The Importance of a Plaform-Agnosting Bytecode for a Decentralized Network

Gabriele Miotti

May 2023

Abstract

How can a distributed network agree on the execution of an arbitrary code in an trust-free way? Currently there is plenty of solution to achieve that, every solution differs by the network's structure and procols but the common ground is the presence of a bytecode that is able to run arbitrary code in a deterministic manner. The following 'paper' will describe the basic characteristics of an Platform Agnostic Bytecode and how is used in Polkadot, a distribuited network that 'aims to provide a scalable and interoperable framework for multiple chains with pooled security'(polkadot-overview paper).

1 Platform-Agnostic Bytecode

1.1 Definition

A Platform-Agnostic Bytecode (PAB) is a bytecode that follows those two main principles:

- Turing Completness
- Support for tooling that makes it exacutable on every machine

A bytecode like this ideally is designed to be executed on a virtual machine that follows general patters. This design should make easier the compilation to another bytecode, understandable by a real machine. Examples of real architectures with specified bytecode are AMD and Intel with x86 or ARM with aarch64.

1.2 Execution

PABs require multiple phases of Compilation, the first one is encountered when you want to compile you High-Level language to the PAB using a Cross-Compiler. Once you have the arbitrary PAB code, you should be able to run it on every machine using another compiler that will create the final executable code.

Re-compiling is not the only way to execute a PAB, another common solution is to implement a VirtualMachine (VM) able to run aribitrary PAB code interpreting it.

1.3 Key features

Every bytecode can be a PAB if tools to make it runnable to different machine exist. If every bytecode, ideally, can become a PAB then there must be some metrics to define which one is the better PAB, those are:

Hardware Independence A bytecode can't be a PAB if tightly related to specific hardware. A PAB can be defined as such if there is no direct connection between bytecode and hardware, the only exception is if there is a small relationship but the execution on different hardware requires only little overhead.

Sandboxing The machine used to execute the PAB is defined as *embedder*. The embedder will execute arbitrary code, possibly malicious, and avoiding any security problem is the aim of the embedder, sandobxing is the solution. Executing the PAB in a sandboxed environment makes impossible to compromise the embedder, the implementation of the sandboxed environment is embedder dependent but a PAB can be more or less suitable to this technique.

Efficency The efficency of a PAB has a lot of meaning, it could be:

- Compiling High-Level Language to the PAB
- Execution of the PAB, it could mean compile to the final bytecode and then execute it, interpret it or more complex solution

Generally the first is not really related to the PAB, but more on the tool used (examples gcc, rustc, etc.). The execution efficiency is the real deal, how fast a PAB can be executed on a machine is crucial.

Tool Simlicity The easyness of compiling an High-Level language and the executing the PAB is very important to make a it usable in the real world.

Support as Compilation Target Writing bytecode by hand (or any text representation) is something really rare and done only in specific cases. Every compiled language has a compiler to make this, and is very important for a PAB to support the compilation from as many languages as possible.

1.4 Current usage

PAB are widely used and the following are a couple of examples:

eBPF Linux brought eBPF into the kernel, enabling arbitrary programs to be executed in a privileged context (OS level).

LLVM IR LLVM IR is the LLVM assembly language, it provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy. (https://llvm.org/docs/LangRef.html#abstract)

WASM WebAssembly is a safe, portable, low-level code format designed for efficient execution and compact representation (https://webassembly.github.io/spec/core/)

1.5 PAB in blockchains

Blockchains are Distributes Systems that needs to agree on the execution of arbitrary code (more or less, TODO: explain better) and the code has to run on different machines. PAB is the solution for both problem, but there is a little caveat in the first problem: the code code execution must arrive at the same result, idependently of the machine the code is running on. What has just been described will be called ëxecution determinismẗrom now on.

2 WASM

2.1 Definitions

(wasm-core) (wasm-polkadot)

WebAssembly, shortened to Wasm, is a binary instruction format for a stack-based virtual machine(wasm-core). It is a platform-agnostic binary format, meaning it will run the exact instructions across whatever machine it operates on(wasm-polkadot).

