Securing the Device Drivers of Your Embedded Systems: Framework and Prototype

Zhuohua Li
The Chinese University of Hong Kong
zhli@cse.cuhk.edu.hk

Jincheng Wang
The Chinese University of Hong Kong
jcwang@cse.cuhk.edu.hk

Mingshen Sun
Baidu X-Lab
sunmingshen@baidu.com

John C.S. Lui
The Chinese University of Hong Kong
cslui@cse.cuhk.edu.hk

ABSTRACT

Device drivers on Linux-powered embedded or IoT systems execute in kernel space thus must be fully trusted. Any fault in drivers may significantly impact the whole system. However, third-party embedded hardware manufacturers usually ship their proprietary device drivers with their embedded devices. These out-of-tree device drivers are generally of poor quality because of a lack of code audit. In this paper, we propose a new approach that helps third-party developers to improve the reliability and safety of device drivers without modifying the kernel: Rewriting device drivers in a memory-safe programming language called Rust. Rust's rigorous language model assists the device driver developers to detect many security issues at compile time. We designed a framework to help developers to quickly build device drivers in Rust. We also utilized Rust's security features to provide several useful infrastructures for developers so that they can easily handle kernel memory allocation and concurrency management, at the same time, some common bugs (e.g. use-after-free) can be alleviated. We demonstrate the generality of our framework by implementing a real-world device driver on Raspberry Pi 3, and our evaluation shows that device drivers generated by our framework have acceptable binary size for canonical embedded systems and the runtime overhead is negligible.

CCS CONCEPTS

• **Security and privacy** → *Security in hardware*; *Embedded systems security*;

KEYWORDS

Rust, Device Drivers, Linux Kernel

ACM Reference Format:
Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2019. Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES 2019) (ARES '19), August 26–29, 2019, Canterbury, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3339252.3340506

ARES '19, August 26–29, 2019, Canterbury, United Kingdom

ACM ISBN 978-1-4503-7164-3/19/08…$15.00
https://doi.org/10.1145/3339252.3340506

1 INTRODUCTION

Modern computer systems are often connected with many peripheral devices (like mouses, keyboards, wireless adapters and USB flash drives), or integrate special hardware infrastructures (like SGX/TrustZone). Operating systems usually contain device drivers which control the hardware connected to computers, and provide an universal well-defined interface for userspace applications to interact with different devices. This way, applications do not need to know about the low-level details, and can reuse the hardware manipulation code in device drivers.

Unfortunately, device drivers have become one of the most significant source of complexity and vulnerability for computing systems. Device drivers account for more than 60% of the code base of Linux kernel[1]. Recent study also shows that device drivers are more vulnerable than other code in the kernel[3][12][9][2]. What further complicates the situation is that the monolithic architecture of Linux kernel implies that drivers will run with kernel permissions and can do anything it wishes. This situation becomes worse for out-of-tree third-party drivers because they may not receive enough scrutiny and generally have lower quality. In 2016, Google reported that about 85% of the Android kernel bugs happened in vendor drivers[13]. Numerous research projects were proposed to tackle this problem, such as hardware-based isolation[14], software-based isolation[11], language-based isolation[17], microkernel architecture[8][6], user-space device drivers[1], etc. However, although these techniques protect device drivers against common exploits, they either need to modify the kernel or require extra hardware support. Some of these proposed methods even have unacceptable overheads. In this paper, we propose a new approach to protect device drivers: Using a safe programming language called Rust. The idea is to leverage the Rust compiler's powerful type system to prevent device driver developers from making mistakes, which become attack vectors exploited by hackers. Since most of the security checks are done at compile time, many errors can be detected before drivers are deployed, meanwhile, this method introduces negligible runtime overhead and requires no modifications of the kernel.

Rust is a strongly-typed programming language that focuses on memory-safety, concurrency and performance. Rust enforces most

[1]We ran `cloc` against Linux 4.19.27, `drivers/` directory contributes 12074795 out of totally 18911993 lines of code.

of its safety guarantees at compile time, resulting in little runtime overhead comparing with other memory-safe languages like Java or C#, which need to manage memory at runtime (i.e. garbage collection). Rust is also designed for safe system-level programming. Firstly, Rust has a unique ownership system that prevents aliasing, hence avoids potential bugs such as use-after-free. Secondly, the borrow checker in Rust keeps track of all the references to make sure they are always valid, eliminating dangling pointers and many other memory corruptions. Thirdly, Rust does not need to perform garbage collection for memory management, instead, it records the lifetime of each allocation and statically determines when to deallocate resources. If implemented correctly, Rust can achieve comparable performance to C/C++. With all of the above features, Rust becomes a promising language to carry out high-performance low-level system programming tasks.

In this paper, we introduce our work that leverages Rust's security features for Linux device driver development. We propose and build a framework that integrates the Rust programming language with the Linux kernel build system, and automatically re-exports kernel functions and data structures. We also reimplement parts of the Rust standard library so that device driver developers can take advantage of Rust's safety guarantees to manage kernel memory and handle synchronizations. We provide a general but concise interface for device driver developers to quickly build safe device drivers in Rust. We also implement a real-world device driver to demonstrate the capabilities and evaluate its performance.

We have open-sourced our source code[2] on GitHub. Our work is partially based on some previous efforts towards integrating Rust into kernel programming. The files `kernel-cflags-finder`, `src/printk.rs` and `src/c_types.rs` are from Geoffrey Thomas and Alex Gaynor's `linux-kernel-module-rust` project[15].

In summary, our contributions are listed as follows:

- **Methodology:** We propose a new approach to mitigate common security issues in Linux device driver development. Our approach is of great benefit for third-party hardware manufacturers to develop reliable and safe device drivers with low runtime overhead, without modifying the Linux kernel.
- **Implementation:** We implement a framework that assists developers to build Linux device drivers in Rust. The framework integrates the package manager of Rust (i.e. Cargo) and the build system of the Linux kernel (i.e. Kbuild), and provides several useful libraries for developers to easily manage kernel memory and handle synchronizations, etc. We also implements a real-world device driver for LAN9512, a USB to Ethernet controller, to show that our framework is capable enough to deal with requirements in reality.
- **Security:** We discuss common security issues in Linux device drivers and demonstrate that Rust can help programmers to either eliminate or mitigate several vulnerabilities during the development.
- **Evaluation:** We design and perform several benchmarks to test the device drivers that we implement based on our framework. The results show that the Rust compiler is able to generate kernel modules with acceptable binary size for embedded systems, and writing device drivers in Rust incurs negligible runtime overhead.

