A Rust-based Alternative for BPF

Bachelor Thesis Colloquium

30.08.2024

Ben Kowol

RUB - ITS

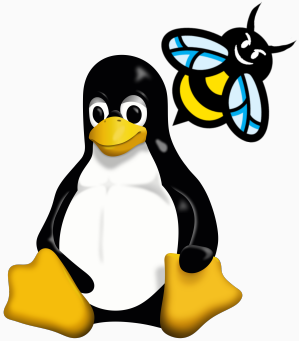
Motivation

What Makes eBPF so Important?

- Fast and safe kernel extensions
- Important for very time sensitive tasks
  - Profiling
  - Network packet rerouting
  - Security
  - ...

[?]

Why Might an Alternative to eBPF be Necessary?

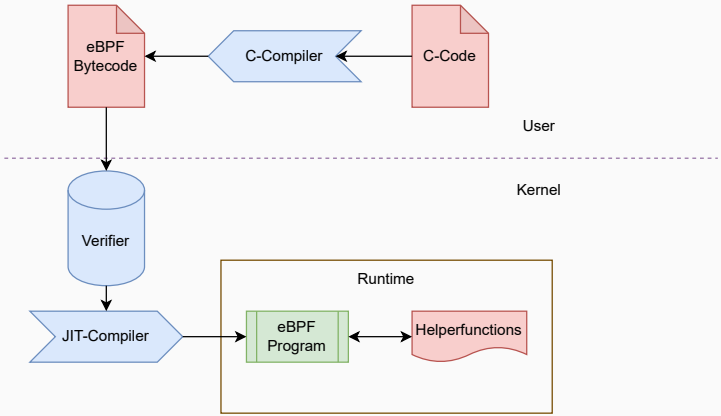


- Very complex
- Restrictive
- Safety and security concerns

Structure

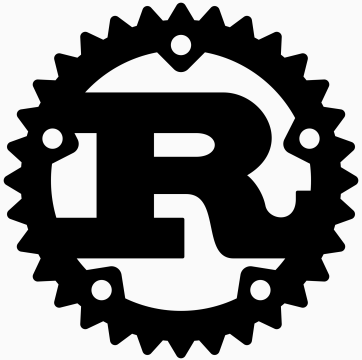
Necessary Background on Important Technologies

High-Level View of eBPF



Rust

- Growing kernel support
- Only other language besides C and ASM used for Linux kernel
- Memory and type safe [?, ?, ?]
  - Resource Acquisition Is Initialization (RAII)
  - Ownership and borrowing
- Similar performance to C [?, ?, ?, ?]



[?]

Parallels Betweeen eBPF and Rust

Feature	eBPF	Rust
Calling other programs	✓	✗
Dead code elimination	(✓) only privileged programs	✓
Bounded loops	(✓) limited in complexity	✗
No deadlock	✓	✗
Memory Safety	✓	✓
Stack Safety	✓	(✓) with compiler pass
Stack Security	✓	✓

In-depth View of the Proposed Alternative

How Safety is Achieved

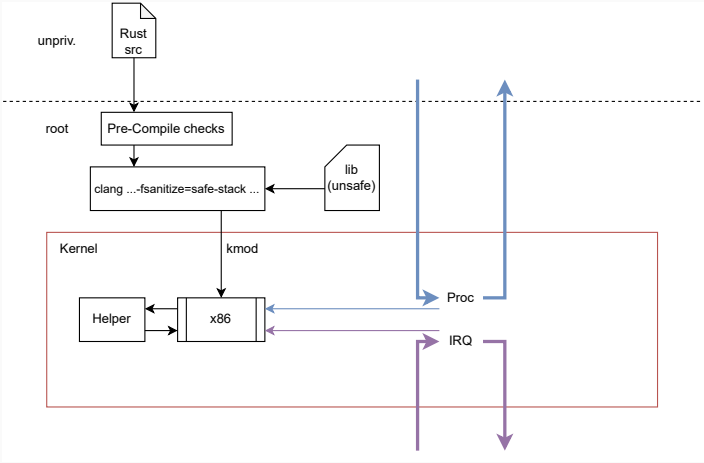
- Memory/type safety $\rightarrow$ Rust language features
- Stack security (e.g. Stack buffer overflow)$\rightarrow$ compiler passes [**?**, **?**]
- Bounded stack size $\rightarrow$ compiler [**?**], 3rd party [**?**], guard page
- Termination $\rightarrow$ pre-emption timer
- Kernel locks $\rightarrow$ exclude from provided library
- Less helper functions $\rightarrow$ more expressive

