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

Hongyu Li, *Beijing University of Posts and Telecommunications;*
Liwei Guo, *University of Electronic Science and Technology of China;*
Yexuan Yang, Shangguang Wang, and Mengwei Xu, *Beijing University of Posts and Telecommunications*

https://www.usenix.org/conference/atc24/presentation/li-hongyu

Motivation

- Rust-for-Linux (RFL)
  - We know it exists
  - We know Rust is memory safe

- However,
  - What is the status quo of RFL?
  - Does RFL live up to the hype?
  - What are the lessons learned from RFL?

Summary

- RFL is rarely studied
  - Does it solve the kernel dilemma of safety vs performance?
- First empirical study on RFL
- Collect and analyze 6 key RFL drivers
  - Hundreds of issues and pull requests
  - Thousands of commits and mail exchange (Linux mailing list)
  - 12K discussion on Zulip (online forum)

Rust Safety Model

- Ownership and Lifetime
  - Each memory location have a single owner
  - Each owner has its scope as its lifetime
- Move and Borrow
- The “unsafe” keyword

How do you implant Rust into Linux?

- **Preprocess kernel APIs we need**
- ***rust-bindgen* generates Rust API from kernel API**
  - Rust API is unsafe (as it maps to kernel address space; unchecked by Rust)
- **RFL wraps it with a safe abstraction layer**
  - It is proven by properly wrapping unsafe code under safe APIs it is possible for the whole program to still enjoy the safety guarantee of Rust.

Rust for Linux

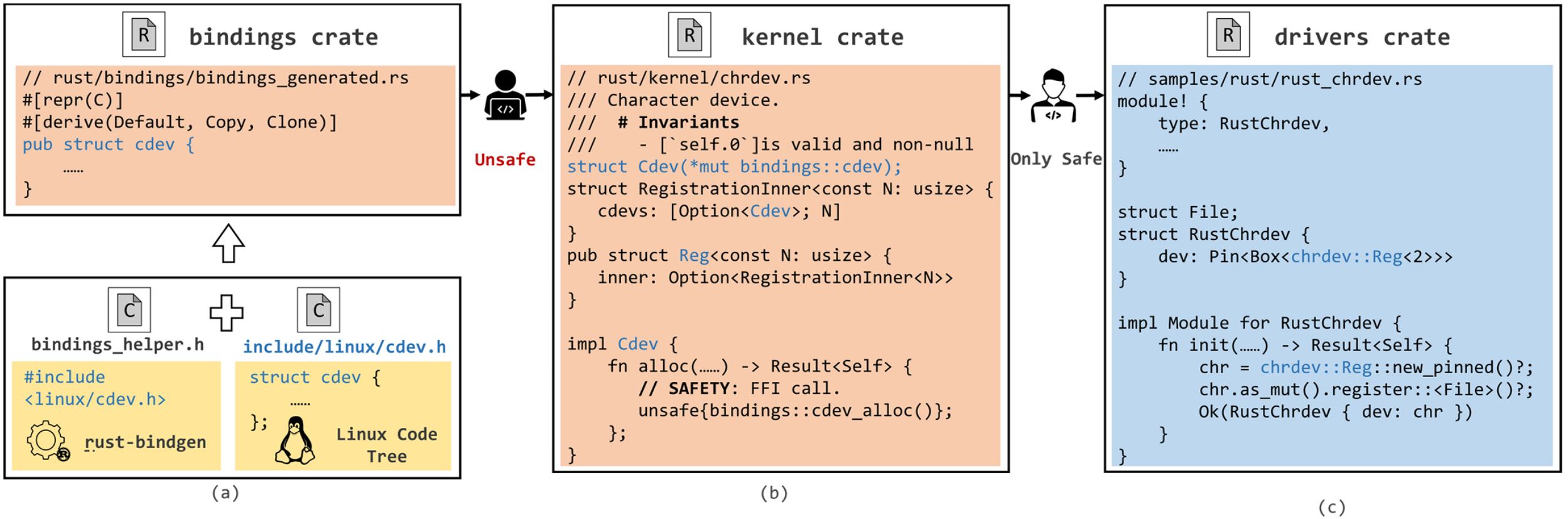

Figure 2: The architecture of Rust-for-Linux. (a): Rust-bindgen generates FFI bindings of kernel data structures and interfaces. (b): The developer constructs safe abstractions (i.e. the kernel crate) by wrapping around the unsafe FFI bindings. (c): Drivers (i.e. the drivers crate) invoke *RFL* safe abstractions to enjoy zero-overhead safety.