[2]https://github.com/lizhuohua/linux-kernel-module-rust/

2 BACKGROUND

In this section, we present the background knowledge of Linux device drivers and the Rust programming language, and show how Rust can help to solve some security bugs in device drivers using its security features.

2.1 Linux Device Drivers

Device drivers are an important software component that provide an interface between the operating system and hardware devices. In general, a driver is responsible for three tasks: (1) configure and manage one or more devices; (2) convert requests from the kernel into requests to hardware; and (3) deliver the results from hardware to the kernel.

Although the Linux kernel is monolithic, it is able to extend its functionalities at runtime, by dynamically inserting loadable kernel modules. A kernel module is an object file and it will be linked with the kernel when it is inserted. Many third-party vendors take advantage of this feature to distribute their device drivers as individual kernel modules instead of merging the source code to the mainline kernel. As stated in the previous section, this form of device drivers introduces many security threats. Therefore, in this paper, we only address the case where drivers are compiled as kernel modules.

For the Linux kernel, all devices can be classified into three fundamental types[4]: *character* devices, which are byte-stream oriented (like a file); *block* devices, which support random accesses; and *network* devices, which manipulate streams of packets. Figure 1 depicts the general architecture of Linux device drivers. Device drivers play a role in communicating with hardware and hence applications only need to interact with the system call interface provided by the kernel. This way, although there are many different devices in practice, they all implement an unified interface provided by the kernel so that the differences between them are transparent to users. Writing a device driver involves defining several "*callback functions*" for the targeted device and registering them to the kernel. When the kernel needs to operate this device, it will invoke the corresponding callback functions. Also, developers often make use of functions and data structures defined in the kernel, and build device drivers on top of other modules or subsystems.

However, writing a "safe" device driver is non-trivial. The monolithic architecture means the entire kernel, both the core kernel and device drivers, runs in the same address space with the same privilege level. This implies that there is no mechanism that can prevent a driver from mistakenly invoking kernel functions or manipulating critical kernel memory, which may cause a kernel panic. Comparing with micro-kernels, this architecture is more efficient but it also exposes a larger attack surface. It forces users to trust all the device drivers, including those provided by third-party vendors. As a result, any bug in a device driver can lead to severe vulnerabilities, which may further compromise the entire system.

Typical bugs that happen in device drivers include: incorrect boundary checks, null pointer dereference, information leakage,

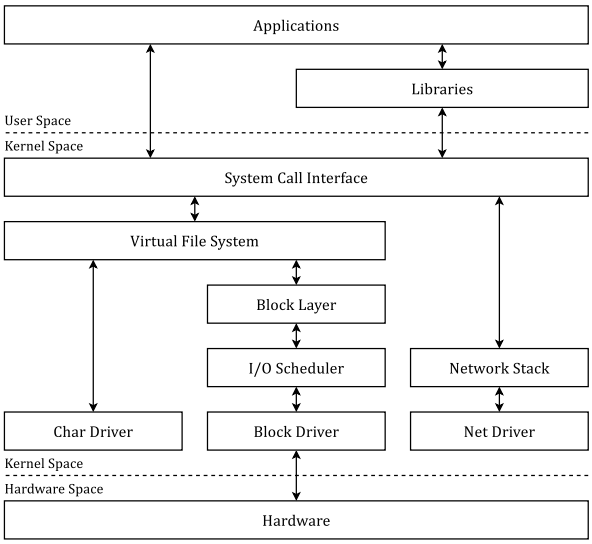

Figure 1: General Architecture of Linux Device Drivers

use-after-free, etc. Many of them can be easily exploited by attackers. These bugs can be attributed to the fact that Linux is written in C, an unsafe programming language[2]. This fact motivates us to come up with a way to take advantage of a safe programming language for device driver development.

2.2 The Rust Programming Language

As a new language, Rust integrates both practical experience and research outcomes of C/C++ and ML/Haskell in the past several decades, and has attracted attention from both academia and industry. The most unique features that truly distinguish Rust are the "*ownership*" system and *traits*. The former enables Rust to guarantee memory safety without performing garbage collection, and the latter declares a set of methods that a type must implement.

2.2.1 The Ownership System. The main idea of the ownership system is derived from concepts of "linear logic"[7] and "linear types"[16], which means all values must be used exactly once. In such a system, neither reference counting nor garbage collection is needed because linear resources would not be duplicated or discarded. It also enables safe parallel computation because shared resources do not exist. However, the linear type system is too constrained to be practical. Therefore, Rust uses the concept of ownership, which relaxes the constraint of pure linearity.

Under the ownership system, each value (e.g., an integer on the stack) has an unique owner. The value is destroyed when the owner of the value goes out of scope. Ownership can be moved (transfered) between variables. Once a value is moved, it is no longer accessible from the original variable binding. Rust also allows references that temporarily borrow a value without invalidating the original binding. The behavior of references is restricted. On the one hand, to prevent dangling pointer bugs, the lifetime of a reference cannot exceed the value that it points to. On the other hand, it is safe to have more than one reference to a value, as long as the value is being read; when the value is written, only one single reference is allowed to exist.

The following code snippet shows the rules we mentioned above. (a) Line 4 defines a string whose ownership is binded to the variable `x`. After assigning `x` to `y` at line 5, the ownership is moved to `y` and the original binding `x` no longer has access to the string value at line 7. (b) Line 8 and line 9 show that it is not allowed to define multiple mutable references. (c) `y` is constrained to live in a scope that is defined by a pair of curly brackets. At line 13, `y` is invalid since it goes out of the scope. (d) `r` is a reference to `y`, reading `r` outside the scope of `y` is an error because the lifetime of `y` has ended and `r` becomes a dangling pointer at line 14. Note that the above mentioned problems can be checked at compile time to reduce security faults.

```
1  fn main() {
2      let r;
3      {
4          let mut x = String::from("hello");
5          let mut y = x;      // NOTE: x moved here
6          println!("{}", y); // OK
7          println!("{}", x); // ERROR: use of moved value x
8          let r1 = &mut y;
9          let r2 = &mut y;    // ERROR: cannot borrow y as mutable
10                             // more than once
11         r = &y;
12     }                       // NOTE: the scope of y ends here
13     println!("{}", y);      // ERROR: y is out of the scope
14     println!("{}", r);      // ERROR: borrowed value y does not live
15                             // long enough
16 }
```

In summary, Rust's ownership system eliminates aliasing, and guarantees that every reference is valid. This prevents many memory corruption bugs like double-free and use-after-free. Note that developers gain more benefits from this design: since there is only one owner for each value, when the owner is out of scope, the compiler can ensure that it is safe to deallocate this value. Therefore, no garbage collection is needed. Also, since mutable reference is unique, race condition is also avoided because it is impossible to have more than one thread accessing the same variable.

