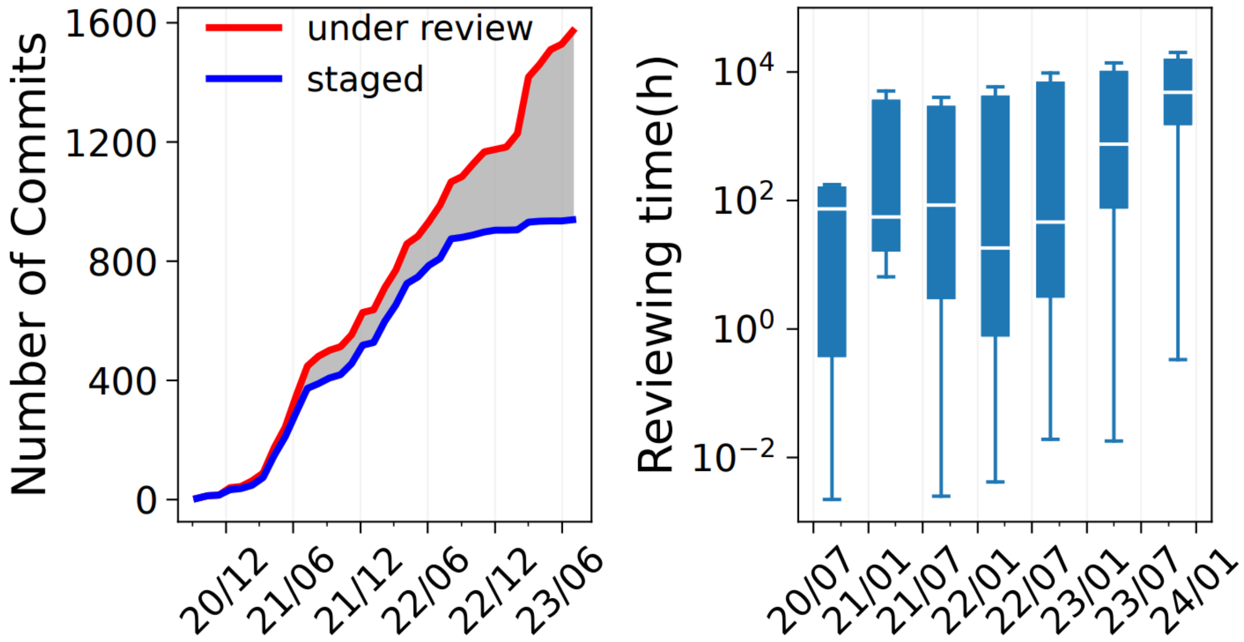
1. Development progress
2. Patch distribution



Insight 2: RFL infrastructure has matured, with safe abstraction and drivers being the next focus.

Q1: RFL development status (RQ1)

1. Development progress
2. Patch distribution
3. The trend



<u>Insight 3: RFL is bottlenecked by code review but not by code development.</u>

Q2: How to construct safety abstraction (RQ1)

1. Structs safety abstraction
   - rust-bindgen: same layout in memory

```c
struct llist_head {
    struct llist_node *first;
};
```

```rust
#[repr(C)]
#[derive(Copy, Clone)]
pub struct llist_head {
    pub first: *mut llist_node,
}
```

```rust
impl Default for llist_head {
    fn default() -> Self {
        unsafe { ::core::mem::zeroed() }
    }
}
```

Q2: How to construct safety abstraction (RQ1)

1. Structs safety abstraction
   - rust-bindgen: same layout in memory
     - Bit field/union

```
struct fs_parameter {
    const char       *key;        /* Parameter name */
    enum fs_value_type  type:8;      /* The type of value here
*/
    union {
        char        *string;
        void        *blob;
        struct filename *name;
        struct file *file;
    };
};
```

Q2: How to construct safety abstraction (RQ1)

1. Structs safety abstraction
   - rust-bindgen: same layout in memory
     - Bit field/union

```
pub struct __BindgenBitfieldUnit<Storage, Align> {
    storage: Storage,
    align: [Align; 0],
}
```

```
impl<Storage, Align> __BindgenBitfieldUnit<Storage, Align> {
       pub fn get(&self, bit_offset: usize, bit_width: u8) -> u64
       pub fn set(&mut self, bit_offset: usize, bit_width: u8, val: u64)
}
```

Q2: How to construct safety abstraction (RQ1)

1. Structs safety abstraction
   - rust-bindgen: same layout in memory
   - Abstraction
     - Deref is valid: *ptr -> foo<*mut ptr>

```rust
impl File {
    /// Creates a reference to a [`File`] from a valid pointer.
    /// # Safety
    /// The caller must ensure that `ptr` is valid and remains valid for the lifetime of
    /// the returned [`File`] instance.
    pub(crate) unsafe fn from_ptr<'a>(ptr: *const bindings::file) -> &'a File {
        // SAFETY: The safety requirements guarantee the validity of the dereference,
        // while the `File` type being transparent makes the cast ok.
        unsafe { &*ptr.cast() }
    }
}
```

Q2: How to construct safety abstraction (RQ1)

