WASM + Dela

smart contract execution env.

Master Semester Project

Responsible : Prof. Bryan Ford
Supervisor : Noémien Kocher
Student : Maxime Sierro
Introduction
The project

Implementation of a new WebAssembly smart contract execution environment for the DEDIS Ledger Architecture
Motivation

2 big disadvantages with the current native execution module:

1. Necessary recompiles of the entirety of the node’s environment

2. Support for Go smart contracts only
Node environment (in Dela)

Smart contract execution

execute

...
WebAssembly (WASM)

- Binary format obtained from higher level "source languages"
- Introduced in 2017 for web browser use
- Sandboxed execution
- eWASM: Ethereum’s proposed execution layer redesign
Goals

- Fully functional alternative to the native module
- Simultaneous support of multiple source languages
- Determinism analysis
- Ideally: automated smart contract loading
Design
Dela ledger node

module X

... execution module

module Y

WASM execution env.
(the ‘box’)

in

exec

out
3 possible solutions

1. Web browser application

2. Web server

3. Unix daemon
Main factors

- Amount of relevant resources
- Ease of communication
- WASI transition
WASI

- WebAssembly System Interface, extension of WASM to the OS level
- Newer: 2019
- Less languages are WASI-compatible
Final choice: Node.js application

Best of both worlds:

- Easy communication with the framework
- JavaScript’s “standard WASM” API
C/C++ support arguments

- State of the art binary translation
- 2 languages with one stone
- Prevalence
Go support arguments

- Accurate comparisons with native executions
- Frequently used by the lab
Differences

- C/C++ treat WASM as a library, Go treats it as an application.

- Very different implementation issues
Results
CPU : 2.5 GHz
Incrementation of a randomized counter
Ed25519 crypto operations

Single executions of smart contracts containing sequential operations

- Go : DEDIS’s Kyber library
- C : Libsodium library
Randomized scalar multiplications of the \((x, \frac{4}{5})\) base point
Ed25519 Point Multiplication

Runtime (ps)

Number of operations

CPU: 2.5 GHz

Randomized scalar multiplications of points
Ed25519 Point Addition

CPU : 2.5 GHz

Randomized additions of two points
Trustworthy takeaways

- Very low overhead
- Similar order of magnitude for all executions
- Go to WASM up to 3 times slower than native
Determinism
Sources of nondeterminism

1. Nondeterministic imports

2. NaN result from a floating point operator

3. Resource exhaustion
WASM is “almost deterministic”

OK for experiments, **not** for a realistic use case

What should be done if anyone can submit a smart contract?
eWASM’s solution : Sentinel

- Smart contract validator released in “Alpha0” state
- No documentation, written in Rust
- Last commit 2.5 years ago 😞
Sentinel strategy

1. Nondeterministic imports  =>  Reject if there is an illegal import

2. NaN result   =>   Reject if there is a floating point operator

3. Resource exhaustion   =>   Fix a limit on the stack size
Sentinel strategy

Theoretically applicable: Sentinel validates contracts after the WASM compilations

Reverse engineer without the metering injector?

Does it really guarantee strict determinism?
Automated smart contract loading
Current smart contract loading

1. Compile the smart contract to WASM

2. Add the binary in the environment’s correct folder

3. Add ~10 lines of JavaScript code

4. Recompile & relaunch
WASM compilation automation

Unfeasible for C/C++ : unpredictable fixes required in practice

Probably not worth the trouble anyway
Automatic handling of new smart contracts

First idea: automatically handle new binaries

Problem: the API does not expect a large nor variable number of them

Solution: switch from “1 binary per smart contract” to “1 binary per source language”
Conclusion
Suitable for experiments: multiple languages, on par with native module, satisfying performance...

But can it become more?
Future work

Immediately useful, should be quick :

- Rust support

Crucial but would take a lot of time :

- Strict determinism (if possible) & automated smart contract loading

Maybe slightly better in many years :

- Support WASI and use the relevant Node.js API
Questions
Backup Slides
Strong candidate: Rust

- Top tier WASM support

- Similar to C/C++: Emscripten
Counter Increase

CPU : 2.5 GHz

Incrementation of a randomized counter
Automatic handling of new smart contracts

Switch from “1 binary per smart contract” to “1 binary per source language”

Achieved differently depending on the language

Simplified and scalable JavaScript code

Drawback : users must update and compile larger files