2.2.2 Trait. Traits define interfaces for types. Different from the canonical object-oriented programming (OOP) languages (e.g. C++) that use inheritance to derive properties between classes, Rust uses traits to define shared behaviors in an abstract way. A trait defines a group of method signatures to depict a set of behaviors necessary to accomplish some purpose. Implementing a trait on a type requires the programmer to provide all the definitions of these methods. This mechanism enables generic programming: a generic function can accept different types of arguments as long as they implement the corresponding traits. Trait mechanism can be regarded as another design pattern that changes the way people construct their code. Comparing with the traditional OOP design pattern, traits not only can achieve generic programming, but also provide additional information for the compiler to perform type checking. In this paper, we take advantage of this feature to define a unified interface for each type of device drivers.

3 DESIGN

In this section, we first present the architecture of our system. Then we discuss some essential components and their design principles.

3.1 Architecture

We build a framework to empower developers to write efficient and safe device drivers using the Rust programming language. The main task that our system needs to accomplish is to utilize Rust's compiler and type system to develop safe kernel modules. The following challenges have to be considered: (1) How to instruct the Rust compiler to generate bare-metal machine code that can run in kernel space; (2) How to invoke functions defined inside the kernel; (3) How to link Rust code with the kernel and generate kernel modules.

Figure 2 illustrates the architecture of our system. The framework consists of a series of Rust source files that implement several useful infrastructures (such as kernel memory allocator), and provide an API for Rust device driver development. Developers can use the API by simply importing the framework library in their device driver projects. A JSON specification file is used to create a bare-metal target for the Rust compiler. Some Rust's own libraries also need to be recompiled so they can be used in kernel space. Kernel functions and data structures are re-exported using `bindgen`. The Rust compiler compiles both the framework library and the developer's library, and generates an object file. The build system then takes charge of the linking process, during which the object file is linked as a kernel module.

Being a middleware between developers and the Linux kernel, our system solves the challenges we mentioned above: (1) The build system instructs the Rust compiler to generate bare-metal code and recompiles a series of pre-compiled libraries so that they can be used in kernel space; (2) By using `bindgen` to transform Linux kernel's C interface into Rust's foreign function interface (FFI), Rust programs are able to invoke C functions defined in the kernel; (3) The Kernel Build System (Kbuild) is integrated to construct kernel modules (*.ko).

3.2 Interface of Device Drivers

In our framework, each type of device drivers is modeled as a trait, which strictly stipulates the interface that every driver of this type has to implement. Developers should define their own struct to represent a device driver, then implement the corresponding trait for this struct. Using this design pattern, every type of device drivers has the same interface so that they can be easily managed using a unified framework. This provides a more elegant interface for developers to implement their drivers, and reduces some boilerplate code such as device driver registration and unregisteration.

A device driver usually consists of two classes of code: (1) the device's logic that is implemented within the driver; (2) the interactions between the driver and the Linux kernel. The former should be written by developers in pure Rust and the latter is provided by our framework through FFI bindings. Developers only need to invoke the corresponding routines to communicate with the kernel.

3.3 Tools and Infrastructures

To follow the conventional Rust coding style and provide more useful tools for kernel programming, parts of the Rust standard library are reimplemented to be fitted into the Linux kernel. The memory allocator is implemented based on Rust's *Global Allocator*, which allows programmers to customize the default memory allocator

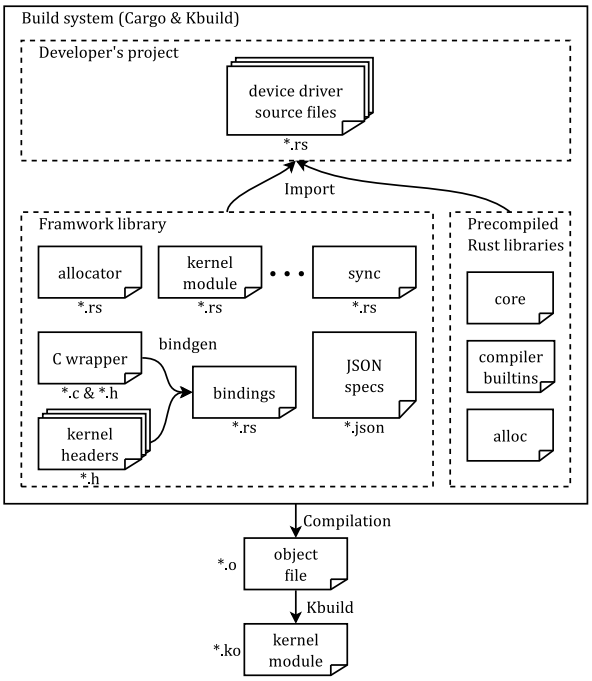

Figure 2: The Architecture

and redirect all the memory allocation requests to it. This way, types defined in the `core` library which need memory allocation (e.g. `Box<T>` and `Vec<T>`) still work in kernel space. Also, based on the synchronization primitives provided by the kernel, our framework implements `Mutex` and `Spinlock` imitating the Rust standard library. Since all the definitions of these tools mimic the standard library, proficient Rust programmers will not find it hard to use our framework.

4 IMPLEMENTATION

In here, we discuss our implementation. Section 4.1 presents the technical details on how Rust is integrated into the Linux kernel build system. This lays the foundation for Section 4.2, in which we details the implementation of the interface and infrastructures of our framework.