1. Structs safety abstraction
2. Functions safety abstraction
   - Functions as the members of the struct
     - /// # Invariants
     - /// # Safety
     - // SAFETY:

```
impl File {
    /// Returns the flags associated with the file.
    ///
    /// The flags are a combination of the constants in [`flags`].
    pub fn flags(&self) -> u32 {
        // SAFETY: The file is valid because the shared reference guarantees a nonzero
refcount.
        unsafe { core::ptr::addr_of!((*self.0.get()).f_flags).read() }
    }}
```

Q2: How to construct safety abstraction (RQ1)

1. Structs safety abstraction
2. Functions safety abstraction
   - Functions as the members of the struct
   - Function pointers as *trait*

```
impl<T: Operations> OpsTable<T> {
    const VTABLE: bindings::dev_pm_ops = bindings::dev_pm_ops {
        suspend: Some(suspend_callback::<T>),
        resume: Some(resume_callback::<T>),
        freeze: Some(freeze_callback::<T>),
        restore: Some(restore_callback::<T>),
    };
    ……
}
```

Q3: How to rustify device drivers (RQ1)

1. Workflow
   - Device probe
   - Driver logic
   - Device cleanup
2. Rust/RFL abstraction influences programing inflexibility

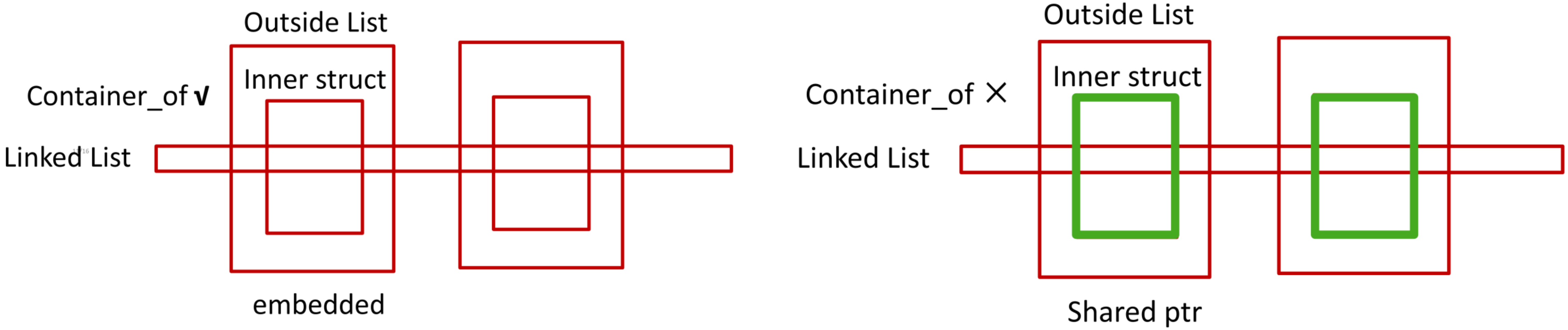
```
// In C
struct elements { int len; void* inner; };
struct factory {
          struct elements inner;
};

// In Rust with Fixed N
struct elements<const N: usize> {
          inner: [foo; N],
}
struct factory { inner: elements<256/8> }

// In Rust with Dynamic change N
struct thread/proxy<const N: usize>{
         thread/proxy_elements: elements< N >,
}
impl dyn_num for thread<256>/proxy<8> {}
trait dyn_num { // fn use_elements(&self); }
struct factory { inner: &'static dyn dyn_n }
```

Q3: How to rustify device drivers (RQ1)

1. Workflow
2. Rust/RFL abstraction influences programing flexibility
   - ➢ Container_of



<u>Insight 4: The major difficulty of writing safe drivers in Rust is to reconcile the inflexibility of Rust versus kernel programming conventions.</u>

RQ2: does RFL live up to the hype?

Q1: Does Rust help Linux become safer?
Q2: Does Rust bring additional overhead?
Q3: How does Rust improve Linux development?

Fields	Goal
Safety	Memory-safe and thread-safe drivers.
Performance	Zero overhead on abstraction.
Tools	Better documents and CI test quality.
Efficiency	Higher development efficiency.
Community	More developers in the kernel.

Q1: Does Rust help Linux become safer? (RQ2)

1. There exist *soundness bugs* in the safety abstractions
   - Wrapping unsafe APIs needs manually review
   - Bugs may not disappear, just hide deeper[1]

Source	Compilation bug	Soundness bug
GitHub [22]	4(1/3)	7(3/4)
Intel LKP [41]	8(6/2)	0
Mailing List [44]	4(4/0)	2(1/1)

[1] https://lwn.net/Articles/953116/

Q1: Does Rust help Linux become safer? (RQ2)

1. There exist soundness bugs in the safety abstractions
2. The RFL drivers use *unsafe* blocks
   - The driver itself still needs *unsafe* due to complex logic
   - The safety abstraction is hard to maintain pure safe[1]

Driver	Number of Unsafe usage	
	Driver logic	**Safety abstractions**
GPU [67]	107	7
NVME [69]	44	16
Null block [68]	0	0
E1000 [62]	4	2
Binder [59]	45	9
Gpio_pl061 [64]	0	3
Semaphore [70]	0	4

Insight 5: with RFL, Linux becomes more "securable" but still cannot be fully secure.

