A WebAssembly Runtime for the Linux Kernel

Design, Implementation, and Evaluation in Comparison with eBPF and Kernel Modules in the Linux Kernel

Bertold Brödner[1], Simon Cyrani[1], Tobias Görgens[1],
Robert Oleynik[1], Malte Stellmacher[1], Sebastian Wilke[1],
Lukas Pirl[2], and Clemens Tiedt[2]

[1] Professorship for Operating Systems and Middleware
Hasso Plattner Institute at University Potsdam
{Bertold.Broedner,Simon.Cyrani,Tobias.Goergens,Robert.Oleynik,Malte.Stellmacher,Sebastian.Wilke}@student.hpi.de
[2] Professorship for Operating Systems and Middleware
Hasso Plattner Institute at University Potsdam
{Lukas.Pirl,Clemens.Tiedt}@.hpi.de

Modern general-purpose operating systems require extensible kernel architectures to support software closer to the hardware. Yet existing methods, such as traditional kernel modules and extended Berkeley Packet Filter (eBPF), present distinct trade-offs among security, portability, and computational expressiveness. This project addressed these challenges by evaluating the WebAssembly Stack Machine (WASM) as an isolated, Turing-complete, and microarchitecture-independent alternative for Linux kernel extensions.

We developed *wasm-kernel-runtime*, an execution environment implemented as a kernel module using the *Rust for Linux* framework. The methodology utilized a pull-based decoder for memory-efficient parsing and a stack-machine executor that manages value, call, and control stacks to maintain strict isolation. To facilitate user space interaction, a Virtual File System (VFS) interface named `kwasmrt` was implemented in C, enabling module loading and invocation through standard Unix file operations. Technical validation was conducted using a custom testing harness, achieving a 71.20% compliance with the WebAssembly 1.0 Core Specification, and a VM-based CI/CD pipeline. The runtime successfully executes non-trivial logic, such as a prototypical Netfilter-based networking application. Still, the current state lacks a formal module verification component, leaving potential semantic vulnerabilities that are only caught at runtime. Future research could also implement WebAssembly specification extensions or expand the host function interface to mirror the eBPF helper function ecosystem.

1. Motivation & Introduction

Modern general-purpose operating systems, like *Linux*, *Windows*, and *macOS*, allow kernel behavior to be extended. On Linux, this can be achieved using kernel modules, on Windows using kernel-mode drivers, and on macOS using kernel extensions. These kernel extensions are necessary for implementing software closer to the hardware, such as device drivers, firewalls, or CPU schedulers. In this master's project, we focus on the Linux kernel. Even before Linux's first stable release in 1994, kernel modules have been supported. However, kernel modules must be compiled for each Linux kernel version, as breaking changes could have been introduced in the executing version. Additionally, loading third-party kernel modules is dangerous, as they can gain full system privileges and crash, destroy, or compromise the operating system.

Therefore, the Linux community developed an alternative approach for applying kernel modifications more securely. In 1992, the *Berkeley Packet Filter* (BPF) (also called *BSD Packet Filter*) has been introduced by researchers at *Lawrence Berkeley Laboratory* [6]. It enables the execution of kernel-based network packet filtering programs, providing performance benefits over user space filtering. Starting with kernel 3.18, Linux implemented *extended Berkeley Packet Filter* (eBPF) [3], which supports non-networking use cases. An eBPF allows the development of cross-microarchitecture, cross-version kernel modifications. Writing eBPF requires its own programming language, which implements a subset of the C language. Before an eBPF program is executed, it is passed to a verifier that performs static analysis to determine whether the code could crash or hang the kernel. If it succeeds, the kernel executes the program event-driven when a so-called eBPF hook triggers. They are implemented in various parts of Linux, e.g., when network packets are received from a network interface card. Additionally, eBPF programs are not sandboxed and gain full access after passing the verifier. Unlike kernel modules, they are not *Turing*-complete because the verifier enforces termination and limits the number of instructions. However, they are the only cross-version, cross-microarchitecture alternative for kernel modifications so far.

This project developed another approach to kernel modifications that relies on *WebAssembly*. WebAssembly Stack Machine (WASM) [8] is a stack-based virtual instruction set architecture. Table 1 compares WASM to kernel modules and eBPF programs. Every WASM program (called a module) is executed in full isolation by design, with controlled access to external resources via the host function interface. Moreover, WASM is designed to be platform-, hardware-, architecture-, and runtime-independent. Every compliant WASM runtime can execute every WASM module. Therefore, a WASM runtime built as a kernel module can run arbitrary modules without requiring other software or hardware. Additionally, WASM can be compiled from an increasing number of programming languages, including C, Rust, Zig, TypeScript, Go, and more, offering greater flexibility in kernel development. Lastly, unlike eBPF, WASM is *Turing*-complete, enabling the execution of more complex programs.

Methods	Cross-architecture	Cross-version	*Turing*-complete	Languages	Isolation
Kernel module	×	×	✓	C, Rust	×
eBPF	✓	✓	×	Subset of C	×
WASM	✓	✓	✓	All with WASM target	✓

Table 1: Comparison of kernel extension methods regarding architecture- and version-independence, *Turing*-completeness, usable programming languages, and isolation features. Kernel modules are *Turing*-complete but lack cross-architecture and cross-version support. Additionally, they do not provide isolated execution. eBPF has cross-architecture and cross-version support but misses *Turing*-completeness. WASM offers cross-architecture and cross-version execution, as well as *Turing*-completeness. Furthermore, WASM provides isolated execution by design and an increasing number of usable programming languages.

This project introduces *wasm-kernel-runtime*, a WASM runtime developed as a kernel module that runs WASM modules adhering to Core Specification 1. The modules are executed in kernel space, and user space communication is handled via a *Virtual File System* (VFS).

The remainder of this report is organized as follows: section 2 discusses related research to our project. Afterward, our approach is introduced in section 3 as a high-level overview of the WASM runtime components, which are then detailed further in section 4. In section 5, we delve into the inner workings of the runtime, focusing on our kernel API integration via host functions, our VFS-based communication approach, and the module instantiation process. The section 6 outlines our *Continuous Integration* (CI) pipeline and testing framework, while section 7 assesses the success of our approach and the suitability of using *Rust for Linux* (R4L). Finally, section 8 identifies potential avenues for further development.

2. Related Work

The idea of using WASM for kernel space applications has already been explored in a few other papers. For example, one of these papers [1] adopted a very similar approach to ours, building a Linux kernel module that includes a WASM runtime. However, this project relies on a two-year-old, unmaintained third-party runtime, which could make its use in kernel space potentially dangerous. Additionally, the author used a different method for user-to-kernel-space communication: a character device. Nevertheless, Abdelmonem implemented *Kprobes*[3] for the host

[3] `https://docs.kernel.org/trace/kprobes.html`

function interface. *Kprobes* enable hooking into most kernel routines, allowing for the simulation of eBPF hook functionality. We see that this is an interesting idea to hook into the Linux kernel, similar to eBPF. Because of that, we would suggest using it in future versions of this project as well.

Wasmer developed a very similar project — `kernel-wasm`[4] is another WASM runtime as a kernel module that claims to be faster than native execution for tested applications [12]. Interestingly, the authors chose to implement parts of the *WebAssembly System Interface* (WASI) (to, e.g., enable file manipulation). Unfortunately, this project has been unmaintained for more than seven years. Requests from different users regarding the project's activities remain unanswered.

Another project called `Wasmachine` [10] takes a more radical approach to executing WASM in kernel space: The authors built a minimal WASM-optimized kernel that executes WASM modules natively. Such an approach seems advantageous for non-Linux systems, e.g., embedded microcontrollers. For Linux systems, complex kernel modifications may be necessary.

Other approaches focused more on the eBPF ecosystem. For instance, `Wasm-bpf` [11] is another approach that uses WASM to develop kernel extensions by transpiling WASM to eBPF and deploying the eBPF program. This, however, has the same restrictions as eBPF for kernel space, but offers greater flexibility in choosing the programming language for the eBPF program. Additionally, the WASM runtime running in user space can use eBPF hooks to execute code in kernel space partially.

Lastly, in Kröning et al., the RWTH Aachen's Rust-based unikernel project `Hermit`[5] received support for WASM-based kernel extensions to execute sandboxed code using the third-party `Wasmtime`[6] runtime. However, most *Unix*-like systems are based on the Linux kernel. Thus, this work does not help with our focus.

In conclusion, executing WASM in kernel space has attracted increasing research interest in recent years. However, most related projects did not implement the WASM runtime without third-party dependencies, which could be a security problem. Especially in cases where the WASM runtime is left unmaintained for years. In contrast, in our solution, we are aware of the complete supply chain of the kernel module.

