

Master Projekt WiSe 2025/26: WebAssembly im Linux-Kern

"Usage of WebAssembly in the Linux Kernel: Evaluation of a WebAssembly Runtime Against eBPF and Kernel Modules in the Linux Kernel"

Bertold Brödner, Malte Stellmacher, Robert Oleynik,
Sebastian Wilke, Simon Cyrani, Tobias Görgens

Operating Systems and Middleware
Prof. Andreas Polze, Clemens Tiedt, Lukas Pirl

**Design IT.
Create Knowledge.**

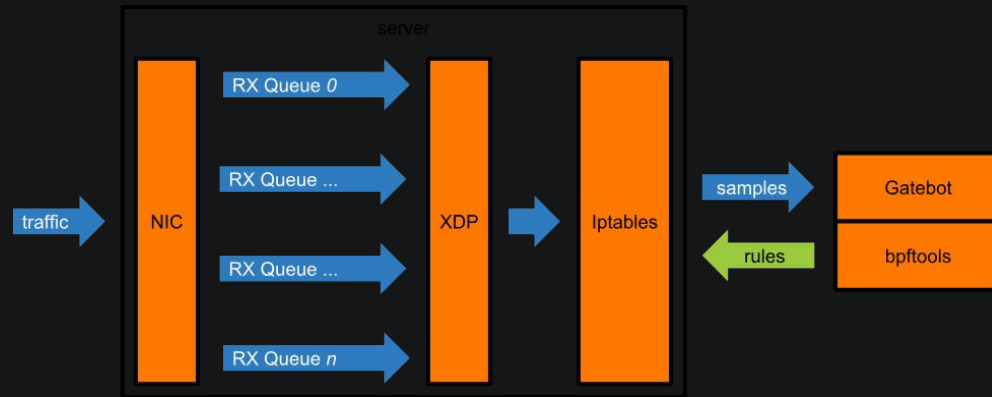
www.hpi.de



Recap: Unser Projekt + Anwendungsbeispiel



- **eBPF (extended Berkeley Packet Filter)** ist im Linux Kernel verbreitet, um dynamisch Programmcode in Hardwarenähe auszuführen, zum Beispiel:
 - XDP (eXpress Data Path): eBPF Hooks, die direkt nach Interruptverarbeitung des NICs aufgerufen werden
 - z. B. Implementierung komplexer Regeln als Programmcode zur Erkennung von DoS-Angriffen in Firewalls



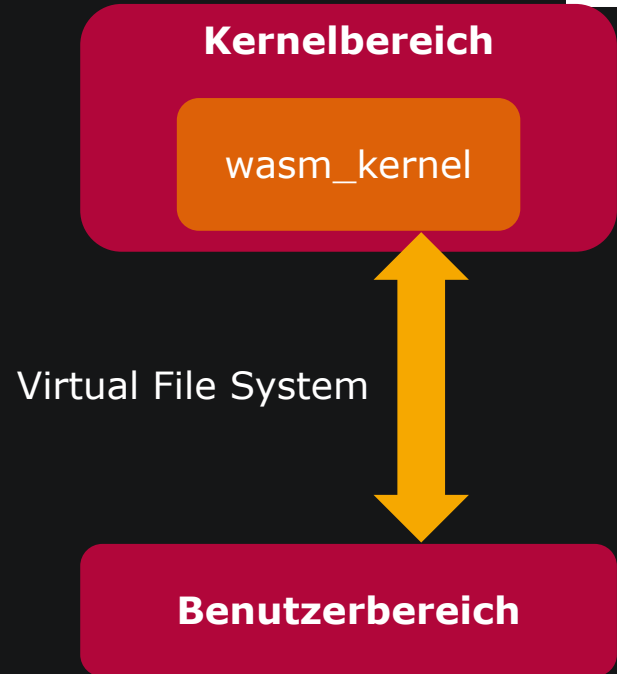
Cloudflare L4Drop Architektur [\[Quelle\]](#)

Recap: Anwendungsfall und Projekt

- **eBPF** ist basiert auf automatischer Verifizierung vor Ausführung
 - keine Isolation notwendig, aber Limitierung auf max. 1 Mio Instruktionen pro Ausführung
- **WebAssembly Stack Machine (WASM)** erlaubt isolierte, Turing-vollständige Ausführung von Programmcode
- Was, wenn wir eBPF-Anwendungsfälle mittels WebAssembly lösen könnten?

