HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

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"

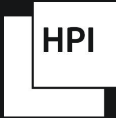
Bertold Brödner, Malte Stellmacher, Robert Oleynik, Sebastian Wilke, Simon Cyrani, Tobias Görgens

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

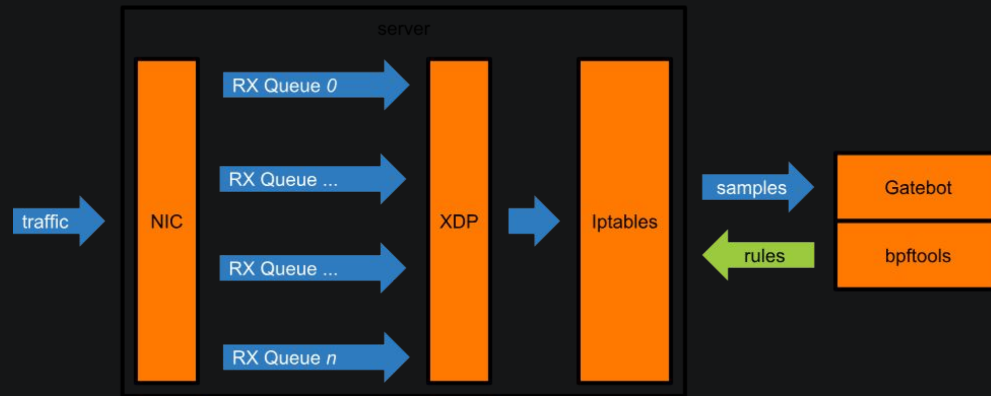
Design IT.
Create Knowledge.

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



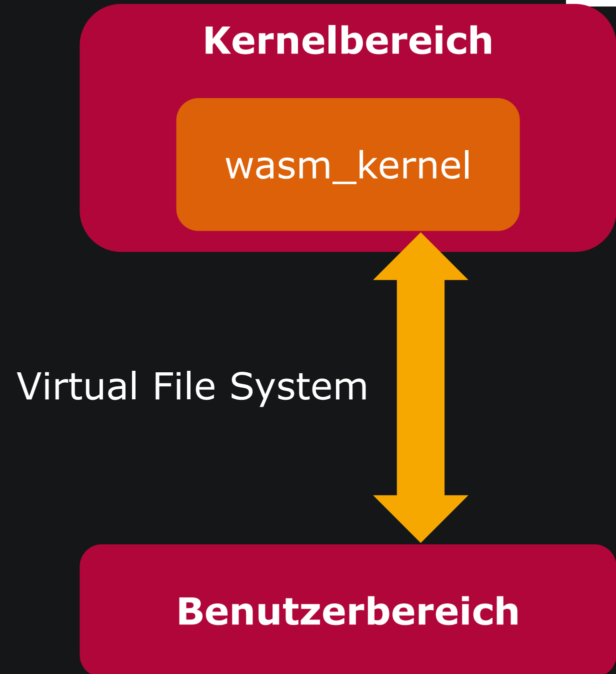
Cloudflare L4Drop Architektur [Quelle]

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

	Architekturun abhängig	Versions unabhän gig	Turing-vollständig	Entwicklungssprachen	Tests	Isolierung
Kernel Modul			✓	C, Rust	VM, KUnit für C	keine
eBPF	✓	✓		Eingeschränktes C	VM, Unit-Tests	keine
WA	✓	✓	✓	Rust, C (**alle** mit *WASM* als Zielarchitektur)	VM, Unit-Tests	**Sandbox**

Recap: Roadmap für das Masterprojekt

Laden von WASM Binärdateien als Module

Kernel-API unterstützen

Masterprojekt-Ende (Q1/2026)