4.1 Build System

The build system of our project integrates Rust with the Linux kernel. The integration involves two technical aspects. First, due to the monolithic design of the Linux kernel, device drivers, as a part of the kernel, are unable to use libraries that exist in the user space (e.g. the Rust standard library) or perform operations that are forbidden inside the kernel (e.g. floating point operations). Therefore, the default target of the Rust compiler needs to be changed to generate statically linked and OS-independent machine code without using floating point instructions. Second, to leverage the existing data structures and functions defined inside the kernel, the kernel

headers need to be translated into Rust bindings. Our system includes the scripts and makefiles to automate the whole procedure.

4.1.1 Compile OS-independent Rust code. By default, the Rust compiler automatically links the standard library and the startup routine of the C runtime. While this default behavior is proper in user space, it does not work in kernel space, where the only available "library" is the kernel itself. Therefore, we enhance the Rust compiler, to generate code that runs directly on hardware, without relying on the standard library. This is done by adding a new target to the Rust compiler.

Rust supports defining new targets by using JSON specification files, which describe the properties of a compilation target, for example, its architecture, its operating system and the default linker. We create a custom target based on the default specification of a built-in target[3], and modify it to match the needs. The main modification includes disabling dynamic linking and the use of floating-point hardware, etc.

Note that the Rust compiler is shipped with a collection of precompiled libraries (`core`, `compiler_builtins`, `alloc`). To use these libraries for the new target, they need to be recompiled. The `core` library provides necessary Rust types and structs such as `Result` and `Option`. The `compiler-builtins` library contains compiler intrinsics that LLVM (the codegen backend of the Rust compiler) may call when generating machine code. The `alloc` library provides smart pointers and collections for managing heap-allocated values. There is an existing tool called `cargo-xbuild`[4] that can automatically cross-compile those libraries.

4.1.2 Generate bindings for kernel headers. The Linux kernel defines a large number of functions and data structures. Usually, kernel developers are recommended to reuse the code by including the kernel headers, which cannot be directly read by the Rust compiler. To solve this problem, our system re-exports the symbols defined in the kernel headers by generating Foreign Function Interface (FFI) bindings so that developers can call external C function from Rust. While writing bindings by hand allows us to create more elegant APIs, we prefer to automatically generate them for two reasons:

(1) The amount of kernel functions is too large so manually writing each FFI binding for all the functions is not practical.
(2) Linux kernel does not guarantee ABI stability, meaning that the data structures may vary in different versions of the kernel.

We use the `bindgen` tool, which takes the kernel headers as input and outputs the Rust version type definitions and function declarations. It invokes `clang` to construct the abstract syntax tree (AST) of the C headers, then translates them to Rust. In order for `clang` to correctly handle kernel headers, we extract the compiler flags from Kbuild. One drawback of using `bindgen` is that it cannot generate bindings for inline functions or macros because they will be expanded during compilation and hence do not generate any symbols. As a result, there is no way Rust can call into them. One can resolve this problem by manually writing C functions that wrap up inline functions or macros.

4.1.3 Implement various Rust primitives. The standard library defines several *language items*[5] that are essential for the compiler to work properly. For example, the panic handler function defines the behavior when a program crashes. Without these, the compiler is unable to generate the corresponding machine code. Consequently, we need to implement these ourselves. We do not implement any recovery mechanism in the panic handler since it is invoked when a device driver goes terribly wrong and is unlikely to recover. Instead, we assert failure using the `BUG` macro defined in the kernel, which outputs the contents of registers and the stack trace, then stops the current process.

4.2 APIs and Infrastructures

We design and implement an interface for developers to quickly build device drivers in Rust. Several infrastructures like the kernel memory allocator and synchronization primitives are re-implemented to utilize Rust's security guarantees.

4.2.1 Interface. Device drivers source code often includes boilerplate code such as driver registration and initialization. These common behaviors can be summarized as a unified interface for all device drivers. In our system, each type of device drivers is modeled as a *trait*, which stipulates an interface for this type of drivers. The following code snippet presents the definition of the `CharDevice` trait. It enforces that every character device driver must implement an initialization function `init()` which takes a string as the name of the driver, and a destruction function `cleanup()`.

```
// For character device drivers
trait CharDevice: Sized {
    fn init(name: &'static str) -> KernelResult<Self>;
    fn cleanup(&mut self);
}
```

Driver developers need to implement their drivers according to this interface. By using *generic programming*, our framework can easily manage different drivers because they have the same interface. The boilerplate code can then be reduced.

4.2.2 Kernel Allocator. To allocate kernel memory in Rust, we re-implement the default allocator using the `GlobalAlloc` trait by wrapping `kmalloc()` and `kfree()` defined in the kernel. This way, all the heap allocation requests will be routed to our customized allocator. The compiler automatically requests memory when programmers use types that need memory allocation, such as `Box<T>`. `Box<T>` is a smart pointer type which contains a generic type `T`, it can be used to allocate kernel memory for any other types. Then Rust's ownership system is able to keep track of the lifetime of the allocated memory and automatically deallocate it when its lifetime ends. Therefore, developers are liberated from delicate and error-prone kernel memory management. The following code shows an example of allocating an array in kernel memory.

```
fn allocation_example() {
    let v = Box::new([1, 2, 3]);  // Allocate kernel memory using kmalloc()
    println!("v = {:?}", v);      // Output "v = [1, 2, 3]"
    // Here, v is automatically deallocated using kfree()
}
```

[3]Use command `rustc -Z unstable-options --print target-spec-json`
[4]https://crates.io/crates/cargo-xbuild

[5]Some pluggable operations or functionalities that are not hard-coded into the language, but are implemented in libraries.

4.2.3 Synchronization Primitives. In order to support synchronizations in kernel space, we also implement the synchronization primitives `Mutex` and `Spinlock` mimicking the Rust standard library. The underlying lock primitives are the `spinlock` and `mutex` provided by the kernel. By design, `Mutex` and `Spinlock` contain a pointer to the protected data, but do not provide any interface to access the data. The only method they have is called `lock()`, which returns a `MutexGuard` or `SpinlockGuard`, respectively. These two guard types implement the `Deref` and `DerefMut` traits so they can be dereferenced to get a either mutable or immutable reference to the shared data. Therefore, developers cannot access shared data unless they explicitly invoke the `lock()` method. This eliminates the case where developers forget to lock the data before using it. Also, by leveraging the ownership system, the compiler will help us release the lock once the guard is not used anymore.