Unsere Lösung:

- WebAssembly-Laufzeitumgebung als Linux Kernelmodul
- Basierend auf Rust-for-Linux-Projekt
- Interaktion mit Kernelmodul / WebAssembly Modulen über virtuelles Dateisystem



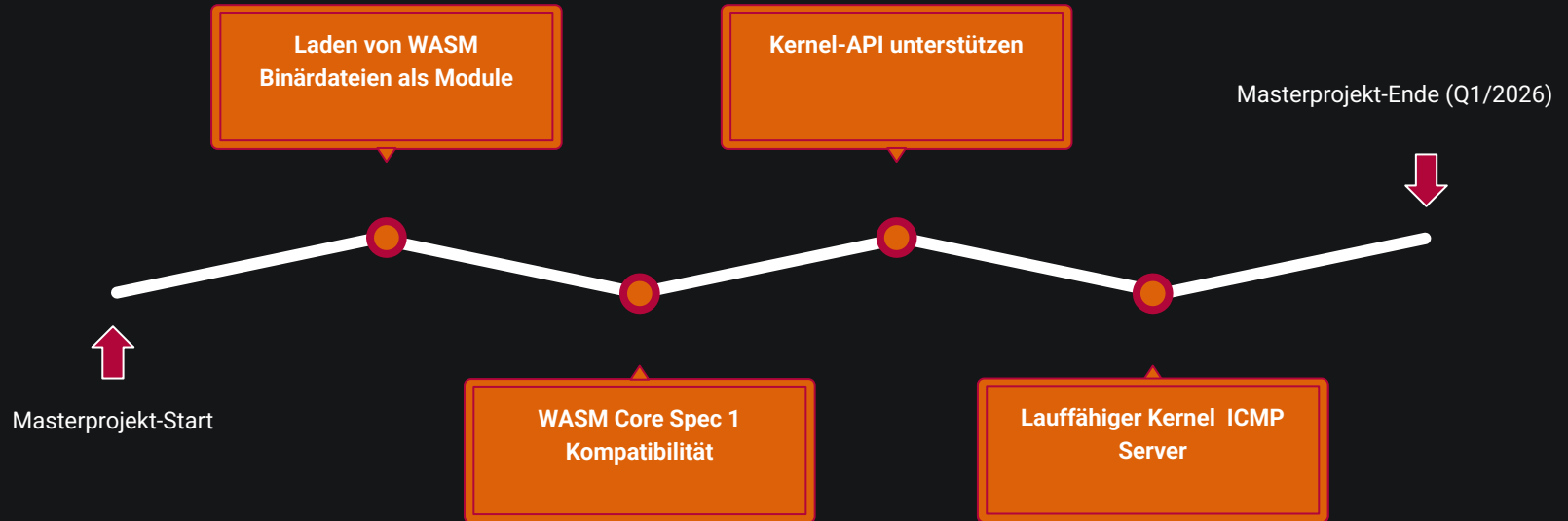
Recap: Alternativen



Kernelerweiterung: zwei Optionen

	Architekturunabhängig	Versionsunabhängig	Turing-vollständig	Entwicklungssprachen	Tests	Isolierung
Kernel Modul			✓	C, Rust	VM, KUnit für C	keine
eBPF	✓	✓		Eingeschränktes C	VM, Unit-Tests	keine
	✓	✓	✓	Rust, C (alle mit WASM als Zielarchitektur)	VM, Unit-Tests	Sandbox

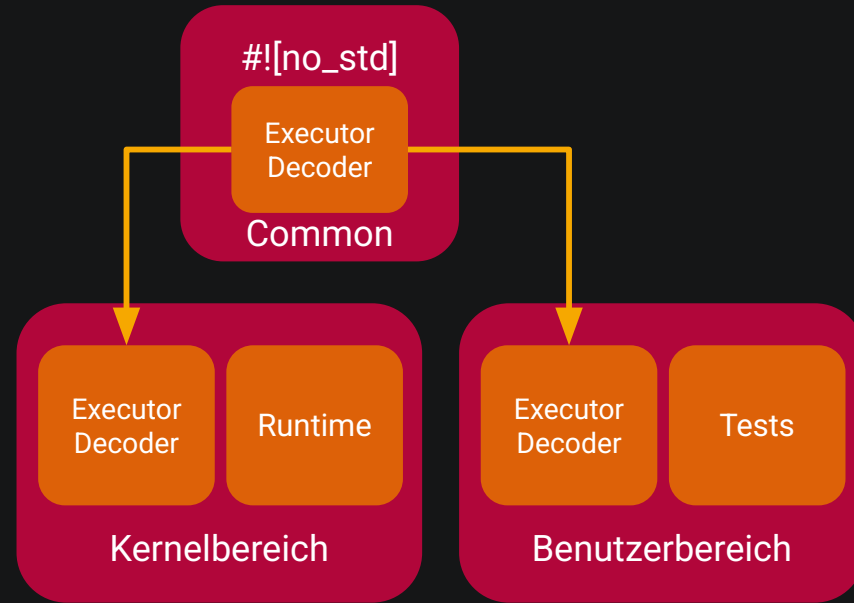
Recap: Roadmap für das Masterprojekt



Architektur

Überblick und neue Entwicklungen

- **Decoder:** Interpretiert WebAssembly Binärformat und konstruiert daraus Rust-Datenstrukturen
- **Runtime:** Schirmstruktur über Executor, verwaltet Zustand für WASM-Module
- **Executor:** führt Instruktionen aus
- **Refactoring zur Unterstützung von Kontrollstrukturen (Branches, etc.)**
 - **Decoder** zergliedert Funktionen in Blöcke
 - **Executor** arbeitet Kontrollstruktur-Stack ohne explizites Wissen über interne Implementierung ab
 - **Runtime** verwaltet inneren Zustand unter Berücksichtigung der Tragweite von verschiedenen Blöcken (lokale Variablen, Branches, etc.)



