Rust in the Linux Kernel

This paper was prepared as part of my MEng project.

Contents

MEng in Electronic & Computer Engineering

Ege Bilecen
ege.bilecen2@mail.dcu.ie
Dublin City University
Dublin, Ireland

***Abstract*—The improper use of C has resulted in many security problems in the Linux kernel, most of which are related to memory. Strong security is crucial for the Linux kernel because of its wide use. Various techniques have been proposed to improve C; however, each technique has overhead and complexity. It is possible to achieve better performance, safety, and reduced complexity by using a memory-safe language. In late 2022, Rust was added as a second language to the Linux kernel. I assess the properties of two kernel modules implementing the same algorithm, one written in C and the other in Rust. In addition, I also assess whether the Rust code requires any restrictions for interoperability with C and whether Rust's safety features are advantageous or disadvantageous. The comparison of the C and Rust modules demonstrates that the performance of the Rust module is very close to that of the C, along with the advantageous safety features, and does not require any restrictions for interoperability with the C owing to safe abstractions.**

***Keywords*—Linux kernel, kernel module, Rust, C, PRESENT-80.**

I. Introduction

C is one of the most popular programming languages used to develop system-level software. There are many reasons for its popularity, but the most prominent are that it is simple, flexible, portable, and very fast. However, owing to its simplicity and flexibility, it is also prone to developer mistakes [14], which may be fatal for system-level software and result in security vulnerabilities.

Throughout history, the Linux kernel has suffered from security problems owing to the improper use of C, most of which are memory related [1][2], such as use-after-free and memory overflow. Strong security is even more crucial for the Linux kernel, because it is commonly used in many applications. There have been various proposals to add new system-level programming languages to the kernel in the past to improve the overall security; however, none of them have been successful for various reasons, including performance inefficiency and language complexity.

In early 2021, a proposal with an emphasis on improved overall security along with other modern language features that Rust provides was made by Miguel Ojeda [3] under the "Rust for Linux" project name to add the Rust programming language as a second language to the kernel. This proposal also included bindings, safe abstractions, and example modules written in Rust. Subsequently, a real NVMe driver was written in Rust to compare its performance against C and evaluate the Rust language [4][5].

In late 2022, official support for the Rust programming language was added to the Linux kernel [13]. The added Rust support is minimal, as not all of the safe abstractions in the initial proposal are made into the kernel. However, it is still possible to create custom bindings and safe abstractions.

This study aims to assess the following using two custom loadable kernel modules implementing the same algorithm, one written in C and the other in Rust:

1. Rust safety features and whether they are advantageous or disadvantageous when writing a kernel module.
2. Properties of the same module written in Rust and C in terms of performance and footprint.
3. The interoperability of Rust and C code to determine whether Rust's language features may be freely used without requiring any restrictions.

II. Existing Techniques To Improve C

It is possible to improve C, instead of adding a new language to the kernel. In fact, the following techniques have been proposed in the past, and some have found their way into the kernel:

- Address Space Layout Randomization (ASLR)
- Control Flow Integrity (CFI)
- Static Analysis Tools
- eBPF
- Code Isolation

However, each technique has an associated overhead that may reduce the overall efficiency of the kernel. In addition, most of these techniques utilize runtime checks to prevent security problems, resulting in increased complexity. With the use of a memory-safe language, such as Rust, it is possible to mitigate these problems [15], even before runtime, owing to their strict type system and lack of undefined behaviors, thereby reducing the overall performance overhead and complexity without compromising performance.

III. Design of the Kernel Module

As technology improves rapidly, each device becomes more computationally capable. This advancement has led to the emergence of new applications of these devices. One of the most popular applications is the Internet of Things (IoT), in which devices create a large, interconnected network to communicate and exchange data. Security is arguably more crucial in these systems because one compromised device may affect the entire network. Therefore, I decided to implement the PRESENT-80 encryption algorithm. This algorithm is not implemented in the kernel and is heavily dependent on bit manipulation, which allows the use of Rust specific safe features to perform these operations, whereas it has undefined results in standard C. It should be noted that to evaluate Rust in the Linux kernel, it is necessary to write a custom algorithm that is independent of kernel features itself. Otherwise, we will only call the safe Rust abstractions that are basically wrappers around the C functions.

