Veld: Verified Linux Drivers

Xiangdong Chen
University of Utah

Zhaofeng Li
University of Utah

Jerry Zhang
University of Utah

Anton Burtsev
University of Utah

Abstract

Device drivers and kernel extensions have long been considered one of the main sources of defects in the kernel. In the past, complexity of driver execution environment and their internal logic kept them beyond the reach of formal verification. We argue, however, that recent advances in systems programming languages, and automated verification make a leap forward toward enabling practical development of verified kernel code. Verified Linux drivers (Veld) is a new device driver framework for Linux that leverages Rust and Verus for development of formally correct device drivers. High-level of automation offered by Verus allows us to sidestep traditional burden of verification and instead focus on challenges related to verification of driver code: expressing complex model of the driver, kernel and hardware interfaces, support for verification of concurrent driver code, and integrating with the low-level interface of the kernel. We develop all code in Rust and prove its functional correctness, i.e., refinement of a high-level specification with Verus. Our early experience with developing Veld and verifying parts of the model-specific register (MSR) driver demonstrates the possibility of device driver verification.

CCS Concepts: • **Security and privacy** → **Operating systems security**; **Logic and verification**; • **Software and its engineering** → *Runtime environments*; **Formal software verification**.

ACM Reference Format:
Xiangdong Chen, Zhaofeng Li, Jerry Zhang, and Anton Burtsev. 2024. Veld: Verified Linux Drivers. In *Kernel Isolation, Safety and Verification (KISV '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3698576.3698766

1 Introduction

Device drivers have long been considered one of the main sources of defects and vulnerabilities in the kernel [11, 28]. Today's Linux 6.5 kernel contains around 9,921 device drivers that account for 70% of its source code [25], a number that has doubled since 2014. While the core kernel is relatively stable, the number of device drivers is large and continues to grow with every new generation of hardware. An empirical study of faults in the Linux kernel by Chou et al. found that in 2001 device drivers contained seven times more faults compared to other subsystems [11]. In the last two decades, improved automated testing, fuzzing [12, 15, 9, 29, 33, 30, 26, 34, 38, 37, 26] and static analysis [6, 3, 4, 5, 32] improved device driver flaw density significantly. Nevertheless, device drivers remain one of the three kernel subsystems with the highest flaw density along with architecture layers (`./arch`) and file systems (`./fs`) [28].

Historically, complexity of the driver code kept device drivers beyond the reach of formal verification [1]. Drivers execute in a concurrent and reentrant environment, expose asynchronous communication interfaces require implementation of complex object lifetimes which combine reference counting and manual memory management, implement concurrent dynamic unplug and replug protocols, and employ a range of low-level optimizations to achieve optimal performance. While several projects attempted verification of the driver code [31, 17, 10, 2, 36, 27] it largely remained impractical.

We argue, however, that recent advances in automated formal reasoning significantly lower the burden of verifying low-level systems code hence creating an opportunity for practical development of verified device drivers. Specifically, recent verifiers combine properties of linear types with automated verification based on satisfiability modulo theories (SMT) [20, 24, 19]. Linear types significantly lower the burden of reasoning about the heap due to a strict pointer aliasing discipline. Moreover, verifiers like Verus [20, 19] natively support verification of Rust, a programming language designed for systems development, hence enabling practical verification of driver code that can then be compiled and executed on bare metal.

Our work leverages Verus [20, 19], a new SMT-based verifier for Rust, for exploring the possibility of developing verified device drivers in the Linux kernel. Verus translates Rust code into an SMT formula that is then checked by the Z3 solver [13]. Similar to previous SMT-based verifiers [8, 21, 35], Verus provides a high level of proof automation and verification speed – in many cases the SMT solver can prove the verification condition automatically, but in some cases requires hints in the form of assertions and inductive proof functions.

High-level of automation offered by Verus allows us to sidestep traditional burden of verification and instead focus on challenges related to verification of the driver code. First, we develop techniques of integrating Verus with the

KISV '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1301-9/24/11
https://doi.org/10.1145/3698576.3698766

Linux kernel. Device drivers communicate with the kernel by exchanging a collection of data structures that utilize a number of low-level programming idioms: sharing hierarchical data structures by reference, utilizing polymorphism via `void*` and union types, relying on unsafe type casts for nested data structures and `container_of`, managing lifetimes with reference counting, and maintaining endless low-level invariants about semantic properties of each data structure. The majority of the kernel-driver interface is unsafe and has to be explained to Verus, which is limited to reasoning about the safe subset of Rust. We develop techniques for expressing safety and correctness of low-level accesses to raw C types via a collection of safe Rust wrappers and Verus specifications.

