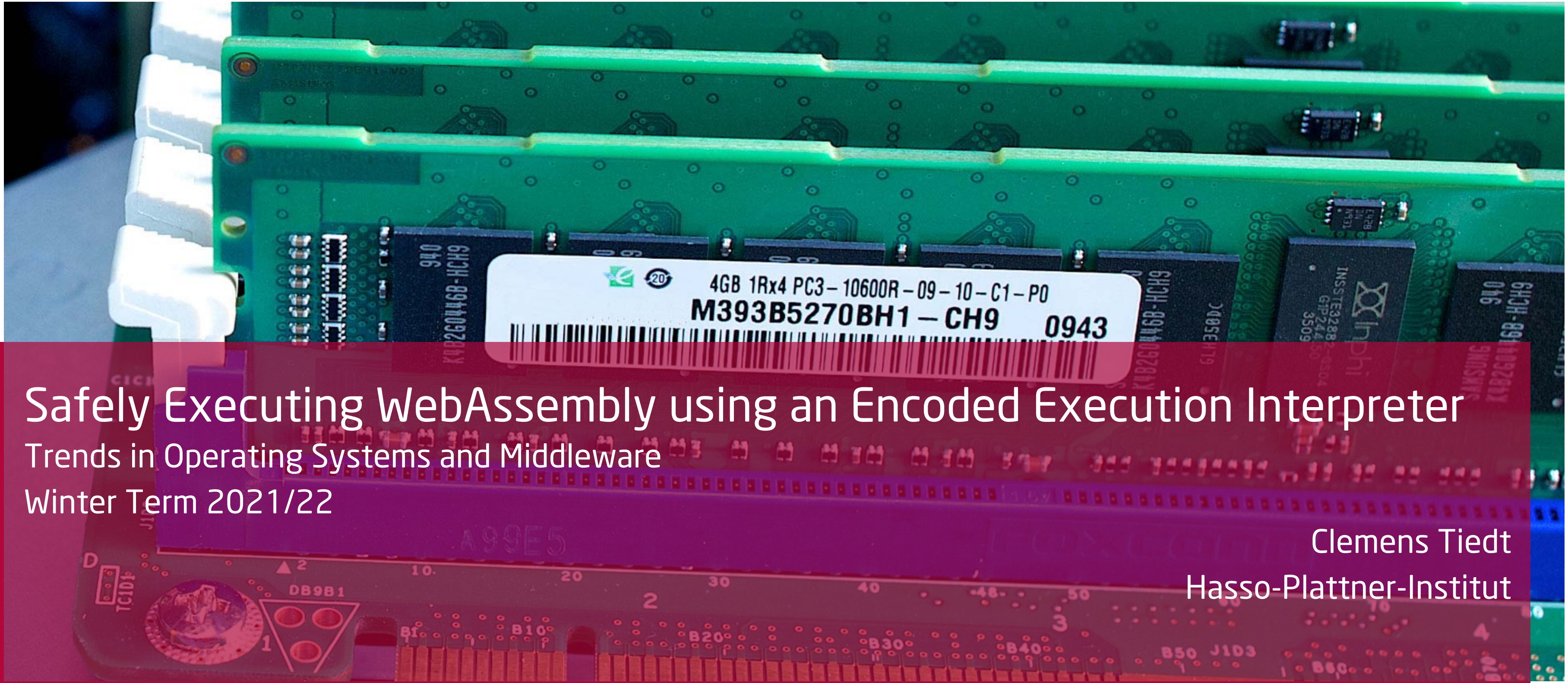


Safely Executing WebAssembly using an Encoded Execution Interpreter

Trends in Operating Systems and Middleware

Winter Term 2021/22

Clemens Tiedt

Hasso-Plattner-Institut

Motivation

- Building certified systems costs a lot of time and money
- Hardware used in certified systems is often less performant than consumer-grade hardware
- Consumer-grade hardware is inexpensive and readily available, but (potentially) unsafe
- Need to mitigate hardware errors due to e.g. environmental effects
- Common solution: Redundancy in hardware and/or software
- Alternative/complementary approach: Encoded Execution