- WebAssembly-Binärdateien bestehen aus **Sections**
- **Sections** sind meist nach WebAssembly-Modulkomponenten getrennt
 - ähnlich zu ELF Sections
- Erlaubt theoretisch paralleles Kompilieren oder Stream-Kompilierung
 - *wasm-kernel* unterstützt bisher nur sequentielle Decodierung
- **Textformat ist grundlegend anders strukturiert!**
 - Nicht unterstützt von *wasm-kernel*

Custom

Type

Import

Function

Table

Memory

Global

Export

Start

Element

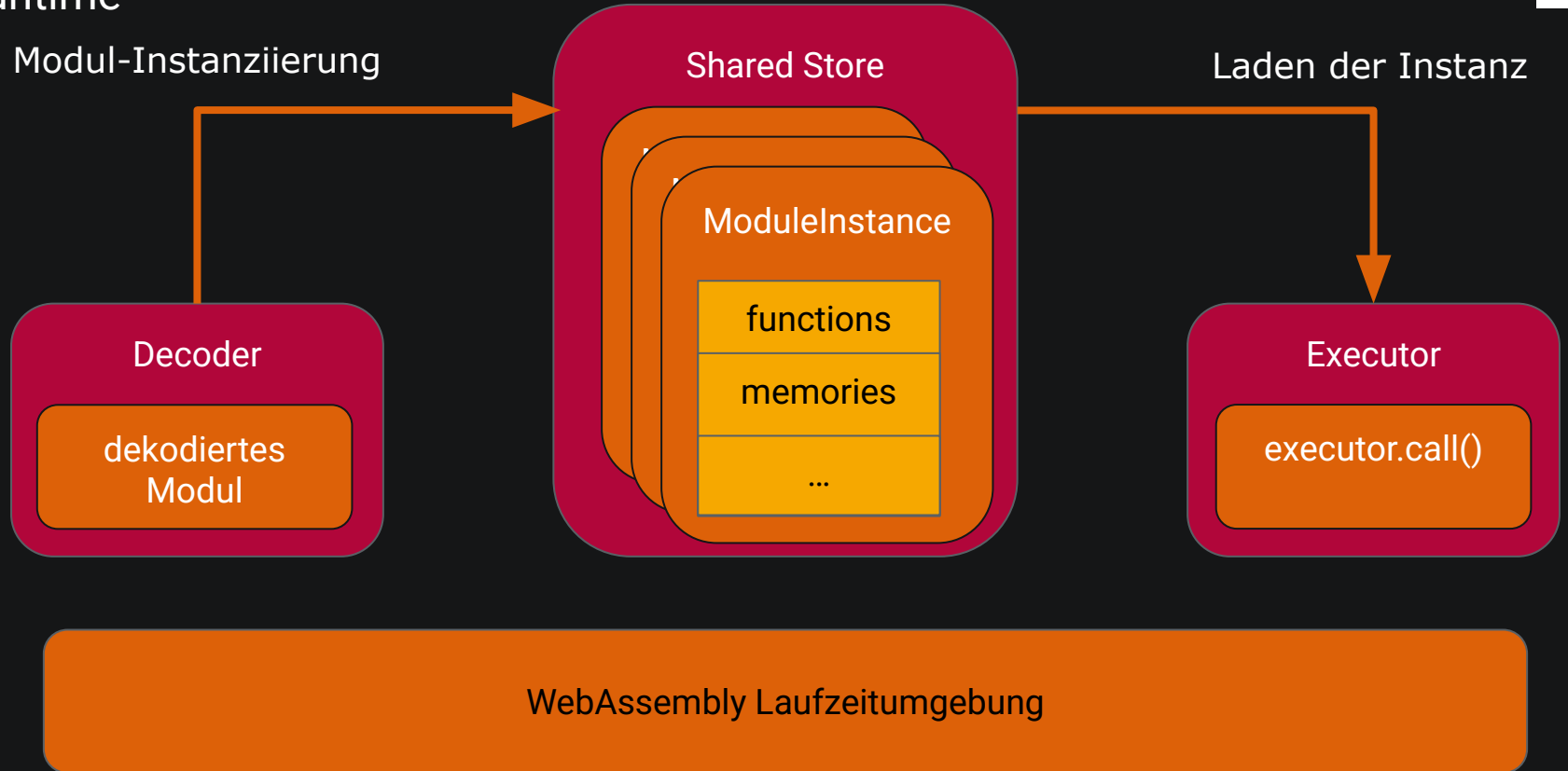
Code

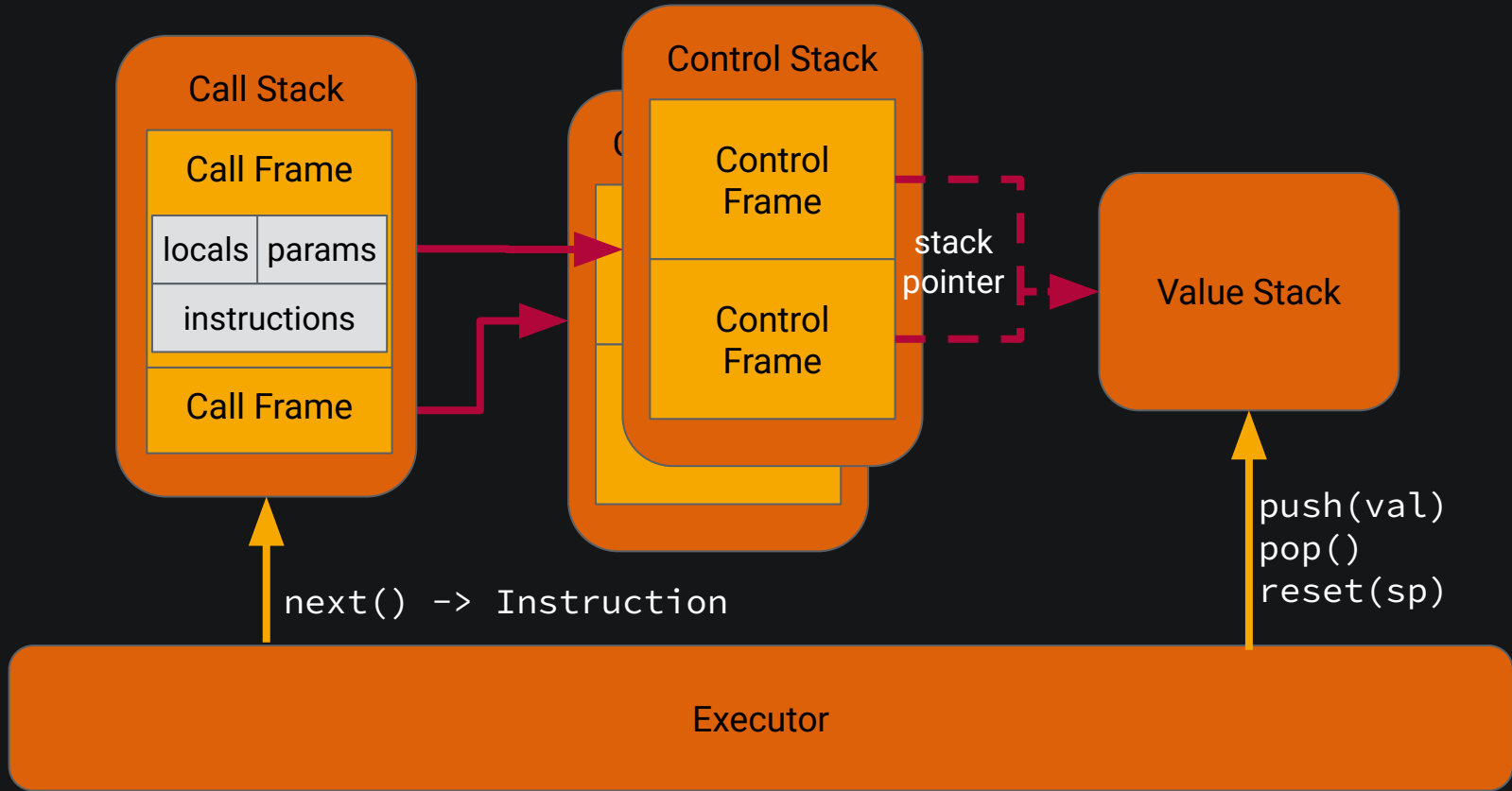
Data

Architektur Runtime

Modul-Instanziierung

Laden der Instanz





1. **Entwurf:** Minimales eigenes Testing-Framework in Kernelbereich
2. **Entwurf:** Rust Unit-Tests über das *common*-Crate
 - dadurch mind. im Benutzerbereich problemlos nutzbar
3. **Entwurf:** *spectest-harness* (mehr dazu gleich)

```
pub fn i32_add() -> Test {
    let mut runtime = build_binary_op (Instruction::I32Add).unwrap ();
    let mut tests = KVec::new ();
    let mut success = true;

    let first = (Value::I32 (2), Value::I32 (3));
    let second = (Value::I32 (10), Value::I32 (5));
    let third = (Value::I32 (1), Value::I32 (1));

    tests.push (first, GFP_KERNEL).unwrap ();
    tests.push (second, GFP_KERNEL).unwrap ();
    tests.push (third, GFP_KERNEL).unwrap ();