Second, we explore the ways of modeling execution environment of the driver and its interaction with asynchronous and concurrent invocations and updates from the operating system and the hardware. Device drivers execute in a concurrent and re-entrant environment of the kernel. After registration, the operating system can invoke any function of the driver interface concurrently from multiple threads at any moment as long as the invocation is allowed by the protocol of the kernel-driver interface. User threads access device drivers via the system call interface. Kernel threads invoke device drivers to implement interrupt processing and a range of periodic tasks. Moreover, the hardware can change its state concurrently with respect to the execution of the driver. To reason about correctness of the driver we develop a model of its execution environment that captures concurrent execution of the kernel and the hardware.

Finally, to support development of concurrent driver code, we explore Verus abstractions for read-write locks that allow us to reason about correct accesses to the state of the driver shared across multiple threads as well as to avoid deadlocks.

Our work presents an early prototype of a device driver framework, Veld (verified Linux drivers), designed to provide environment for development of verified device drivers for Linux. We develop a partial implementation of a verified model-specific register (MSR) driver. Specifically, we build all code in Rust and prove its functional correctness, i.e., refinement of a high-level specification with Verus. Our initial experience shows that development of verified device drivers is practical.

2 Background

Modern operating systems implement device drivers as dynamically loaded kernel extensions. For example, Linux device drivers are object files (ELF binaries) compiled and linked separately from the core kernel. While some systems deploy well-contained driver frameworks like IOKit in MacOS [22], in Linux device drivers are free to call any function exported by the kernel (e.g., anything from memory allocation and device registration to specialized subsystem-specific helper functions). The kernel, however, does not invoke the functions of the driver directly. Instead, each driver is responsible for registering a collection of function pointers with the kernel that implement the driver's interface.

```
1  static ssize_t msr_read(struct file *file, char __user *buf,
2       size_t count, loff_t *ppos) {...};
3
4  static const struct file_operations msr_fops = {
5    .owner = THIS_MODULE,
6    .read = msr_read,
7    .write = msr_write,
8    .open = msr_open,
9    .unlocked_ioctl = msr_ioctl,
10 };
11
12 static const struct class msr_class = {
13   .name  = "msr",
14   .devnode = msr_devnode,
15 };
16
17 static int __init msr_init(void)
18 {
19   int err;
20
21   if (__register_chrdev(MSR_MAJOR, 0, NR_CPUS, "cpu/msr", &
         msr_fops))
22   {
23     pr_err("unable to get major %d for msr\n", MSR_MAJOR);
24     return -EBUSY;
25   }
26   err = class_register(&msr_class);
27   if (err)
28     goto out_chrdev;
29
30   err = cpuhp_setup_state(CPUHP_AP_ONLINE_DYN, "x86/msr:online",

31         msr_device_create, msr_device_destroy);
32
33
34   if (err < 0)
35     goto out_class;
36   cpuhp_msr_state = err;
37   return 0;
38
39 out_class:
40   class_unregister(&msr_class);
41 out_chrdev:
42   __unregister_chrdev(MSR_MAJOR, 0, NR_CPUS, "cpu/msr");
43   return err;
44 }
```

Figure 1. The MSR driver's `init` function and interface.

To illustrate a typical driver boundary and the challenges it presents for driver verification, we consider an example of a simple device driver that provides a high-level interface to access model-specific registers (MSRs) on Intel CPUs in the Linux kernel (Figure 1). Execution of every driver starts with the `init()` function that is called by the kernel when the driver is loaded. The `init()` function of the MSR driver (lines 17–44) registers an interface of a "character" device (lines 4–10). In general, interfaces encapsulate device driver functionality, e.g., all network drivers hide device-specific packet handling code behind a general network interface. The interface of the driver (lines 4–10) consists of a collection of function pointers that implement its interface. A device driver and the kernel communicate by exchanging references to hierarchical data structures, e.g., the interface of a character device is centered around the `struct file` data structure that represents a file. For example, to read an MSR, the driver uses a file pointer `struct file *file` to retrieve the CPU identifier from the minor

number of the file, and the MSR address from the offset within the file `loff_t *ppos`. The driver returns the value of the MSR to the user via a buffer pointer `char __user *buf`.