Status quo of RFL

- Methodology
  - Collect PR/commits in GitHub
  - Patches on Linux mailing list

  - Categorize RFL code into three categories
    - Pending: 500+ commits (186K LoC)
    - Staged: 1300+ commits (112K LoC)
    - Merged: 160+ commits (19K LoC)

Development Progress

- In terms of LoC, merged code (7.1%) constitutes of 0.125% of kernel code
- Insight 1: “Driver, file, netdev, and filesystems are the long tail of RFL code”
  - These systems account for most kernel code (78% in Linux v6.2)

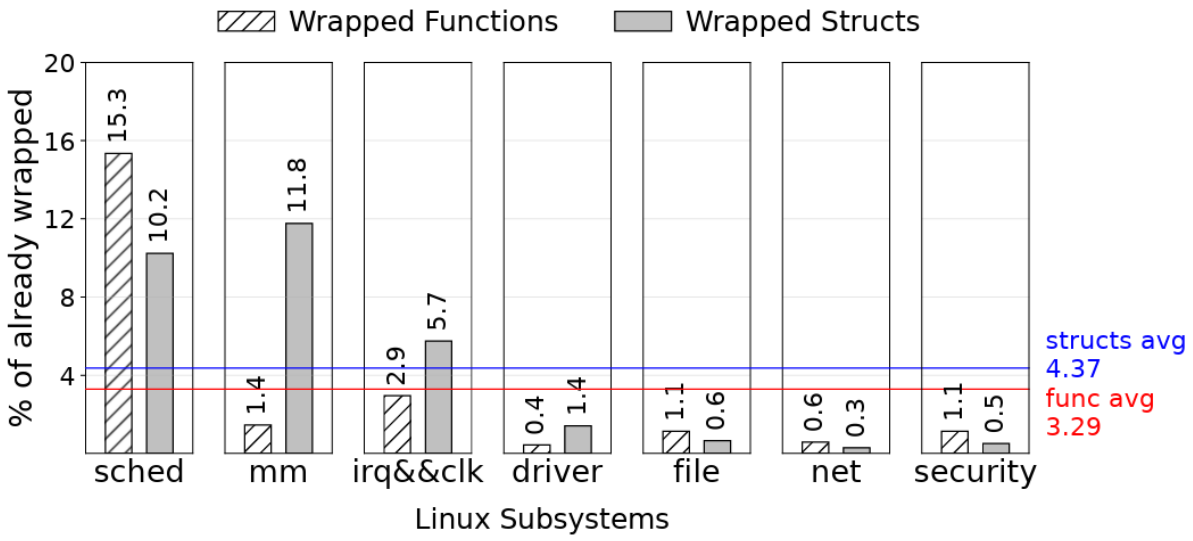

Figure 3: The progress of wrapping APIs.

Patch Distribution

- Insight 2: “RFL infrastructure has matured, with safe abstraction and drivers being the next focus”
  - Foundation of RFL has been laid (Kbuild’s recession)

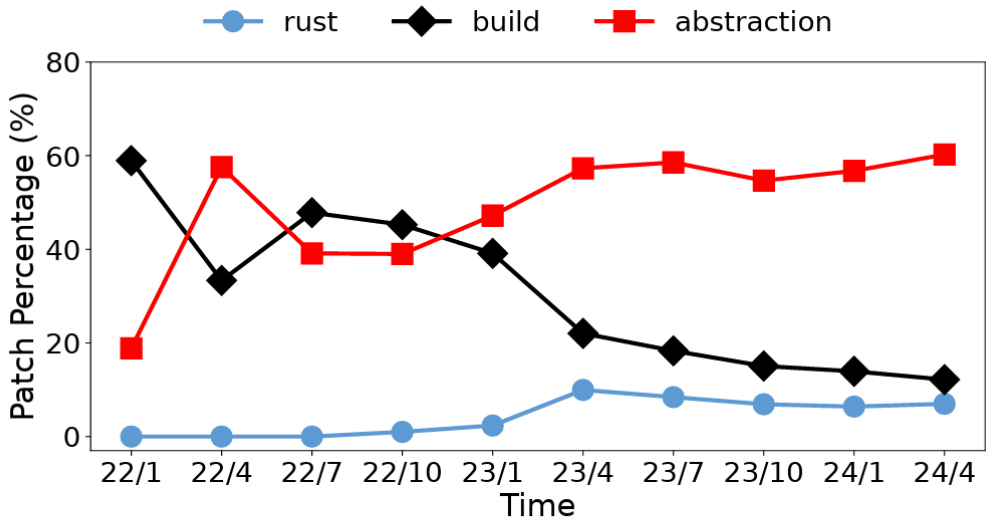

Figure 4: *RFL* patch distribution over time. *Rust*, *Kbuild*, *abstraction* are patches for modifying Rust compiler, constructing KBuild system, and the safe abstraction, respectively.