PRESENT-80 is a simple and lightweight block cipher designed for use in resource-constrained devices [6], such as IoT devices. It uses an 80-bit key and operates on a 64-bit block, also known as state. The encryption process consists of four steps, with each step but the first step repeated in each round:

1. **Round Key Generation:**
   This step is executed only once, prior to the beginning of the encryption. The given 80-bit key is expanded to thirty-two 64-bit keys to be used in each encryption round.
2. **Round Key Addition:**
   The state is updated with the XOR operation using the 64-bit round key.
3. **Substitution Layer:**
   Each 4-bit in the state is updated with a different 4-bit specified in the PRESENT-80 substitution box.
4. **Permutation Layer:**
   Each bit in the state moved into a different bit position specified in the PRESENT-80 permutation box.

The main approach in both loadable kernel modules is to keep the algorithm implementations largely the same, with the only differences being the use of language-provided features to perform the same operations.

To receive and send data from and to the user space, I decided to create two misc. character devices, named `present80_key` and `present80_encrypt`, respectively. In Linux, character devices are special files in which file operation calls such as read and write are redirected to the file owner [11], which is the kernel module in this case. The creation of misc. character devices relies on kernel features; therefore, I need to use safe Rust abstractions to call the related C functions. The user may now set the encryption key by writing ten bytes to the `present80_key` device, initiate the encryption of the block by writing eight bytes to the `present80_encrypt` device, and then obtain the encryption result by reading the same device. Whenever these write or read operations are performed by the user, the actions are handled by callback functions defined in the code.

Although it is possible to use integer types (`uint32_t`, `uint64_t`, etc.) in both languages for encryption calculations to simplify the overall code (and possibly increase the overall speed), I decided to use byte arrays and perform bitwise operations on each byte itself instead of integer types, with the expectation of using more language features of Rust. For example, bits of the 80-bit key stored in two separate integer types may be rotated in the following way using C:

```c
uint64_t t;
uint64_t kl;
uint64_t kh;

...

//The key register is rotated by 61 bit
positions to the left
t = kh & 0xFFFF;
kh = (kl >> 3) & 0xFFFF;
kl = (kl << 61) | (t << 45) | (kl >> 19);
```

Fig. 1. PRESENT-80 round key generation step, key rotation using integer type [16].

However, in the byte array approach, the bytes that make the integer are rotated one at a time as follows using Rust:

```rust
fn bytes_rotate_right(bytes: &mut [u8],
bit_count: usize) {
    let size = bytes.len();

    ...

    let fpb = bytes[0];
    let mut pb;
    let mut npb = 0;

    for ib in 0..size {
        let inb = if ib == size - 1 { 0 } else
{ ib + 1 };

        if ib == 0 {
            pb = bytes[ib];
            npb = bytes[inb];
        } else if ib == size - 1 {
            pb = npb;
            npb = fpb;
        } else {
            pb = npb;
            npb = bytes[inb];
        }

        bytes[inb] = ((pb &
preserve_mask).checked_shl((8 - shift_count) as
u32)).unwrap_or(0)
            | (npb.checked_shr(shift_count as
u32).unwrap_or(0));
    }

    ...
}
```

Fig. 2. PRESENT-80 round key generation step, key rotation using bytes of an integer.

IV. Comparison of the C and Rust Kernel Modules

All results below were obtained using the following hardware and operating system:

- **CPU:**
  Intel® Alder Lake Core™ i7-12700H 14C/20T, 24MB L3, E-CORE Max 3.50GHZ P-CORE Max 4.7GHZ, 45W, 10nm SuperFin
- **RAM:**
  8GB DDR4 3200MHz

- **OS:**
  Linux 6.3.0-microsoft-standard-WSL2+ #4 SMP Sat May 25 17:35:42 BST 2024 x86_64 GNU/Linux (custom-built Rust-enabled kernel using the WSL2 default configuration, which may be found in [10])

The encryption tests below were performed using the same key and data pair on both modules, which consists of randomly generated 1000 keys and 1000 blocks of data, resulting in a total of 1,000,000 encryptions.

In addition, because the added Rust support is minimal and does not have support for device creation, I use the initial safe abstractions of the "Rust for Linux" project, which may be found in [9].