The operation of the driver is fully concurrent and asynchronous. For example, the kernel can open the interface of the MSR driver multiple times (via the `open()` function, line 8) and invoke the driver concurrently from multiple threads on multiple CPUs. Moreover, the hotplug interface (lines 30–31) which driver registers with the kernel allows the kernel to change hardware availability by taking CPUs down or bringing them back up.

Verus Verus is a new SMT-based verifier for Rust [20]. To ensure a high degree of ergonomics, Verus supports development of executable code, specifications, and proofs directly in Rust. Verus supports a large subset of Rust and introduces minimal extensions for development of specifications and proofs (for example Verus disables checking for linearity to allow free use of linear variables, e.g., multiple times, in specifications).

The key advantage of Verus is that it uniquely leverages the properties of the linear type system [20]. First, Verus relies on the linear type system to simplify its SMT encoding and to dramatically lower lower complexity of reasoning about the heap [20, 24]. For example, Verus encodes immutably borrowed references and owned heap pointers as plain values instead of references to memory. Moreover, linear types significantly lower the burden of verification by avoiding a large number of memory invariants that are not interesting in a typical case of unaliased objects [24].

Second, Verus provides an elegant way to reason about raw pointers through an idea of linear permissions [20]. To ensure the practicality of an otherwise linear language, Rust allows escape from the ownership rules through its unsafe subset, e.g., to implement doubly-linked lists, aliases, concurrent primitives, etc. Historically, reasoning about the unsafe subset of Rust remained challenging [16] thus limiting verification to safe Rust [7]. In contrast, Verus relies on a combination of linear ghost permissions and SMT verification to provide a safe and verified alternative to traditionally unsafe pointer operations. Typically, operations on pointers, e.g., dereferencing a pointer, are unsafe, i.e., in Rust such operations bypass the borrow checker and break the rules of the linear type system, which makes reasoning about them problematic. Verus, however, offers an elegant abstraction of *permissioned pointers*, i.e., `PPtr<T>`, and a linear tracked permission type `PointsTo<T>` that allow proofs about raw pointers in a convenient manner [20]. Specifically, to read from a raw pointer one requires an immutable reference to the permission corresponding to that pointer (writing requires a mutable reference to the permission). Hence, accesses to raw pointers follow the normal ownership model in Rust. This allows us to use raw pointers like C, i.e., construct cyclic data structures like linked lists. The ghost permission types however allow us to prove that operations on pointers are safe and match the abstract specification.

3 Veld Architecture

Verus allows us to carry development and verification in an almost complete subset of Rust. Moreover, the verified code that can be compiled and executed on bare-metal (or in our case in the environment of the kernel). Furthermore, it is possible to integrate Verus with the kernel build system to ensure native kernel development: we implement a verified device driver in the kernel tree (i.e., verify, compile, and install it) in a manner identical to traditional driver development. The verified driver is dynamically loaded into the kernel as a kernel extension (i.e., Linux kernel module), which starts execution with the `init()` function. Several design choices, however, are important for enabling development and verification of driver code.

Kernel interface Verus can only reason about safe Rust types. Hence it requires an intermediate Rust layer to access unsafe C types that are used by the kernel interface. Specifically, the interface of the kernel has to be explained to Verus as a combination of safe Rust wrapper types, specifications (pre and post conditions), and linear permissioned pointers that allow Verus to establish correctness of unsafe pointer operations at the level of the proof.

Different approaches are possible for developing such safe wrappers. For example, it is possible to implement thin, C-like interfaces that follow the design of existing kernel interfaces replacing C pointers with Verus linear permissioned pointers. Such an approach enables development of a verified Rust driver that closely follows C interfaces, and C code.

Alternatively, it is possible to embrace the idiomatic Rust style and hide idiosyncrasies of low-level C behind high-level Rust abstractions. The practical advantage of such approach is that it has already been adopted by the Rust-for-Linux (RFL) project, a recent device driver framework for Linux that enables development of device drivers in Rust. RFL implements Rust bindings for various kernel subsystems, e.g., network, block, non-volatile memory on PCIe (NVMe), etc. Rust bindings expose *safe* interface to the unsafe interface of the kernel. Specifically, RFL relies on bindgen [18] to automatically generate foreign function interface (FFI) bindings from the kernel header files. Bindgen takes in a C header file and automatically translates the C structures and function declarations to Rust-compatible structures and signatures.