[1] https://github.com/Rust-for-Linux/linux/commit/90e53c5e70a69159ec255fec361f7dcf9cf36eae

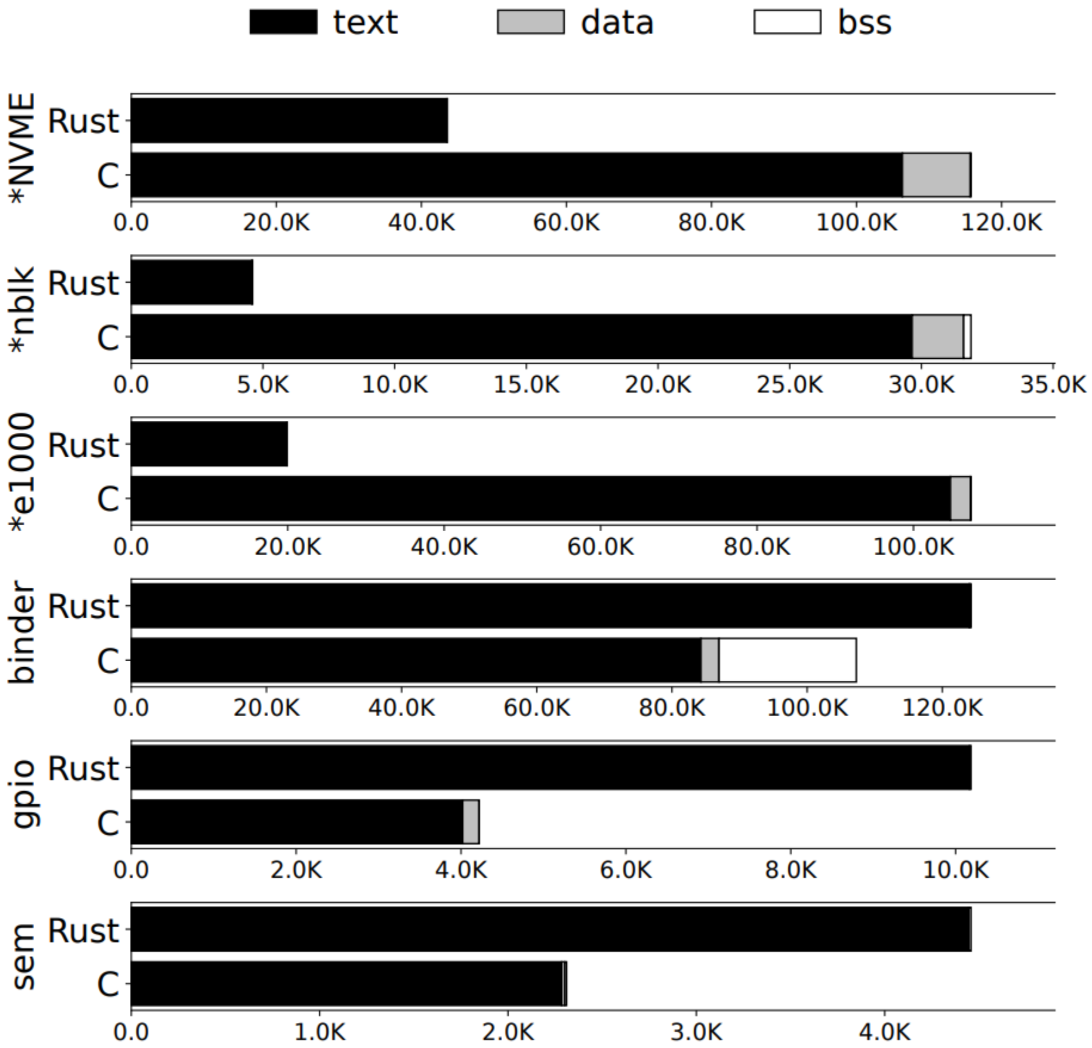
Q2: Does Rust bring additional overhead? (RQ2)

1. Setup
   - NVME and binder are considered the first batch of drivers to be merged in the Linux mainline
   - Others: File system/Network/Driver

Driver	Benchmark	Metrics		Device
NVME	fio	driver size	throughput	PC
Null Block	fio		throughput	PC
E1000	ping		latency	PC
Binder	ping		latency	Raspi4b
Gpio_pl061	-		-	-
Semaphore	-		-	-

