KTH ROYAL INSTITUTE
OF TECHNOLOGY

Safe Kernel Programming with Rust

Johannes Lundberg

July 19, 2018

Kernel programming (in C)

Consequences of software bugs

The Northeast blackout of 2003

A race condition in the software caused a several days long electric blackout in Northeast USA.[1]

Therac 25

A race condition in the software of a radiation therapy machine caused the machine to give massive overdoses of radiation. The bug caused several deaths.[1].

[1] Source: Wikipedia

Three common bugs in C

Data races
Failure to use proper locking mechanism.

Double free
Freeing memory that has already been freed.

Use after free
Access memory that has been freed.

Userland vs Kernel

For userland programming there exist many high level languages like C#, Java, etc, that abstract away manual memory management.

Userland vs Kernel

For userland programming there exist many high level languages like C#, Java, etc, that abstract away manual memory management.

However, this is not the case for the kernel, which is often written in C.

Userland vs Kernel

For userland programming there exist many high level languages like C#, Java, etc, that abstract away manual memory management.

However, this is not the case for the kernel, which is often written in C.

Most things outside the kernel depends on the kernel so it does not solve the underlying problem.

How to make kernel programming safe?

OKE, Open Kernel Environment

Safety is guaranteed by trust management, language customization and a trusted compiler.

How to make kernel programming safe?

OKE, Open Kernel Environment

Safety is guaranteed by trust management, language customization and a trusted compiler.

TAL, Typed Assembly Language

TAL is an assembly language which ensures memory safety and control flow safety of machine code through a type system.

How to make kernel programming safe?

OKE, Open Kernel Environment

Safety is guaranteed by trust management, language customization and a trusted compiler.

TAL, Typed Assembly Language

TAL is an assembly language which ensures memory safety and control flow safety of machine code through a type system.

P-Bus

Programming interface layer for safe OS kernel extensions. New extensions are verified with a model checker to ensure safety of code run in the kernel.

Rust

- Rust 1.0 released in 2015.

Rust

- Rust 1.0 released in 2015.
- Strongly typed, compiled language.

Rust

- Rust 1.0 released in 2015.
- Strongly typed, compiled language.
- Claims same performance as C/C++ but without the bugs.

Rust

- Rust 1.0 released in 2015.
- Strongly typed, compiled language.
- Claims same performance as C/C++ but without the bugs.
- No garbage collector means it is possible to use in the kernel.

Rust

- Rust 1.0 released in 2015.
- Strongly typed, compiled language.
- Claims same performance as C/C++ but without the bugs.
- No garbage collector means it is possible to use in the kernel.
- Little or no runtime overhead.

Rust eliminates bugs

Rust eliminates bugs

~~Data races~~

Only unique mutable references. Mutexes protect the data, not the code. `Mutex<MyStruct>`

Rust eliminates bugs

~~Data races~~

Only unique mutable references. Mutexes protect the data, not the code. `Mutex<MyStruct>`

~~Double free~~

Memory is released automatically when an owning variable goes out of scope (because data can only have one owning variable).

Rust eliminates bugs

~~**Data races**~~

Only unique mutable references. Mutexes protect the data, not the code. `Mutex<MyStruct>`

~~**Double free**~~

Memory is released automatically when an owning variable goes out of scope (because data can only have one owning variable).

~~**Use after free**~~

References are bound by lifetimes. A reference can't outlive the origin data.

Rust safe and unsafe code

Safe code

Ownership model. Safe code is guaranteed to be free of **data races**, **dangling and null pointers** and **segfaults** through static analysis.

Unsafe code

Unsafe code unlocks the power of raw pointers and the ability to call unsafe functions. Unsafe code is marked with `unsafe {...}` block.

Goals

Safeness

How much unsafe code is required to write a kernel device driver? Can we do it completely in safe code?

Goals

Safeness

How much unsafe code is required to write a kernel device driver? Can we do it completely in safe code?

Performance

Boundary checks and similar safety additions mean slower code. How will the Rust driver perform compared to the C implementation?

Methodology

- Setup cross compile environment.

Methodology

- Setup cross compile environment.
- Create a Rust kernel programming interface - RustKPI.

Methodology

- Setup cross compile environment.
- Create a Rust kernel programming interface - RustKPI.
- Port a network device driver to Rust.