    for (left, right) in tests.iter () {
        let mut args = KVec::new ();
        args.push (*left, GFP_KERNEL).unwrap ();
        args.push (*right, GFP_KERNEL).unwrap ();

        let result = runtime.call (MemString::from_str ("binary_op").unwrap (), args).unwrap ();
        success = success && result.is_some () && result.unwrap () == (*left + *right).unwrap ();
    }

    Test {
        name: "i32_add",
        success,
        message: None,
    }
}
```

```
#[test]
fn test_add () {
    let instance = testfn!((num_params=2, num_locals=0, returns=true) [
        LocalGet (0),
        LocalGet (1),
        I32Add
    ]);
    test_instance!(instance(42, 13) => Some(Value::I32(55)));
    test_instance!(instance(42, 0) => Some(Value::I32(42)));
    test_instance!(instance(0, 0) => Some(Value::I32(0)));
}
```

1. Entwurf

2. Entwurf

- WebAssembly Core 1.0 Standard enthält **172 Instruktionen**
- WebAssembly Core 3.0 Standard über **400 Instruktionen**
 - SIMD, Garbage Collection, Exception Handling, ...
 -
- **Tests für alle Instruktionen selbst schreiben?**
- WebAssembly bietet offizielle WebAssembly Test-Moduldateien!

spec

This repository holds the sources for the WebAssembly specification, a reference implementation, and the official test suite.