3. Architecture

An in-kernel WASM runtime can be implemented in multiple ways. For Linux, it might be possible to include the runtime directly in the kernel source code, develop a kernel module, or use a mixture with eBPF as discussed in section 2. Additionally, Linux offers at least two programming languages for kernel development: C and Rust. Lastly, communication between the user and kernel space can be handled

[4] `https://github.com/wasmerio/kernel-wasm`
[5] `https://github.com/hermit-os/kernel`
[6] `https://github.com/bytecodealliance/wasmtime`

in several ways, for example, using a VFS, or a character device, as in the related work of Abdelmonem.

Our approach of designing the WASM runtime as a kernel module still requires it to be microarchitecture- and version-specific. However, this approach allows building the kernel module against all supported Linux kernel microarchitectures and versions once and then executing arbitrary WASM modules cross-microarchitecture and cross-version in the kernel module. In addition, WASM does not make any assumptions regarding microarchitecture, so every WASM module has cross-microarchitecture support by design.

We decided to develop the kernel module using the emerging *Rust for Linux*[7] (R4L) project. As of 2026, Rust in the Linux kernel is no longer considered experimental and is generally available [2]. Using Rust as a programming language allows us to leverage Rust's memory-safety guarantees, which is advantageous compared to C. Beyond memory safety, Rust offers several language features that are beneficial for a project of this complexity. Modern constructs such as pattern matching, algebraic error types, and functional programming paradigms improve code clarity and reduce repetitive implementation patterns. Compile-time type inference and a trait-based type system enable target-agnostic abstractions, as demonstrated by the `#[cfg(MODULE)]` bindings pattern discussed in subsection 7.1. Additionally, Rust's integrated package manager and built-in documentation tooling (`rustdoc`) reduce development overhead, even though the kernel build environment required custom adaptations to both, as outlined in subsection 7.1. For example, there is currently no native support for Rust unit tests or procedural macros in kernel modules.

The Rust ecosystem is built on binaries and libraries called *crates*. Crates are the smallest amount of code the Rust compiler will consider. In some areas (e.g., out-of-memory handling), the Rust standard crate has semantics that differ from those of the Linux kernel. Therefore, R4L modules cannot use the standard crate. Instead, they use a separate kernel crate. For that reason, Rust kernel modules have access to the `kernel` crate, which re-implements data structures as safe to fail, so they do not trigger a kernel panic (also called fallible). As a result, only a small subset of data structures is implemented.

To dynamically load WASM modules into the kernel, user space processes must communicate with the kernel module. This is possible via our own VFS, `kwasmrt`. The system is explained and compared to alternative approaches in detail in subsection 5.3. Processes can invoke, control, monitor, and deploy WASM binaries using the file system structure. Additionally, interaction with WASM modules is possible by manipulating memory objects exposed as files.

Not only is interaction for loading or managing modules in the kernel necessary, but interaction with the kernel interfaces is as well. As explained in subsection 5.2, we decided to implement a subset of the C kernel API bindings as WASM host functions for kernel integration. However, this is only prototypically implemented for now and remains future work, see section 8.

[7] `https://rust-for-linux.com/`

Figure 1 gives an overview of the relationships among all components of our architecture. The components are described in-depth in section 4.

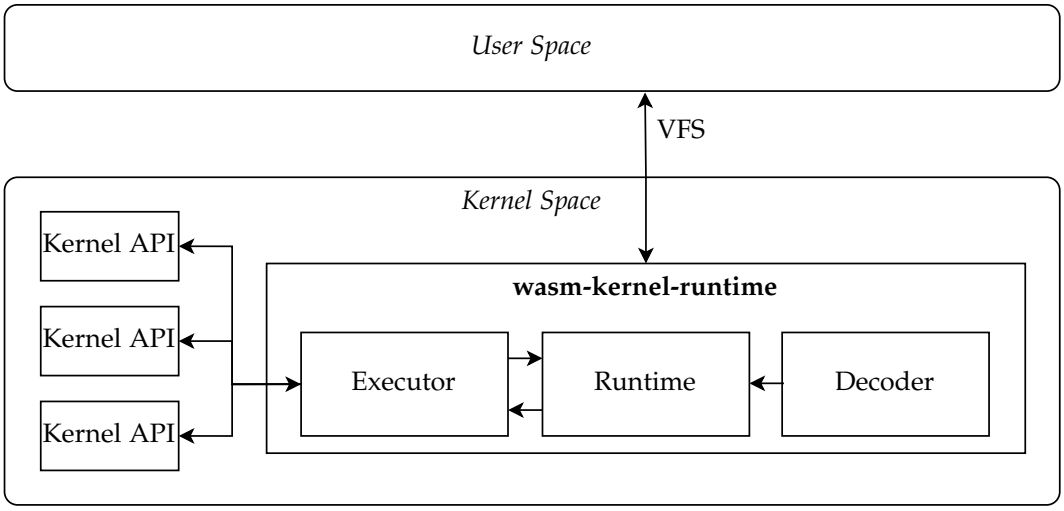

Figure 1: An overview of our architecture for the WASM kernel module. User space processes communicate with the kernel module via the VFS. The kernel module loads WASM modules into the decoder and passes the parsed modules to the runtime, which instantiates them. The executor executes the instructions described by the WASM module and communicates with the runtime to maintain the module instance's state. Additionally, the executor integrates the host function interface to communicate with kernel APIs.

4. Components

We designed a component-based architecture for our kernel module:

Decoder is the parsing component for WASM binary files. The decoder is described further in subsection 4.1.

Executor is the instruction-level executing component. It executes WASM instructions for WASM module instances. Described in subsection 4.2.

Runtime functions as an umbrella on top of the executor. It instantiates WebAssembly modules and maintains the internal state of all running WASM module instances. Module instantiation is explained further in subsection 5.1.

The decoder and executor components are part of the `common` crate. This crate relies only on Rust's `core` library, allowing it to be compiled for Linux kernel modules and user space Rust projects. This allows us to use Rust's unit-test framework

in user space components based on the `common` crate. The runtime component uses advanced interfaces provided by either the `kernel` crate in kernel modules or the `std` library in user space. Thus, the runtime has to be implemented twice (for tests and the kernel module) for now.

4.1. Decoder

To allow compatibility between kernel and user-space, our initial decoder implementation was build without dependency to Rust's or the kernel's standard library, and we initially also decided to not allocate any memory as part of the decoding process. This approach allows us to easily port the decoder between both kernel and user-space. At the same time, it is still necessary to allocate our structures. This forces us to split the decoder into two parts. The first part reads the binary encoded modules and returns the corresponding runtime constructs and the second part which stores the decoded constructs inside the runtime's module representation. To avoid maintaining to implementation of the second part, we used the array implementation inside the `common` library as an abstraction above the standard library. Additionally, we also needed to relax the no-allocation inside the decoder part. This is because of WASM use of blocks to represent if conditions and loops. Allocations are necessary, to keep track of the read instructions while reading a nested loop. To ensure that we are compatible with both kernel and user-space, we again used the abstractions from the `common` library.

4.2. Executor

The executor implements the execution logic for each WebAssembly instruction. To maintain the state of the runtime we use multiple logically decoupled stacks, namely the *value stack* and the *call stack*. In this sense it is different from the description in [8, Runtime Structure] which only uses a singular stack to model the state. The call stack consists of *call frames*, each of which maintains a *control stack* of *control frames*. The diagram in Figure 2 provides a visualization. In addition, the executor has access to the *module instance* in which it is executed via an interface that allows it to modify global variables, memory, or tables (see subsection 5.2 for more details). Instructions can modify the value stack (e.g., arithmetic operations), the call stack (e.g., branching operations), or the module instance (e.g., memory, table, or global variable operations).

The *value stack* keeps track of all values used for program execution. For example, a WASM module reads two local values from the call frame and pushes them onto the value stack, then performs an addition by popping those two values, and pushing the sum back to the stack, and finally returns the value by popping it again.

