RUHR-UNIVERSITÄT BOCHUM

RUB

Ben Kowol

A Rust-based Alternative to BPF

Bachelorarbeit im Fach IT-Sicherheit

4. September 2024
Bochum Operating Systems and System Software Group (BOSS)

RUB Faculty of Computer Science

Ruhr-Universität Bochum (RUB) ·
Fakultät für Informatik
Operating Systems and System Software Group
Universitätsstraße 150 · 44801 Bochum · Germany

A Rust-based Alternative to BPF

Bachelorarbeit im Fach IT-Sicherheit

vorgelegt von

Ben Kowol

geb. am 28. Dezember 2002
in Essen

angefertigt an der

Bochum Operating Systems and
System Software Group (BOSS)

Fakultät für Informatik
Ruhr-Universität Bochum

Betreuer*innen:	**Benedikt Herzog, M.Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Timo Hönig**
Beginn der Arbeit:	**15. Mai 2024**
Abgabe der Arbeit:	**15. August 2024**

Titel meiner Abschlussarbeit / title of the final thesis

A Rust-based Alternative to BPF

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich erkläre mich des Weiteren damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Statutory Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university to obtain an academic degree.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I furthermore agree that the digital version of this thesis will be used to subject the paper to plagiarism examination.

Not this English translation but only the official version in German is legally binding.

Bochum, 4. September 2024

(Ben Kowol)

Abstract

The extended Berkeley Packet Filter (eBPF) is a toolchain used to load extensions into the kernel. It promises both safety and performance. While kernel modules can also act in a comparable way, they lack both the safety checks as well as the ability to support many kernel versions without having to change the program.

But over time, many security and safety concerns where discovered, which has shown that eBPF is far from being as safe as it should be. And while the performance is good, when it comes to the execution speed of its compiled machine-code, eBPF's restrictions make it hard to write efficient programs.

In this thesis, I introduce an alternative to the eBPF based upon safe Rust and its build-in safety features. I present a way to create an eBPF alternative with Rust as a base, which is at least as safe as eBPF while being simpler and more expressive. To this end, I designed a Rust-macro prototype that prohibit the use of variable length loops and conducted theoretical research with a special focus on replacing eBPF programs for Interrupt Request (IRQ) context.

What I found was that creating an eBPF like construct with Rust is possible and simpler than eBPF, since this new construct can rely on already existing tools and language features. Additionally, I made Rust-macros out to be a powerful and simple tool to enforce further restrictions on the code at compile time.

Kurzfassung

Der extended Berkeley Packet Filter (eBPF) ist ein beliebtes Mittel um die Funktionalität des Kernels zu erweitern. Es verspricht dabei sowohl Stabilität und Sicherheit, als auch Performanz. Auch wenn Kernel Module sich sehr ähnlich verhalten, fehlt ihnen doch die Überprüfungen durch den Verifier. Außerdem müssen diese für jede neue Kernel Version angepasst werden.

Mit der Zeit wurden jedoch mehr und mehr Sicherheits- und Stabilitätsprobleme gefunden, was zeigte, dass eBPF doch nicht so sicher ist wie es eigentlich sein sollte. Und auch wenn die Ausführungsgeschwindgkeit des von eBPF kompilierten Maschinen Codes noch immer sehr gut ist, führen die Einschränkungen, denen eBPF-Programme unterliegen, dazu, dass es schwierig ist, effiziente Programme zu schreiben.

In dieser Arbeit stelle ich eine alternative Lösung zu eBPF vor, die sich safe-Rust und die dort eingebauten Sicherheitsfunktionen zu Nutze macht. Ich stelle dabei einen Weg für diese Alternative vor, sodass dieses neue Konstrukt so sicher ist wie eBPF, aber deutlich einfacher und mit weniger Einschränkungen. Dafür habe ich einen Rust-Makro Prototypen entwickelt, der es erlaubt die Verwendung von Schleifen variabler Länge zu unterbinden. Zusätzlich habe ich theoretisch geforscht, wie weitere Funktionen von eBPF ersetzt werden können, mit einem besonderen Fokus auf eBPF-Programme im Interrupt Request (IRQ)-Kontext.

Dabei habe ich herausgefunden, dass es in der Tat möglich ist, ein solches System zu entwickeln. Dieses ist einfacher zu konstrieren als eBPF, da man sich dabei auf bereits existierende Werkzeuge und Eigenschaften von Rust verlassen kann. Zusätzlich konnte ich feststellen, dass die Makros von Rust ein mächtiges Tool sind um ein Programm auf unerlaubte Syntax, wie Schleifen, zu überprüfen.

Contents

Chapter 1

Introduction

The emergence of eBPF has revolutionized the development of kernel extensions. Due to eBPF, it became possible to load code into the kernel safely and with exceptional performance. While eBPF still produces programs with good performance, its safety guarantees have become rather disputed. These safety guarantees are given by an overly complex verifier and Just-In-Time (JIT)-compiler [eBP; Com]. These place heavy restrictions on the eBPF-Program, limiting its expressiveness. To combat this, helper functions have been introduced. But both the complexity of the eBPF-safety-system, verifier and compiler, and the helper functions lead to loopholes in the safety promises [NISa; NISb; NISd; NISe; NISf; NISc]. And even the helper functions do not extend eBPF to such a degree, that complex computations would be efficiently implementable. eBPF works very much like a language build on C that tries heavily to circumvent the many bugs and problems one can create with bad C code. Many of the safety problems stem from the buggy and incomplete verifier. Over time, the verifier had to be extended to account for newly found faults. This way, its size increased dramatically and thus leading to more potential for bugs [Sun+24]. This is a problem, since developers need to trust eBPF to ensure the safety of their program, after all it will be loaded and executed within the kernel space. A crashing program here could lead to the entire kernel crashing or to reveal secret kernel data.

