Safely Executing WebAssembly using an Encoded Execution Interpreter

Clemens Tiedt

***Abstract*—The process of building certified systems is difficult and expensive. This project examines the possibility of using the concepts of encoded and diversified execution to create a safe WebAssembly interpreter. It extends *wain* to run on embedded platforms and import external functions. The interpreter encodes WebAssembly programs at the instruction level to detect hardware errors. Diversified execution is implemented as an additional safety technique through a new *DiversifiedRuntime* type which uses a similar interface to the existing *Runtime* type.**

I. Introduction

Safety-critical systems have very specific and strict requirements for their hardware and software. This leads to high costs and time necessary to build them. One issue in the development of safety systems is that they are generally not portable. Even in cases where safety software could be reused on new hardware, both software and hardware would need to be certified again. A possible approach to make safety software portable is to introduce a runtime environment or virtual machine that the safety software is developed for. This runtime could then be certified for different hardware platforms. Safety software would not interact directly with the hardware, but only with the interfaces provided by the runtime. Therefore, the safety software would only require certification in the context of the runtime.

Additionally, the hardware used in safety systems is commonly more expensive and less performant than available consumer hardware. This is the case because safety hardware needs to be more tolerant to environmental influences such as vibrations or radiation. However, these performance limitations impose constraints on the size and complexity of software that can be executed on these safety systems. Using consumer hardware would in many cases allow for more complex programs or higher performance in existing programs, but is not possible without strategies to mitigate possible hardware errors.

One possible approach to use unsafe hardware for safe systems is redundancy. This can be implemented both in hardware and software, for example by executing multiple instances of the software in parallel and comparing the results. Another approach is *Encoded Execution*. With Encoded Execution safety functions do not work with their normal input values, but instead use encoded input values. After a safety function has finished execution, it is checked if the function's return value is decodeable with the encoding scheme used for the input values. If this is not the case, a hardware issue must have occurred during execution.

It should be noted that the safety guarantees provided by the runtime are not sufficient to fulfill required standards on their own. For example, safety applications in a railway context would still need to fulfill the EN 50128 standard which includes development processes for the targeted Safety Integrity Level (SIL). While the runtime can allow these applications to run on less safe hardware, it cannot fully replace safety mechanisms in the software and safe development processes. Interactions of the software outside the systems are also not protected by the safe runtime. There is, for example, no way for the runtime to verify that data received from a network interface is correct. The runtime can only ensure that the data stays intact after it has been received and while it is being processed inside the software.

This project explores the possibility of embedding techniques for safe execution in a WebAssembly interpreter. This approach promises to enable safe execution of portable software while allowing for high performance.

II. Related Work

The concept of coded processing was introduced by Forin [1] and has been built upon by different projects since then. The AN encoding procedure used by the safe WebAssembly interpreter is taken from the AN encoding compiler [2]. This compiler works on LLVM intermediate representation to replace native operations with encoded ones. This includes, for example, arithmetic and logical operations.

```
int multiply_encoded(int x, int y, int A)
{
    x_a = A * x;
    y_a = A * y;
    int result_encoded = (x_a * y_a) / A;
    if (result_encoded % A == 0) {
        return result_encoded / A;
    } else {
        eprintf("Encoding was
  violated\n");
        exit(1);
    }
}
```

Listing 1: Example of manually encoded integer multiplication in C.

AN encoding works by multiplying all operands with a number A and replacing native operations with encoded versions. An example of this can be seen in Listing 1. The

parameters x and y are encoded by multiplying with the additional parameter A. The encoded multiplication requires dividing the result by A, as it would be $A \cdot A \cdot x \cdot y$ otherwise. Then, the encoding is checked using the modulus. If it is zero, the result is valid and the decoded result is returned. If the modulus is not zero, a hardware error (such as a bit flip) must have occurred and the program exits with an error code.

AN encoding also forms the basis of [3] which presents the concept of diversified execution to increase the safety provided by AN encoding. Diversified execution uses a diversity framework to execute safety functions twice, natively and encoded. Comparing the results of both executions helps to detect more errors, leading to better safety guarantees than just AN encoding.