The kernel developer uses these bindings to construct a Rust library, i.e., a crate, which acts as a trusted layer, potentially containing unsafe operations. This layer is written in idiomatic Rust, utilizing Rust features such as traits and templates to create convenient and *safe* abstractions for the upper layers (i.e., the actual Rust driver which can be implemented entirely in safe Rust).

From RFL to VFL In Veld we leverage RFL but change it in several important ways. First, to expose safe Rust interfaces,

```
impl VerifiedMajorRegistration {
    pub fn register(&mut self, kernel: &mut KernelState) -> (ret:
          bool)
        requires
            !kernel.registered(old(self).major, old(self).
                base_minor, old(self).count) && ...
        ensures
            ret ==> kernel.registered(self.major, self.base_minor,
                 self.count) && ...
    { ... }
}

fn init(kernel_state: &mut KernelState)
    ...
{
    let mut registration = VerifiedMajorRegistration::new(cstr!(
          "msr"), MSR_MAJOR, 0, NR_CPUS);
    registration.register(kernel_state);
    assert(kernel_state.registered(MSR_MAJOR, 0, NR_CPUS));
    ...
}
```

Figure 2. Registering a character device through wrapped RFL types

RFL relies on a collection of run-time checks and human, natural language arguments about the safety of the code, e.g., correctness of reference counters, synchronization primitives, iterators, etc. Such approach requires verified code to trust RFL which sometimes requires complex reasoning about the safety and logical correctness guarantees. In Veld we use RFL wrapper types as minimal thin abstractions that allow access to the C kernel interface but establish safety and correctness of such accesses via a proof. For example, in contrast to RFL, we avoid putting device deregistration logic in the `Drop` methods and instead require the driver to explicitly unregister with the kernel when requested.

Second, we make simplifications and extend RFL types with Verus specifications that describe the expected behavior of each type. For example, each character device (cdev) in Linux is identified by a major number and a minor number. The `__register_chrdev` and `__unregister_chrdev` functions, respectively, register and unregister a cdev identified by a given major number and a range of minor numbers. In VFL, character device registrations are modeled as the `MajorRegistration` struct, where we make the simplification to only support character devices that encompass the entire major number (hence the name). Additionally, we assume that no other driver will attempt to unregister the major number: Calling the `register` function will guarantee the caller that the registration remains valid until deregistration. Figure 2 shows a snippet of the `register` function where we encode our assumptions about the kernel state in the `KernelState` struct.

Veld, however, benefits from automation provided by RFL. For example, we utilize procedural macros provided by RFL for generation of virtual tables which abstract C function pointers behind safe Rust traits. For each C function in a given virtual table, RFL implements a trusted trampoline that performs type casting and invokes the corresponding method in safe Rust. The procedural macro exposes missing methods

```
pub enum OSStep {
    CpuOnline {cpu_id:CpuId},
    CpuOffline {cpu_id:CpuId},
    Init {},
    Exit {},
    ...
}
pub enum CPUStep {
    Read {cpu_id:CpuId, addr:MsrAddress},
    Write {cpu_id:CpuId, addr:MsrAddress, value:u64},
}
pub enum DriverStep{
    Read{cpu_id:CpuId, addr:MsrAddress, ret:Result},
    Write{cpu_id:CpuId, addr:MsrAddress, value:u64, ret:Result},
    RegisterCharDev {major:u64, baseminor:u64, minorct:u64, name:
        String, ret:Result},
    UnRegisterCharDev {},
    RegisterClass {name:String, ret:Result},
    UnRegisterClass {},
    RegisterCpuHotPlugEvent {event:cpuhp_state, name:String, ret:
        Result},
    UnRegisterCpuHotPlugEvent {},
    RegisterDev {cpu_id:CpuId, name:String, number:u64, ret:
        Result},
    UnRegisterDev {cpu_id:CpuId, name:String, ret:Result},
    RegisterDevNode {cpu_id:CpuId, name:String, number:u64, ret:
        Result},
    UnRegisterDevNode {cpu_id:CpuId, name:String, ret:Result},
    SetAllowWrites {value:AllowWriteMsrs},
    Open {file:File},
    Close{file:File},
    ...
}
```

Figure 3. Partial steps of the state machines modeling operating system, CPU, and the driver

in the Rust implementation so they can be represented as null pointers in the virtual table.