But luckily, for a fiew years Rust has made its way into the kernel development and support is slowly growing. Rust has many of the safety concerns associated with C already covered with its language safety functions [MK14; BA22], all while being close in terms of performance and features. This makes it an ideal language to use when constructing an alternative to eBPF.

1.1 Related Work

Rust has only recently made its way into the Linux kernel, prompting a lot of effort to find new ways to make it even saver. Baranowski et al. extended SMACK [On3] to work as a verifier for Rust [BHR18]. With “Prusti”, Astrauskas et al. argued that Rust’s special memory system simplifies the verification [Ast+22]. In another paper, “Galeed” was introduced, which protects safe Rust from the interference of unsafe program parts that are possibly even written in C [Riv+21]. Ling et al. developed a transpiler to turn C into Rust, while turning as much of the program into safe Rust as possible [Lin+22].

In their paper, Balasubramanian et al. showed the advantages of using Rust in system programming [Bal+17].

Rust was proposed to be used for the safety of kernel extensions in general, with the main focus on type and memory safety and the compatibility problems that might come with them [Mil+19]. Jia et al. try to significantly decrease the usage of helper functions in eBPF utilizing Rust and runtime protections [Jia+23].

In an effort to improve the security of eBPF, a method to defend against malicious eBPF programs from within the hypervisor was proposed [WLC23]. Additional work has been done to improve eBPF's defenses against Spectre [LGH24]. In [Sah24], it is claimed that termination of an eBPF program is not enough, runtimes still matter, and proposes a runtime estimation.

Chapter 2

Background

2.1 Extended Berkeley Packet Filter (eBPF)

eBPF is the successor of the Berkeley Packet Filter (BPF), which, as the name suggests, was intended for efficient packet routing. Nowadays BPF and eBPF are used as synonyms, while BPF is referred to as classic Berkeley Packet Filter (cBPF). eBPF advances from cBPF as being more versatile and for a broader area of use cases. With these new use cases, like performance profiling, came complexity. In the following, eBPF's inner structure and especially the verifier are taken a closer look at. Furthermore, some issues with eBPF are worked out.

2.1.1 Architecture

First, it is necessary to understand the components of the eBPF subsystem. Figure 2.1 provides a high-level overview that can be used to understand eBPF's architecture.

The Kernel expects the eBPF program to be delivered in bytecode. While it is possible to write direct bytecode, it is much more useful to use some sort of abstraction. The easiest being some sort of pseudo-C-Code that can then be compiled to eBPF-bytecode with the `bpf` target for clang as seen in listing 2.1.

Due to its vastly different syntax and restrictions, this C-Code is often referred to as its own language. Projects like bcc [On0] or bpftrace [Rob] can be utilized to aid in the development of eBPF programs, by providing an even higher level of abstraction.

To load this eBPF-Bytecode via a syscall, one needs to either have root privileges or the CAP_BPF capability for unprivileged users. It is important to mention that eBPF-Programs loaded from an unprivileged process are checked with much more restrictive rules, which limit their functionality and kernel access. Afterwards, the verifier checks for safety concerns and the compliance with certain restrictions. eBPF Programs are compiled by a JIT-compiler, thus constant blinding [RBA17] is conducted to protect against vulnerabilities that are unique to JIT-compilers. This aforementioned

```
clang -O2 -target bpf \
-c my_ebpf_program.c -o my_ebpf_program.o
```

Listing 2.1 – Compiling an eBPF program

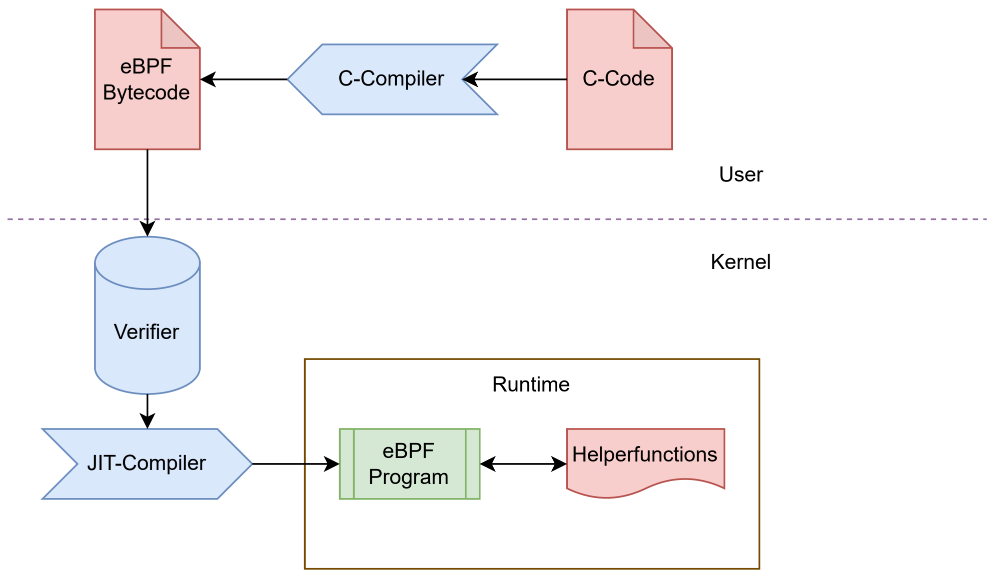

Figure 2.1 – eBPF's components and how they interact as well as where in the kernel they are running.