Generally, our call and control stack together resemble a typical call stack implementation, so we only mention some noteworthy details: The call stack maintains the order of called function, whereas each control stack maintains the order of executed control blocks within a function. Internally, call frames reference all in-

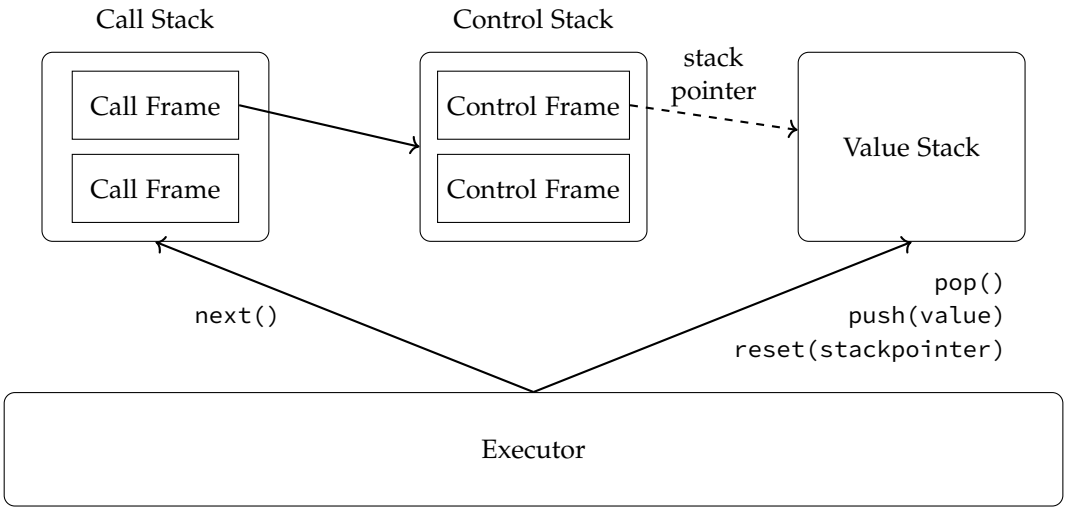

Figure 2: Architecture of the executor including its most important functions. The executor maintains a call stack and value stack. Each call frame of the call stack maintains a control stack, and each control frame stores a stack pointer referencing a position in the value stack. The executor fetches instructions by calling `next()` on the call stack, and modifies the value stack by calling one of the respective functions.

structions split into their respective control blocks. A control block is a contiguous array of instructions that reflects a specific branching path. Executing a branching instruction like `loop`, `while`, `if-else`, etc., therefore adds a new *control frame* that references the corresponding control block within the call frame by index. Additionally, the control frame is used to reset the value stack to its initial height after the control frame is exited.

This architecture ensures that instruction execution and instruction management are decoupled: During the main execution loop, the executor only fetches the current control block index of the top-most control frame to retrieve the next instruction of the top-most call frame until the call stack is empty. In particular, the executor itself does not maintain any program counters, which is done by the implementation of the call frame instead. The actual instructions are loaded into memory only once during module creation, eliminating unnecessary copy operations. Additionally, we provide our own stack implementation since in kernel space we do not have access to the `std` stack.

5. Runtime Implementation

In this section, we present the most important aspects regarding the functionalities of the kernel module and their design decisions.

5.1. Module Instantiation

A WebAssembly module is the stateless, compiled representation of a WebAssembly program. It represents a self-contained binary artifact that declares types, functions, imports, and exports, and is independent of any specific runtime environment. Module Instantiation is the process by which the runtime derives a Module Instance, an executable representation, from a stateless module. As shown in Figure 3, the instance does not hold runtime state directly, but instead contains address references into the store, the central runtime state container, resolving all function, table, memory, and global definitions. While a module remains immutable, it is the module instance that the runtime operates on during execution.

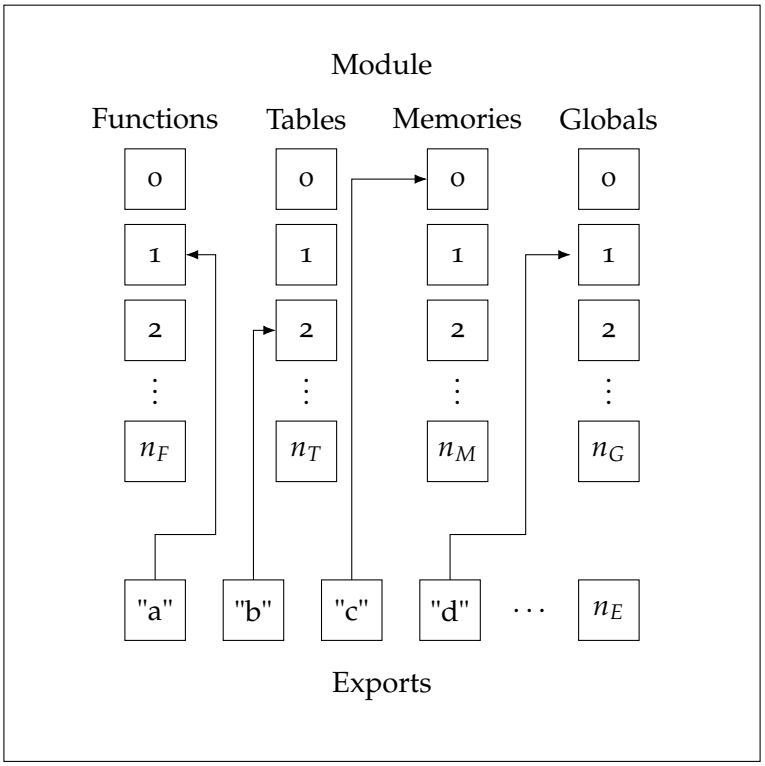

Figure 3: Structure of a decoded module. It consists of four lists for storing functions, tables, memories, and globals. While our runtime supports multiple memory and table sections the implemented specification expects at most one memory and one table. Additionally, it also contains a map of names (strings) and indices into one of the other lists (e.g., an index into the function list). Each export can reference only one function, table, memory, or global. This allows other modules to import parts of this module during instantiation.

Since a module is stateless and immutable, the WebAssembly specification permits a single compiled module to be instantiated multiple times, each producing

an independent module instance with its own store. Although the current implementation does not support concurrent module instances, the architecture was designed with this in mind, establishing the foundation for a future extension in this direction.

Our implementation follows the WebAssembly specification model closely. The `Module` struct serves as the stateless, decoded representation of a binary artifact and is not directly executed. Instantiation allocates all runtime entities into a central state container (`Store`), holding function, memory, global, and table instances as well as data segments, from which an executable module instance is derived.

While the current implementation supports multiple module instances within a single store, full multi-threading support remains a conscious simplification made within the scope of an MVP and is a candidate for future work.

Instantiation is performed by the instantiate method on `Module`. It begins by invoking `ModuleInstance::allocate`, which resolves all declared imports. For this, it looks up the corresponding exported entities from already-registered instances in the shared store and initializes all function, memory, table, and global references within the new instance. Subsequently, active element segments are written into their corresponding tables and active data segments are written into linear memory, both written to their statically defined offsets. The resulting instance is then registered in the shared store. Two aspects of the specification are intentionally not implemented within the project scope. First, module validation prior to instantiation is omitted. Second, the specification prescribes that an optional start function, if declared, is invoked automatically as the final step of instantiation. In our implementation, the start field is recorded on the instance but not automatically executed. Instead, the entry point is selected explicitly by the host after instantiation, as omitting module validation makes automatic start function execution unsafe at this stage.

5.2. Host functions

WebAssembly is meant to be executed in an isolated and sand-boxed environment. Thus, a pure WebAssembly module does no depend on platform-specific capabilities. It cannot perform I/O, allocate host memory, or interact with the OS directly. To bridge the execution environment with the host world, WebAssembly defines a system of *imports* and *exports* that have to be declared explicitly.

Imports define the external entities a WebAssembly module requires from its host environment to function. The host has to satisfy all dependencies during the instantiation phase:

- **Functions:** Callable host code. In our kernel runtime, these are wrapper functions exposing internal Linux kernel APIs (e.g. `pr_info` for logging, or network socket operations).
- **Tables:** Arrays of opaque values (typically function references) used primarily for indirect function calls and dynamic dispatch.

- **Memories:** A contiguous array of raw bytes serving as the module's linear memory. The runtime environment (referred to as *host*) can provide this memory to copy data structures to the interpreted language (referred to as *guest*).
- **Globals:** Scalar values that can be read by both the guest and the host, useful for passing configuration states.

Exports are the inverse of imports. They are the internal WebAssembly entities made available to the host environment. The host can optionally choose to bind them during initialization but does not have to.

- **Functions:** The primary entry points into the WebAssembly logic. The host invokes these exported functions to execute the guest application (e.g., by invoking the entry-point of a application or calling a *Netfilter* hook callback).
- **Memories, Tables, and Globals:** Similar to imports, memories, tables, and globals can be exported so the host environment can inspect or mutate internal state of the WebAssembly instance.

As WebAssembly's origin lie in the web browser world, it is crucial for a browser-based WebAssembly runtime to also expose functions that JavaScript can invoke like `console.log`, while importing JavaScript functions to interact with the broader JavaScript ecosystem. However, when bringing to the privileged WebAssembly of an out-of-tree Linux kernel module, our host function interface must act as a bridge between native C kernel space and the WebAssembly virtual machine. Data passed across this boundary, such as pointers to kernel memory mapped into the WebAssembly linear memory, must be rigorously managed to maintain the integrity of the kernel and prevent system panics.