Trend

- Insight 3: "RFL is bottlenecked by code review but not by code development"
  - Lack of qualified reviewers who must be familiar with both Rust and kernel programming
  - Mismatch of collaboration conventions between the RFL and Linux subsystem communities
  - Deadlock of RFL development
    - Subsystem community unwilling to review abstractions without real Rust drivers
    - Without Abstractions RFL community is unable to construct drivers in Rust

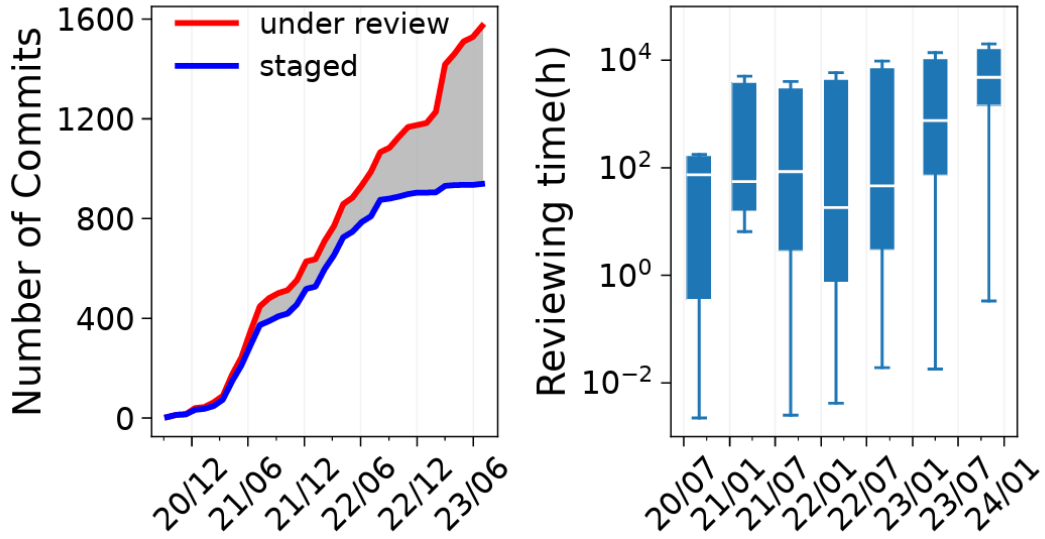

Figure 5: The trend of RFL commits and reviews over time.

Rustify Linux with safe abstractions

- Kernel Programming Paradigm
  - Extensive use of typecasting, pointer arithmetic, bit operation
- Converting kernel data structures
  - RFL leverage bindgen (rule-based) to generate Rust bindings

Table 2: The translation rules from C to Rust in the rust-bindgen.

Type	C	Rust
Primitive types	foo	core::ffi:c_foo
Typed pointers	foo *	*mut foo
Attributes	aligned	#repr(c) (with caveats [7, 9])
	unused	ignored
	weak	ignored
	randomize _layout	ignored
Function pointer	fn	option<fn>

Table 2: The translation rules from C to Rust in the rust-bindgen.

Type	C	Rust
F		c_foo
T		foo
		(c)
		ts [7, 9])
		ed
		ed
	randomize _layout	ignored
Function pointer	fn	option<fn>

Not every C type translates into a corresponding Rust primitive

Insight 4: "Kernel's initiative to control memory in fine granularity conflicts Rust philosophy, which incurs overhead for RFL"

Binding kernel data to Rust

- Generated bindings have identical data layout as their C counterparts
  - Bindings involve numerous raw pointers (unsafe to use)

- RFL uses **helper types** to manage kernel data

Helper Type

- Type and Deref coercion
  - E.g,. For void* pointers RFL implement deref traits that coerces the dereference to result in a correct type

- Automate life cycle management
  - Implement three new low-level types to manage kernel structs (ScopeGuard, ARef, opaque)

  - These types execute custom stub functions upon entering/exiting specific scopes
    - ScopeGuard frees allocated resources of a Task by executing its drop traits when the Task's life cycle ends