Standard library and memory management Verus supports use of the standard library via a collection of trusted bindings to unverified standard library types. While the use of the core types from the standard library is a practical choice (e.g., recent Verus-based verification projects like Verismo [39] choose to trust the standard library), we avoid such trust in Veld and instead, develop the driver with a collection of types that we verify. We, however, choose to trust the kernel memory allocator.

Synchronization and concurrency To support verification of concurrent and reentrant driver code, i.e., drivers that can be concurrently accessed from multiple threads of execution on multiple CPUs, and be preempted by interrupts, we leverage Verus support for tracked permissions. Specifically, each thread is given a unique, unforgeable tracked thread identifier permission before entering the driver. We then protect access to the shared driver state with a read-write lock that returns a thread-local read/write permission that is tied to the lock. The thread must provide the lock permission before accessing the shared driver state.

4 Verification

Specifications We use specifications to capture both possible behavior of the driver environment (i.e., concurrent

changes of the hardware, concurrent invocations of the driver by the kernel) and correct behavior of the driver itself. We model the environment of the driver – the operating system and the hardware – as a collection of state machines that each can take arbitrary steps (Figure 3), e.g., change the state of the hardware, change the state of the kernel, and invoke functions of the device driver, etc. To model the expected behavior of the driver, we develop a model of the driver's abstract state and define how it is evolving on each state transition, i.e., reflecting the change in the kernel, hardware, or the driver through invocation of its methods.

To define the abstract model of the system, we utilize Verus ghost variables, i.e., `Set<Driver>`, that represent the abstract model of the concrete system state, a set of registered device drivers in the kernel. We then define the high-level behavior of the system as a collection of specification functions which describe how this abstract state is updated on each state transition, i.e., invocation of the driver, hardware update, etc.

To establish the functional correctness of the driver, we prove a refinement theorem, i.e., the implementation of the driver refines its abstract state. That is, invocation of the driver results in a state of the driver that is equivalent to the change in the high-level specification. Specifically, to prove refinement, we establish equivalence between the abstract and concrete state of the system, and proof that equivalence holds on each state transition.

Operating system Our model captures the high-level behavior of the operating system with respect to the driver. For example, the character driver like MSR follows the kernel registration protocol. After the driver is registered, its interface can be invoked from any thread of the system. Specifically, a thread can open a file that reserves a handle for future read and write operations. Moreover, even if the driver is unregistered, all opened file descriptors remain functional, allowing the threads to access the driver. The kernel will not allow to open new file descriptors. The kernel implements a reference counting protocol that uses two counters to keep track of the references to the character device and its backing kernel module.

Our model captures registration, hotplug, and file open protocols. We represent the state of the kernel as a high-level abstract state, e.g., an arrays of drivers registered for a specific major and minor numbers. We also model the kernel reference counters to properly reflect the behavior of the kernel.

We also use the abstract state of the kernel to properly define the correct behavior of the driver. For example, the post-condition for the initialization function of the driver requires the driver to be registered (or leave the kernel in a clean state in case of an error). The driver hence has to invoke functions of the kernel interface, e.g., register character driver, register class, etc. Each of these functions updates the abstract state of the kernel.

The state machine of the kernel can take any of the allowed steps (i.e., open the driver multiple times from multiple threads as long as the driver is registered, unplug CPUs, and change properties of the driver via the kernel module parameter interface, etc.).

Hardware The hardware operates asynchronously and concurrently with the drivers' execution. In our model, the hardware can take any of the possible transitions while the driver code is running (i.e., it can read MSRs written by the driver, and update the values of other MSRs to reflect the change). In the case of the MSR driver, the values of the model-specific registers can change between reads as long as the change matches the expected behavior of the hardware.

Arguably, a complete hardware model of the MSR interface should model the functional behavior of each MSR as well as the dependencies between them. For example, the `IA32_APIC_BASE` MSR holds the base address of the Advanced Programmable Interrupt Controller (APIC) as well as a bit to toggle the x2APIC (Extended xAPIC) feature. Clearing this bit will make MSRs related to x2APIC (e.g., `IA32_X2APIC_APICID`) inaccessible.