Our runtime implements host functions under the module name `kwasmrt` (**K**ernel **WebAssembly R**unt**i**me). In `host_imports.rs`, the resolver maps imported names such as `read_fd` into Rust callback methods on our `ModuleInstance` (see subsection 5.1). It checks that the imported WebAssembly signature matches the expected host signature, which prevents accidental binding of incompatible imports. If unknown imported environments or methods are found the instantiation of a module fails, since our runtime might not be able to execute the given binary properly.

5.2.1. Minimum Viable Product

To get our demonstration application[8] working, we implemented a minimum API of host functions that can be imported. Namely for sending and receiving network traffic on a socket buffer (`create_tun()`, `read_fd()`, `write_fd()`) and also appending to the kernel log (`log()`). For hooking into *Netfilter*, during initialization, we are resolving if a module specifically exports a function of name and signature `nf_hook_callback(buf: *mut u8, len: usize) -> u32`.

[8] `https://gitlab.com/hpi-potsdam/osm/webassembly-kernel-runtime/demo`

At execution time, each host function is implemented as a method of `ModuleInstance`. The runtime keeps a small table of host-side file descriptors (`host_fds`) and assigns them monotonically. The socket-create procedure (`host_create_tun`) creates such an entry and returns the corresponding integer descriptor to the WebAssembly guest. `host_read_fd` and `host_write_fd` validate their arguments, access guest linear memory and then copy data between guest memory and the host-side object. `host_log` reads a byte slice from guest memory and emits it through kernel logging (`pr_info`).

The callbacks call C functions from `host_io.c`. That layer allocates a host I/O object around a kernel raw ICMP socket, performs non-blocking `kernel_recvmsg`/`kernel_sendmsg`, and normalizes transient `EAGAIN/EWOULDBLOCK` conditions to zero-length operations.

The host interface remains narrow. Only basic packet I/O and logging are implemented, while richer host services and more general networking abstractions are still missing.

5.3. User Space Interaction Through a File System API

A WebAssembly runtime on its own is not usable as long as no modules are loadable. This is also true for other kernel runtimes. We could identify two common approaches for kernel interactions.

Requests The first approach is using requests for communication. This is done by writing requests to a character device or `netlink` sockets. We also based our initial implementation on this approach using `netlink` sockets send module load request to the kernel runtime. Additionally, other implementations register a character device to do the same[9].

`ioctl` The second approach also proposes character devices, but instead of writing to these devices, the runtime handles `ioctl` requests[10] to load modules.

While both approaches are suitable, we propose using a virtual files system based on the UNIX philosophy of representing everything as a file. We can use this, to expose runtime internals to user space applications. This comes with the benefits, that no special utilities are required to inspect or modify the runtime (e.g., ls can be used to list all running instances). Additionally, we can also stick to common Linux security practices for protecting access to the runtime, like file permissions and ownership, or SELinux policies.

For implementing this file system, we decided to expose the stored modules, instances, and memories as directories. Additionally, we decided against exposing tables, because in WASM Spec 1.0 tables only contains references to functions[11]. Because no VFS Rust bindings exist, we decided to write our file system in C. On

[9]https://github.com/cisco-open/camblet
[10]https://github.com/Faisal-Saleh/kernel-wasm-runtime
[11]https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/#table-types

the other hand, we need to bridge between the C file system and the Rust runtime now. For the C file system, we do this by exposing the initialization and destruction functions for the file system. To ensure that these functions are easily portable between Rust and C, we expose them in a separate header without any includes. This is necessary, as it is not trivial to call `bindgen` with the correct arguments including all defined preprocessor values and because including kernel headers may cause duplicate definitions inside the Rust part (e.g., some C structs are imported as part of the kernel bindings as well as part of the file system bindings). For the Rust runtime, we create a set of functions to allow the manipulation of runtimes, modules, and instances. To access these C structures, we expose them as opaque C pointers. This setup allows us to expose a header file without other includes or exposed rust structs and therefore makes runtime implementation details transparent to the file system implementation. Additionally, we base our implementation on Linux implementation of `ramfs`[12]. We modify this file system by implementing custom `inodes` and file system operations for the *modules*, *memories*, and *instances* directories.

5.3.1. The Modules Directory

We expose the *modules* directory as interface to the runtime's module store. Inside this directory, we map file and `inode` operations to the corresponding store operations. For that, we implement custom handler for the file creation workflow, see Figure 4.

On lookup, we call the runtime to find a module with the same name as the file to lookup. If this function does not find a module, we let the lookup call return with no file found. This causes the Linux kernel to call the create function. Because we do not know the binary encoded module yet, we create an empty module entry for our module. After that, it is guaranteed that a module with that name exists. This simplifies the implementation of the open function, as we can store a pointer to our module with the file. We use this pointer, to read from this module or to write a new module. If the file is closed by the user space application, the kernel calls the release function. After this function is called, we know that the binary module is complete. Therefore, we can decode the module and instantiate a new instance. At the end, we update the file system to show the correct files inside the `instances` and `memory` directory. To handle concurrent read and write, we decided on a copy-on-write (cow) approach, which is used by Linux in different places. Using this approach, we first store a reference to the original encoded binary module data and copy this data if write is called.

5.3.2. The Instances Directory

The second directory implemented as part of the file system is the `instances` directory. This directory is used to expose status information about all instantiated

[12] https://github.com/torvalds/linux/tree/master/fs/ramfs

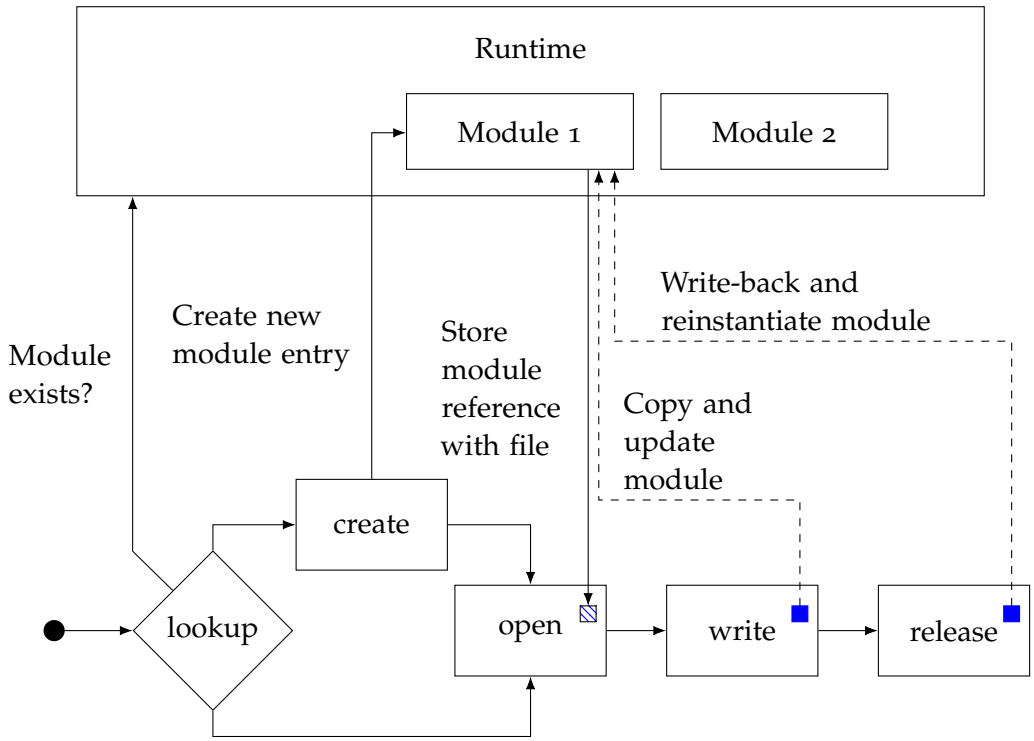

Figure 4: Access to the runtime during the handling of IO-operations on files inside the *modules* directory. While lookup and create access the runtime to check modules or create new ones, both do not reference a module. This is done inside the `open` callback by storing a reference with file's private metadata. We use this reference to copy the module's data on write and also to update the old module if we release the file.

modules and is used to invoke function inside that module. Therefore, this directory exposes for each module the following files:

- The `status` file is used to expose the state of the instance. Therefore, it contains the last error encoded as number and short description separated by a space.
- The `invoke` file is used to start a new function. In the following, we use invocation as name for a running function. To invoke a new function the name of the corresponding export and the required parameters must be written as text into the file. Therefore, the following syntax must be used: The name of the function separated by a white space character followed by a list of space separated parameters, which are encoded as type (e.g., `i32` or `i64`) followed by colon and a text encoded number. This call must be submitted as part of a single write call, and the result of that invocation will be readable from the file after the invocation completes. Unlike the invocation, the result is binary-encoded. It consists of a magic value `KWR1` followed by one byte containing the invocation status. At last the return value is encoded by one

type identifier followed the number encoded as byte string using native byte order.