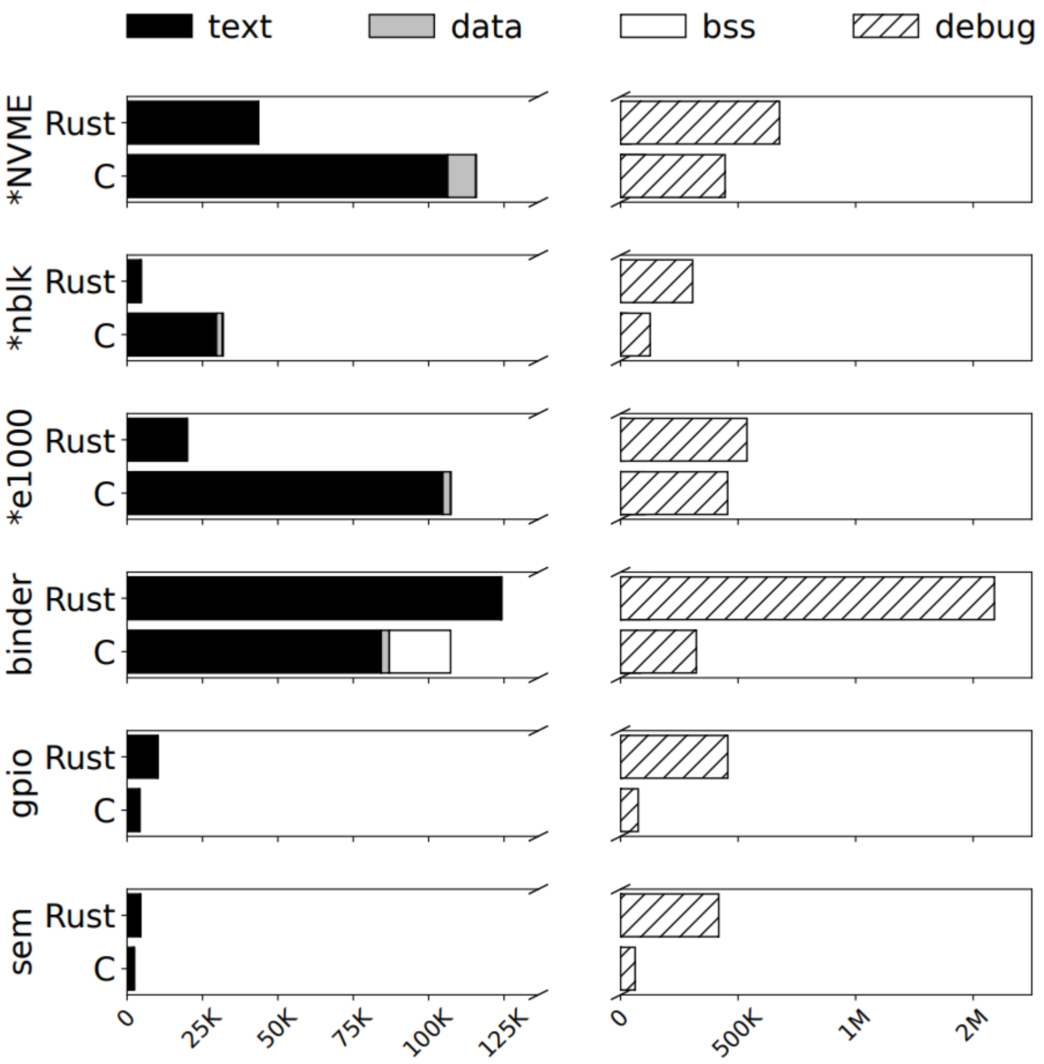
Q2: Does Rust bring additional overhead? (RQ2)

1. Setup
2. Binary size overhead
   - ➢ **1.2×** for binder, **2.4×** for gpio, and **1.9×** for sem (* means full feature in Rust)



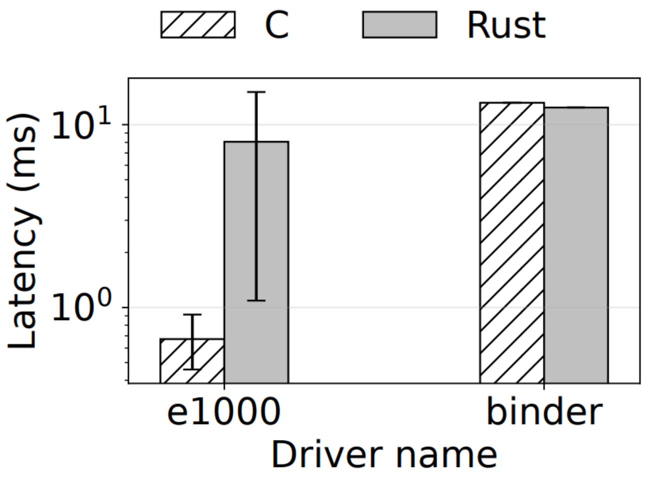
Q2: Does Rust bring additional overhead? (RQ2)

1. Setup
2. Binary size overhead
   - 1.2× for binder, 2.4× for gpio, and 1.9× for sem (* means full feature in Rust)
   - Rust brings overhead especially in the *debug* segmentation: **3.9×–6.6× larger**



Q2: Does Rust bring additional overhead? (RQ2)