A formatted version of the spec is available here: webassembly.github.io/spec,

- WebAssembly Testdateien werden in unser **spectest-harness** Projekt geladen
- Rust-Projekt; importiert unsere WebAssembly Laufzeitumgebung (*Decoder, Runtime, Executor*)
- Berechnet Spezifikationskonformität aus
 - **S_{total}**: Summe an Gesamt-Tests ohne Gleitkommaoperationen
 - **S_{success}**: und Summe an bestandenen Tests

$$\text{Compliance} = \frac{\mathbf{S_{success}}}{\mathbf{S_{total}}} = \mathbf{71,20\%}$$

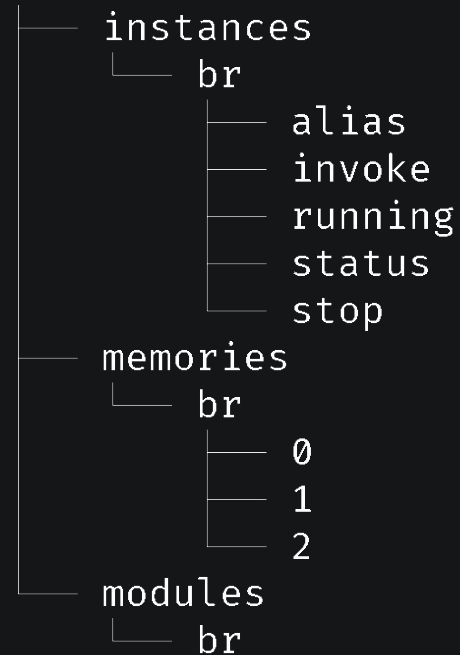
```
memory_size: 36 | 2 | 0
memory_trap: 123 | 0 | 48
names: 481 | 1 | 0
nop: 83 | 4 | 0
return: 53 | 20 | 10
select: 94 | 16 | 0
skip-stack-guard-page: 10 | 0 | 0
stack: 3 | 0 | 0
start: 1 | 10 | 0
store: 16 | 51 | 0
switch: 26 | 1 | 0
table: 3 | 0 | 0
token: 2 | 0 | 0
traps: 22 | 0 | 10
type: 2 | 2 | 0
unreachable: 58 | 0 | 5
unreached-invalid: 0 | 110 | 0
unwind: 29 | 0 | 20
utf8-custom-section-id: 0 | 176 | 0
utf8-import-field: 0 | 176 | 0
utf8-import-module: 0 | 176 | 0
utf8-invalid-encoding: 176 | 0 | 0
SPEC-COMPLIANCE: 71.20%
JUnit report written to target/junit.xml
Overall summary: 3982 passed, 1611 failed, 1121 policy-skipped
```

Tooling

VFS



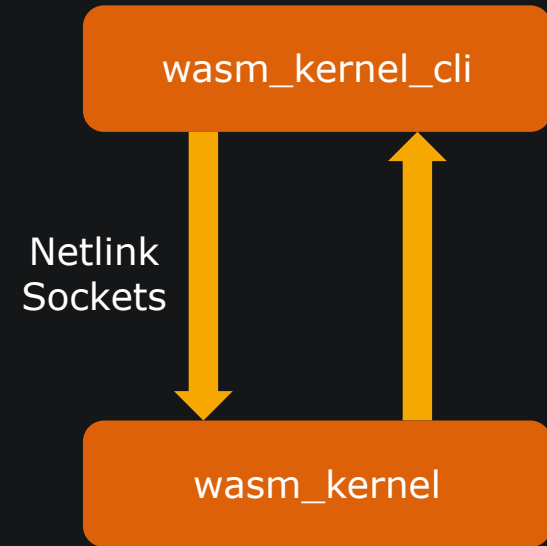
- **Linux Virtual File System (VFS) API** zur Übertragung und Kommunikation des WebAssembly-Moduls
- Vorgegebene Ordner:
 - **Modules** - enthält WASM-Binärdateien
 - **Memories** - enthält WASM-Memory-Objekte zum Auslesen / Einfügen von Daten
 - **Instances** - enthält Meta-Dateien zu Instanzen
 - → Kommunikation zwischen Benutzerbereich-Prozess und Kernelmodul
- *Runtime* wird von VFS gestartet
- *Decoder* lädt Binärdatei aus VFS