III. Concept

To explain how we ensure the safe execution of WebAssembly code, an overview of WebAssembly's execution model as well as limitations in the context of safety software are provided here.

A. WebAssembly Overview

WebAssembly programs are referred to as *modules*. A module can contain

- Type definitions
- Functions
- Global Variables (commonly referred to as *globals*)
- Memories and tables

Besides these, modules can also import and export functions, globals, memories and tables.

WebAssembly functions are executed in a stack machine which means that all instructions interact with values by pushing them to the stack or popping them from the stack.

```
(func $add (param $x i32) (param $y i32)
  (result i32)
    local.get $x
    local.get $y
    i32.add
)
```

Listing 2: Addition function in WebAssembly text format

We give an explanation of this behaviour using the addition function in Listing 2. In the declaration we see two 32-bit integer parameters `$x` and `$y`. When called, the function will then put them on the top of the stack using the `local.get` instructions. Then, the `i32.add` instruction is executed. All arithmetic instructions contain the type of number they operate on (namely 32-bit or 64-bit floats or integers), since the stack is an array of bytes and data types are only interpreted by instructions. The `i32.add` instruction will pop the top 8 bytes of data from the stack and interpret it as two 32-bit integers. It then adds them together and pushes the result to the stack. Instead of an explicit return from the function, the return type is specified as `(result i32)` in the signature. Therefore the caller would know that the top 4 bytes of the stack are the return value and should be interpreted as a 32-bit integer.

B. A safe subset of WebAssembly

Within the context of safety systems, some common constructs and techniques cannot be used. For example, the imprecisions and complexity of using floating point numbers (as explored for example in [4]) may not be tolerable in certain applications. Floating point numbers also prove to be an issue for the AN encoding used by the safe interpreter. Due to the definition of the modulus operator for floating point numbers (which is given as `x - (x / y).trunc() * y` by the Rust documentation) the assumption `(x * y) % y == 0` does not hold if `x` is a floating point number. This means that floating point values cannot be encoded and decoded naïvely. While it is possible to address these issues, floating point support adds a significant amount of complexity that would not be used in safety applications. By default, floating point and other instructions not useable in safety software should therefore raise a trap when called. It would also be possible to lock their implementations behind a *cargo* feature which could be enabled to use them for systems that run in the encoded runtime, but are not safety-critical.

IV. Implementation

The safe WebAssembly interpreter is based on the *wain* project[1] by GitHub user rhysd. *wain* is a WebAssembly interpreter written in Rust with no external dependencies. To add the capability for encoded execution, we made a number of changes and additions.

A. Support for Embedded Systems

While *wain* does not use any external dependencies, it is dependent on the Rust standard library which links against a libc and can therefore only be used on devices running a full operating system. However, this may not be feasible in safety applications. However, a subset of the Rust standard library is made available through the *core* and *alloc* libraries which can be used in embedded contexts. Using the `#![no_std]` crate-level attribute, the automatic import of the standard library can be disabled and the *core* and *alloc* crates can be manually imported. In many cases, only import paths need to be changed. For example, the `std::mem::size_of` function is a re-export of `core::mem::size_of`. Functions and data types requiring dynamic memory allocation are available through the *alloc* crate. This includes for example the `Vec` data type for growable lists.

While most changes only required changes of import paths, some operations are unavailable in `#![no_std]` contexts. This includes many operations on floating point numbers such as calculating the square root or rounding. On non-embedded systems these are implemented as *libm* calls. Since this is not possible on embedded systems, these functions are not available in Rust's *core* library. However, there is an effort by the Rust language team to write a *libm* implementation in pure Rust. This implementation already contains all of the mathematical functions required for the WebAssembly

[1] https://github.com/rhysd/wain

standard, meaning that methods defined on primitve types in the standard library can simply be replaced by the respective *libm* crate functions.

B. External functions

In the WebAssembly standard, it is possible to import functions from other WebAssembly modules or the execution context. However, at the time of writing, this feature is not implemented within *wain*. Since it is required to execute more complex software that relies on external libraries, it was implemented as part of this project.