JIT-compiler ensures optimal performance by compiling to machine specific instruction set also known as machine code. Afterwards the eBPF Program can be safely executed when the attached hook is passed.

2.1.2 Verifier

With our alternative, we want to explore whether it might be possible to remain expressive while still complying with eBPF's safety standards, which makes it necessary to take a closer look at the verifier.

After ensuring that the calling process has the required privileges, the verifier checks the program for safety violations. These checks include a check for run to completion, which means no unbounded loops and no blocking. The program is also required to conduct safe memory accesses, thus it needs to acquire and release resources correctly, initialize memory before usage, and never access memory which is out of bounds. Furthermore, it is verified that the eBPF Program does not exceed a given size limit. In addition to this size limit, there is also a limit in complexity that is enforced by evaluating all possible execution paths. But it is important to remember that the verification process only ensures safety not security.

In the next step the eBPF program code is hardened against known vulnerabilities. To this end the memory containing the eBPF program is made read-only, so that any modification will lead the kernel to crash and not to execute the potentially dangerous program. Furthermore, security measurements against Spectre [Koc+19], an attack based on CPUs branch-prediction, are added.

2.1.3 Issues

There are, however, a few problems with this system. eBPF and its verifier have been growing [Cha] ever since their introduction and have now reached a size that makes them hard to fix and prone to many bugs [May]. Another problem comes from the copious number of restrictions. Thus, limiting the expressiveness of eBPF severely.

This leads to decreased performance for computationally complex programs, like checksums. To combat this issue, helper functions were introduced. Helper functions provide a stable and safe interface for eBPF programs. Nowadays, where kernel functions or kfuncs are more stable, they can be used for eBPF too. While these functions help extend the restricted functionality of eBPF to a certain degree, they also bypass some of the safety guarantees given by eBPF originally [NISf].

2.2 Rust Programming Language

Rust is a relatively new programming language centered around safety and performance. In the next sections, this interesting programming language is explained in more detail, with a special emphasis on the elements that make it interesting for replacing eBPF, as well as what might be a problem to that end.

2.2.1 Language Design

Rust is a general-purpose programming language. It is often compared with C, due to them being similar in performance and use cases. While its main use cases are cli tools, WebAssembly, networking and embedded systems, Rust has become the only other language besides C and assembly used in the Linux kernel development. What makes it so interesting for my idea are the safety features that are built in. Rust is memory and type safe, which makes it great for kernel development, since unsafe or incorrect memory access often leads to errors and bugs. To combat resource leaks and other bugs related to the handling of resources, Rust utilizes a complex scoping system. This system is based on Resource Acquisition Is Initialization (RAII) and a new system of ownership.

2.2.2 Memory Safety

RAII is a system that was originally developed for C++. It demands that all resources are acquired before the corresponding object can be initialized, both are handled by the constructor. But more importantly, when the object is destroyed, which means its destructor is called, all resources held by the object are released. This guarantees that as long as the object is initialized and finalized correctly, no resource leaks happen. [CMM18]

However, this necessitates dealing with variables a bit differently and pay attention to the variables scope. If the variable is given to another scope, it is moved and therefore no longer accessible from the original scope. One-way to get around this would be to create a copy of this variable either with the copy trait for datatypes that can be placed on the stack or with clone for most other datatypes that are placed on the heap. But this requires additional memory and is very inefficient for bigger data. For datatypes that should not exist twice, like mutex, this is not even an option. Instead, you can let a variable be borrowed. This way not the data itself is passed but rather a pointer, thus keeping the ownership of the data in the original scope. By creating this datatype as mutable and passing it as mutable, it can be changed in the new scope and the changed value is then available in the original scope. However, only one mutable borrow is allowed at a time, in opposition to immutable borrows which have no such limit. Examples for Rusts ownership system can be found in Listing 2.2. In this listing, the second example will fail, because `num` is moved in line 7 and is destroyed when the function `is_even_str()` terminates. Thus, it is no longer available in line 8. For the first example this does work since `is_even()` only gets a copy of `num_primitive()`. In the third example, something similar is done manually with `.clone()`. Example 4 uses borrowing, thus only passing a reference to `is_even_str_brw()`, which is what Rust is known for.

Responsible that the code adheres to the rules is the borrow checker [On2b]. It operates on the mid-level intermediate representation (MIR), which is a simplified version of Rust somewhere between source code and LLVM. This is built from the Abstract Syntax Tree (AST) generated in the previous compile step. At this level of IR, it becomes easier to construct control-flow graphs and conduct dataflow analyses. This enables checking the type safety and the ownership system reliably. As already pointed out, Rust creates all variables as immutable by default. If one wants to change any data after its initialization, it needs to be explicitly allowed by using the mut keyword when creating a variable.

2.2.3 Limitations

While Rust is considered a programming language that focuses heavily on memory safety, there are still other areas of a program's safety, that are not covered by Rust. And for normal use cases, fixing these concerns is not necessary, but for kernel extensions they need to be addressed.