Tooling

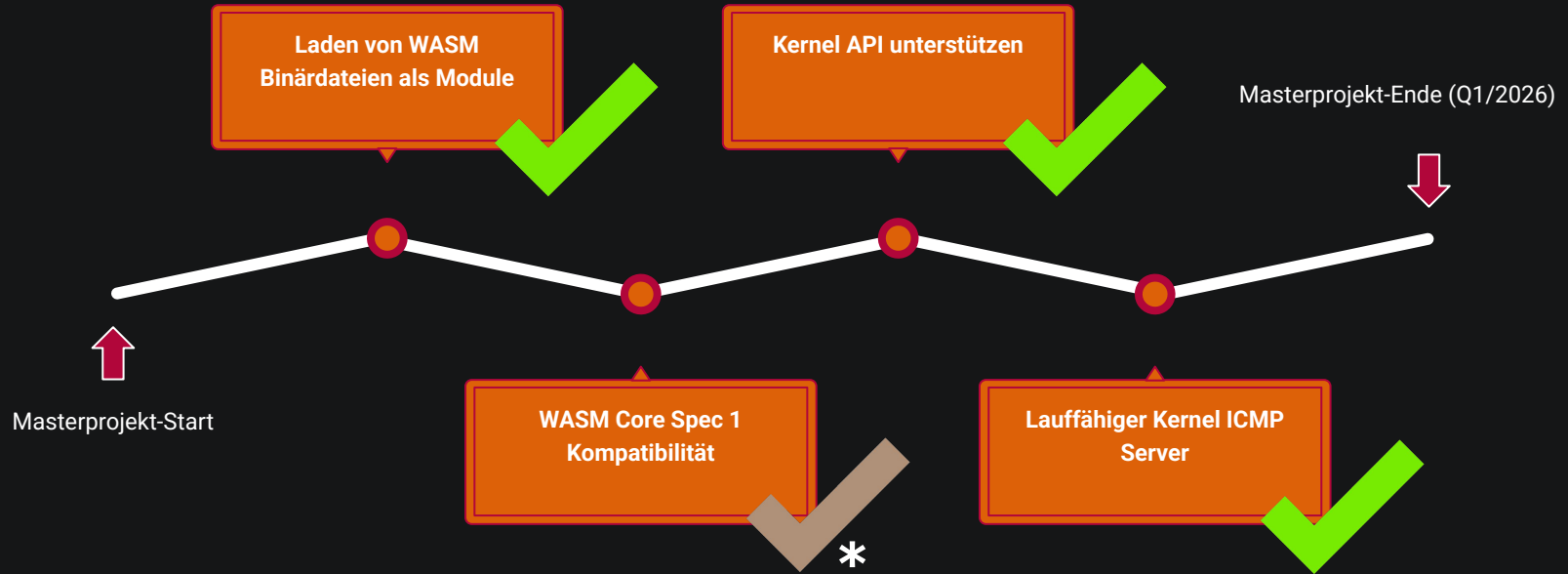
Alternative für VFS

- Zuvor: **Linux Connector API** zur Kommunikation zwischen Kernel und Benutzerbereich-Prozess
- Basiert auf Netlink Sockets
- **Problem:** Eigenes Protokoll für Datenübertragung aus Memory-Objekten (die de-facto Dateien sind) notwendig; VFS fühlt sich “ergonomischer” an



- ICMP-Server in Benutzerbereich
- WebAssembly Modul im Kernelbereich fängt eingehende ICMP-Pakete vorher ab
- Basiert auf in Rust geschriebenen WebAssembly-Modul mit Zielarchitektur wasm32v1-none
 - WASM Spec 1, keine Erweiterungen

Recap: Roadmap für das Masterprojekt



* Dazu gleich mehr

Offene Probleme / Zukünftige Arbeit

WebAssembly Spec Compliance erfüllen

Spec Compliance aktuell nicht bei 100% trotz

Implementierung aller Instruktionen, da:

1. **WebAssembly-Module können syntaktisch korrekt, aber semantisch falsch sein**
 - Dafür: Validierung innerhalb des Decoders notwendig
2. **Gleitkommazahlen werden nicht unterstützt**
 - Grundsätzlich im Kernelbereich nicht erwünscht

```
(assert_invalid
  (module
    (memory 1)
    (func
      (result i32)
      (memory.grow (f32.const 0))
    )
  ) "type mismatch"
```



Erwartet i32

Offene Probleme / Zukünftige Arbeit

Bessere Integration mit Kernel APIs

WebAssembly bietet nur wenige standardisierte primitive Datentypen:

- Int32, Int64, Float32, Float64

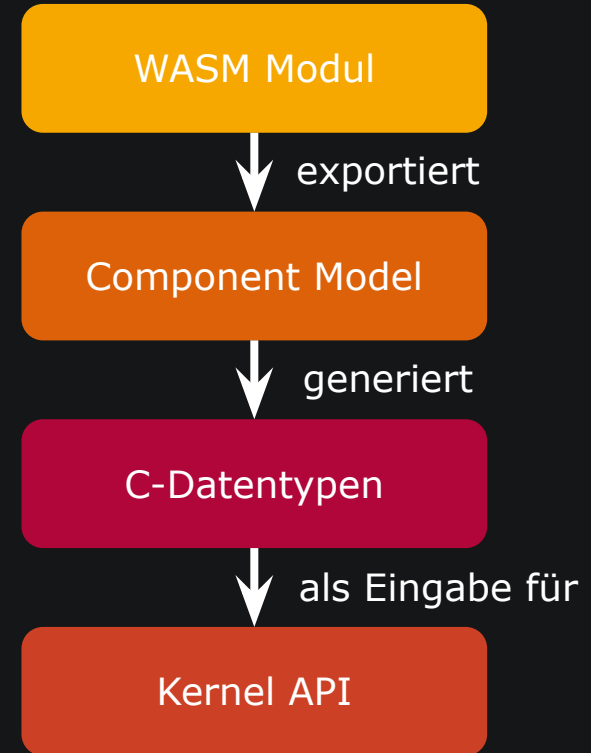
Wie werden Strings, Structs, Listen, etc. abgebildet?