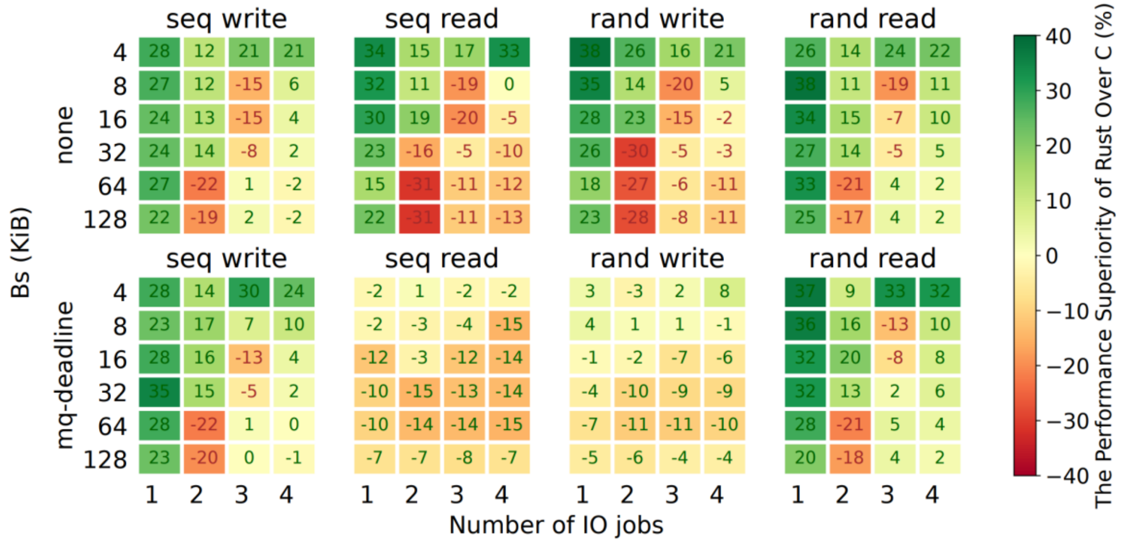
1. Setup
2. Binary size overhead
3. Runtime overhead
   - Latency
     - Rust e1000 driver lacks features, e.g., prefetch

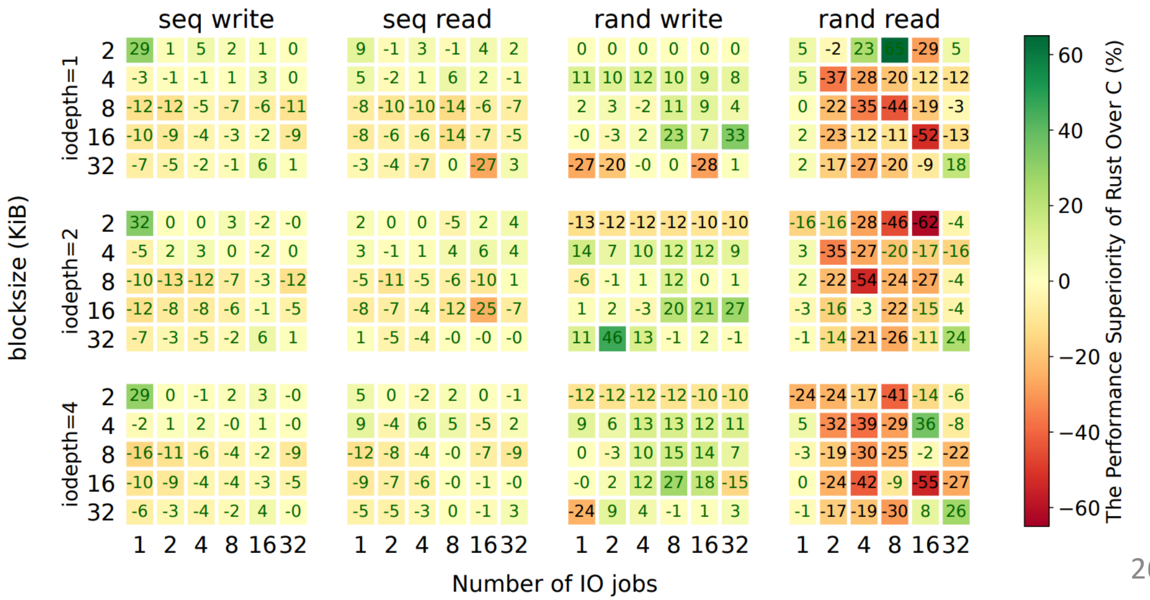


Q2: Does Rust bring additional overhead? (RQ2)

1. Setup
2. Binary size overhead
3. Runtime overhead
   - Latency
   - Throughput (poor)
     - Rust has higher cache miss rate due to smart pointer

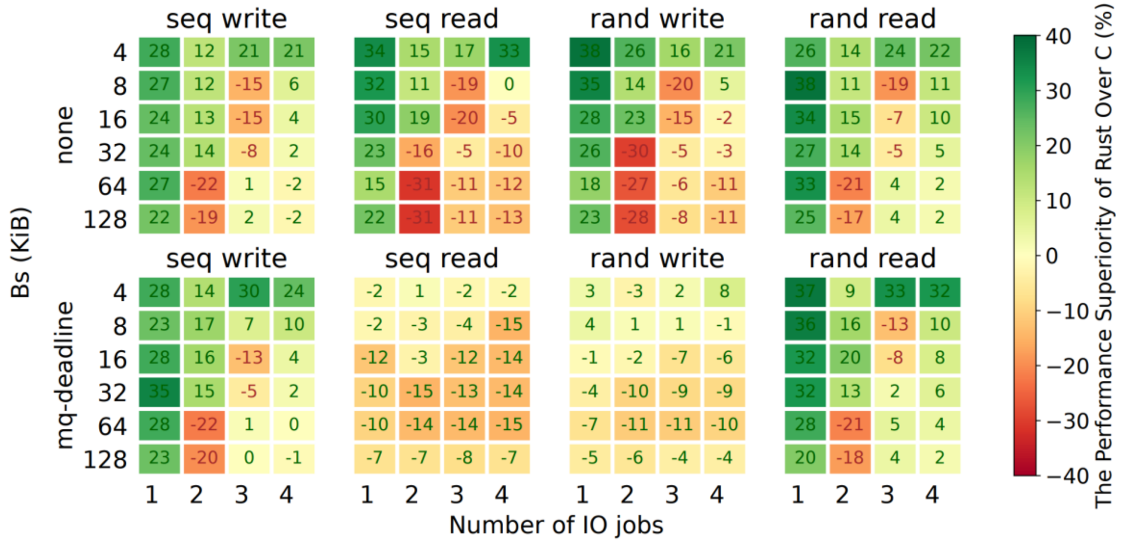
Rust:	78,280,692	L1-icache-load-misses
	2,240,713	dTLB-load-misses
C:	52,976,908	L1-icache-load-misses
	1,312,452	dTLB-load-misses