The following example shows the use of `Spinlock` to achieve mutual exclusion. `GLOBAL` is a `Spinlock` that encloses shared data inside. The shared data can be only accessed through `global`, a `SpinlockGuard` returned from the `lock()` method. This guarantees that shared data is only accessed when the mutex is locked. When `global` goes out of scope, the lock is released automatically.

```
lazy_static! {
    static ref GLOBAL: Spinlock<i32> = Spinlock::new(0);  // Shared data
}

fn synchronization_example() {
    let mut global = GLOBAL.lock(); // Lock the data
    *global = 1;
    // Here, the lock is released automatically
}
```

Three benefits are gained from this design: First, `Spinlock` locks data, instead of the control flow. Second, developers are required to lock the data before using it because `Spinlock` does not provide any interface to touch the data. Otherwise, the compiler will throw an compilation error. Third, thanks to the ownership system, developers do not need to remember to release locks, since they will be automatically unlocked when they go out of scope.

5 COMMON BUGS IN DEVICE DRIVERS

In this section we first present and categorize some common bugs in Linux device drivers, then we illustrate how Rust can mitigate those bugs. We categorize bugs into three groups: (1) *Language-Specific Security Issues*: For this kind of bugs, we can solve them by using a high-level programming language. (2) *General Security Issues*: This kind of bugs are intrinsically inevitable, but the Rust compiler may provide extra help for developers. (3) *Logic Errors*: In this case, it is the responsibility of the developers.

To show where the bug is and how it is fixed, we use the `diff`[6] format to display the difference between the original code and the patched code. Security patched lines are in "green" and preceded by a "+" sign, while security problematic lines are in "red" and preceded by a "-" sign. We will not go into the details about how to exploit a bug due to page limit.

5.1 Language-Specific Security Issues

Many security issues can be ascribed to the use of unsafe programming languages like C. By using a safe programming language, this kind of bugs can be trivially solved. The following examples illustrate this kind of issues.

5.1.1 Array-based buffer overflow. The following code snippet shows the patch of CVE-2017-1000363. At line 9, the array `parport_nr` is indexed by an integer `parport_ptr` whose value is not carefully checked, which may cause a buffer overflow. This bug happens because in C, the boundary of an array is not checked. Even if adversarial inputs overflow the array, the program silently continues so it is possible for attackers to inject forged data or even hijack the control flow. At line 10, the index `parport_ptr` is checked before use, thus the bug is fixed.

```
diff --git a/drivers/char/lp.c b/drivers/char/lp.c
index 565e4cf..8249762 100644
--- a/drivers/char/lp.c
+++ b/drivers/char/lp.c
@@ -859,7 +859,11 @@ static int __init lp_setup (char *str)
        } else if (!strcmp(str, "auto")) {
            parport_nr[0] = LP_PARPORT_AUTO;
        } else if (!strcmp(str, "none")) {
-           parport_nr[parport_ptr++] = LP_PARPORT_NONE;
+       if (parport_ptr < LP_NO)
+           parport_nr[parport_ptr++] = LP_PARPORT_NONE;
+       else
+           printk(KERN_INFO "lp: too many ports, %s ignored.\n",
+       str);
        } else if (!strcmp(str, "reset")) {
            reset = 1;
        }
```

In contrast, Rust guarantees that any access to an array will be checked at runtime so this kind of vulnerability can be avoided. The following Rust example tries to read an out-of-bound value. Running this program directly will trigger the panic handler, stopping the execution immediately, so that there is no way for attackers to exploit buffer overflow to overread/overwrite memory.

```
let mut xs:[i32; 5] = [1, 2, 3, 4, 5]; // 5 elements in total
xs[5] = 6;                             // This will panic!
```

5.1.2 Using unsafe functions. Although it is well-known that some deprecated functions like `gets`, `strcpy`, and `sprintf` are dangerous and should be eliminated, they still occasionally appear in the kernel source code. CVE-2010-1084 shows an example of using `sprintf` at line 14, which does not check the length of the string it copies, resulting in memory corruption. The workaround replaces `sprintf` with `snprintf`, which is the safe version of the former with length checks.

```
diff --git a/net/bluetooth/sco.c b/net/bluetooth/sco.c
index f93b939..967a751 100644
--- a/net/bluetooth/sco.c
+++ b/net/bluetooth/sco.c
@@ -960,13 +960,22 @@ static ssize_t sco_sysfs_show(struct class *dev,
    struct sock *sk;
    struct hlist_node *node;
    char *str = buf;
+   int size = PAGE_SIZE;

    read_lock_bh(&sco_sk_list.lock);

    sk_for_each(sk, node, &sco_sk_list.head) {
-       str += sprintf(str, "%s %s %d\n",
+       int len;
+
+       len = snprintf(str, size, "%s %s %d\n",
            batostr(&bt_sk(sk)->src), batostr(&bt_sk(sk)->dst),
            sk->sk_state);
+
+       size -= len;
+       if (size <= 0)
+           break;
+
+       str += len;
    }

    read_unlock_bh(&sco_sk_list.lock);
```

[6] https://en.wikipedia.org/wiki/Diff

Instead of naively regarding contiguous sequences as arrays, Rust provides a higher level abstraction such as `str`, `slice` and `array`. They strictly define the behaviors when they are copied. For example, function `clone_from_slice` clones elements from one slice to another, and it panics if two slices have different sizes. Therefore, programmers will be alerted of potential bugs during driver development.

5.1.3 Uninitialized data. The kernel usually copies data from kernel space to user space. If the unused fields are not zeroed out, sensitive kernel information might be leaked to user space. CVE-2010-3876 shows that the struct member `sll_pkttype` is not initialized, potentially leaking the kernel stack contents to user processes. At line 9, the bug is fixed by initializing the struct member as zero.

```
1  diff --git a/net/packet/af_packet.c b/net/packet/af_packet.c
2  index 3616f27b..0856a13 100644
3  --- a/net/packet/af_packet.c
4  +++ b/net/packet/af_packet.c
5  @@ -1742,6 +1742,7 @@ static int packet_getname(struct socket *sock,
   ↪    struct sockaddr *uaddr,
6      sll->sll_family = AF_PACKET;
7      sll->sll_ifindex = po->ifindex;
8      sll->sll_protocol = po->num;
9  +   sll->sll_pkttype = 0;
10     rcu_read_lock();
11     dev = dev_get_by_index_rcu(sock_net(sk), po->ifindex);
12     if (dev) {
```

Note that the above error will not happen in Rust because Rust's memory-initialization check guarantees that memory is always initialized. Therefore, uninitialized data does not exist.