A. *Module Size*

When a C or Rust code is compiled, the compiler generates an object file that contains metadata and data regarding the overall code. The information inside this object file is structured in a specific format, such as the Executable and Linkable Format (ELF) for UNIX-like systems or Portable Executable (PE) for Windows, based on the compiler and target platform, and is separated into various sections [7][8]. The total size of a module is the aggregated size of each individual section of the object file, along with additional metadata. The sizes of both modules are shown in **Fig. 3**.

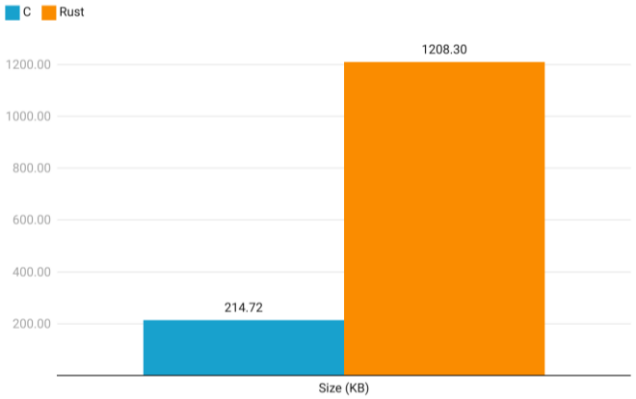

Fig. 3. Size of the C and Rust modules in kilobytes (KB).

B. *Section Size*

In accordance with the previous section, the size of the specific sections of both modules is shown in **Fig. 4**.

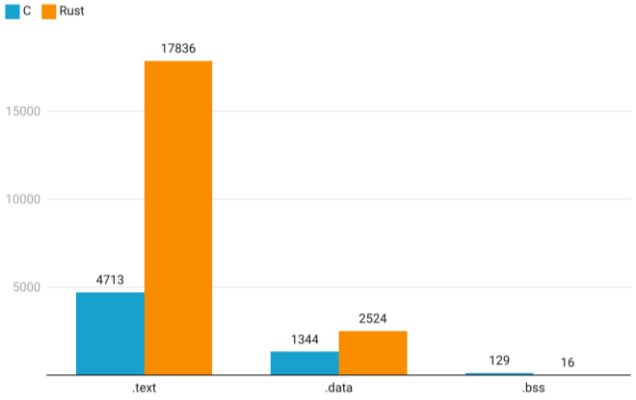

Fig. 4. Size of the `.text`, `.data`, and `.bss` sections of the C and Rust modules in bytes. The sizes of these sections also include the related relocation section size.

C. *Number of Symbols*

The total number of symbols in both modules is shown in **Fig. 5**. These numbers include debug symbols; however, they do not include absolute sections if their symbol names represent a file.

Fig. 5. Total number of symbols in the C and Rust modules.

D. *Average Encryption Time of a Block*

The average time required to encrypt a block in both modules is shown in **Fig. 6**. To reduce variance, five different benchmarks were performed.

Fig. 6. Average time required for the C and Rust modules to encrypt a block. The results are in µs.

E. *Total Encryption Time of 1000 Blocks with 1000 Different Keys*

The total time required to complete 1,000,000 encryptions in both modules is shown in **Fig. 7**. To reduce variance, five different benchmarks were performed.

Fig. 7. Total time it takes for the C and Rust modules to perform 1,000,000 encryptions. The results are in seconds.

F. Build Time

The total time required to build both modules is shown in **Fig. 8**.

Fig. 8. Total time it takes for the C and Rust modules to be built. The results are in seconds.

V. Analysis of the Both Modules' Properties

A. Module Size

Based on **Fig. 3**, it is clear that the Rust module is significantly larger (+139.64%). To determine what causes the Rust module to be larger, we need to look at the sizes of the other sections that are not shown in **Fig. 4**.

B. Section Size

Based on **Fig. 4**, we already know that the Rust module has two sections that are larger than the C module. However, even if these two larger sections of the Rust module are added together, there is still a large gap in the total module size. This means that some of the omitted sections are significantly larger.

When kernel modules are built, they include symbols solely for debugging purposes. If we look at **Fig. 9**, we may see that the size of the debug symbols in the Rust module is significantly larger (+142.28%) than that in the C module and makes up the majority of the Rust module.