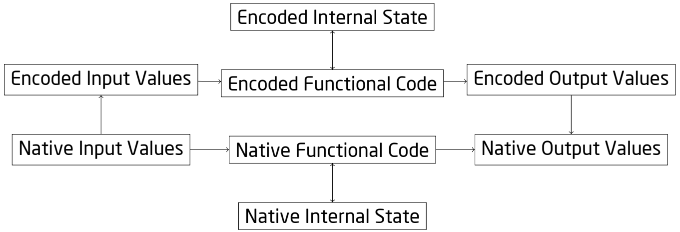
Encoded Execution (1/2)

- Also known as *Coded Processing*, i.e. using software codes to detect errors at runtime
- Related works:
  - Forin, 1989: Vital Coded Microprocessor Principles and Application for Various Transit Systems
  - Fetzer et. al., 2009: AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware
  - Süßkraut et. al., 2015: Safe Program Execution with Diversified Encoding
- Determine encoding/decoding procedure
- Encode all inputs when calling a function
- Execute function using encoded operations
- Is function return value decodeable? $\rightarrow$ no hardware errors occurred

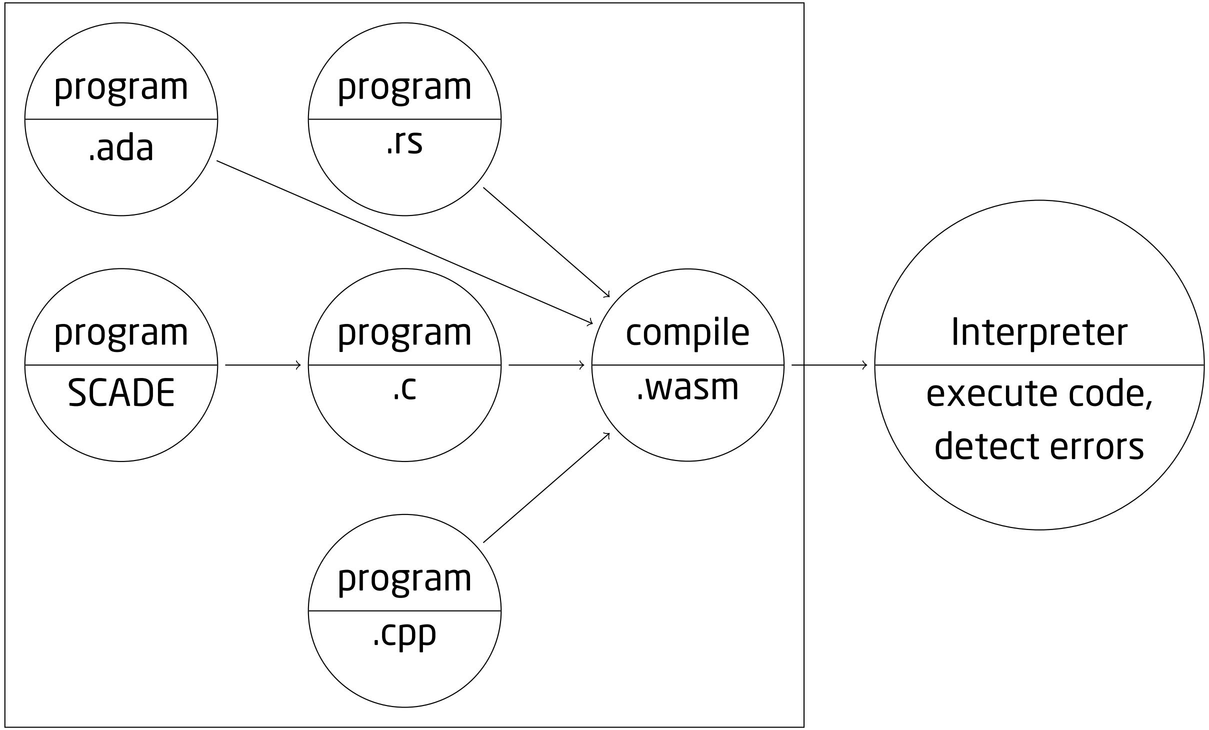
Encoded Execution (2/2)