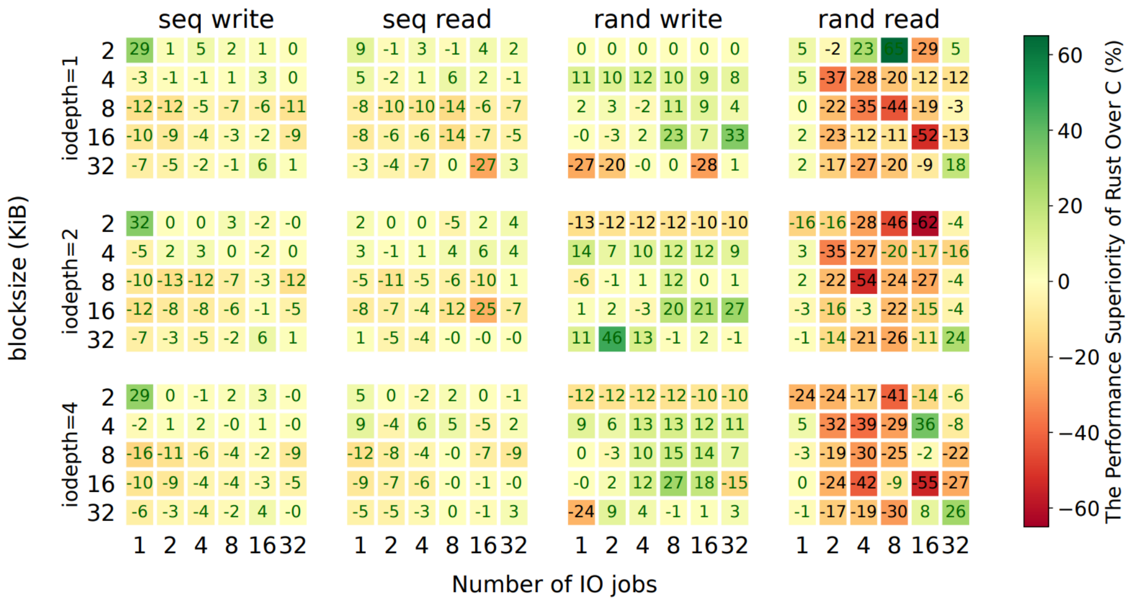




Q2: Does Rust bring additional overhead? (RQ2)

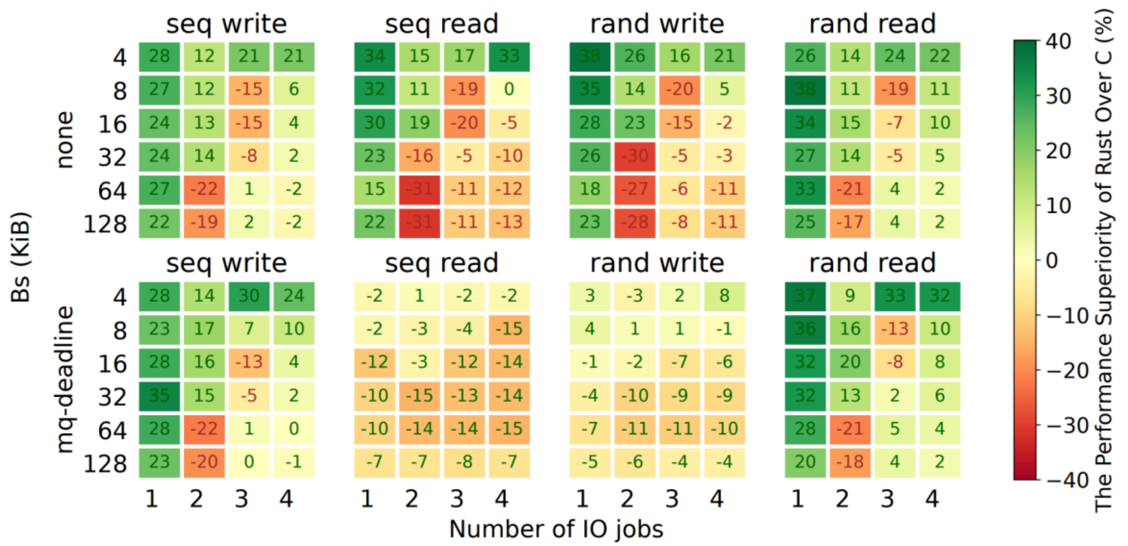
1. Setup
2. Binary size overhead
3. Runtime overhead
   - ➢ Latency
   - ➢ Throughput (poor)
     - ➢ Rust has higher cache miss rate due to smart pointer
     - ➢ Rust runtime checks/bit field translation