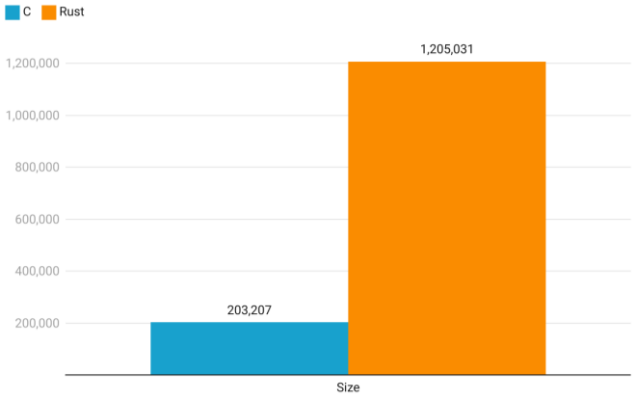

Fig. 9. Total size of debug symbols in the C and Rust modules. The sizes are in bytes. The sizes of these sections also include the related relocation section size.

Table 1 lists the percentages of the total size occupied by each section of the module.

Module	Section	Size %
C	.text	2.14%
C	.data	0.61%
C	.bss	0.059%
C	.debug	92.42%
C	Other	~4.77%
Rust	.text	1.44%
Rust	.data	0.20%
Rust	.bss	0.001%
Rust	.debug	97.27%
Rust	Other	~1.09%

Table 1. Table of which section occupies how much space in the module.

According to **Table 1**, the majority of the size is contributed by debug symbols in both modules. As Miguel Ojeda mentioned in [3], the most probable reason for the Rust module being significantly larger than the C module is that the unused parts of Rust's standard library are imported into the module, along with their debug symbols.

In addition, if we look at the undefined symbols in the C module, we may see two symbols, `misc_register` and `misc_deregister`, which are functions defined in "`drivers/char/misc.c`" and exported as symbols in the kernel. As they are exported as symbols, this allows other C modules to import them dynamically. However, Rust modules require safe abstractions, and we may see that the safe abstraction for `misc_register` and `misc_deregister` is included in the Rust module, which is called `<kernel::miscdev::Registration<...>::new_pinned`, in contrast to the C module, where it is dynamically included. We may understand that another reason for the Rust module being large is the necessity to include these abstractions in the module itself.

C. Performance

Based on **Fig. 6**, the Rust module is ~0.16µs slower than the C module when encrypting a block. Based on **Fig. 7**, the Rust module is ~0.17s slower than the C module when both modules complete a total of 1,000,000 encryptions.

D. Safety

Throughout the development of the Rust code, Rust's compiler caught many potential problems owing to short-lived references during compile-time checks and prompted me to fix them. In C, because memory management is left to the developer, it could result in access to a dangling reference, which is an undefined behavior. In addition, the compiler constantly warned me about other types of potential problems, such as the use of potentially uninitialized variables and moved values, which might result in a double-free problem if it were C code.

Because the implementation of the algorithm relies on heavy bit manipulation, there are cases where the bits are shifted by more than the width of the type (e.g., 8-bit integer). In the C standard, the result of this operation is undefined [12]. This requires an additional code to prevent this undefined behavior from occurring. However, in Rust, owing to the standard language functions, this operation may safely be handled with a variety of functions, such as `checked_shl`, `overflowing_shl`, and `wrapping_shl`.

During the testing of the code, there were cases where kernel crashed due to improper copy operations, that is, trying to copy more bytes than the size of the target buffer. In C, this may have resulted in a memory-overflow problem in the worst case or crash the kernel in the best case. However, in Rust, this copy operation is checked, which results in crashes in every case and prevents memory-overflow attacks. There has been another instance of kernel crash due to access to out-of-bounds of an array,

which is prevented by Rust by generating panic, thus crashing the kernel. In C, access to out-of-bounds is an undefined behavior [12].

Overall, development in Rust was safer because of the compile time checks, language's safe (checked) functions, and use of the special type `Result`, as all error cases must have been handled. It is likely that the potential problems detected by the Rust compiler will not be the case in a small C project like this module; however, as the project grows, the chances for these problems to occur also increase.