Masterprojekt-Start

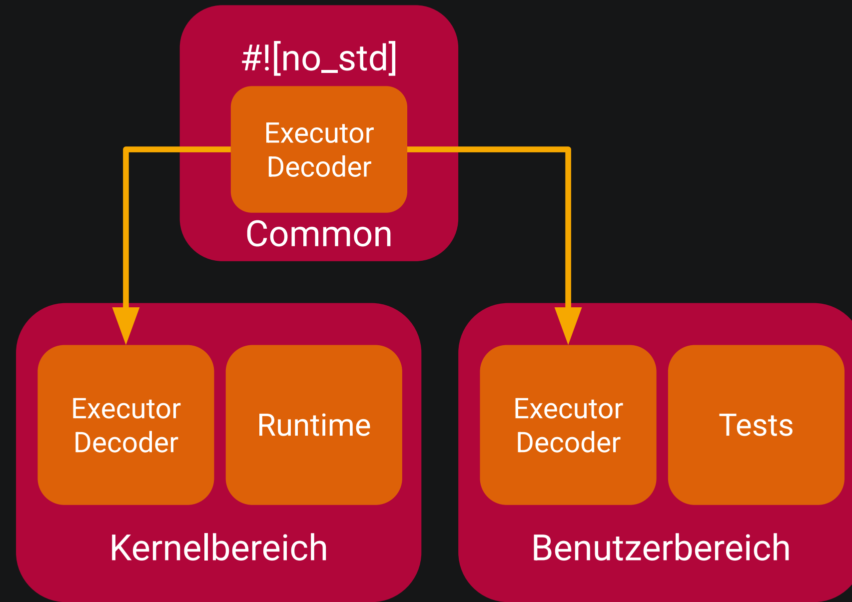
WASM Core Spec 1 Kompatibilität

Lauffähiger Kernel ICMP Server

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.)