Extension: Diversified execution

- Run multiple program instances with different encodings in parallel
- Compare instance states e.g. at every function entry/exit using checksums
- Additional safety from software-side redundancy
- Diversified code can be generated by *Diversity Framework*, in this case modified WebAssembly interpreter



Approach


program
.ada
program
.rs
program
SCADE
program
.c
compile
.wasm
program
.cpp
Interpreter
execute code,
detect errors

WebAssembly Example (1/2)

```
int add(int a, int b) {
    return a + b;
}
```

- Compile using e.g. *clang* with target *wasm32*
- Example shows a library, executable would have _start function as entry point
- Interpreter is a Rust library that can be embedded in other code

```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "add" (func $add))
 (func $add (; 0 ;) (param $0 i32) (param $1 i32) (result
  (i32.add
   (get_local $1)
   (get_local $0)
  )
 )
)
```

WebAssembly Example (2/2)

```
let source = include_bytes!("add.wasm");

let tree = parser::parse(source.as_slice()).expect("Could not parse source");

let mut runtime = DiversifiedRuntime::instantiate(
    &tree.module,
    DefaultImporter::new(),
    DefaultImporter::new(),
    A,
).expect("Failed to instantiate runtime");

let ret = runtime.invoke_encoded("add", &[Value::I32(4), Value::I32(2)])
    .expect("A trap occurred during execution")
    .unwrap();
```

The WebAssembly Interpreter

- This project is based on *wain* (**W**eb**A**ssembly **In**terpreter) by GitHub user *rhysd*
- Written in pure Rust (a systems programming language that ensures memory safety through a concept of ownership)
- No external dependencies
- No unsafe code
- Some modifications necessary before implementing Encoded Execution

Compatibility with embedded platforms

- Rust's standard library links against *libc* and cannot be used on embedded platforms
- Most components from standard library (e.g. dynamically allocating data structures) can be replaced by *core* and *alloc* crates
- Mathematical operations in standard library use system *libm*, but can be replaced with pure Rust *libm* implementation

Using external functions

- Not yet implemented in *wain*, but prerequisites exist
- We can instantiate a *Runtime* with our own implementor of the *Importer* trait that can delegate WebAssembly calls to Rust functions
- *wasm_fn!* macro to ergonomically write WASM functions
- Still not perfect: Order of arguments matters, WebAssembly exeuction model operates only on numeric types

Listing: Macro invocation and expansion

```
wasm_fn!(square, |v: i32| -> i32 { v * v });

fn square(stack: &mut Stack, _memory: &mut Memory) {
    let v = stack.pop::<i32>();
    let ret = { v * v };
    stack.push::<i32>(ret);
}
```

Encoding of WebAssembly values

```
pub trait Encodeable: Sized {
    type Output;

    fn encode(self, code: i32) -> Self::Output;
}

pub trait Decodeable: Sized {
    type Output;

    fn decode(self, code: i32) -> Result<Self::Output, DecodeError>;
}
```