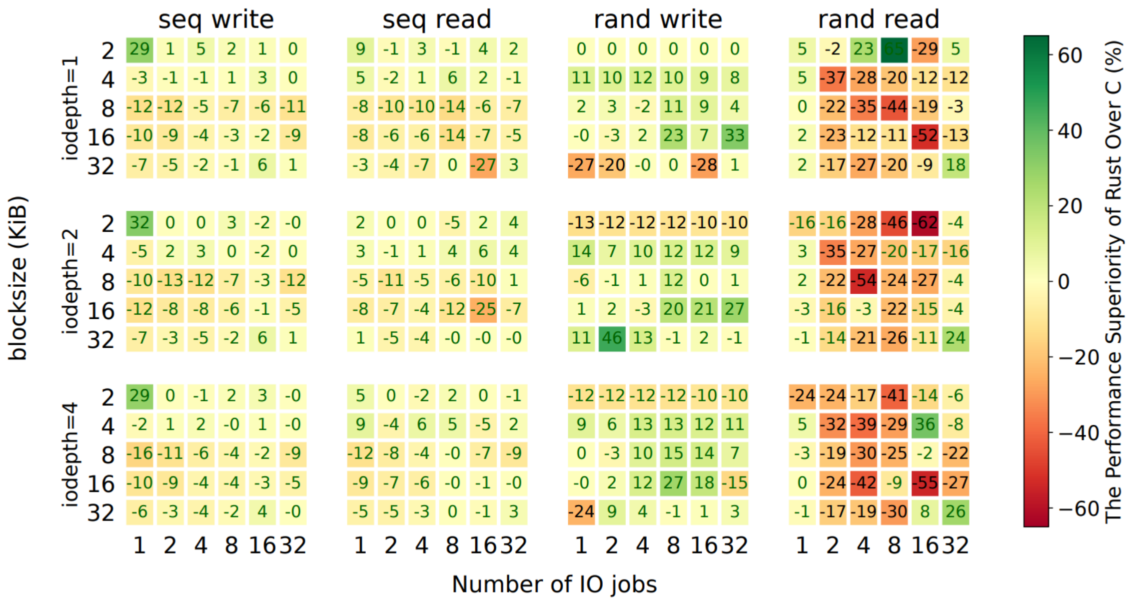




Q2: Does Rust bring additional overhead? (RQ2)

1. Setup
2. Binary size overhead
3. Runtime overhead
   - Latency
   - Throughput (better)
     - Rust use less cache lines (*pahole*)
     - Less code path





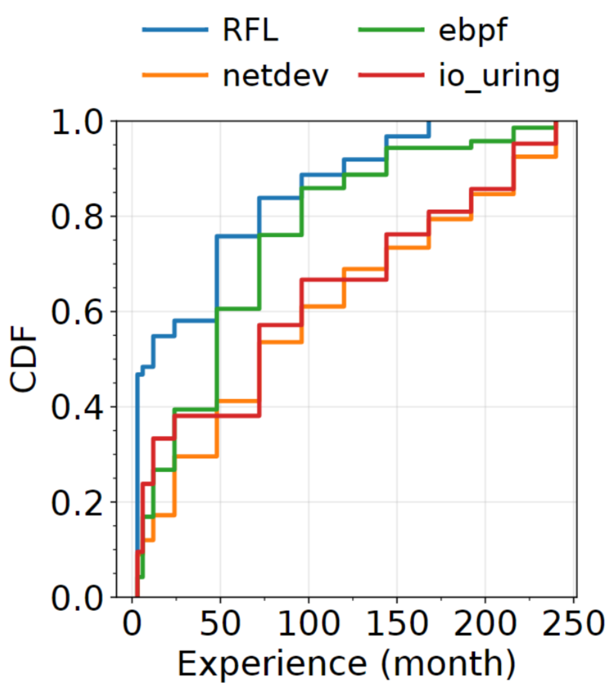
Q2: Does Rust bring additional overhead? (RQ2)

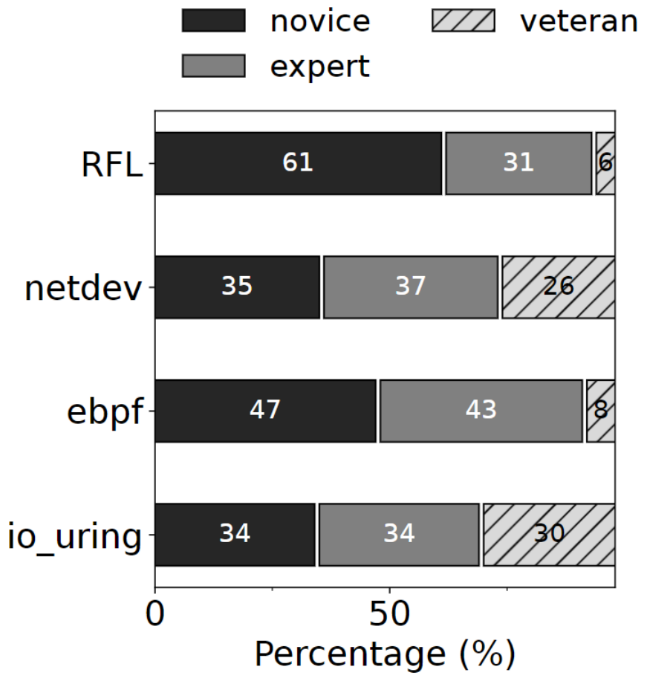
1. Improved code quality and readability
   - RFL improves the Linux documentation coverage by *rustdoc*
   - RFL has the built-in CI system which improves the code quality