- The `running` file is used to provide the number of running invocations. This number is not exposed as human-readable number, but as machine-readable number with native byte order.
- The `stop` file is used to stop all running invocations by writing to this file.
- The `alias` file is used to set a different name for instantiated module and is used by the spec tests for compatibility reasons (see section 6.1.1).

5.3.3. The Memories Directory

The last directory exposed as part of the runtime's file system is the `memories` directory. We implemented this directory to expose all memories requested by a module. Meaning for each instantiated module, we create a list of file named after the index of the memory. Each of these memory index file can be read and written to. In this function it is similar to the `/proc/<pid>/mem` file, provided by `procfs` and comes with similar security risks. These files should only accessible by privileged processes, because they allow modifications of an instances state. Unlike normal processes, these files do not contain memory with WASM instruction or the working stack. This is due to the separation of runtime stack, instructions, and memory inside the WASM runtime. Because of that, we argue that it can only cause the unexpected behavior while executing an WASM instance, but no escape from runtime to kernel.

6. Tooling

In this section, we present our main tooling components that we developed next to the kernel module itself and how they work.

6.1. Specification Tests

To systematically evaluate the compliance of our implementation with the standards, we have developed a dedicated *Spectest harness*. It executes the official WebAssembly 1.0 specification test suite against our runtime, enabling a structured assessment of functional correctness, unsupported features, and remaining compliance gaps.

6.1.1. Implementation and execution model

The official tests are published in the official WebAssembly specification repository on GitHub [4]. These tests are provided as scripts in the `.wast` format, i.e., small specification-driven programs that describe module definitions, invocations, and expected outcomes such as return values, traps, or validation failures. For execution in our harness, these tests are converted with `wast2json` into a JSON description

together with the referenced `.wasm` modules. This representation allows the harness to process the test commands sequentially in a machine-readable form.

The harness interprets this JSON representation of the official test suite. More concretely, each JSON file contains an ordered list of commands corresponding to the original specification script, such as `module`, `register`, `action`, `assert_return`, or `assert_trap`. The `types` module deserializes these commands into Rust enums, and the `harness` module then executes them sequentially while collecting detailed statistics. Along with the total number of tests passed or failed, it distinguishes between setup commands (e.g., loading or registering a module), assertion commands, and other commands. In addition, the harness tracks *policy-skips*. This category is used when a test is not considered a semantic error of the current implementation, but lies outside the intentionally supported subset, particularly for floating-point functions.

A small synthetic `spectest` module is created at startup from an embedded WebAssembly Text (WAT) representation stored as a string literal. It exports the imports expected by the official suite, including printing functions, integer and floating-point globals, a table, and a memory. As a result, these imports do not need to be treated as special cases in the runtime itself, and the test configuration largely corresponds to the conventions of the upstream WebAssembly tests.

6.1.2. Backend abstraction

A useful architectural feature of the current harness is the trait `SpectestRuntime`. It abstracts the operations that the harness requires from a backend:

1. loading a module from bytes,
2. registering an alias for a module,
3. invoking an export of a module.

Owing to this abstraction, the harness logic itself remains generic. Consequently, the exact same spectest interpreter can be used for different execution backends. Although these components represent backends from the perspective of the harness, the term `Runtime` is used consistently in the implementation because each backend provides the runtime interface required by the trait.

This project currently implements two distinct backends:

`HostRuntime` This backend embeds the normal runtime environment directly into the user space process. Modules are registered through `Runtime::register_from_bytes`, aliases are forwarded to `Runtime::register_alias`, and exported functions are invoked through `Runtime::invoke_export`. This path is convenient for fast local testing and continuous integration because it requires no running kernel module while still exercising the project's runtime, decoder, and executor. It is used by default unless the kernel-VM backend is selected explicitly.

`KernelVmRuntime` This backend uses the actual kernel-module execution path through a running Linux virtual machine. It wraps a `KernelVmClient`, uploads modules into a running VM, ensures that the kernel module is loaded there, reads the module status from the mounted `kwasmrt` file system, registers aliases through the file system interface, and invokes exports through it. The kernel-VM backend can be selected by passing the `--kernel-vm` command-line flag when executing the spectest harness.

6.1.3. Command execution and result classification

For each JSON file, the harness runs the commands in the order in which they appear in the source. A `module` command loads a module from disk and optionally saves its name as the current default module. If the test does not specify a name, the harness assigns generated names such as `$anon0`, `$anon1`, and so on. A `register` command creates an alias either for an explicitly named module or, if no name is specified, for the most recently loaded module.

Execution commands are processed directly. Simple `invoke` and `action` commands must be executed successfully without trapping. `assert_return` compares the returned value with the expected result, currently only for calls with a single result. `assert_trap` is successful if the runtime reports a matching trap category. `assert_return_canonical_nan` and `assert_return_arithmetic_nan` directly check the returned IEEE-754 bit pattern.

For negative tests such as `assert_malformed`, `assert_invalid`, and `assert_unlinkable`, the test harness does not execute the module in the same way as it executes modules in positive tests. Instead, it first classifies the module. The current classification logic analyzes the module structure, decodes sections, checks some index bounds, validates certain properties of the instruction stream, deliberately ignoring certain opcodes for policy reasons, and then assigns one of four classes: malformed, invalid, unlinkable, or valid. This is a practical compromise, but also the main reason for the remaining compliance gap documented in subsection 7.2: the classifier is not a complete WebAssembly validator.

6.1.4. Summary

The *Spectest harness* fills a gap in the development tooling. Unit tests can verify individual statements or helper functions, and application demonstrations can show that a specific use case works. However, neither of these approaches provides systematic evidence that the runtime behaves according to the WebAssembly specification in many edge cases. The *Spectest harness* provides precisely this form of evidence.

Because it reuses the project-specific runtime implementation, its errors are actionable. Each error indicates either a concrete error in runtime behavior or a missing validation rule. This makes the harness not only a regression tool for CI, but also a road map for future work on standards compliance.

6.2. Continuous Integration

After introducing the spec test harness, it is useful to show how this tooling is integrated into the development workflow of this project. A central aspect of this integration is our Continuous Integration (CI) setup, which we will present in this chapter.

6.2.1. Goals of the CI pipeline

The CI pipeline is designed to cover the most important quality control checks for the project:

- **Basic code quality checks:** Formatting is automatically checked so that style issues are detected early on.
- **Reproducible builds:** The kernel module is built in a clean container environment with a fixed Rust toolchain and all necessary system dependencies.
- **Regression testing on multiple levels:** In addition to the usual tests at the Rust level, the pipeline also performs tests that start a virtual machine, load the kernel module, and check the behavior in the kernel.
- **Specification-oriented testing:** The spectest harness runs automatically in CI, so changes to the decoder, executor, runtime environment, or harness logic are immediately reflected in the measured WebAssembly 1.0 conformance.
- **Documentation deployment:** The pipeline also creates project documentation and publishes it automatically.

This combination is particularly important in this project, as errors can occur at different levels: in pure Rust code, in the module build process, in interaction with the Linux kernel, or in WebAssembly semantics.

6.2.2. Multi-architecture support

The project uses *GitLab CI* and structures the pipeline into the phases `check` (consistency checks), `compile` (module compilation), `unit-test` (functional testing), `spec-test` (specification-oriented testing), and `deploy` (documentation deployment). These stages reflect the intended development workflow.

An important property of this setup is its multi-architecture design. The pipeline is configured for both `x86_64` and `arm64`. At the same time, the exact test coverage differs between microarchitectures: while both targets are built and tested at the Rust level, jobs that rely on virtualization support are currently only available on `x86_64`, since the available `arm64` runner does not support nested virtualization. In principle, additional microarchitectures can be integrated by providing the corresponding toolchain, target configuration, and runner infrastructure.

6.2.3. CI stages

As illustrated in Figure 5, the central CI stages of the project are `compile`, `unit-test`, and `spec-test`. In the `compile` stage, the jobs `build-kernel-module:x86\`

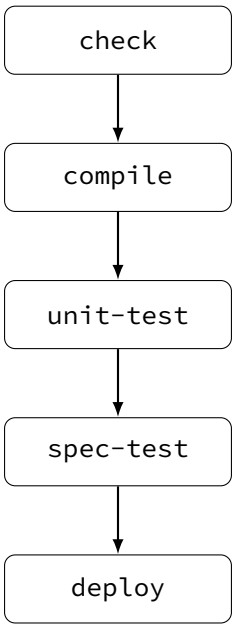

Figure 5: Simplified workflow of the CI pipeline stages.