5.1.4 Incorrect kernel memory management. Similar to the dynamic memory allocation using `malloc()` and `free()` in user space, the Linux kernel also has a general-purpose memory allocator called `kmalloc()` and `kfree()`. The device driver developers are responsible for managing kernel memory, which is delicate and error-prone. Many vulnerabilities are caused by incorrect kernel memory management, such as null-pointer dereference, memory leakage, double-free, and use-after-free.

CVE-2018-8087 is an example of memory leakage caused by missing deallocation. The bug is fixed by adding a `kfree()` at line 11. This example also reflects the fact that manually managing kernel memory is potentially dangerous and becomes a burden of developers.

```
1  diff --git a/drivers/net/wireless/mac80211_hwsim.c
   ↪    b/drivers/net/wireless/mac80211_hwsim.c
2  index 6bf063a..66c2ac0 100644
3  --- a/drivers/net/wireless/mac80211_hwsim.c
4  +++ b/drivers/net/wireless/mac80211_hwsim.c
5  @@ -3197,8 +3197,10 @@ static int hwsim_new_radio_nl(struct sk_buff *msg,
   ↪    struct genl_info *info)
6      if (info->attrs[HWSIM_ATTR_REG_CUSTOM_REG]) {
7          u32 idx = nla_get_u32(info->attrs[HWSIM_ATTR_REG_CUSTOM_REG]);
8
9  -       if (idx >= ARRAY_SIZE(hwsim_world_regdom_custom))
10 +       if (idx >= ARRAY_SIZE(hwsim_world_regdom_custom)) {
11 +           kfree(hwname);
12             return -EINVAL;
13 +       }
14         param.regd = hwsim_world_regdom_custom[idx];
15     }
```

The above problem does not exist in Rust because Rust's ownership system ensures that each allocation will eventually be freed as long as it is not used anymore. So developers do not need to explicitly deallocate any resources, it will be released automatically when it goes out of scope. Thus, if implemented correctly, memory leakage, double-free and use-after-free should not happen in Rust. Also, since the ownership system ensures that a reference cannot outlive the value it points to, hence bogus references such as null references do not exist.

5.2 General Security Issues

In this section we discuss some general security issues in device driver programming. They are fundamentally inevitable but Rust does provide some help to mitigate some of them.

5.2.1 Integer Overflow. An integer overflow[5] happens when the result of an arithmetic operation does not fit into the fixed size integer. The result can be too small or too big to be representable with a given number of digits. For Rust, integer overflow is checked at runtime. However, for performance considerations, Rust will only perform overflow checks for debug builds by default. That means Rust can still help developers to find and mitigate integer overflows during the development stage. In release builds, overflow checks are disabled to achieve better performance.

5.2.2 Concurrency. There are more causes of concurrency in kernel space than in user space[4]. First, modern CPUs usually have multiple cores to support symmetric multiprocessing (SMP), so kernel code may be executed simultaneously on different processors; Second, kernel code is preemptible, meaning that driver code may lose CPU at any time, and the process that replaces it may also run in the same driver code; Third, device interrupts occur asynchronously, causing concurrent execution of driver code. All of the above lead to the data race problem.

Rust claims itself to be thread-safe, however, Rust can not prevent general race conditions because this is fundamentally impossible. After all, hardware resources are intrinsically shared. In user space, Rust uses threads to run code concurrently and uses the ownership system to prevent shared mutable states between threads. However, this does not work in the kernel space because the kernel itself is preemptible, in other words, every piece of kernel code is potentially shared by all the user processes. This is agnostic to the Rust compiler since in the compiler's point of view, it just compiles a single-threaded program. As a result, Rust cannot prevent kernel concurrency issues as easy as in the user space. Luckily, Rust's rigorous type checking provides some help.

Device driver developers still need to carefully determine whether a variable might be accessed simultaneously and thus it needs explicit concurrency management. In our system, we implement two synchronization primitives: `Spinlock` and `Mutex`, which mimic the behaviors of Linux kernel's `spinlock` and `mutex`.

5.3 Logic Errors

Developers should be responsible to minimize and handle any logic error during the program development. Nevertheless, they still benefit from Rust's security features.

5.3.1 Deadlock. A deadlock is a state in which a group of locks are waiting for each other and no one is able to proceed. It happens because of incorrect concurrency management. Although Rust claims that it guarantees thread safety, it does not consider deadlocks unsafe, because deadlocks cannot be statically prevented. Also, unlike Go which detects deadlocks at runtime, Rust tries to keep its runtime as simple as possible to guarantee performance. As a result,

developers must be careful to avoid deadlocks in their software development.

5.3.2 Error Handling. Failures are common in device drivers, but if handled properly, most of them can be detected and resolved without leading to kernel panics. Nevertheless, it is always challenging to do error handling. Firstly, programmers tend to focus on functionalities instead of carefully considering all possible cases of errors; Secondly, missing error handling may not be noticed until the product has been shipped to customers and some disasters have happened. Thirdly, programmers usually use inconsistent and implicit placeholders as return values (e.g. 0 for success and -1 for failure). Rust's `Result` type provides an elegant way to handle errors. The type `Result<T, E>` is an enum with two possible variants, `Ok(T)`, which represents success and contains a value of type `T`, and `Err(E)` which indicates an error of type `E`. Functions with return type `Result` are able to return either a value or an error. Therefore, Rust helps developers to be more disciplined in the code development.