Unfortunately, constructing a complete hardware model for MSRs is challenging. Historically, Linux sidesteps the problem by allowing the MSR driver to execute illegal MSR accesses that result in exceptions. The kernel, however, has a specific path to catch such exceptions turning them into I/O errors returned from the low-level MSR read and write functions. This allows the driver and the entire kernel to avoid a crash.

We develop a partial model for a small subset of MSRs. This allows us to develop a driver that avoids illegal hardware accesses resulting in hardware exceptions.

Synchronization primitives To support verification of concurrent code that synchronizes access to shared state, we develop abstraction of a read-write lock (Figure 4). Our read-write lock follows the design of a C-style read-write lock which allows us to implement critical sections. In contrast to a guard returned by a typical Rust mutex, our lock returns a read or write permission to access the objects protected by the lock. To acquire a lock, the thread provides a thread identifier permission which is unique and unforgeable (generated by the TCB for a specific thread). This permission allows us to distinguish the steps taken by different threads. To distinguish different locks, each lock maintains a unique identifier (this allows us to prove that access permission acquired from one lock cannot be used in place of another lock). Moreover, the lock permissions cannot be passed between threads.

We divide the specifications of synchronization primitives into two categories: global and local. For example, globally, we know that a variable does not change unless a thread holds a write lock allowing it to mutate the variable. Local specifications ensure the correct use of locks by one thread, e.g., the thread cannot acquire the lock twice, it has to release

```
pub fn acquire_read(&mut self, Tracked(t_id_perm): Tracked<&
      ThreadIDPerm>)
    -> (ret: Tracked<ReadLockPerm>)
requires
    // local specs
    old(self).thread_holds_no_lock(Tracked(t_id_perm)),
    // global specs
    old(self).reading_threads().contains(t_id_perm@) == false,
ensures
    // local specs
    self.thread_holds_read_lock(t_id_perm),
    self.thread_holds_write_lock(t_id_perm) == false,
    self.is_owner_of_read_perm(&ret@),
    self@.thread_id() == t_id_perm@,
    self.view_from_thread(t_id_perm).wf(),
    // global specs
    old(self).writing_thread().is_None(),
    self.reading_threads() =~= old(self).reading_threads().
          insert(t_id_perm@)
    self.view_from_system() =~= old(self).view_from_system()
```

Figure 4. Specifications of the function that acquires a read lock

all the locks, etc. Note, however, that the moment the thread releases the lock it cannot make assumptions about the state of the variables protected by the lock as they may be changed by other threads. Global specs though allow us to reason about the global state of the system, e.g., which threads can read and update which variables, their state, etc.

Driver In our model, an invocation of the driver is not atomic (i.e., the hardware can change its state asynchronously, the operating system can update the shared state of the driver from another thread, etc.). Therefore, we model driver invocations as multiple steps of the state machine. This allows us to model all possible executions of the operating system, hardware, and the driver as their smallest steps and treat the driver as a state machine that can take any valid step for the current state.

5 Implementation

Build system Verus extends the Rust compiler to perform verification. Naturally, this ties Verus to a specific version of Rust. Moreover, similar to Rust, the Verus standard library and supporting crates are built using the *Cargo* build system. Linux, however, integrates Rust (and its `rustc` compiler) with its Makefile-based build system. Moreover, RFL relies on the most recent version of Rust.

To decouple the toolchain versions and minimize changes to the kernel build system, we perform the build in two passes. In the *verification pass*, we build the annotated `kernel` crate using the Verus toolchain with ghost code preserved. The drivers are verified against this crate. During the *compilation pass*, we rebuild the Verus standard library, the `kernel` crate and the drivers with ghost code elided using the kernel toolchain.

6 Status

We completed development of the Veld framework, i.e., integration with the kernel via VFL and the kernel build system. We verified parts of the MSR and `virtio` drivers. For the MSR driver, Verus finishes verification in under a minute. We do not see a measurable slowdown compared to the unverified C driver on a server-grade Intel Xeon machine although complex Rust-based drivers are known to demonstrate some overhead compared to C counterparts [14, 23].