`_64` and `build-kernel-module:arm64` build the kernel module for the respective target microarchitecture. The subsequent `unit-test` stage contains both host-side tests, such as `test-runtime:x86\_64` and `test-runtime:arm64`, and VM-based kernel tests such as `test-kernel:x86\_64`. Finally, the `spec-test` stage integrates the *Spectest harness* through jobs such as `test-wasm-spec:x86_64` and `test-wasm-spec:arm64`.

Together, these stages ensure that the project is checked on several levels: successful compilation of the kernel module, correctness of host-side runtime functionality, execution in a realistic virtualized kernel environment, and compliance with the WebAssembly 1.0 specification.

In-kernel tests in a virtual machine. Another noteworthy part of the CI configuration is `test-kernel:x86_64`. This job installs QEMU and associated VM tools, starts a custom Debian guest image, copies the compiled kernel module to the guest via SSH/SCP, unloads all previous module instances, inserts the current module with `insmod` (kernel module insertion), and then checks the kernel log with `dmesg`. The test is only successful if the module reports a successful summary in the kernel log.

This is an important aspect of the pipeline, as it tests the system in a configuration that is much closer to the actual deployment scenario than ordinary host-side unit tests.

Architecture-specific limitation on arm64. The corresponding `arm64` kernel test job is present in the CI file, but is currently disabled. The reason for this is that the available `arm64` runner does not support nested virtualization.

6.2.4. Integration of the spectest harness

The job `test-wasm-spec:x86_64` (and analogously `test-wasm-spec:arm64`) integrates the *Spectest harness* into our CI setup by downloading the WebAssembly test suite and executing the harness afterwards. In addition, the pipeline extracts a numerical compliance value directly from the harness output using a regular expression. This converts the printed summary into a machine-readable CI metric. The harness also writes a JUnit report to `target/junit.xml`, which is uploaded as a GitLab artifact and as a structured JUnit test report. This means that the results of the specification tests are not only visible in the raw console log, but also integrated into GitLab's test report interface.

6.2.5. Documentation and deployment

Beyond testing, the pipeline also includes a small deployment path. A `rustdoc` job generates documentation for internal crates, including private elements, using the Rust-for-Linux target configuration. A `pages` job then publishes the generated documentation via *GitLab Pages*. This deployment step is limited to the `main` branch.

6.2.6. Summary

Overall, the CI configuration is well suited to the requirements of the project and achieves the following:

- It is not limited to ordinary unit testing, but also includes VM-based in-kernel validation.
- The WebAssembly specification test harness is integrated as an explicit pipeline stage and delivers results that are readable by both humans and machines.
- The pipeline is architecture-aware and already prepared for `x86_64` and `arm64`, even though the latter currently has infrastructural limitations for kernel-level testing.
- Generating documentation is part of the same automated workflow.

In summary, the CI pipeline supports the engineering methodology of the project. It not only continuously checks whether the code compiles, but also whether the kernel module behaves correctly in a virtualized Linux environment and whether the implementation continues to progress towards WebAssembly 1.0 compliance.

6.3. Module Runner

As a supplement to automated `spectest`-based evaluation and the CI pipeline, the project also includes a small *module runner* in user space. Its purpose is not large-scale conformity testing, but rather the quick manual execution of individual WebAssembly modules during development. In this sense, the module runner bridges the gap between low-level unit tests and end-to-end scenarios. A developer can compile or customize a single module, load it into the runtime environment, call a selected export, and check the result directly.

Architecturally, the module runner reuses the existing project-internal execution components. Instead of implementing a separate execution environment, it uses the same project-internal components that are also used by the *Spectest harness* presented in subsection 6.1, namely the decoder, the runtime environment, the executor, the value representation, and the error handling. This avoids a second, different code path just for manual execution of modules.

From the user's perspective, the tool provides a concise command line interface. A module path is required, while the exported function can be selected optionally. Function arguments can be passed in typed form, such as `i32:1` or `i64:-2`. In addition, the runner supports a `--kernel-vm` mode and an optional invocation timeout. This makes the tool useful in two different situations. In standard mode, it quickly executes a module on the host using the common runtime implementation, while in kernel VM mode, it executes the realistic deployment path via a running virtual machine.

In host mode, the module runner reads the input module, translates the textual `.wat` representation into binary WebAssembly, registers it in the runtime environment, and then calls either an explicitly selected export or an implicit default entry point. The default call path tries the conventional entry functions `_start` and `main`. This mode is particularly useful for quick debugging, as it does not require a booted VM and provides immediate feedback on instantiation and execution errors.

The path for more comprehensive testing is the kernel-supported execution mode. Here, the module runner uses the same SSH-based VM client infrastructure as the *Spectest harness*. Details on the interaction with the kernel can be found in subsubsection 6.1.2.

For practical development, the module runner also improves diagnostics. If a module fails to load because imports cannot be resolved, it derives a human-readable hint by checking the import section of the module. Kernel-side status codes are translated back into runtime errors, and optional invocation timeouts prevent the tool from blocking indefinitely during debugging sessions.

Overall, the module runner is a useful development tool. It is particularly relevant for ad-hoc debugging of host imports and for validating the complete path from a command in user space to execution within the kernel-supported runtime environment.

7. Evaluation

Evaluation was conducted by deploying the kernel module inside a QEMU-based virtual machine running a rust-enabled linux kernel. The VM is managed via a set of shell scripts: `start.sh` boots the guest image, and `build-and-reload.sh` compiles the kernel module and installs it into the running VM via SSH and SCP. Correctness was assessed on two levels: host-based integration tests exercise the decoder and executor without kernel overhead, while in-kernel tests load the module via `insmod` and verify the kernel log output with `dmesg`. Specification compliance

was measured using the *spectest harness* described in Section 6.1, executed against the official WebAssembly 1.0 test suite.

7.1. Rust for Linux

The kernel module builds on Rust for Linux (R4L), a project that enables writing Linux kernel modules in Rust by providing safe wrappers around kernel C APIs [9]. The module targets a custom fork of the R4L tree (branch `wasm-runtime`, commit `e30eed90`), built with Rust toolchain version 1.91.1.

At the beginning of this project, R4L was still under active development and imposed several constraints on what was available to kernel module authors. Notably, R4L transitioned from an experimental addition to an established part of the kernel over the course of this project. First merged in Linux 6.1, it was formally declared no longer experimental at the Linux Kernel Maintainers Summit in December 2025 [7]. The limitations described below reflect the state of R4L as encountered during development.

Limited C API coverage. Only a subset of kernel subsystems has received Rust bindings. For example, the VFS does not yet expose a Rust interface, yet a file-based interface for loading `.wasm` binaries and inspecting runtime state is desirable. Rather than writing unsafe C bindings by hand, we implemented the file system layer in C and provided thin wrappers on the Rust side to interact with the runtime as described in Section 5.3.

Incomplete standard library. R4L exports the `core` crate in its entirety, but provides a custom implementation of `alloc` that differs from the standard library counterpart. Some data structures are missing or behave differently. For instance, `BTreeMap` is not implemented. The kernel provides a red-black tree (`RBTree`) instead. More subtly, types such as `Vec`, `Arc`, and `Futex` exist in both environments but carry different allocation semantics. Kernel allocations are fallible by design and require explicit `GFP_KERNEL`[13], whereas `std` allocations panic on failure by default.

Limited tooling support. Unlike typical Rust projects, the kernel build system does not use Cargo but `make` with `Kbuild`. As a consequence, rust-analyzer cannot discover the crate graph automatically, and IDE features such as autocompletion and inline diagnostics do not work out of the box. To address this, the build scripts were extended in two ways. First, `devel.sh` generates a `rust-project.json` file by constructing a crate-graph template from kernel build artifacts, pointing rust-analyzer at the correct sysroot and kernel crates. Second, `ra-check.sh` runs the build with `KBUILD_RUST_ANALYZER=1`, a flag enabled by modifications to the kernel Makefiles, which causes `rustc` to emit JSON-formatted diagnostics. These are filtered by `splice-json.py` and

[13] GFP (Get Free Pages) flags specify the allocation context in the Linux kernel. `GFP_KERNEL` denotes a normal, potentially blocking allocation.

path-normalized by `ra-normalize-paths.py` before being consumed by rust-analyzer via `rust-analyzer.check.overrideCommand`. This approach has limitations: paths may be incorrect if `make` is invoked from within a container, and rust-analyzer auto-completion remains partially unreliable due to the non-standard build environment.