- **Stack Protection:** Rust can protect against stack buffer overflow attacks. Most compilers support something like clangs SafeStack [Teab] or rustc's own exploit mitigations [On2a] to prevent attacks on the stack. However, clang's compiler pass is optimized for C and might not work for Rust. Most of Rusts mitigations only work for nightly builds of the compiler. Nightly builds are usually in an earlier stage of development and thus might be unstable, which is counteractive to the development of a safe tool.

```
 1 fn main() {
 2     //Ex. 1
 3     let num_primitive:i32 = 2; //i32 does implement copy trait
 4     let mut even:bool = is_even(num_primitive); //num_primitive copied here
 5     println!("is_even({:?})={:?}",num_primitve,even); //does work
 6
 7     //Ex. 2
 8     let num = String::from("2"); //String does not implement the copy trait
 9     even = is_even_str(num); // num is moved here
10     println!("is_even_str({:?})={:?}",num,even); //does not work
11
12     //Ex. 3
13     even = is_even_str(num.clone()); //num cloned here
14     println!("is_even_str({:?})={:?}",num,even); //does work
15
16     //Ex. 4
17     even = is_even_str_brw(&num); //num borrowed here
18     println!("is_even_str_brw({:?})={:?}",num,even); //does work
19 }
20
21 fn is_even(num:i32) -> bool{...}
22 fn is_even_str(num:String) -> bool{...}
23 fn is_even_str_brw(num:&String) -> bool{...}
```

Listing 2.2 – Example for Rust's memory system

- **Termination:** This becomes especially problematic for IRQ context, where the kernel module cannot be preempted. The most direct solution to this are coding restrictions.

Chapter 3

Design

Our approach, as seen in figure 3.1, looks quite similar to eBPF from a high-level perspective. But taking a closer look, it becomes clear that it is actually different in many ways. Instead of developing something completely new, it extends the functionality of kernel modules.

The system starts with Rust source code, which is then checked for restrictions like the usage of potential infinite loops. Afterwards the code is compiled, where well-chosen compiler settings provide additional safety and security. During compiling, the Rust code is linked to a specialized library. The resulting file can then be loaded into the kernel while it is running. There it can utilize helper functions, which now only need to provide access to kernel functionalities. Since this is simply a safer kernel module, it can be attached to hooks inside the kernel like any other module. It is important to note that the requirements in terms of safety differ between a module used in normal process context and one used in IRQ context.

3.1 How to Achieve Safety

When loading an extension into the kernel, many safety regulations must be followed. This way, stability and security of the kernel can be ensured. Responsible for this, in the case of eBPF, is the verifier. For the Rust-based alternative, I want to utilize a set of different solutions to achieve a comparable result.

First, some safety concerns can be protected against using Rust's language features. These are memory safety as well as type safety. The correctness of each memory access is guaranteed by Rust's memory safety features, as explained in 2.2.2. Type safety is simply assured by Rust being type safe, which is not always directly visible, since one can declare a variable without assigning a type, but at initialization, Rust automatically sets the type if possible. This is checked during compilation.

Additionally, it must be guaranteed that the stack is used in a safe and secure manner. For the security of the stack, compiler passes can be used. Both rustc nightly [On2a] and clang [Teab] provide those. In terms of safety, it is necessary to make sure that the stack has a bounded size and does not grow over this limit. clang provides a tool to estimate the needed stack space of a program [Teaa] but it is not clear how well it works for Rust, as it was developed for C. Alternatively, projects like Cargo Call Stack [Apa] can be used, however they might need further development. If it turns out that both are unfit for the task, a combination of a guard page and page fault handler should do the job.