E. Build Time

Rust is often criticized for its slow build times. This is because Rust projects often depend on modules that may also depend on other modules. Because the project and dependency modules are built from source, it may take a long time to build, depending on the size of the project. However, the kernel's safe Rust abstractions only depend on Rust's standard modules `core` and `alloc`, and not any other modules. Based on **Fig. 8**, the build time of the Rust module is very close to that of the C module because the Rust module does not depend on modules other than those mentioned above and on safe Rust abstractions.

VI. Conclusions

Even though the Rust module is significantly bigger when compared to the C module, this may be fixed in the future with a similar approach taken in the C modules by making the safe abstractions dynamically includable.

Rust safety features were advantageous, many possible problems were detected in compile time checks even before running the module.

The test results showed that the Rust code has a performance very close to that of the C code.

The written Rust code did not require any restrictions for interoperability with the C code, owing to the safe abstractions provided by the kernel.

References

[1] "Linux Linux Kernel : CVE Security Vulnerabilities, Versions and Detailed Reports." www.cvedetails.com, https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html. Accessed 8 June 2024.

[2] Chen, Haogang, et al. "Linux Kernel Vulnerabilities." Proceedings of the Second Asia-Pacific Workshop on Systems - APSys '11, 2011, https://pdos.csail.mit.edu/papers/chen-kbugs.pdf, https://doi.org/10.1145/2103799.2103805. Accessed 8 June 2024.

[3] Ojeda, Miguel. "[PATCH 00/13] [RFC] Rust Support - Ojeda." Lore.kernel.org, 14 Apr. 2021, https://lore.kernel.org/lkml/20210414184604.23473-1-ojeda@kernel.org/. Accessed 7 June 2024.

[4] "Rust for Linux - NVMe Driver." Rust-For-Linux.com, https://rust-for-linux.com/nvme-driver. Accessed 8 June 2024.

[5] Hindborg, Andreas. Linux (PCI) NVMe Driver in Rust. 2022.

[6] Bogdanov, A., et al. "PRESENT: An Ultra-Lightweight Block Cipher." Cryptographic Hardware and Embedded Systems - CHES 2007, pp. 450–466, https://doi.org/10.1007/978-3-540-74735-2_31.

[7] "Object File." Wikipedia, 1 May 2024, https://en.wikipedia.org/wiki/Object_file. Accessed 12 June 2024.

[8] "Data Segment." Wikipedia, 21 May 2023, https://en.wikipedia.org/wiki/Data_segment. Accessed 12 June 2024.

[9] "GitHub - Rust-For-Linux/Linux at Rust." GitHub, https://github.com/Rust-for-Linux/linux/tree/rust. Accessed 13 June 2024.

[10] "WSL2-Linux-Kernel/Arch/Arm64/Configs/Config-Wsl-Arm64 at edee386af1c313fc6e5f136fbb9bfea8379cc2de · Microsoft/WSL2-Linux-Kernel." GitHub, https://github.com/microsoft/WSL2-Linux-Kernel/blob/edee386af1c313fc6e5f136fbb9bfea8379cc2de/arch/arm64/configs/config-wsl-arm64. Accessed 13 June 2024.

[11] "Character Device Drivers — the Linux Kernel Documentation." Linux-Kernel-Labs.github.io, https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html. Accessed 18 June 2024.

[12] "ISO/IEC 9899" ISO, 1999, https://www.open-std.org/jtc1/sc22/wg14/www/standards. Accessed 19 June 2024.

[13] "Linux_6.1 - Linux Kernel Newbies." Kernelnewbies.org, 12 Dec. 2022, https://kernelnewbies.org/Linux_6.1. Accessed 22 June 2024.

[14] P. C. van Oorschot. "Memory Errors and Memory Safety: C as a Case Study." IEEE Security & Privacy, vol. 21, no. 2, 1 Mar. 2023, pp. 70–76, https://doi.org/10.1109/msec.2023.3236542.

[15] Caballar, Rina Diane. "The Move to Memory-Safe Programming - IEEE Spectrum." Spectrum.ieee.org, 20 Mar. 2023, https://spectrum.ieee.org/memory-safe-programming-languages. Accessed 7 July 2024.

[16] Oryx Embedded. "present.c Source Code - PRESENT Encryption Algorithm." www.oryx-Embedded.com, https://www.oryx-embedded.com/doc/present_8c_source.html#l00231. Accessed 7 July 2024.