To bridge this gap without duplicating the executor logic, we introduced the `bindings`[14] crate. It abstracts these differences using `#[cfg(MODULE)]` conditional compilation: under a kernel build, the types resolve to their R4L counterparts;[15] in a host build, they resolve to thin wrappers around standard `alloc` types. All allocation methods accept a GFP flags parameter, which the host wrappers ignore. This allows the decoder and executor[16] to be compiled and tested on the host without modification. Listing 1 illustrates this pattern for `KVec`: The `bindings` selects the appropriate implementation at compile time, and `common`[17] re-exports both `KVec` and `GFP_KERNEL` under a single path (line 3). All shared code — including the host-based tests — can therefore use a single import regardless of the build target.

```
// kernel::alloc::kvec (Rust for Linux)
pub fn push(&mut self, v: T, flags: Flags) -> Result<(), AllocError> {
    self.reserve(1, flags)?;  // may fail: propagates AllocError
    // ...
    Ok(())
}
// crates/bindings/src/vec.rs (host wrapper)
pub fn push(&mut self, value: T, _gfp: ()) -> Result<(), Infallible> {
    self.0.push(value);  // infallible: panics on OOM like std
    Ok(())
}
// shared executor code -- identical call site on both targets
vec.push(element, GFP_KERNEL)?;
```

Listing 1: Adaptation of `KVec::push`: the host wrapper accepts `()` and returns `Infallible`, while the kernel implementation takes `Flags` and may return `AllocError`. Because `GFP_KERNEL` resolves to `()` on the host and to the actual flags constant in the kernel, the call site is syntactically identical across both targets; the `?` operator handles the differing error types via `From` conversions.

The approach supersedes an earlier design in which a common trait had to be declared for every kernel type and then implemented twice: once against the

[14] `crates/bindings`
[15] `kernel::prelude::KVec`, `kernel::sync::Arc`, etc.
[16] `crates/common`
[17] `crates/common`

R4L type and once against the `std` equivalent. Maintaining consistent behavior across two implementations proved error-prone and introduced repetitive code that obscured the actual logic. The `#[cfg(MODULE)]` re-export pattern eliminates this entirely and may be a useful pattern for other Rust-for-Linux projects that require user space testing.

The two signatures are not type-compatible — `()` (the unit type) and `Flags` are distinct types, as are `Infallible` and `AllocError`. This pattern works because `#[cfg(MODULE)]` selects different type definitions per build target before the call site is type-checked, allowing the same source to compile against either implementation without modification. Shared code calls `vec.push(val, GFP_KERNEL)?`, where `GFP_KERNEL` is `()` on the host and the actual flags constant in the kernel. Because the types are selected by the `#[cfg(MODULE)]` re-exports before the call site is type-checked, the compiler sees a consistent signature in each build. The `?` operator requires only that the callee's error type (`Infallible` or `AllocError`) implements `From` for the enclosing function's error type, which is satisfied in both builds, as `From` is implemented for the project's `WasmError` type. The syntax of the call site is therefore identical across targets, while the semantics differ.

One limitation worth noting is the host wrappers always return `Infallible` as the error type for allocation operations, meaning allocation failures can never be simulated. If a project requires testing out-of-memory behavior, the host wrapper would need to be extended with a configurable failure mode. Rust's unstable `allocator_api` would provide a cleaner approach via `Vec::try_with_capacity`, though it remains nightly-only at the time of writing.

7.2. WASM Specification Compliance

The current compliance reported by the spectest harness, presented in section 6.1, is 71.20%, with 3982 passed assertions, 1611 failed assertions, and 1121 assertions skipped due to policies. Since assertions skipped due to policy are excluded from the conformance denominator, this number should be interpreted as "conformance within the currently intended subset" rather than full conformance with WebAssembly 1.0. When the policy-skipped assertions are included in the denominator, the overall conformance with respect to the complete WebAssembly 1.0 test suite is approximately 22% to 23%.

The error distribution shows a very clear pattern: the predominant problem is missing or incomplete validation, not the basic execution of already accepted modules. Most errors (903) are `assert_invalid` checks where the harness classifies the module as `Valid` even though the official suite expects a rejection. 791 (87.60%) of these invalidity errors are due to *type mismatches*. This cluster is spread across many folders, including `block`, `if`, `i32`, `i64`, `store`, `load`, `local_set`, and `local_tee`. This is due to the current classifier only performing structural validation and some targeted semantic checks, but does not perform a complete type check of the instruction sequences, labels, local variables, and control flow links.

A second cluster concerns the processing of malformed modules. In particular, the UTF-8-related suites (`utf8-custom-section-id`, `utf8-import-field`, and

utf8-import-module) fail completely. Here, the expected result is assert_malformed, but the observed classification is often Unlinkable. This is caused by the fact that malformed names currently flow too far into the subsequent import resolution instead of being rejected during decoding.

A third cluster concerns linking and instantiation semantics. All assert_unlinkable errors are currently observed as Valid. The current classification logic only marks a module as unlinkable if an import cannot be resolved, but does not yet model more complex errors at instantiation time, such as incompatible import types or out-of-bounds initialization of data/elementsegments. This explains the repeated failures in imports, data, elem, and linking.

There are also a few smaller but still instructive semantic gaps:

- **get actions:** several failures in exports and linking are caused by the fact that assert_return currently only supports invoke-actions, not global get-actions.
- **Start semantics:** the start folder contains both validation failures (start.0–start.2) and state mismatches after instantiation (start.3, start.4), caused by incomplete handling of start-function validation and/or execution.
- **Table/element behavior:** several failures in elem, imports, and linking are caused by mismatches between the expected trap text "uninitialized element" and the currently reported runtime error TableElementNotFound, as well as wrong return values after element initialization.
- **Name transport corner case:** the names suite almost passes completely, but one remaining failure with a control-character-heavy export name highlights a remaining edge case in name transport or invocation.

Overall, the evaluation shows that the WASM runtime already provides broad support for the instruction set of the WASM 1.0 specification. The main remaining limitations are not located in the execution of valid modules, but in the surrounding runtime features, in particular module validation, instantiation checks, and parts of the linking semantics. These gaps currently have a significant impact on formal specification compliance, as reflected by the failed negative tests and the reduced overall conformance score. At the same time, the results indicate that semantically correct and structurally valid WASM modules can already be executed reliably in the current system.

8. Future Work

The current implementation successfully demonstrates the feasibility of sand-boxed, architecture-independent kernel extensions. But still critical milestones remain to label this prototype a production-ready platform. Future research must expand on the host function interface to achieve functional parity with the eBPF helper ecosystem, alongside the integration of module validation to address semantic

vulnerabilities. Additionally, improving the runtime through the WebAssembly System Interface (WASI) Component Model, JIT or AOT compilation, and per-process isolation will be essential for robust data marshaling, low-latency execution, and secure multi-user environments. The following sections detail these prospective developments across the architectural, functional, and infrastructural domains of the project.

8.1. Kernel APIs

To demonstrate a working WebAssembly runtime in a kernel environment, we built a minimal prototype application that can intercept and handle network requests. Our efforts went into providing this minimum viable product with the required API for intercepting and responding to network traffic. Our environment only offers the functionality to interface with the kernel *Netfilter* hook API[18] to respond to intercepted traffic and additionally create sockets to send and receive packets.

To constitute a useful alternative to eBPF programs however, a WebAssembly runtime should offer a broad range of available kernel APIs comparable to the eBPF helper function ecosystem[19]. This would allow the runtime to become a general-purpose kernel extension platform.

An advantage of using the WebAssembly specification is its explicit import and export declarations. We think of a capability-based security model for future versions. The runtime can perform validation before loading a module into the runtime and by inspecting a module's `import` section and comparing these to the allowed capability level. Loading of the module can be interrupted, if an API is to be imported that is not explicitly permitted (e.g., a packet-filter module that requests file system write functionality). This would offer a more flexible approach than current kernel-module development, while on the other hand still retaining the sandbox-approach offered by alternatives.

8.2. Component Model

After adding the means to invoke functions from the host and the reverse to a runtime it then has to permit transferring data between them. Since our MVP just needs to work on socket buffers, copying raw bytes from host to guest and vice versa suffices for our use case, but it has security implications as we are allowing the sandboxed application to handle raw bytes. The Component Model [20] that is specified since WebAssembly System Interface (WASI) Preview 2 [21] can be used for exactly this. The runtime's canonical ABI would then handle the marshaling of kernel data into the WebAssembly sandbox, ensuring that the module never accesses out-of-bounds kernel memory.