- Aufgabe des Transpilers (z. B. Rust / C)

Inkonsistent für Interaktion mit Host-Funktionen! (hier: Kernel-APIs)

Besser: **WebAssembly Component Model**

- standardisierte Datenstrukturen wie:
 - Booleans, Chars, Strings, Lists, Records (Structs), Tuples, ...



Offene Probleme / Zukünftige Arbeit

Encoded Execution



WebAssembly-Laufzeitumgebung, aber verlässlich(er):

- zur Vermeidung von Rechenfehlern werden z. B. **Instruktionen mehrfach ausgeführt und anschließend verglichen**
- Gesamte “Wertschöpfungskette” der Laufzeitumgebung liegt in unseren Händen (da keine Drittanbieter-Komponenten genutzt)
 - macht vielfältige Ansätze möglich, z. B. auch verteilte Ausführung eines WebAssembly-Moduls mit Votingmechanismen für korrekte Ergebnisse
- **Einsatz in verlässlichen eingebetteten Systemen, z. B. Bahnumfeld**

```
i32.add (i32.const 1) (i32.const 2)
```

Exec 1: $1 + 1 = 2$

Exec 2: $1 * 7 + 1 * 7 = 30 / 7$ 

Exec 3: $1 * 11 + 1 * 11 = 22 / 11 = 2$

Return 2

Safe Kernel Extensibility and Instrumentation With Webassembly

Faisal Abdelmonem

CMU-CS-25-123
August 2025

Carnegie Mellon University
Computer Science Department
School of Computer Science
Pittsburgh, PA, 15213

THESIS COMMITTEE
Anthony Rowe (Chair)
Benjamin Titzer

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Copyright © 2025 Faisal Abdelmonem

Safe Kernel Extensibility
and Instrumentation With
Webassembly

The slide is titled "Running WebAssembly on the Kernel" and is part of a presentation by Yuhseng Zheng. It features a diagram showing the relationship between Applications, Kernel, CPU, Memory, and Devices. The Applications layer is at the top, followed by the Kernel layer, which contains a blue icon of a kernel. Below the Kernel are three boxes representing CPU, Memory, and Devices. The slide also includes an abstract and an introduction section.

Running WebAssembly on the Kernel

This is the story of our journey running Wasmer on the Linux kernel

Yuhseng Zheng
eunomia-bpf Community
China
yuzheng156@gmail.com

Yiwei Yang
UC Santa Cruz
China
yyang@soe.ucsc.edu

Tong Yu
eunomia-bpf Community
China
yutong@ustc.edu.cn

Andrew Quinn
UC Santa Cruz
USA
aquinn@ucsc.edu

Applications

Kernel

CPU **Memory** **Devices**

We have been obsessed with making the `Wasmer` WebAssembly runtime faster: first by minimizing the compilation time by using caching and then by adding different compiler flags into the runtime.

As time progressed, we started asking ourselves... What is the fundamental cause of VM based programs being slower than native ones? Is there any way can we take it for some specific use-case?

In this article we will overview what we did to make `Wasmer` (optionally) run on the kernel... achieving over 10% speedup on a WebAssembly toy echo server over native code!

Background

"The Second OS"

Many languages and runtimes, including WebAssembly (WASM) implementations and Java/Byte (HotSpot and browsers), have been trying to build another sandboxed "OS" on top of the real operating system. The

Wasmer kernel-wasm

WASM-BPF: Streamlining eBPF Deployment in Cloud Environments with WebAssembly

Yuhseng Zheng
eunomia-bpf Community
China
yuzheng156@gmail.com

Yiwei Yang
UC Santa Cruz
China
yyang@soe.ucsc.edu

Tong Yu
eunomia-bpf Community
China
yutong@ustc.edu.cn

Andrew Quinn
UC Santa Cruz
USA
aquinn@ucsc.edu

ABSTRACT

The extended Berkeley Packet Filter (eBPF) is extensively utilized for observability and performance analysis in cloud-native environments. However, deploying eBPF programs across a heterogeneous cloud environment presents challenges, including compatibility issues across different kernel versions, operating systems, runtimes, and architectures. Traditional deployment methods, such as standalone containers or tightly integrated core applications, are cumbersome and inefficient, particularly when dynamic plugin management is required. To address these challenges, we introduce Universal BPF (Wasm-BPF), a lightweight runtime in WebAssembly (Wasm) and the WebAssembly System Interface (WASI). Leveraging Wasm's platform independence and WASI's standardized system interface, with enhanced relocation for different architectures, Wasm-BPF ensures cross-platform compatibility for eBPF programs. It simplifies deployment by integrating with container toolchains, allowing eBPF programs to be packaged as Wasm modules that can be easily managed within cloud environments. Additionally, Wasm-BPF supports dynamic plugin management in WebAssembly. Our implementation and evaluation demonstrate that Wasm-BPF introduces minimal overhead compared to native eBPF implementations while simplifying the deployment process.