Traditionally, C handles errors by explicitly checking the return values, which is clumsy and error-prone. In the following C code snippet, `smsc95xx_read_reg` loads the content of a register into a buffer `read_buf`. If failure happens, this function returns a negative integer. Line 2 explicitly checks whether this function succeeds or not.

```
1  // C style error handling
2  int ret = smsc95xx_read_reg(dev, COE_CR, &read_buf);
3  if (ret < 0)
4      return ret;
```

Note that it is a common problem that programmers may forget to check the return value, thus fail to handle the error. Rust stipulates that to get the underlying value contained in a `Result`, the error case must be handled. The above code may be rewritten in Rust as follows. The `ret` variable is of type integer if and only if the `Result` is consumed by the `match` statement. And the Rust compiler will check whether the `match` statement handles both the success case and the error case.

```
1  // Rust style error handling
2  let ret = match smsc95xx_read_reg(dev, COE_CR) {
3      Ok(val) => val,
4      Err(e) => return e,
5  };
```

6 EVALUATION

This section we report our experiment on evaluating the runtime cost of Rust device drivers. We build a driver for LAN9512, a USB 2.0 to 10/100 Ethernet Controller. The device used in our experiment is Raspberry Pi 3, which runs Linux kernel 4.19.29 provided by Raspberry Pi Foundation. The kernel is compiled by ourselves using `clang`. We want to address the following questions:

- Does Rust generate huge kernel modules?
- What is the performance impact of our system on real-world device drivers?

6.1 Binary Size

Since Rust is designed for system programming and its runtime is deliberately kept minimal, it should be able to generate binaries with acceptable sizes for embedded devices. To demonstrate the code size generated by the Rust compiler, besides the real-world driver, we also implement several "toy" drivers in both C and Rust. All the Rust versions enable the *release build* and *link-time optimization*, and C versions are compiled according to the default kernel configuration used by Raspberry Pi 3. Table 1 shows the binary size comparison of these kernel modules.

Table 1: The Size of Kernel Modules (Unit: Byte)

Driver Name	C Version	Rust Version	Overhead
hello_world	3632	4252	17.07%
yes	5340	22312	317.83%
sysctl	4804	20288	322.31%
smsc9512	10880	34224	214.56%

Table 2: Transmission Bandwidth (Unit: KBytes/sec)

Attempts		1	2	3	4	5
Rust	Send	11554	11555	11555	11555	11554
	Receive	11469	11470	11469	11470	11468
C	Send	11554	11561	11555	11555	11561
	Receive	11469	11470	11468	11468	11469

Except for the simplest "hello world" driver, kernel modules written in Rust are generally 2-3 times larger than the corresponding C versions. After examining the contents of the ELF sections, we found that the large size mainly comes from the linkage of Rust libraries, for example, the `core` library. And it can be predicted that the binary size will be even larger if more libraries are linked. Although this may be an issue in systems with restricted resources, it is important to note that the binary size is still within "*several kilobytes*", and this is usually acceptable for modern devices that run on Linux.

6.2 Performance

We evaluate the runtime performance of the LAN9512 Ethernet controller driver. For simplicity reasons, we only implement necessary functions and omit the functionalities that are unrelated to performance tests such as power management, suspend/resume, and hardware initialization. We connect the Raspberry Pi to a desktop through an Ethernet cable. Both of them install a network speed test tool called `iPerf`[7]. Note that, different from common performance evaluation methods, we do not test the execution time since device drivers are usually not CPU-bound. Instead, we use `iPerf` to measure the bandwidth, jitters and packet loss rate of the network connection.

6.2.1 Transmission Bandwidth. Table 2 gives the bandwidth measured by `iPerf`. Note that there is little difference between C and Rust because both of them have reached the maximum bandwidth. This reflects the fact that most device drivers are simply an interface between hardware and the kernel, so that usually they are not the performance bottleneck.

[7] https://iperf.fr/

6.2.2 Jitters. Jitter quantifies the differences between consecutive transmit times. It measures the stability of network delays. We set the target bandwidth by sending packets in different rates. Figure 3 presents the change of jitters along with different bandwidths. As the figure shows, in general the C version has lower jitters, while the difference between C and Rust is quite small. Rust is even better in some cases (bandwidth = 60 and 70).

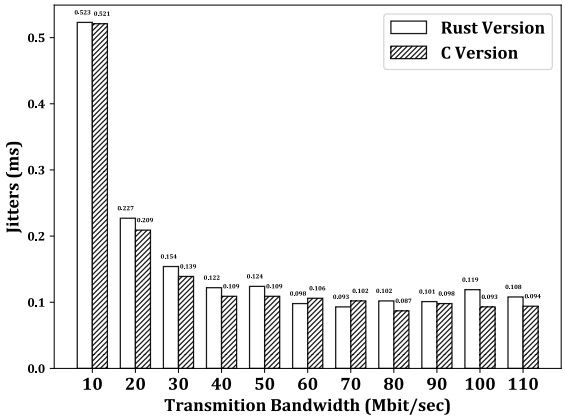

Figure 3: Jitters

6.2.3 Packet Loss Rate. When the sending rate exceeds the capacity of the receiver, packet loss will happen. `iPerf` also measures the packet loss rate for different bandwidths. From Figure 4, again, there is no significant difference between C and Rust.

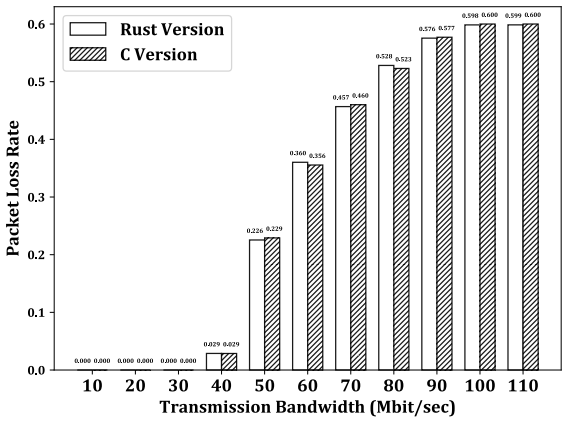

Figure 4: Packet Loss Rate

6.2.4 Summary. The above performance evaluation shows that the difference between C and Rust is negligible. This is expected because Rust is designed for low-level programming and most of its security checks are done at compile time. We conclude the Performance evaluation results in the following aspects: (1) The Rust compiler is capable enough to generate machine code with comparable performance to the C compiler. (2) Device drivers usually work as an interface between the kernel and hardware, so they generally are not the performance bottleneck in real-world applications.