[18] `https://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-4.html`
[19] `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`
[20] `https://github.com/WebAssembly/component-model/blob/main/README.md`
[21] `https://github.com/WebAssembly/WASI/blob/main/docs/Preview2.md`

8.3. Per-Process Isolation

In the current state any user on the system can read, modify, and execute WebAssembly modules in the kernel using User Space Interaction Through a File System API since `other` is permitted `rwx` on every file. This can be easily restricted with file permissions on the directory structure. In our opinion this can be even more elegant by attaching the WebAssembly module to an owner process. Since we already provide a file based interface for interacting with modules (see The Modules Directory) it should be straight forward to move the current mount point into a process specific directory (e.g. `/proc/<pid>/kwasmrt/`).

8.4. Encoded Execution

For critical applications, correct computation is of high priority. In respective use cases, increased resource usage can be traded-in for increased fault tolerance. One approach to achieve increased fault tolerance for computations is encoded execution. In encoded execution, e.g., arithmetic operations are executed redundantly with different representation of the data. For example, summands can be multiplied with a factor before addition, where after the sum of the addition would have to be divided by the same factor. If the result equals the same operation without the multiplications, the result is correct with a higher probability compared to non-redundant execution. Because our implementation has full control over the module execution, it also offers prospects to implement encoded execution.

8.5. Performance

Optimizing latency and throughput has not been prioritized during planning and implementation. We think it would be interesting to evaluate whether either ahead-of-time compilation (As other, like WasmEdge [22] and WAMR [23] implemented) or just-in-time compilation (see Wasmtime [24], WAMR [23], Wasmer [25] and V8 [26]) would fit the constraints in a kernel context better.

8.6. CI/CD and Development Infrastructure

While the previous sections focus on the architectural opportunities for future improvements, the current development workflow also highlights possible enhancements. Currently, our CI/CD pipeline sets up a whole development environment reinstalling and building every dependency for every job. While this promotes

[22] https://wasmedge.org/
[23] https://bytecodealliance.github.io/wamr.dev/
[24] https://wasmtime.dev/
[25] https://wasmer.io/
[26] https://v8.dev/

simplicity and reproducibility, it leads to inefficient resource usage and longer job runtime.

A future version should implement artifact caching to reuse pre-built kernel headers. Additionally, while specification tests (see subsection 6.1) are validated on `x86_64`, it would be beneficial to enable full kernel-level testing on `arm64` as soon as a suitable runner infrastructure is available to truly support both architectures.

9. Conclusion

In this project we have explored the feasibility of using WebAssembly as a mechanism for extending the Linux kernel. By implementing our own WASM runtime based on the *Rust for Linux* project, we have shown that WebAssembly is usable as an alternative for safe and portable kernel extensibility when compared to established approaches, such as traditional kernel modules or eBPF (see Table 1 for an extensive comparison).

Our architecture follows a modular design consisting of a decoder, executor, and runtime. This enables a clear separation between parsing, execution, and state management. By sharing core components between kernel and user space, the implementation achieves a high degree of code reuse and testability as well as improved development speed. Also, testing against the official WASM specification test suite has provided a measurable notion of correctness that improved development speed as well. In addition, integration with user space via a virtual file system further demonstrates that interaction with WebAssembly modules running in kernel space can follow established Unix principles, allowing intuitive inspection and control without specialized tooling.

As has been demonstrated by our networking-related use case, the runtime successfully executes WebAssembly modules inside the kernel and supports host interaction via kernel APIs. However, several limitations remain. First and foremost, our implementation lacks a module verification component as prescribed by the standard. As a result, syntactically valid but semantically invalid modules might be executed, which compromises security as long as such a component is missing. This issue also explains most of the missing compliance with the WebAssembly 1.0 specification, which is caused by validation tests that cannot be handled by our implementation. Additionally, our host function interface still needs to be extended to support a more exhaustive set of kernel capabilities.

When compared to eBPF, WebAssembly presents a clear trade-off: eBPF is highly integrated into the Linux kernel, efficient, and supported by a mature ecosystem. However, it is constrained in expressiveness and lacks strong runtime isolation. In contrast, WebAssembly offers greater flexibility, language support, and inherent sandboxing, but our runtime currently lacks deep kernel integration, comprehensive APIs, and full specification compliance. Compared to kernel modules, WebAssembly significantly improves portability (and security once the verification component is implemented) albeit at the cost of additional runtime overhead and architectural complexity.

Overall, WebAssembly has the potential to evolve into a unifying abstraction for safe kernel extensibility. While it does not yet fully replace existing mechanisms, it provides a strong foundation for future research and development toward more secure, portable, and flexible kernel development.

Appendix

A. Glossary

Glossary

BPF Berkeley Packet Filter. 5, 6, 34

CI Continuous Integration. 7, 21–24, 31

eBPF extended Berkeley Packet Filter. 5–8, 29, 30, 32

Netfilter a framework within the Linux kernel that provides a standardized set of hooks for intercepting and manipulating network packets. 5, 15, 30

R4L Rust for Linux. 7, 9, 26–28, 32

VFS Virtual File System. 5, 7, 9, 10, 16, 26

WASI WebAssembly System Interface. 8, 30

WASM WebAssembly Stack Machine. 5–11, 16, 19, 28, 29, 32

B. Code Contributions

Category	Feature	Author(s)
Spec Implementation	Decoder	Robert, Sebastian, Tobias
	Memory handling	Sebastian
	Tables	Malte
	Branching (Decoder, Executor)	Simon, Tobias
	Instructions	Simon, Tobias, Malte, Sebastian, Bertold
	Module Instantiation	Robert, Bertold
I/O	File based interface	Robert
	Host $\rightarrow$ Guest invocation	Malte, Robert
	Host Functions (Create Tun, Read/Write FD, Log)	Tobias
	Host Functions (Netfilter callback)	Malte, Robert
	File based interface (Invocation + cooperative stop)	Tobias
Misc	port to out-of-tree module	Robert
	kernel-/user space bindings	Bertold
Tooling	VM reload setup	Robert
	Cargo-less Rustanalyzer	Robert, Bertold
	Guest Module / Demo	Malte
	Module Runner (kernel/user space)	Tobias
CI/CD	CI-Pipeline	Tobias, Sebastian
	Cargo-less Rustfmt, rustdoc	Malte
	Spec Test Harness (kernel/user space)	Tobias
	Spec Test Harness (GitLab MR test report)	Malte

References

[1] F. Abdelmonem. "Safe Kernel Extensibility and Instrumentation With Webassembly". Master's thesis. Pittsburgh, PA, 15213: Carnegie Mellon University, Aug. 2025.

[2] J. Corbet. *The (successful) end of the kernel Rust experiment*. Dec. 2025.

[3] B. Gregg. "BPF Internals". In: USENIX Association, June 2021.

[4] Y.-Y. He et al. *WasmEdge/wasmedge-spectest*. original-date: 2025-02-24T07:51:32Z. Mar. 2026. URL: https://github.com/WasmEdge/wasmedge-spectest (visited on 2026-04-03).

[5] M. Kröning, S. Lankes, J. Klimt, and A. Monti. "From Browser to Kernel: Exploring a Lightweight Sandboxed Approach for Unikernel Extensions". en. In: *Proceedings of the 13th Workshop on Programming Languages and Operating Systems*. Seoul Republic of Korea: ACM, Oct. 2025, pages 127–135. ISBN: 979-8-4007-2225-7. DOI: 10.1145/3764860.3768334.

[6] S. McCanne and V. Jacobson. "The BSD packet filter: a new architecture for user-level packet capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, 1993, page 2.

[7] M. Ojeda et al. *Rust for Linux*. URL: https://rust-for-linux.com (visited on 2026-03-22).

[8] A. Rossberg, editor. *WebAssembly Core Specification*. Version 1.0. Dec. 5, 2019. URL: https://www.w3.org/TR/wasm-core-1/.

[9] *The (successful) end of the kernel Rust experiment*. LWN.net. 2025. URL: https://lwn.net/Articles/1049831/ (visited on 2026-03-22).

[10] E. Wen and G. Weber. "Wasmachine: Bring the Edge up to Speed with A WebAssembly OS". In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. Beijing, China: IEEE, Oct. 2020, pages 353–360. ISBN: 978-1-7281-8780-8. DOI: 10.1109/CLOUD49709.2020.00056.

[11] Y. Zheng, T. Yu, Y. Yang, and A. Quinn. *Wasm-bpf: Streamlining eBPF Deployment in Cloud Environments with WebAssembly*. Version Number: 1. 2024. DOI: 10.48550/ARXIV.2408.04856.

[12] H. Zhou. *Running WebAssembly on the Kernel*. May 2019. URL: https://blog.wasmer.io/running-webassembly-on-the-kernel-8e04761f1d8e (visited on 2026-03-27).