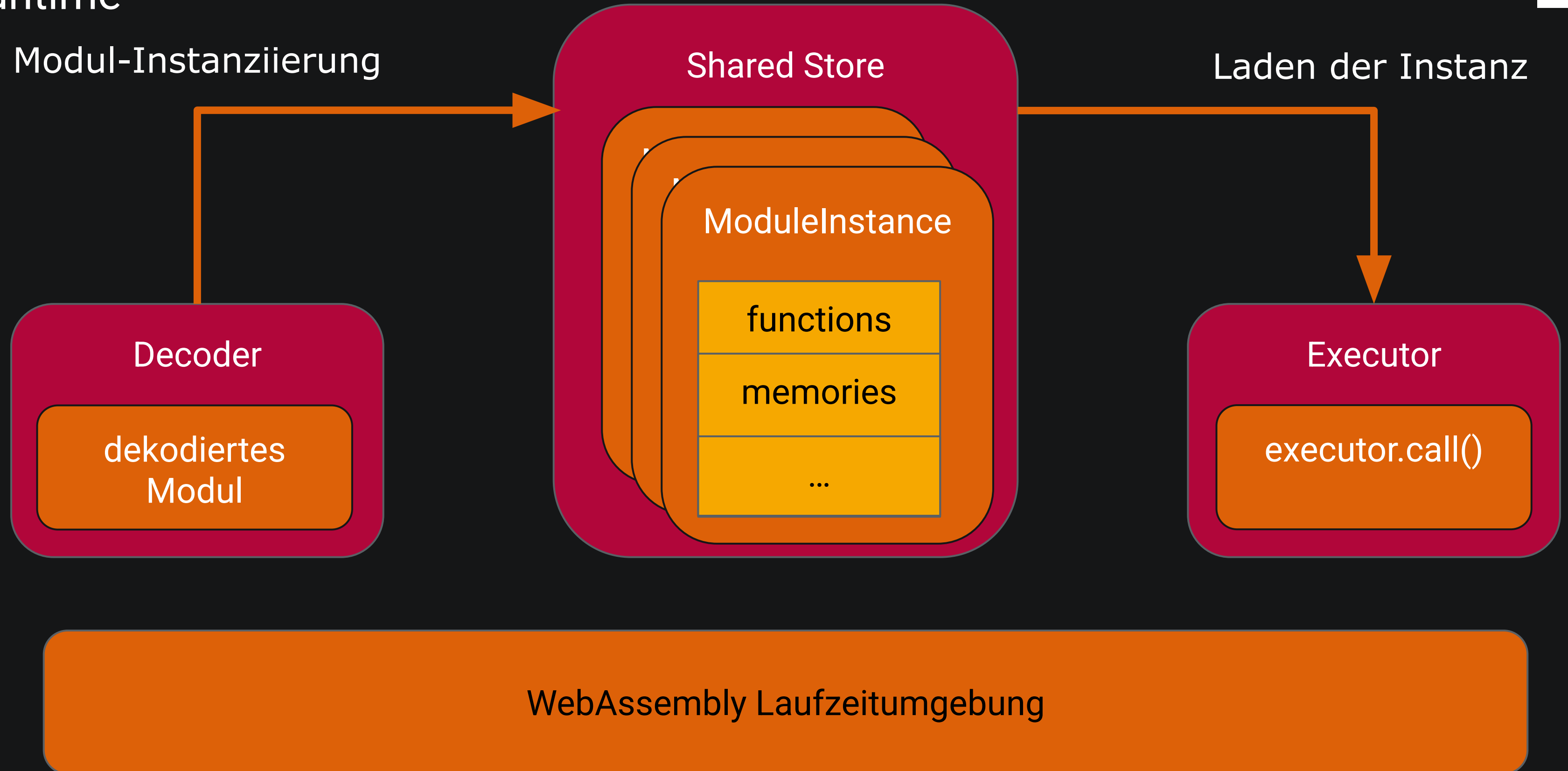


Architektur
Decoder

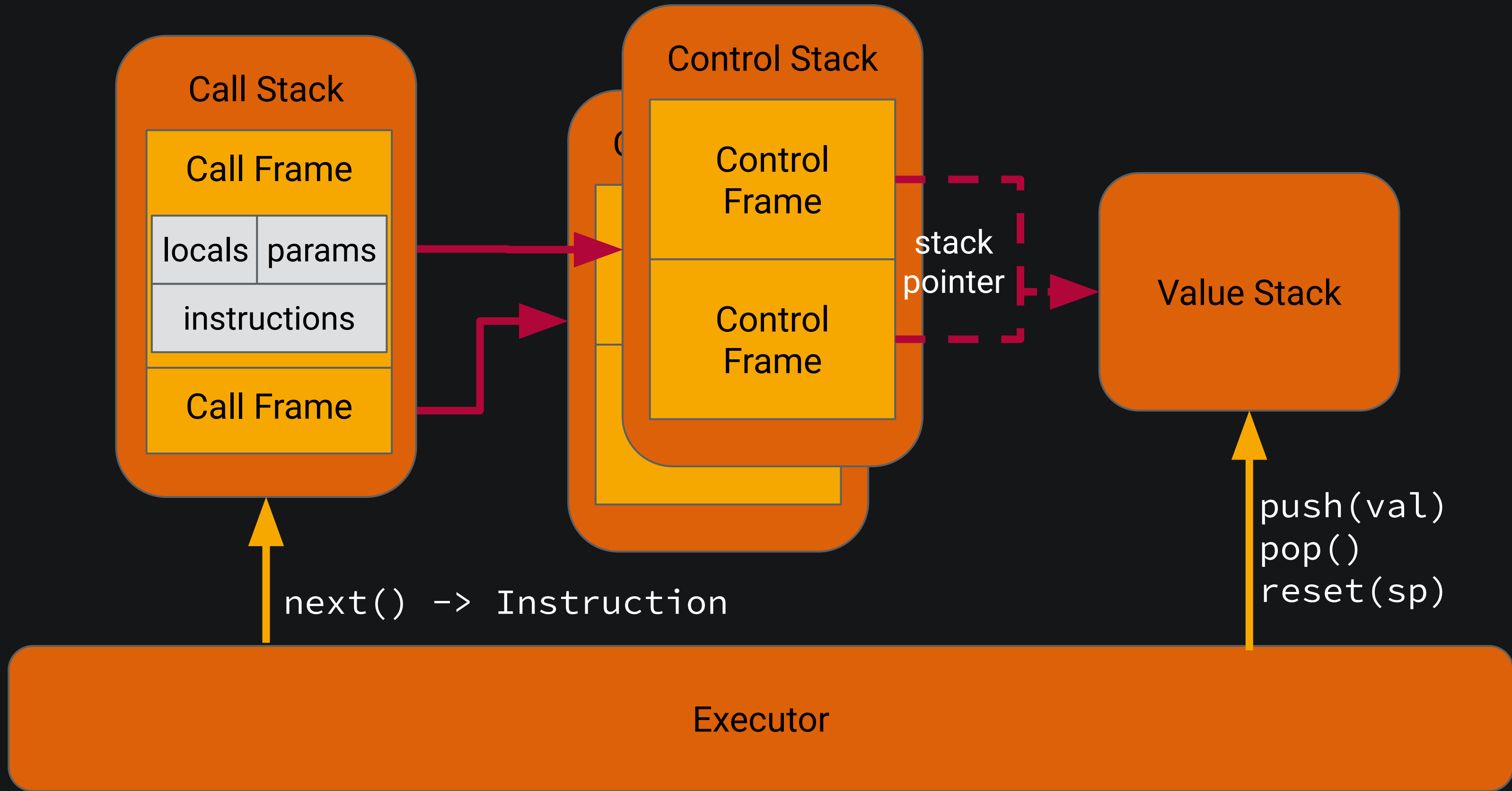
- WebAssembly-Binärdateien bestehen aus **Sections**
- **Sections** sind meist nach WebAssembly-Modulkomponenten getrennt
  - ähnlich zu ELF Sections
- Erlaubt theoretisch paralleles Kompilieren oder Stream-Kompilierung
  - *wasm-kernel* unterstützt bisher nur sequentielle Decodierung
- **Textformat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