There exists already code in *wain* to later implement calling external functions, namely the `Importer` trait. Each `Runtime` has an importer that could be used to integrate external functions. Using the `Importer::call` method, external functions that are given access to the runtime's stack can be called. The only implementation of `Importer` in *wain*, `DefaultImporter`, has hard-coded implementations of the C functions `memcpy`, `abort`, `putchar`, and `getchar`.

The safe WebAssembly interpreter introduces the `EnvImporter` type which is initialized with mappings of function names to Rust function pointers and WebAssembly function signatures. At runtime, the WebAssembly instruction `Call` will then delegate to `EnvImporter` if the called function is not defined within the current module.

C. Encoded Execution of WebAssembly Instructions

For encoded execution, we use the AN encoding scheme as described by Fetzer et. al.[2]. In this encoding, all values are multiplied by a constant A. To decode, an encoded value is divided by A. If the remainder of an encoded value and A is zero, the encoded value is valid.

```
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>;
}
```

Listing 3: The Encodeable and Decodeable trait definitions

In the interpreter, this is represented using two traits, *Encodeable* and *Decodeable*. Since the number of valid encoded values of a specific data type is smaller than the number of values of this data type without encoding, the encoding and decoding operations return a differently sized data type. For example, a 32-bit integer (*i32*) is encoded into a 64-bit integer (*i64*) and an *i64* is decoded into an *i32*.

As established in subsection III-B, floating point arithmetic is generally not used in safety software. Since the interpreter presented in this project is based on an existing interpreter, floating point arithmetic is still supported. In the future, it may however be disabled by default. 64-bit floating point values have to be encoded using 128-bit types which are not part of the Rust standard or core libraries. Therefore, we needed to use an external library that provides these types. For this project, we decided on *qd*[2] as it already implemented many traits required for arithmetic as well as type conversions. However, it is by default not `no_std`-compatible and misses some methods related to converting numeric types to and from raw byte data. We therefore extended the *qd* library to be `no_std`-compatible by using *libm* functions and added analogous conversion functions to the ones implemented on *f64*.

Within the interpreter, the execution is encoded at the instruction level. This means that values are saved without encoding on the stack and encoded by instructions that use them. Then, the instruction decodes the value that is written back to the stack and raises a trap if the encoding was violated.

```
fn binop_trap<T, F>(&mut self, op: F) ->
  Result<()>
where
    T: StackAccess + LittleEndian,
    F: FnOnce(T, T) -> Result<T>,
{
    let c2 = self.stack.pop();
    let c1 = self.stack.top();
    let ret = op(c1, c2)?;
    self.stack.write_top_bytes(ret);
    Ok(())
}
```

Listing 4: Generic implementation of fallible binary operation instructions

The execution of instructions is handled generically by the interpreter. This implementation can be seen in Listing 4. This method of the *Runtime* type takes a function as an argument that implements a binary operation on a type that can be taken from the stack and may fail (this is used e.g. to implement division). The arguments to the function are taken from the stack and the result (if a trap did not occur) is written back on top of the stack.

Our encoded version of this method is shown in Listing 5. It introduces more parameters and trait bounds. The type T that the binary operation operates on must now be encodeable into a new type U which itself can be decoded into T. The parameters of the binary operation are taken from the stack as values of type T and encoded. Then, the binary operation is executed and the result decoded back from type U to T. If the encoded value is invalid, a trap is raised.

[2]https://github.com/barandis/qd

```rust
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(())
}
```

Listing 5: Encoded implementation of fallible binary operation instructions

D. The Diversified Runtime

The safe WebAssembly interpreter implements diversified execution as described in [3]. This method executes a function twice, once as a native function (i.e. without encoding) and in parallel as an AN-encoded function. This introduces an additional layer of safety over just encoded execution as the encoded result can be compared to the native result to find discrepancies.

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

Listing 6: The diversified runtime type definition

In the interpreter, this is implemented through a new *DiversifiedRuntime* type. As Listing 6 shows, this type wraps two WebAssembly runtimes, one of which is AN-encoded using the *code* field.

When a WebAssembly function is invoked in the diversified runtime, it is first invoked in the encoded runtime. If an error occurs there, the `DiversifiedRuntime::invoke_encoded` method (shown in Listing 7) will immediately return an error. Otherwise, the native function is also executed and the results are compared.