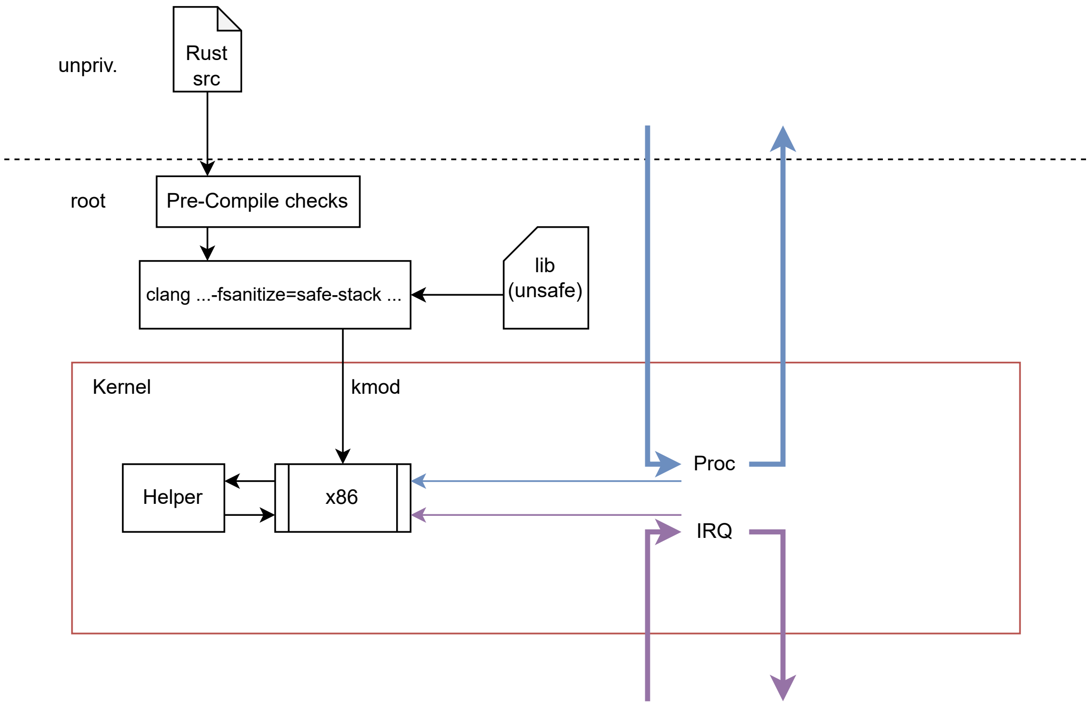

Figure 3.1 – Schematic view of the proposal

To ensure termination of the module, a simple pre-emption timer can be used. While this is a drastic measure, it might be enough for most kernel extensions running in normal process context, without placing further restrictions on their capabilities. This timer should not use time as a form of measurement, since it might be inconsistent depending on the system it is running on. By using an instruction counter, this problem can be avoided.

Furthermore, no module should be in possession of a kernel-owned lock. If a kernel module would hold such a lock while being pre-empted, it would poison the lock. To fix this issue, a limited library is provided that excludes locks. Alternatively, a syntax check could eliminate lock usage.

To further increase the expressiveness of the Rust modules, unsafe code could be used, but loaded into another address space and run with some independence from the rest of the module. However, how the safety of this idea can be guaranteed is highly debatable and up for future work.

3.2 Additional Checks for IRQ Context

When the kernel extension is attached to an interrupt, and thus executed in IRQ context, it becomes necessary to check further properties and apply additional restrictions. This is mainly because programs running in IRQ context cannot be pre-empted.

One such property that must be fulfilled in IRQ context is a bounded computation time. This can be achieved by checking the loops and recursion. Loops can be easily checked for a constant iteration, by ensuring that every bound is static and the variable that is iterated over is not changed inside the loop in a way that would impact the execution time negatively. For more complex examples, where the number of loop iterations is not static, it needs to be assured that the break condition is reached.

This can be done for some cases, while being impractical or impossible for others [SKH16; Mar+24]. For recursion, something similar is the case. However, forcing the implementation of a recursion limit in each module that uses recursion might be an easy fix that is controllable with syntax checks.

Additionally, programs in IRQ context are not allowed to do blocking calls. Again, this is because they cannot be pre-empted. Solutions, for this might be to analyze the call graph, which is time consuming and might not be reliable enough [Ott+23]. Another solution might be to prohibit the usage of sync functions. A relatively simple solution would be to exclude sync functions from the library usable by kernel modules in the presented construct, as done for locks.

Chapter 4

Implementation

The first idea was to create a working prototype that could run a simple version of this system. But it turned out that Rust is still somewhat new in the kernel development community making it difficult to create something like that from nothing, with the limited time at hand. Thus, I opted to recreate some components [Kowa] from the construct, to show the simplicity with which they can be achieved. One is a macro that can check any Rust source file in search of loops. The second is a simple script that is theoretically able to check, compile and load a Rust kernel module.

4.1 Tool for Compiling and Loading of a Module [Kowe]

This script is actually composed of five separate scripts that together provide all functionalities to check, build and load a module. By splitting this into multiple scripts, they can be called independently in case only certain steps of the process are needed.

The main script is the "start" script. It calls all other scripts and takes input parameter that decide which subscripts are to be executed as well as where the module source code file is located.

The first script called in the "start" script is "config.sh". It takes the path to the module and whether constant loops are permitted. Then it writes both of them and the current date into the "config.rs" file from where the macro 4.2 can retrieve this information, which is already in the Rust source code format.

Next, the "check" script is called, which is the only call that cannot be skipped when executing "start.sh". It is responsible for executing all pre-compile time tests such as the implemented macro in section 4.2.

Afterwards, the "build" script gets executed. This script is responsible for calling the Makefile that is used for compiling the module. It is also responsible to add certain compile options to ensure safety and security.

At last, the "load" script is executed and loads the module into the kernel using "insmod".

4.2 Rust Macro for Loop Checking [Kowd]

Rust macros are rather powerful and come in two forms. Declarative and procedural macros, where procedural macros come in three forms. For the checks, a function like macro is used. It is important to note that the macro is more an additional program and does not have to be part of the kernel module.

The macro parses an entire given Rust source file and creates an AST from it. Afterwards this AST is searched recursively for the identifiers "for" and "while", which show that there are loops. If wanted, for-loops can be allowed when they are constant, which means of the format as seen in listing 4.1. It is executed by trying to build a test which will panic and return an appropriate message in case the macro finds a prohibited loop.

The macro is configurable via the "config.rs" file. This file contains the path of the source code to be checked, as well as the option to allow or forbid constant loops. This allows to change settings of the macro without changing its own source code. This might be important if it were to be used in a real environment, where the integrity of the macros source code needs to be protected.

A question that still remains is, why we decided to use a macro for this, when it is used more like an external program. This has two main reasons. The first being that the way it works has changed over time and it was not always intended to be used like this. And secondly, most of the parsing and so forth is mainly intended for macros which makes it a bit easier to use. If this should ever be really used, it might be worth to consider making it a full-fledged executable.

```
    for <variable> in <number>..<number> {...}
```

Listing 4.1 – constant for loops format

Chapter 5

Expected Performance

While I was not able to record any real data, it is possible to make some assumptions for the safety and performance of this construct, based on the observation of similar cases.