Helper Type

- Type and Deref coercion
  - E.g,.
    - s the dereference to
    - resu
- Autom
  - Imp
    - ts (ScopeGuard, ARef,
    - opa
  - Thes
    - ting specific scopes
      - ScopeGuard frees allocated resources of a Task by executing its drop traits when the Task's life cycle ends

Insight 5: "RFL uses helper types to delegate *management* of kernel data to Rust while leaving the *operation* to kernel itself"

Rustify device drivers

- Ownership
  - Unlike C, developer must annotate the device data with ownership
    - How the data might be used by what entity?
    - E.g., *Arc* if it might be shared among threads, *Pin* if data should be unmovable
    - *Pin<Box<SpinLock<Boxx<Ring<RxDesc>>>>>*
- Implementation
  - E.g., Unlike C, Rust requires multiple extra layers to implement dynamically-sized arrays (code bloat)

Rustify

- Ownersh
  - Unlike
    - How
    - E.g.
    - *Pin*
- Impleme
  - E.g., U dynam

wnership

e unmovable

nent

```
// 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 }
```

Figure 6: An example showing the inflexibility of writing dynamically-sized arrays in RFL drivers.

Rustify

- Ownersh
  - Unl
    -
    -
    -
- Imple
  - E.g., U
    dynam

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

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

Insight 6: "The major difficulty of writing safe drivers in Rust is to reconcile the inflexibility of Rust versus kernel programming conventions, which is often an oversight by RFL and the Linux community from what we observe"

wnership

e unmovable

nent

Figure 6: An example showing the inflexibility of writing dynamically-sized arrays in RFL drivers.

RFL makes Linux more “securable”

- Compared to C, RFL greatly reduces the attack space
- It is hard, if not impossible to eliminate unsafe blocks
  - E.g., Kernel exploits inline assembly for managing TLB and issuing memory barrier
  - Ownership sometimes introduce twisted implementation (causing long review cycle)
    - Community compromise to using unsafe
- Bugs do not disappear, only hide deeper
  - Semantic bugs caused by subtle difference in Rust and kernel memory allocation methods. (These bugs pass all compiler check)

RFL makes Linux more "securable"

- Compared to C, RFL greatly reduces the attack space
- It is ha ocks
  - E.g., nd issuing memory barr
  - Ow n (causing long revi
    -
- Bugs do not disappear, only hide deeper
  - Semantic bugs caused by subtle difference in Rust and kernel memory allocation methods. (These bugs pass all compiler check)

Insight 7: "with RFL, Linux becomes more "securable" but still cannot be fully *secure*"

Overhead of RFL

- Methodology
  - Collect 4 drivers with serious use cases (NVME, Binder, Null Block, E1000)
  - + 2 toy drivers (gpio and sem)

  - Only 2 toy drivers + E1000 implement full features

Table 5: The benchmarks and metrics used to test Rust/C drivers. The PC configuration: Intel i54590 with 4 cores, q87 mainboard plus PCIE*16 to m.2, 32 GB DDR3, Samsung SSD 850 Evo, WD SN770, intel 82545 NIC.

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	-		-	-

Binary Size

- Rust drivers that fully implement all features are significantly larger
  - Rust generates extra code to support its unique features
    - Generic programming
    - Boundary checks
    - Lifecycle Management
    - …
  - Binder driver introduce less overhead due to the frequent use of function pointers with unsafe keywords

Binary S

- Rust drivers … significantly larger
  - Rust gene … es
    - Generic …
    - Bounda …
    - Lifecycle …
    - …
  - Binder dri … ent use of function p …