Custom
Type
Import
Function
Table
Memory
Global
Export
Start
Element
Code
Data

Architektur
Runtime



Architektur
Executor



Architektur
Testing

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)

```rust
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(WasmString::from_str("binary_op").unwrap(), args).unwrap();
        success = success && result.is_some() && result.unwrap() == (*left + *right).unwrap();
    }

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

1. Entwurf

```rust
#[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)));
}
```

2. Entwurf

Tooling
Testing

- 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,

Tooling
Testing

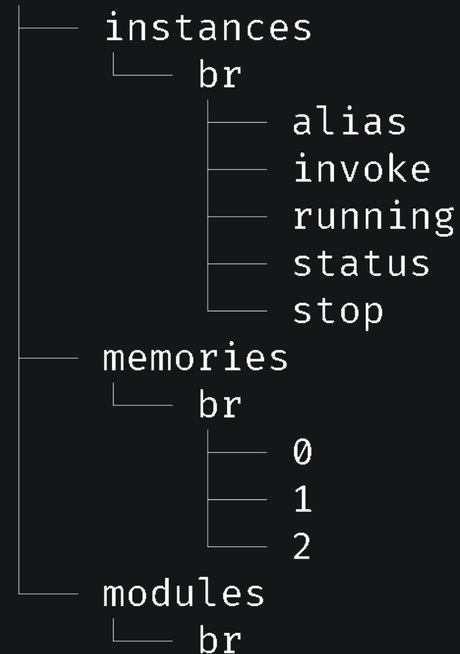
- WebAssembly Testdateien werden in unser **spectest-harness** Projekt geladen
- Rust-Projekt; importiert unsere WebAssembly Laufzeitumgebung (*Decoder, Runtime*, *Executor*)
- Berechnet Spezfikationskonformität aus
    - **S_total**: Summe an Gesamt-Tests ohne Gleitkommaoperationen
    - **S_success:** und Summe an bestandenen Tests

$$\textbf{Compliance} = \frac{\textbf{S_success}}{\textbf{S_total}} = 71{,}20\%$$