5.1 Computation Speed and Efficiency

Using the Rust module construct might lead to better computational performance by compiling to better code and by enabling more efficient programming.

First, way ahead of time or ahead of time compilation is considered to result in a faster running program compared to JIT compilation. However, in case of eBPF's JIT-compiler this might be negligible difference, since the compiled bytecode that eBPF takes as an input is already so close to machine code, at least for x86 architecture that there is hardly any necessity for optimizations. For ARM, the difference between eBPF and a pre-compiled program might be bigger.

The second way this alternative improves performance is by having less restrictions enforced on the code. This way, more efficient code can be written. It might even be possible to use hardware acceleration for certain calculations, which could further increase performance. An example I came across was the implementation of the CRC32C checksum in the SCTP [RS] in LoxiLB [Aut], which had a huge impact on the performance [Apa]. Even if one uses the software implementation of CRC32C, it should still be fast enough that the speed limit of most storage mediums is easily reached [Shi; sta]. With hardware acceleration using SSE4.2, one can achieve even faster speeds [fco]. This leads to the assumption that the eBPF implementation of this checksum is highly inefficient.

5.2 Safety

Even more important than the computational performance of an eBPF program is its safety for the Linux kernel. Thus, any replacement should be at least as secure as eBPF.

Rusts language features guarantee the safety of each memory access and the type safety. Stack safety is guaranteed by the compiler. The pre-emption timer is a very drastic but sure way to stop a program from running indefinitely. All of these are already existing concepts and have thus been proven in practice and should thus provide adequate safety.

However, some safety solutions are not yet guaranteed to work in practice. Memory acquisition and release is handled by predefined constructors and destructors. This does only work if they are tested extensively, such that they will, under no circumstances, cause unwanted behavior. Termination in IRQ is provided by syntax checks, which might leave a few loopholes.

But based on these observations, it can be assumed that the presented concept can become as safe as eBPF. Dividing the safety features on many different tools has the added benefit of enlarging the community involved in bug fixing and safety checking of the overall construct.

Chapter 6

Evaluation

6.1 Test Setup

With the following test set up, it could be determined that the Rust macro, which was developed for this thesis, worked as intended.

To test all features of the macro, [XAM] was chosen because of its size, which is somewhere between too small for testing and unmanageable big. At the end, a simple while loop was added, such that all possible loop types were present. The program's correctness is not necessary for this test, since the code never gets to be compiled. However, to parse the file, it had to have a correct Rust syntax. The macro was configured such that it would check the file and allow only constant for-loops.

All tests a run on a VM using VirtualBox 7.0.8 r156879 for Windows with 8192MB RAM, 4 CPU cores with a base clock of 3,7GHz and running Debian 12 Bookworm.

6.2 Procedure

The macro was tested for the following two things. Its ability to identify every loop correctly and distinguish the allowed constant for-loops from normal ones successfully. And how long it would take to go through the entire file without finding a loop, which should be the macro's worst-case scenario in terms of runtime.

Two phases of testing were conducted. In the first phase, the file was taken as prepared in 6.1 and tried to compile the macro test. Then every for-loop was converted into a constant for-loop such that it fulfilled the form given in 4.1. In the end, only the while loop should bring the macro to panic. If this last loop is removed, no forbidden loops remain [Kowc]. Which enables the second test phase to begin. Now that the macro does not panic, it will run through the entire AST of the file, thus it will have the worst run time.

Additionally, a small bench script [Kowb] was developed to eliminate the overhead of the dependency compilation, when running the speed benchmark. The problem being that once the test was run successfully, it would not run without a clean compilation, which certainly is an issue with this design choice. Thus, the bench script would run the macro compilation three times. Once with a broken Rust file containing the macro call, which lead to the compilation of all dependencies but not the macro and the broken file. Then it would run again, this time with the correct Rust file [Kowg], now only compiling the macro. In the last compilation, everything is cleared, and the compilation should run from beginning to end without errors. For the last two compilations, data is collected by repeating the tests 100 times. For comparison, the same is done with the much smaller `test.rs` file, which calls the macro.

6.3 Results

In these tests, it was found that the macro would falsely find a for loop, when encountering a trait implementation 6.1. But this was easily fixed by adding another match case in the macro and excluding the next “for” identifier from being categorized as a loop. After this fix, it worked flawlessly. It was shown that it could distinguish “for” and “while” in strings and comments, while-loops, for-loops, constant for-loops and now trait implementations too.

The benchmark returned with the result that the worst-case scenario would take around 2,56s including compiling the dependencies and around 0,18s without dependencies. When comparing this with the results from the tests with `test.rs` nothing happens. However, the time overall, including compiling the dependencies, rises to around 2,59s. While this shows that the complexity of the checked file only has a very limited real impact, it is still unclear as to why the overall time rises. But the difference between the overall time and the compilation time of just the macro indicates that there might be a benefit in this case of up to 2,41s or 93% if the macro could be turned into a precompiled binary [Kowf].

```
    impl ... for ... {}
```

Listing 6.1 – trait implementation in Rust

Chapter 7

Conclusion

For many years, eBPF has been the method to go to, when it comes to time sensitive applications, where even the normal OS overhead could be too much. To provide kernel level access, but still ensure the stability of the system, every eBPF program is thoroughly checked and many restrictions applied. But in recent years, many flaws of this system came to light [NISa; NISb; NISd; NISe; NISf]. All this while the kernel support for Rust, a language which is built with safety in mind, is steadily growing. Which is why the possibility to use a construct built on Rust kernel modules to replace eBPF is a rising question.