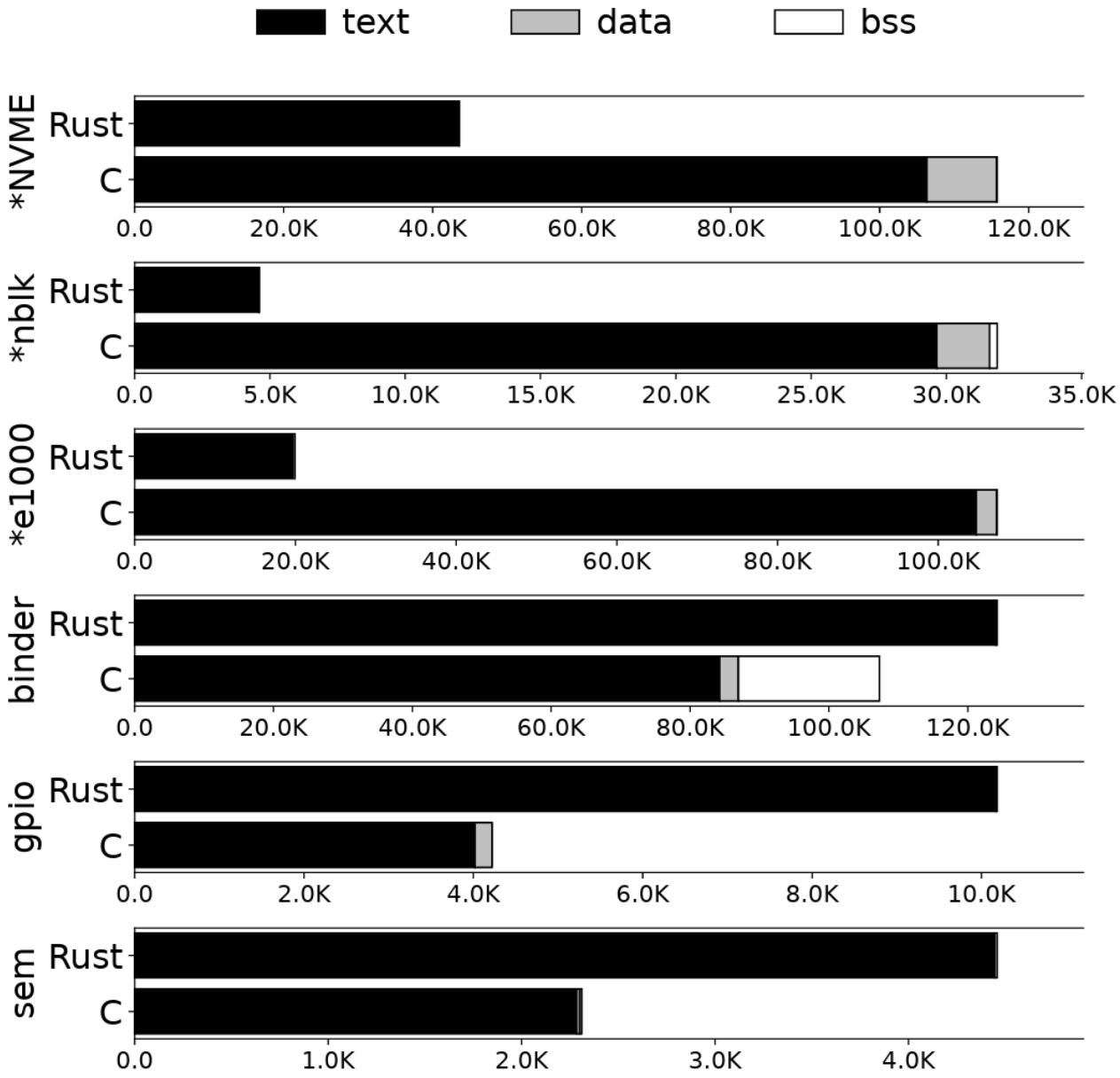

Figure 7: Comparison of Rust and C driver size. * indicates that the Rust driver has not implemented full features as C.

Performance

- Mostly, Rust driver show on par performance

- Wen Rust performs poorly
  - Rust ensure thread safety but does performance depend on the developer
  - Rust performs poorly in memory-intensive workloads (runtime checks in array access)
  - Rust use emulate bit fields as array access (which adds runtime checks)
  - Rust massively use pointers to share ownership (high cache/TLB/branch miss rates)

- When Rust performs better
  - Smart pointers reduce the size of structs
  - Rust does not implement full features; thus some code paths may be omitted

Performance

- Mostly, Rust driver show on par performance

Insight 8: "There is no free lunch for performance – it is the programmer that counts!"

- Wen R
  - Rust
  - Rust
    acce
  - Rust
  - Rust
- When Rust performs better
  - Smart pointers reduce the size of structs
  - Rust does not implement full features; thus some code paths may be omitted

he developer

e checks in array

he checks)

LB/branch miss rates)

Final Thoughts

- Paper raises various questions regarding integration of Rust into Linux (written mainly in C)
- Lots of detail on problems Rust developers face in kernel development
- Paper is organized in RQs but the insights were more interesting (3 RQs but 8 insights; slightly odd structure?)
- Worth taking a look if your interested in the interaction between C and Rust