7 Conclusions

Our early experience with Veld demonstrates that Verus takes a huge step towards enabling practical (low-burden) verification of device drivers in a full-featured Linux kernel. While we verified only parts of relatively simple device drivers, our techniques – approach to modeling concurrent execution environment of the kernel and asynchronous hardware, integration with unsafe kernel interfaces, and ability to support concurrent driver code – are, arguably, general and will enable development and verification of complex, full-featured drivers.

Acknowledgments

We would like to thank KISV'24 reviewers for various insights that helped us to improve this work. This research is supported in part by the National Science Foundation under Grant Numbers 2220410 and 2239615, and Amazon.

References

[1] Alexey Khoroshilov. Linux device-drivers verification challenges, October 2012.

[2] Eyad Alkassar and Mark A Hillebrand. Formal functional verification of device drivers. In *Proceedings of the Second Working Conference on Verified Software: Theories, Tools, and Experiments* (VSTTE), October 2008.

[3] Sidney Amani, Leonid Ryzhyk, Alastair F Donaldson, Gernot Heiser, Alexander Legg, and Yanjin Zhu. Static analysis of device drivers: We can do better! In *Proceedings of the Second Asia-Pacific Workshop on Systems* (APSys), July 2011.

[4] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency Use-After-Free bugs in Linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference* (ATC), 2019.

[5] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. Static detection of unsafe DMA accesses in device drivers. In *Proceedings of the 30th USENIX Security Symposium* (USENIX Security), 2021.

[6] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, (4):73–85, 2006.

[7] Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. Verifying Rust programs with SMACK. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis* (ATVA), October 2018.

[8] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th international conference on Formal Methods for Components and Objects* (FMCO), 2005.

[9] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No grammar, no problem: Towards fuzzing the Linux kernel without system-call descriptions. In *Proceedings of the 30th Network and Distributed System Security Symposium* (NDSS), 2023.

[10] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), Santa Barbara, CA, USA, 2016.

[11] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (SOSP), October 2001.

[12] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS), 2008.

[14] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The case for writing network drivers in high-level programming languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[15] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. SyzDescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy* (S&P), 2023.

[16] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In number POPL, December 2017.

[17] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver–an experience report. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software* (SPIN), Los Angeles, CA, USA, August 2008.

[18] The Rust Programming Language. rust-bindgen. `https://github.com/rust-lang/rust-bindgen`, 2024.

[19] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the 30th Symposium on Operating Systems Principles* (SOSP). ACM, November 2024.

[20] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, (OOPSLA1):85:286–85:315, April 2023.

[21] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16*, Dakar, Senegal, April 2010.

[22] Jonathan Levin. *Mac OS X and iOS Internals: To the Apple's Core*. 2012.

[23] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. An empirical study of Rust-for-Linux: The success, dissatisfaction, and compromise. In *Proceedings of the 2024 USENIX Annual Technical Conference* (ATC), July 2024.

[24] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proceedings of the ACM on Programming Languages*, (OOPSLA1), April 2022.

[25] LKDDb: Linux Kernel Driver DataBase. `https://cateee.net/lkddb/`, 2019.

[26] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. PrIntFuzz: Fuzzing Linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[27] David Monniaux. Verification of device drivers and intelligent controllers: A case study. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, 2007.

[28] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux:

Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), Newport Beach, California, USA, 2011.

[29] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. POTUS: Probing off-the-shelf USB drivers with symbolic fault injection. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies* (WOOT), 2017.

[30] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *Proceedings of the 29th USENIX Security Symposium* (USENIX Security), 2020.

[31] Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O Myreen, et al. Pancake: Verified systems programming made sweeter. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems* (PLOS), 2023.

[32] Hendrik Post and Wolfgang Küchlin. Integrated static analysis for Linux device driver verification. In *International Conference on Integrated Formal Methods*, 2007.

[33] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. SymDrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), 2012.

[34] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *Proceedings of the 31st USENIX Security Symposium* (USENIX Security), 2022.

[35] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL), St. Petersburg, FL, USA, 2016.

[36] Thomas Witkowski. Formal verification of Linux device drivers. *Master's thesis, Dresden University of Technology*, 2007.

[37] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. DEVFUZZ: Automatic device model-guided device driver fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[38] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. Semantic-informed driver fuzzing without both the hardware devices and the emulators. In *Proceedings of the 29th Network and Distributed System Security Symposium* (NDSS), April 2022.

[39] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VeriSMo: A verified security module for confidential VMs. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), July 2024.