I found that indeed Rust can be a good alternative to eBPF because of its inherent traits, powerful macros and growing kernel support.

Memory safety is already a part of Rust due to its RAII and ownership system. Some other things like stack safety have not yet found their way into the stable version of rustc but might be worth considering. Prohibited functions could be excluded by using a specialized limited library, which becomes especially important for IRQ context, where blocking calls are not allowed, since there is no pre-emption. For an ensured termination, I propose a Rust macro that prohibits the use of variable loops, which can also be extended to check for a lot of other things.

In the case of infinite recursion however, work still needs to be done. A full prototype of this system could provide a better overview about which measures might be insufficient or missing. Another unresolved issue is how to protect a checked program from being tempered with.

List of Acronyms

AST	Abstract Syntax Tree
BPF	Berkeley Packet Filter
cBPF	classic Berkeley Packet Filter
eBPF	extended Berkeley Packet Filter
IRQ	Interrupt Request
JIT	Just-In-Time
RAII	Resource Acquisition Is Initialization

List of Figures

List of Listings

References

[Apa] Jorge Aparicio. *Cargo Call Stack*. URL: `https://github.com/japaric/cargo-call-stack`.

[Ast+22] Vytautas Astrauskas et al. “The Prusti Project: Formal Verification for Rust.” In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Cham: Springer International Publishing, 2022, pp. 88–108. ISBN: 978-3-031-06773-0.

[Aut] LoxiLB Authors. *LoxiLB*. URL: `https://www.loxilb.io/`.

[BA22] William Bugden and Ayman Alahmar. “Rust: The Programming Language for Safety and Performance.” In: *2nd International Graduate Studies Congress (IGSCONG’22)*. 2022. DOI: `10.48550/arXiv.2206.05503`. URL: `https://arxiv.org/abs/2206.05503`.

[Bal+17] Abhiram Balasubramanian et al. “System Programming in Rust: Beyond Safety.” In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS ’17. Whistler, BC, Canada: Association for Computing Machinery, 2017, 156–161. ISBN: 9781450350686. DOI: `10.1145/3102980.3103006`. URL: `https://doi.org/10.1145/3102980.3103006`.

[BHR18] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. “Verifying Rust Programs with SMACK.” In: *Automated Technology for Verification and Analysis*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2018, pp. 528–535. ISBN: 978-3-030-01090-4.

[Cha] Paul Chaignon. *Complexity of the BPF Verifier*. URL: `https://pchaigno.github.io/ebpf/2019/07/02/bpf-verifier-complexity.html`.

[CMM18] Guillaume Combette and Guillaume Munch-Maccagnoni. “A resource modality for RAII.” In: *LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages*. Oxford, United Kingdom, July 2018, pp. 1–4. URL: `https://inria.hal.science/hal-01806634`.

[Com] Kernel Developement Community. *BPF-Documentation*. URL: `https://www.kernel.org/doc/html/latest/bpf/index.html`.

[eBP] eBPF.io. *What is eBPF?* URL: `https://ebpf.io/what-is-ebpf/`.

[fco] fcorbelli. *Real life vs max checksum speed*. URL: `https://encode.su/threads/3514-Real-life-vs-max-checksum-speed`.

[Jia+23] Jinghao Jia et al. “Kernel extension verification is untenable.” In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HOTOS ’23. Providence, RI, USA: Association for Computing Machinery, 2023, 150–157. ISBN: 9798400701955. DOI: `10.1145/3593856.3595892`. URL: `https://doi.org/10.1145/3593856.3595892`.

[Koc+19] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[Kowa] Ben Kowol. *Git-Repository*. URL: `https://git.noc.ruhr-uni-bochum.de/kowolb99/anode`.

[Kowb] Ben Kowol. *Git-Repository - Benchscript*. URL: `https://git.noc.ruhr-uni-bochum.de/kowolb99/anode/-/blob/main/tests/benchscript.sh?ref_type=heads`.

[Kowc] Ben Kowol. *Git-Repository - cli_utils.rs*. URL: `https://git.noc.ruhr-uni-bochum.de/kowolb99/anode/-/blob/main/tests/cli_utils.rs?ref_type=heads`.

[Kowd] Ben Kowol. *Git-Repository - Macro*. URL: `https://git.noc.ruhr-uni-bochum.de/kowolb99/anode/-/tree/main/src?ref_type=heads`.

[Kowe] Ben Kowol. *Git-Repository - Scripts*. URL: `https://git.noc.ruhr-uni-bochum.de/kowolb99/anode/-/tree/main/scripts?ref_type=heads`.

[Kowf] Ben Kowol. *Git-Repository - Test Data*. URL: `https://git.noc.ruhr-uni-bochum.de/kowolb99/anode/-/blob/main/tests/Test_data.ods?ref_type=heads`.

[Kowg] Ben Kowol. *Git-Repository - test.rs*. URL: `https://git.noc.ruhr-uni-bochum.de/kowolb99/anode/-/blob/main/tests/test.rs?ref_type=heads`.