```
            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
```

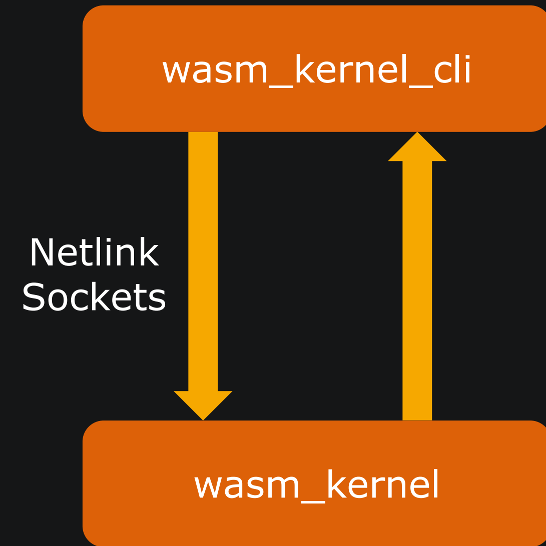
Tooling
VFS

- **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



Demo

HPI

- 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

Demo

HPI

733156

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

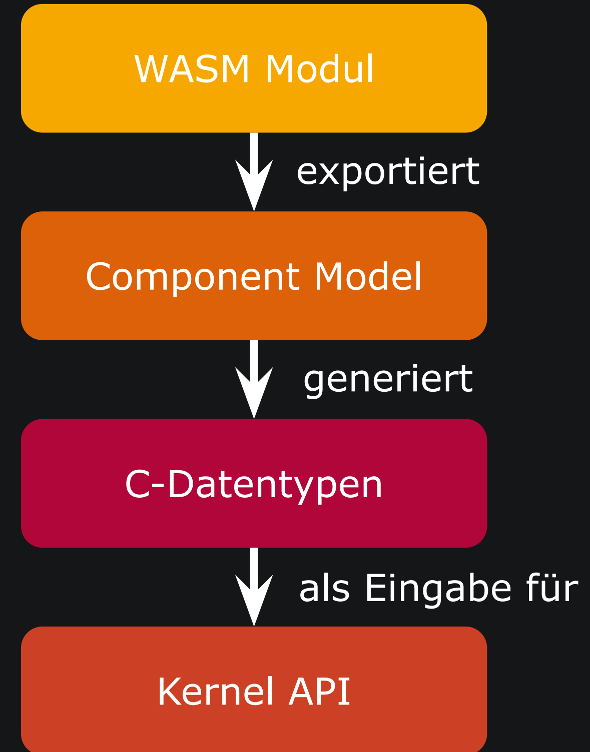
Wie werden Strings, Structs, Listen, etc. abgebildet?

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

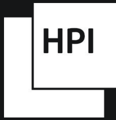
Inkonsistent für Interaktion mit Host-Funktionen! (hier: Kernel-APIs)

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

Return 2

Verwandte Arbeiten

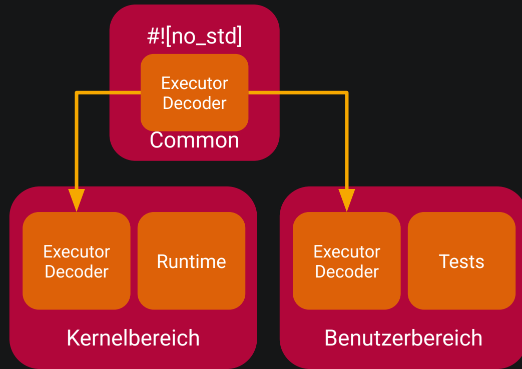
HPI

Safe Kernel Extensibility and Instrumentation With Webassembly

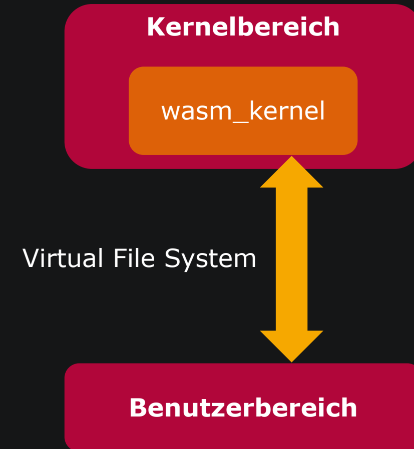
Wasmer kernel-wasm

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

Zusammenfassung und Ausblick




```
           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
```



- Projektziele größtenteils erreicht
- Bald: Paper für die Projektidee und Architekturentscheidungen

- **Danke an alle Beteiligten! <3**