7 DISCUSSION

What can a safe programming language solve? Based on our experience of writing device drivers in Rust, we discuss what kind of problems it can solve for kernel programming. Firstly, language-based security issues such as array-based buffer overflow and unsafe string manipulation, can be trivially solved, because Rust provides higher-level language primitives such as arrays with bounds-checking. Secondly, Rust's type system makes programmers more disciplined. The compiler can help in detecting many errors at compile time, for example, passing arguments with wrong types, accessing shared data without locking the corresponding mutex, etc. Thirdly, Rust reduces the burden of memory management: If a chunk of memory is allocated through Rust's allocator, and it is only used within the driver module, then the Rust compiler is able to trace the lifetime of it and perform deallocation automatically.

What can it NOT solve? There are also several cases that Rust cannot tackle. Firstly, since the kernel is written in C, each invocation of kernel functions must be unsafe. These invocations are inevitable because device drivers must interact with the kernel. Although we encapsulate commonly used kernel functions and provide a safe API, this approach is not sustainable because of the large quantity of kernel functions. Secondly, low-level hardware manipulations are intrinsically unsafe because hardware resources are naturally shared and cannot fit in with the ownership system. Thirdly, Rust cannot in general prevent deadlocks, race conditions and integer overflow. These should be the responsibility of the developers. Finally, most logic errors are due to programmers carelessness, so they cannot be detected by the compiler.

Some limitations of our project. Firstly, even if our framework is used, writing device drivers may require some level of unsafe code. Because (1) writing C-style callback functions, (2) calling kernel functions, (3) manipulating raw pointers are quite common in device driver development, but all of them are unsafe operations in Rust. Passing raw pointers to the kernel makes the compiler lose track of the lifetime and ownership, meaning that memory corruptions like use-after-free are still possible. Secondly, our provided APIs are not general enough to work for every scenario. For example, `Mutex` and `Spinlock` do not work if the kernel requires a pointer of a lock. Thirdly, we only evaluate the performance on LAN9512 Ethernet controller, whose maximum throughput is only 100 Mbps. Finally, we have not devised a failure recovery mechanism, once an error happens, the kernel handler directly stops the current process and prints out a stack trace. This will be our future work.

8 RELATED WORK

8.1 Existing Projects about Rust Kernel Module

The first known attempt of writing kernel modules in Rust is a Taesoo Kim's GitHub repository *rust.ko*[10]. The first commit is in 2013, long before Rust's first stable version was released. Although it is just a toy example, it shows that Rust is able to carry out kernel programming. Geoffrey Thomas and Alex Gaynor's *linux-kernel-module-rust*[15] is a more recent project which has tremendous improvements. It uses `bindgen` to generate bindings. It also tries to provide some safe abstractions for kernel programming, such as a safe interface for user space pointers. Our work depends on Geoffrey Thomas and Alex Gaynor's `kernel-cflags-finder` script which extracts necessary compiler flags from Kbuild in order for `bindgen` to work.

8.2 ixy.rs

Ixy[8] is a userspace network device driver for educational purposes. There are two main features of this project: (1) The whole driver consists only about 1000 lines of C code; (2) It controls network adapters and implements packet processing totally in userspace. Ixy.rs[9] is a Rust reimplementation of the original Ixy. By leveraging the safety guarantees provided by Rust, it achieves high security properties. However, since the driver resides in userspace, it cannot handle interrupts which can be done in kernel space, meaning that only poll-mode is supported. This will consume too much CPU resources and hence it is not suitable for real-world device drivers.

8.3 SafeDrive

SafeDrive[17] consists of a framework that can dynamically detect type violations in Linux device drivers using language-based methods and a recovery system that helps recover from failures. SafeDrive automatically generates runtime checks according to each function's annotations. Although it provides a simple inference mechanism, programmers still need to annotate kernel headers by hand. Also, this project only focuses on bound violations, it does not check other memory safety violations like dangling pointers.

9 CONCLUSION

In this paper, we design and implement a framework for device driver developers to build Linux kernel modules in Rust. We integrate Rust's package manager (Cargo) and Linux's build system (Kbuild), and reimplement parts of the standard library to provide useful tools and infrastructures for developers. We give examples of common security bugs in kernel programming, then show that in our framework, these issues can either be eliminated or mitigated. Our evaluation shows that our framework generates kernel modules with acceptable binary size and the runtime overhead is negligible.

[8]https://github.com/emmericp/ixy
[9]https://github.com/ixy-languages/ixy.rs

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their valuable feedback and assistance. The work of John C.S. Lui is supported in part by the GRF R4032-18.

REFERENCES

[1] Silas Boyd-Wickizer and Nickolai Zeldovich. 2010. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=1855840.1855849

[2] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '11)*. ACM, New York, NY, USA, 5:1–5:5. https://doi.org/10.1145/2103799.2103805

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 73–88. https://doi.org/10.1145/502034.502042

[4] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc.

[5] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 760–770. http://dl.acm.org/citation.cfm?id=2337223.2337313

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[7] Jean-Yves Girard. 1995. Linear Logic: Its Syntax and Semantics. In *Proceedings of the Workshop on Advances in Linear Logic*. Cambridge University Press, New York, NY, USA, 1–42. http://dl.acm.org/citation.cfm?id=212876.212880

[8] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. MINIX 3: A Highly Reliable, Self-repairing Operating System. *SIGOPS Oper. Syst. Rev.* 40, 3 (July 2006), 80–89. https://doi.org/10.1145/1151374.1151391

[9] Rob Johnson and David Wagner. 2004. Finding User/Kernel Pointer Bugs with Type Inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=1251375.1251384

[10] Taesoo Kim. [n.d.]. A minimal Linux kernel module written in rust. https://github.com/tsgates/rust.ko.

[11] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Software Fault Isolation with API Integrity and Multi-principal Modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 115–128. https://doi.org/10.1145/2043556.2043568

[12] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 305–318. https://doi.org/10.1145/1950365.1950401

[13] Jeffrey Vander Stoep. 2016. Android: protecting the kernel.

[14] L. Tan, E. M. Chan, R. Farivar, N. Mallick, J. C. Carlyle, F. M. David, and R. H. Campbell. 2007. iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support. In *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*. 134–144. https://doi.org/10.1109/DASC.2007.16

[15] Geoffrey Thomas and Alex Gaynor. [n.d.]. Framework for writing Linux kernel modules in safe Rust. https://github.com/fishinabarrel/linux-kernel-module-rust.

[16] Philip Wadler. 1990. Linear Types Can Change the World!. In *PROGRAMMING CONCEPTS AND METHODS*. North.

[17] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. 2006. SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1267308.1267312