[LGH24] Peter Wägemann Maximilian Ott Rüdiger Kapitza Luis Gerhorst Henriette Herzog and Timo Hönig. "VeriFence: Lightweight and Precise Spectre Defenses for Untrusted Linux Kernel Extensions." In: *In The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024)*. 2024.

[Lin+22] Michael Ling et al. "In rust we trust: a transpiler from unsafe C to safer rust." In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 354–355. ISBN: 9781450392235. DOI: `10.1145/3510454.3528640`. URL: `https://doi.org/10.1145/3510454.3528640`.

[Mar+24] Emad Jacob Maroun et al. "The Platin Multi-Target Worst-Case Analysis Tool." In: *22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*. Ed. by Thomas Carle. Vol. 121. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 2:1–2:14. ISBN: 978-3-95977-346-1. DOI: `10.4230/OASIcs.WCET.2024.2`. URL: `https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2024.2`.

[May] Dan Mayer. *bugs per line of code ratio*. URL: `https://www.mayerdan.com/ruby/2012/11/11/bugs-per-line-of-code-ratio`.

[Mil+19] Samantha Miller et al. "Practical Safe Linux Kernel Extensibility." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019, 170–176. ISBN: 9781450367271. DOI: `10.1145/3317550.3321429`. URL: `https://doi.org/10.1145/3317550.3321429`.

[MK14] 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. Portland, Oregon, USA: Association for Computing Machinery, 2014, 103–104. ISBN: 9781450332170. DOI: `10.1145/2663171.2663188`. URL: `https://doi.org/10.1145/2663171.2663188`.

[NISa] NIST. *CVE-2021-29154*. URL: `https://nvd.nist.gov/vuln/detail/CVE-2021-29154`.

[NISb] NIST. *CVE-2021-31440*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-31440.

[NISc] NIST. *CVE-2021-3490*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-3490.

[NISd] NIST. *CVE-2021-45402*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-45402.

[NISe] NIST. *CVE-2022-23222*. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-23222.

[NISf] NIST. *CVE-2022-2785*. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-2785.

[On0] *bcc*. URL: https://iovisor.github.io/bcc/.

[On2a] *Exploit Mitigations*. URL: https://doc.rust-lang.org/rustc/exploit-mitigations.html.

[On2b] *MIR borrow check*. URL: https://rustc-dev-guide.rust-lang.org/borrow_check.html.

[On3] *SMACK*. URL: http://smackers.github.io/.

[Ott+23] Maximilian Ott et al. "Towards Just-In-Time Compiling of Operating Systems." In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. PLOS '23. Koblenz, Germany: Association for Computing Machinery, 2023, 41–48. ISBN: 9798400704048. DOI: 10.1145/3623759.3624551. URL: https://doi.org/10.1145/3623759.3624551.

[RBA17] Elena Reshetova, Filippo Bonazzi, and N. Asokan. "Randomization Can't Stop BPF JIT Spray." In: *Network and System Security*. Ed. by Zheng Yan et al. Cham: Springer International Publishing, 2017, pp. 233–247. ISBN: 978-3-319-64701-2.

[Riv+21] Elijah Rivera et al. "Keeping Safe Rust Safe with Galeed." In: *Proceedings of the 37th Annual Computer Security Applications Conference*. ACSAC '21. Virtual Event, USA: Association for Computing Machinery, 2021, 824–836. ISBN: 9781450385794. DOI: 10.1145/3485832.3485903. URL: https://doi.org/10.1145/3485832.3485903.

[Rob] Alastair Robertson. *bpftrace*. URL: https://bpftrace.org/.

[RS] Ed. R. Stewart. *Stream Control Transmission Protocol*. URL: https://www.rfc-editor.org/rfc/rfc4960.html.

[Sah24] Raj Sahu. "Re-thinking termination guarantee of eBPF." In: 2024.

[Shi] Xiaoyi Shi. *fast-crc32c*. URL: https://www.npmjs.com/package/fast-crc32c.

[SKH16] 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)*. 2016, pp. 1–11. DOI: 10.1109/RTAS.2016.7461326.

[sta] stackoverflow. *Implementing SSE 4.2's CRC32C in software*. URL: https://stackoverflow.com/questions/17645167/implementing-sse-4-2s-crc32c-in-software.

[Sun+24] Hao Sun et al. "Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program." In: *Proceedings of the Nineteenth European Conference on Computer Systems*. EuroSys '24. Athens, Greece: Association for Computing Machinery, 2024, 689–703. ISBN: 9798400704376. DOI: 10.1145/3627703.3629562. URL: https://doi.org/10.1145/3627703.3629562.

[Teaa] The Clang Team. *Clang command line argument reference*. URL: https://clang.llvm.org/docs/ClangCommandLineReference.html.

[Teab] The Clang Team. *SafeStack*. URL: https://clang.llvm.org/docs/SafeStack.html.

[WLC23] Yutian Wang, Dan Li, and Li Chen. "Seeing the Invisible: Auditing eBPF Programs in Hypervisor with HyperBee." In: *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*. eBPF '23. New York, NY, USA: Association for Computing Machinery, 2023, 28–34. ISBN: 9798400702938. DOI: 10.1145/3609021.3609305. URL: https://doi.org/10.1145/3609021.3609305.

[XAM] XAMPPRocky. $cli_utils.rsfromTokei$. URL: https://github.com/XAMPPRocky/tokei/blob/master/src/cli_utils.rs.