Methodology

- Setup cross compile environment.
- Create a Rust kernel programming interface - RustKPI.
- Port a network device driver to Rust.
- Analyze unsafe code and compare performance to C implementation.

Methodology

Analyze safeness

- Break down unsafe code into categories.
- Look at safeness of each unsafe block. Some “unsafe” code isn’t necessarily unsafe. Compiler is overly conservative.

Methodology

Analyze performance

- Benchmark network traffic over local network for Rust and C implementations (network bound).
- Benchmark network traffic between host and virtual machine on the same machine for Rust and C implementations (CPU bound).
- Analyze CPU usage on the most interesting case of the above.

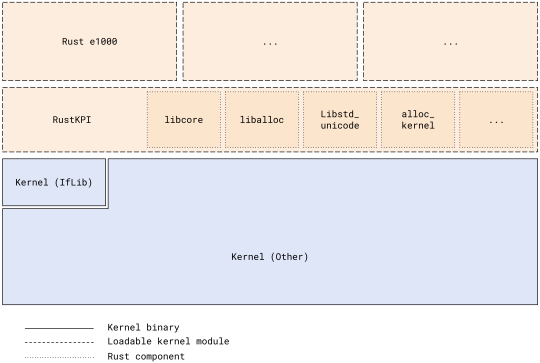
Methodology

Limitations

- Assume that the compiler's guarantees are true.
- Assume that the kernel is bug free and is giving us valid pointers.

Methodology

Kernel diagram



Evaluation

Safeness

- 9897 lines of *driver* code (excluding comments, blank lines and bindgen generated bindings).
- 430 lines of unsafe code that contain:
  - 56 calls to C functions.
  - 43 calls to unsafe Rust functions.
  - 73 access to unions.
  - 37 dereferencing raw pointers.

Evaluation

Safeness

- 56 calls to C functions.
- 43 calls to unsafe Rust functions.
- 73 access to unions.
- 37 dereferencing raw pointers.

Evaluation

Safeness

- ~~56~~ 0 calls to C functions.
- 43 calls to unsafe Rust functions.
- 73 access to unions.
- 37 dereferencing raw pointers.

Evaluation

Safeness

- ~~56~~ 0 calls to C functions.
- ~~43~~ 0 calls to unsafe Rust functions.
- 73 access to unions.
- 37 dereferencing raw pointers.

Evaluation

Safeness

- ~~56~~ 0 calls to C functions.
- ~~43~~ 0 calls to unsafe Rust functions.
- ~~73~~ 0 access to unions.
- 37 dereferencing raw pointers.

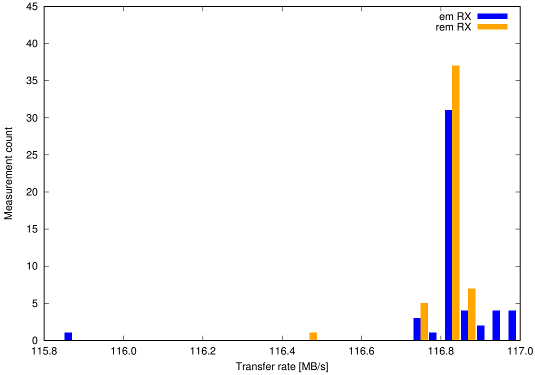
Evaluation

Safeness

- ~~56~~ 0 calls to C functions.
- ~~43~~ 0 calls to unsafe Rust functions.
- ~~73~~ 0 access to unions.
- **37 dereferencing raw pointers.**