Subsystems	Docs%	CI errors/10K LoC
RFL	100%	3.8
ebpf	15%	7.5
io_uring	31%	11.9

Q2: Does Rust bring additional overhead? (RQ2)

1. Improved code quality and readability
2. More young blood to the Linux community
   - RFL has the most novice developers
   - We observe 5 out 6 RFL drivers are developed by the non-novice





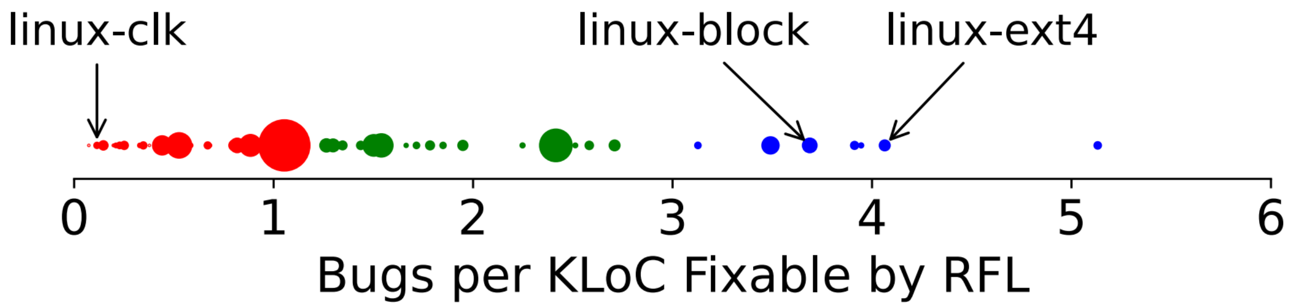
RQ3: what are the lessons learned from RFL?

For the developers using and building RFL

1. use Rustbelt/miri to evaluate the correctness of safety abstraction
2. write your program with ownership in mind
3. accept *unsafe* if you have to

For the developers expanding RFL scope

1. choose the subsystem/drivers that are more fragile



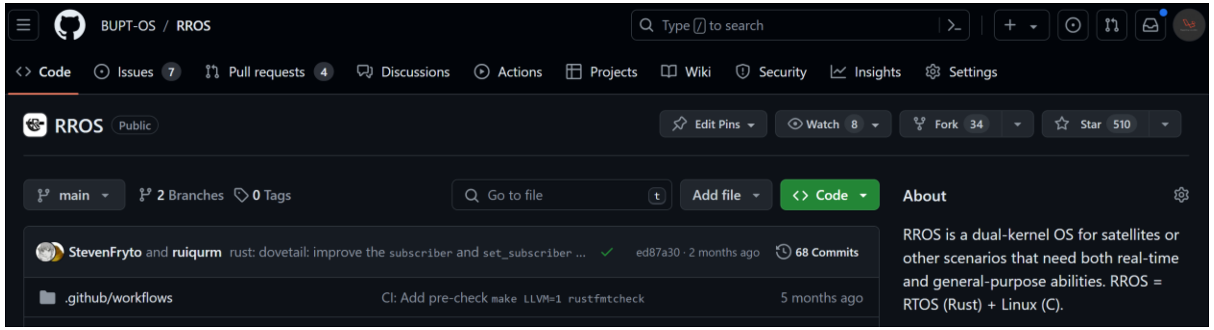
Takeaways

1. Memory safety ***with the price***
2. **No sliver bullet and no guarantee**
3. **Nearly** zero-cost

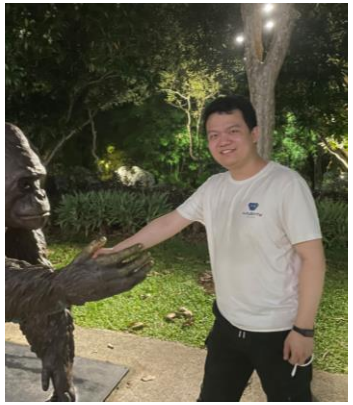
Shout out to the RROS team and my advisors!

RROS

RROS is a Rust dual kernel

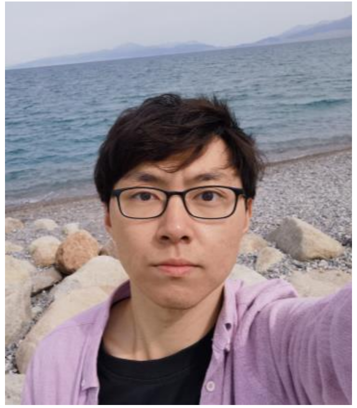


Hongyu Li



Liwei Guo

Yangye Xuan



Shangguang wang

Mengwei Xu

Thanks
Q&A

Source code: https://github.com/Richardhongyu/rfl_empirical_tools

RROS: https://github.com/BUPT-OS/RROS/

Email: lihongyu1999@bupt.edu.cn, lwg@uestc.edu.cn