1 INTRODUCTION

The extended Berkeley Packet Filter (eBPF) has emerged as a powerful technology for observability and performance in cloud-native environments [1, 26, 27, 31–35]. By enabling the execution of custom logic within the kernel, eBPF provides a versatile mechanism for monitoring and modifying system behavior without requiring kernel modifications. This capability is particularly valuable in containerized and cloud ecosystems where dynamism and scalability are critical.

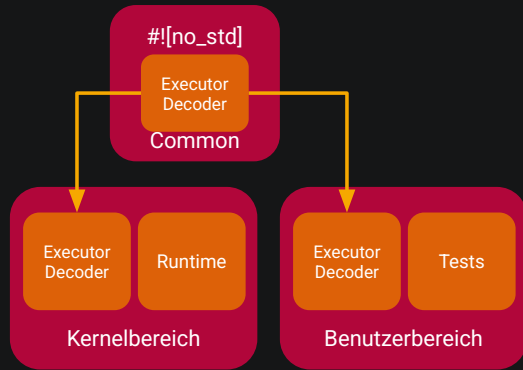
However, deploying eBPF programs heterogeneously at scale presents several significant challenges. eBPF runtimes extend beyond the Linux kernel, encompassing environments such as Windows [7] and FreeBSD [6], as well as various eBPF runtimes like bpf4j [1], bpf4c [24], and bpf4go [28]. Additionally, eBPF must be compatible across multiple operating systems, each with unique kernel data structures, eBPF features, and capabilities, resulting in inconsistent application behavior. Although Compile Once - Run Everywhere (CORE) support can mitigate some issues, it also introduces further complexity to the deployment process.

Current methods for deploying eBPF programs in containers are often inefficient and cumbersome. Standalone containers, commonly used for eBPF deployment, do not optimize for the typically small and monospaced container image of eBPF programs, leading to resource inefficiencies and management difficulties. Large-scale projects like Cilium [2], Falco [12], Tetra [21], and Dropflow [27] often integrate monitoring or management tools directly within the core application or container image, which can complicate management, especially in dynamic environments requiring frequent updates. Comprehensive observability agents like Datadog necessitate dynamic plugin management to adapt to varying scenarios, a feature not adequately supported by traditional tools. Alternatively, some approaches use Remote Procedure Call (RPC) to interface between control plane applications and distributed eBPF daemons, as seen in tools like bpf4j [5], Inspector Galgr [4], and bpf4c [1]. While effective for specific eBPF tools, these methods are better suited for smaller-scale deployments and do not fully address broader compatibility and scalability issues.

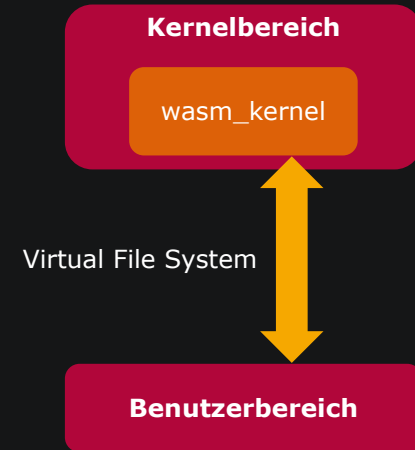
To illustrate, consider a scenario where we need to monitor network traffic across a heterogeneous cloud environment with nodes running different kernels and instruction sets. These nodes might include older versions of Linux that do not natively support eBPF, newer versions of Linux with full

arXiv:2408.04856v1 [cs.OS] 9 Aug 2024

WASM-BPF: Streamlining
eBPF Deployment in Cloud
Environments with
WebAssembly



```
memory_size: 36 | 2 | 0
memory_trap: 123 | 0 | 48
names: 481 | 1 | 0
nop: 83 | 4 | 0
return: 53 | 28 | 10
select: 94 | 16 | 0
skip-stack-guard-page: 10 | 0 | 0
stack: 3 | 0 | 0
start: 1 | 10 | 0
store: 14 | 51 | 0
switch: 26 | 1 | 0
table: 3 | 0 | 0
token: 2 | 0 | 0
traps: 22 | 0 | 10
type: 2 | 2 | 0
unreachable: 58 | 0 | 5
unreached-invalid: 0 | 110 | 0
unwind: 29 | 0 | 28
utf8-custom-section-id: 0 | 176 | 0
utf8-import-field: 0 | 176 | 0
utf8-import-module: 0 | 176 | 0
utf8-invalid-encoding: 176 | 0 | 0
SPEC-COMPLIANCE: 71.20%
JUnit report written to target/junit.xml
Overall summary: 3982 passed, 1611 failed, 1121 policy-skipped
```



- Projektziele größtenteils erreicht
- Bald: Paper für die Projektidee und Architekturentscheidungen
- **Danke an alle Beteiligten! <3**