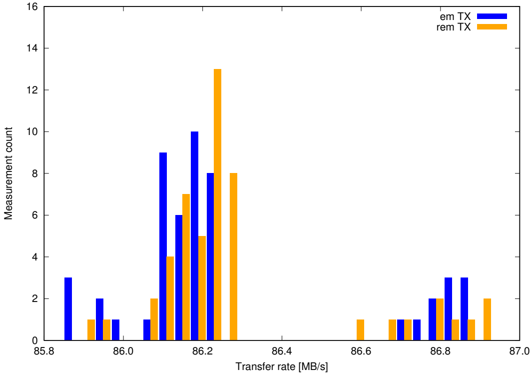
Evaluation

Receive traffic on hardware



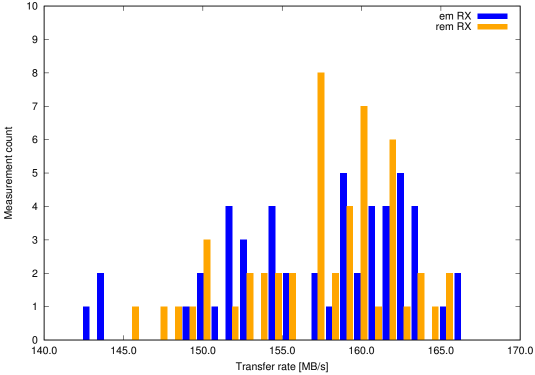
Evaluation

Transmit traffic on hardware



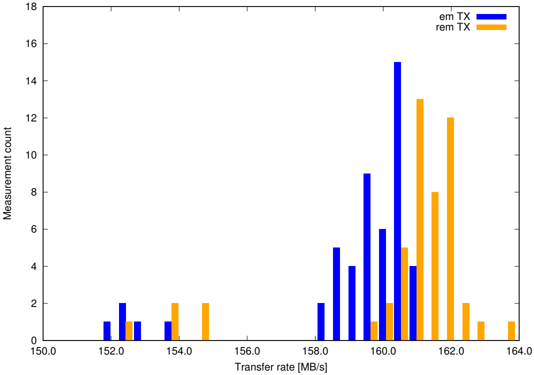
Evaluation

Receive traffic in Bhyve virtual machine



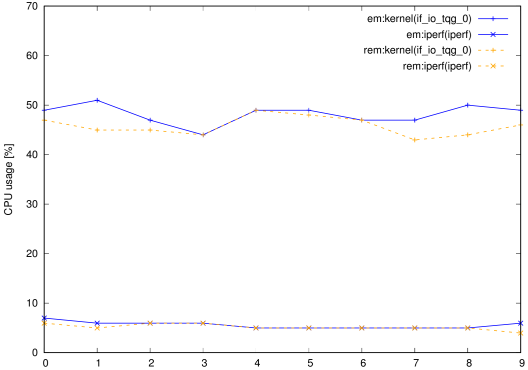
Evaluation

Transmit traffic in Bhyve virtual machine



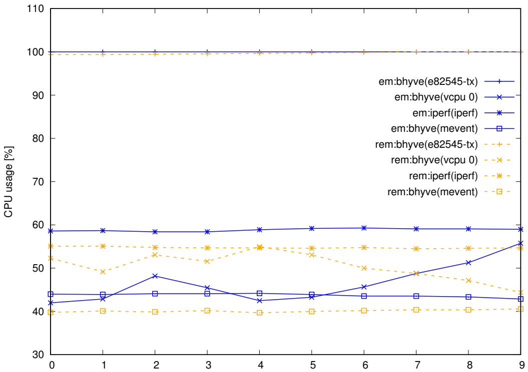
Evaluation

CPU threads in Bhyve virtual machine



Evaluation

CPU threads in Bhyve host



Conclusion

Safeness

It was shown that with little work we can eliminate most unsafe code. With some more effort on a safe interface, a device driver can be completely written in safe code.

Conclusion

Safeness

It was shown that with little work we can eliminate most unsafe code. With some more effort on a safe interface, a device driver can be completely written in safe code.

Limitation

We need to trust the Rust compiler and our safe interface. However, the amount of code that can “behave bad” is limited to the unsafe parts of our interface.

Conclusion

Performance

There was no negative impact on performance for Rust code.

Conclusion

Performance

There was no negative impact on performance for Rust code.

Limitations

IfLib is doing a lot of the heavy lifting. Porting another driver or IfLib itself might reveal some performance penalty in Rust.

Future work

Tracing

DTrace and HWPMC were not able to trace any Rust functions. Make this work would enable more interesting profiling.

Future work

Tracing

DTrace and HWPMC were not able to trace any Rust functions. Make this work would enable more interesting profiling.

IfLib

Thanks to IfLib the driver itself does not interface with that much kernel functions. Porting IfLib itself to Rust would be more challenging and put Rust to the test as a kernel language.

Thank you for listening