V. Performance

For safety-critical use cases, the performance of the safe WebAssembly interpreter is a very important consideration.

```rust
pub fn invoke_encoded(
    &mut self,
    name: impl AsRef<str> + Clone,
    args: &[Value],
) -> Result<Option<Value>> {
    let encoded_res = self
        .encoded_runtime
        .invoke_encoded(name.clone(),
  args, self.code)?;
    let native_res =
  self.native_runtime.invoke(name, args)?;
    if encoded_res == native_res {
        Ok(encoded_res)
    } else {
        Err(Trap::new(
            TrapReason::MismatchedResults,
  0))
    }
}
```

Listing 7: Function invocation in the diversified runtime

We measured the performance of the interpreter using a program that manually calculates the sum from 1 to 10 000 using a for loop. The results are presented in Table I.

TABLE I
100 000 RUNS ON AMD RYZEN 5 3600, EXECUTION TIMES IN μs

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

Firstly, the time difference between encoded and native execution in a singular runtime was negligible, so Table I only shows encoded execuation. Between the singular and diversified runtime, the minimum and median execution times were almost exactly doubled. The maximum execution time varied greatly between runs of the benchmark, implying that it is not a bottleneck in the interpreter, but more likely in the environment (e.g. caused by scheduling).

This means that the overhead added by encoding and diversified execution can in practice be calculated, making the safe WebAssembly interpreter a viable option for real-time tasks.

VI. Future Work

While the safe WebAssembly interpreter already works, there are some remaining topics that could be addressed by future work. This includes using a completely encoded stack for encoded execution. In the interpreter presented here, operands are encoded before an instruction and the decoded value is written back to the stack. This means that the operands are protected during the execution of the instruction, but the remainder of the stack is not protected. Encoding the entire stack would add additional protection, but is difficult to implement in the existing interpreter.

Furthermore, the safety of external functions is a concern. In general, external functions must be treated as black boxes by the interpreter. However, external functions interact with the WebAssembly interpreter's stack. If the entire stack was encoded, this constraint could be used to check for errors even in the return values of external functions. Additionally, a diversity framework as described in [3] could be created for Rust functions and used for external functions which are defined in the same program using the interpreter.

Another step would be to further examine the safety of the WebAssembly interpreter through field tests or fault injection. A possible candidate for field tests is the *RasTA* protocol used for communication between railway hardware by DB. This would, however, first require support for more complex external functions to handle networking.

VII. Discussion and Conclusion

The growing support and ease of implementation make WebAssembly a highly promising tool for safety systems. As this project shows, adding safety techniques to a WebAssembly interpreter is possible. Such an interpreter could be used to run safety-critical software on possibly unsafe hardware.

There are, however, still limitations to the usefulness of WebAssembly for safety software. Firstly, while the use of external functions is already possible, it is not very ergonomic. Secondly, on an STM32 microcontroller with 96Kb RAM, the interpreter runs out of memory. While safety hardware is becoming more powerful, the overhead of the runtime may not be acceptable on current systems.

In the future, the safe WebAssembly interpreter could be certified. A certified interpreter could be used as a target for safety software that only requires the safety software itself to be certified, independently of the hardware where the interpreter is run. At the current time, this is made significantly more difficult by the fact that a certified Rust compiler does not exist. However, AdaCore and Ferrous Systems are currently working to create such a compiler, making the idea of a certified WebAssembly interpreter possible in the future.

References

[1] P. Forin, "Vital Coded Microprocessor Principles and Application for Various Transit Systems," *IFAC Proceedings Volumes*, vol. 23, no. 2, pp. 79–84, Sep. 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017526531

[2] C. Fetzer, U. Schiffel, and M. Süßkraut, "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, B. Buth, G. Rabe, and T. Seyfarth, Eds. Berlin, Heidelberg: Springer, 2009, pp. 283–296.

[3] M. Süßkraut, A. Schmitt, and J. Kaienburg, "Safe Program Execution with Diversified Encoding," p. 9.

[4] B. A. Wichmann, "A Note on the Use of Floating Point in Critical Systems," *The Computer Journal*, vol. 35, no. 1, pp. 41–44, Feb. 1992. [Online]. Available: https://doi.org/10.1093/comjnl/35.1.41