Checks Differ for IRQ-Context

- No pre-emption possible
- Termination
  - Bounded execution time needs to be assured
    - → Check loops and recursion [**?**, **?**]
- Blocking calls
  - Analyze call graph [**?**]
  - Exclude functions with blocking calls from provided library

Implementation of Important Structural Parts

High-Level View of the Proposed Structure



Tool for Compiling and Loading of a Module

- Conceptual implementation
- Provides basic pipeline for
  - Checking
  - Compiling
  - Loading

Rust Macro for Loop Checking

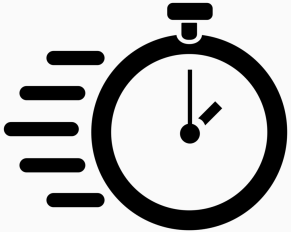
- Checks for *for*- and *while*-loops
- Can recognize *for*-loops with a constant range
  - for <variable> in <number>..<number> {...}
  - Constant range might not be constant runtime
- Can distinguish between *for*-loops and trait implementation
- missing checks for *loop*-loops (infinite loops)
- Easily extensible
- Takes 0,18s for around 500 lines of Code

Example for Rejected Code

```
fn it_works() {
    ...
    for i in 0..1 { \\static range
        for n in 0..j { \\variable range
            println!("{}",n);
        }
    }
}
```

Estimating the Alternatives Capabilities

Expected Computation Speed



[?]

- WAT or AOT compilation results in a faster program than JIT-compilation
- More efficient code due to less restrictions
- Might be possible to use hardware acceleration [?]

Expected Safety and Security

- Memory, type and stack safety are guaranteed by compiler/language features
- Pre-emption timer have been shown to work
- Memory acquisition and release with Drop trait might cause problems
- Checking for termination through AST analyzing might leave loopholes

Conclusion

Conclusion

- eBPF promises safe kernel extensions
- Does not deliver on these promises
- Rust has memory safety build in
- Most other safety concerns can be adressed easily
- Might be easier to work with and result in better programs

References (1)

[1] Exploit mitigations.

[2] alenco.
Stopwatch.

[3] Jorge Aparicio.
Cargo call stack.

[4] William Bugden and Ayman Alahmar.
Rust: The programming language for safety and performance.
In *2nd International Graduate Studies Congress (IGSCONG'22)*, 2022.

[5] Guillaume Combette and Guillaume Munch-Maccagnoni.
A resource modality for RAII.
In *LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages*, pages 1–4, Oxford, United Kingdom, July 2018.

[6] Manuel Costanzo, Enzo Rucci, Marcelo Naiouf, and Armando De Giusti.
Performance vs programming effort between rust and c on multicore architectures: Case study in n-body.
In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10, 2021.

[7] Rust Foundation.
Rust programming language logo.

[8] Kornel.
Rust-c-speed.

References (2)

[9] Lokathor.
safe arch.

[10] Emad Jacob Maroun, Eva Dengler, Christian Dietrich, Stefan Hepp, Henriette Herzog, Benedikt Huber, Jens Knoop, Daniel Wiltsche-Prokesch, Peter Puschner, Phillip Raffeck, Martin Schoeberl, Simon Schuster, and Peter Wägemann.
The Platin Multi-Target Worst-Case Analysis Tool.
In Thomas Carle, editor, *22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*, volume 121 of *Open Access Series in Informatics (OASIcs)*, pages 2:1–2:14, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[11] Nicholas D. Matsakis and Felix S. Klock.
The rust language.
In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[12] oscar6echo.
Rust-vs-c-vs-go-runtime-speed-comparison.

[13] Maximilian Ott, Phillip Raffeck, Volkmar Sieh, and Wolfgang Schröder-Preikschat.
Towards just-in-time compiling of operating systems.
In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, PLOS '23, page 41–48, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Thomas Sewell, Felix Kam, and Gernot Heiser.
Complete, high-assurance determination of loop bounds and infeasible paths for wcet analysis.
In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, 2016.

References (3)

[15] Vadim Shchekoldin.
ebee.

[16] The Clang Team.
Clang command line argument reference.

[17] The Clang Team.
Safestack.

[18] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu.
Towards understanding the runtime performance of rust.
In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

License

License