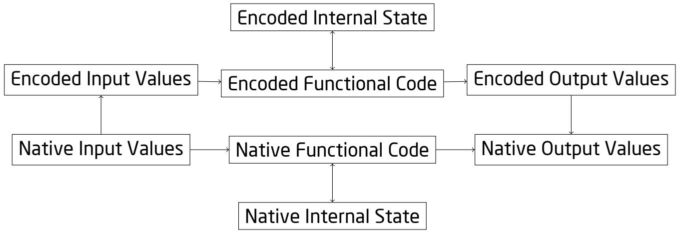
- Our interpreter uses AN Encoding
  - To encode value x with code c: $x_c = x \cdot c$
  - To decode encoded value x_c: $x = x_c / c$
  - Check validity by checking modulus
- WebAssembly values implement *Encodeable* so that a n-bit type is encoded as $2n$-bit (and encoded output types implement *Decodeable*)
- External library for 128 bit floating point numbers required

Encoded execution of WebAssembly instructions

```
fn binop_trap_encoded<T, U, F>(&mut self, op: F, code: i32) -> Result<()>
    where
        T: StackAccess + LittleEndian + Encodeable<Output = U>,
        U: StackAccess + LittleEndian + Decodeable<Output = T>,
        F: FnOnce(U, U) -> Result<U>,
{
        let c2 = self.stack.pop::<T>().encode(code);
        let c1 = self.stack.top::<T>().encode(code);
        let ret = op(c1, c2)?.decode(code)?;
        self.stack.write_top_bytes(ret);
        Ok(())
}
```

- Executes function *op* on the two top values on the stack (used e.g. to implement the i32.add instruction)
- These values are encoded before *op* is called
- If encoding is violated, a *Trap* is raised

Diversified Execution




```
pub struct DiversifiedRuntime<'module, 'source, I: Importer> {
    native_runtime: Runtime<'module, 'source, I>,
    encoded_runtime: Runtime<'module, 'source, I>,
    code: i32,
}
```

- Additional layer of safety by executing each function twice, *native* and *encoded*
- Compare if results after decoding are equal, raise *Trap* otherwise
- Could be parallelized in contexts where multithreading is available

Benchmark Results

- Benchmark application: Sum from 1 to 10 000, no optimizations
- Minimum and median execution times of diversified runtime are consistently twice as long as singular execution
- Maximum execution time fluctuates between $50\mu s$ and $150\mu s$ for both
- Ca. 150Kb footprint in a STM32 binary, similar on x86

	Singular Runtime	Diversified Runtime
Minimum	2.1	4.2
Median	2.3	4.4
Maximum	114.7	54.1

Table: 100 000 runs on AMD Ryzen 5 3600, Execution times in μs

Future Work: Using an encoded stack

```
fn binop_trap<T, U, F>(&mut self, op: F, code: i32) -> Result<()>
where
    T: StackAccess + LittleEndian + Decodeable<Output = U>,
    U: StackAccess + LittleEndian + Encodeable<Output = T>,
    F: FnOnce(U, U) -> Result<U>,
{
    let c2 = self.stack.pop_decode::<T, U>(code)?;
    let c1 = self.stack.top_decode::<T, U>(code)?;
    let ret = op(c1, c2)?.encode(code);
    self.stack.write_top_bytes(ret);
    Ok(())
}
```

- Encode entire stack, only decode for instructions that cannot work on encoded values
- Could provide more safety, but more error-prone during development (e.g. global variables are not written using *Stack::push*, but written at a specific address on the stack)

Future Work: Making safe external function calls

- In general: No assumptions about external functions possible
- However, external functions in the WebAssembly interpreter have to interact with the stack
- If an error occurs, it is only critical if it affects the stack $\rightarrow$ could be caught by encoding the stack
- Native Rust functions could additionally be encoded using compile-time tools (e.g. procedural macros)

Conclusions and Future Work

- Encoded execution of WebAssembly programs is generally viable
  - Overhead of encoding is negligible
  - Overhead of diversified execution is calculable
- Potential to only require certification for programs and allow them to run on different hardware platforms via the WebAssembly runtime (AdaCore and Ferrous Systems are working on a certified Rust compiler)
- More work is necessary to support programs that depend on external functions, e.g. operating system functionality
- Field tests/fault injection could be used to further evaluate safety and performance