'asm.js' was Wasm precursor, but browser vendors like Mozilla, Google, Microsoft, and Apple focus on the Wasm designed. The main goal was to create a binary format where a couple of the main wanted features were: compact, support for streaming complication and sanboxed execution.

Wasm is currently a compilation target for a lot of high level language, this allow languages to enter in the web-world, in client or server applications, but also in completely different application. Examples are plugins, if an application is able to accept, execute and make in interact the application itself then we have a entire set of High-Level languages able to create those plugins having wasm as minumum common divisor. Then we have a entire set of High-Level languages able to create those plugins having wasm as minimum common divisor.

The main design goals in the wasm specification introduction (wasm-core) are:

Fast
It's design allow to create executor with so less overhead that the execution is almost fast as native code

Safe
: It is completely memory-safe as long as the executor is correctly behaving, sandboxing the execution properly

Well-defined
: The definition of the binary format makes easy to create a valid executor that makes the code behave correctly

Hardware-independent
: The compilation process is independent by the architecture that will run the code

Language-independent
: There's no strong influence by other binary format, language or programming model

Platform-independent
: It can be compiled and be executed on all modern architectures, embedded systems or applications as browsers

Open
: There is simple way to interoperate the with executor/environment

Other important consideration are made on the efficiency and portability, the word used to described those two features are: compact, modular, efficient, streamable, and parallelizable. Something not clear at first look is m̈odular¨; it means that the program can be split into smaller parts and those can be transmitted, cahed or consumed separately.

In the following chapters the words executor, embedder or environment have the same meaning.

2.2 Specifications

The specification does not make any assumption on the environment, this makes it completely contraint-less, it just must follows all the defined instruction set, binary encoding, validation, and execution semantics.(wasm-core)

Wasm is stack-based, this means that the instruction set is very different from the standards architecture's bytecode that normally are registered-based. Wasm has also a one-to-one text representation other than the normal binary representation, of course it makes the code less compact but almost human readable.

All the concepts present in the specifications are very high-level even if it is a low level language, those concepts are the following:

Values
: Wasm has only four data type, integers and foaloating points (following IEEE 754 standard) both 32 and 64 bits

Instructions
Being a stack based language every instruction works implicit on a stack but there is a general division between:

- Simple Instructions, performing basic operations on data
- Control Flow, allowing to follow some high-level language control flow having nested blocks

Traps
Those are instructions which immediately aborts the execution, the termination is not endled by wams itself but by the embedder

Functions
Being so new, this assemply-like language, allow users to work with functions abstracting some standards assemply's complexity

Table
NOPE

Linear Memory
This is where the communication between the code and the environment happens, like the name says this is a contiguous area of memory given to the code. This memory is very crucial for the security considerations that we will see later.

Modules
A Module contains everything just explained, this is the logical container of the code. Every wasm code is made by a single module.

Imports
??

Embedder
Of course to be executed wasm needs the embedder, the main jobs are:

- loading and initiate a new module
- provide imports
- manage exports

Other important concepts explained in the specification are wasm phases, they are:

Decoding
Decode the binary format to the specified abstract syntax, the implementation could also compile directly to machine code.

Validation
A decoded module has to be valid, the validation consists in check a set of well-formedness conditions to guarantee that the module is meaningful and safe (wasm-core, un po' troppo copiato questo)

Execution

- Instantiation, set up state and execution stack of a module
- Invocation, calling a function provided by the module to start the effective execution

2.3 Security guarantee

Wasm's aim is to be extremely secure the the specifications describe a lot of aspects to achieve that. The security guarantee depends mostly on the execution, WebAssembly is designed to be translated into machine code running directly on the host's hardware. Being so portable wasm can be sent to someone and be executed freely, examples in every browsers. We are running wasm our machines every day and if would not be so secure then we would had notice a lot of problems.

Executing wasm is potentially vulnerable to side channel attacks on the hardware level(wasm-core) and isolation is the only way to make secure the execution. If the embedder translate one on one every instructions then everything can be computed on your computer, but nothing dangerous if the code has no access to the environment where is executed.

The problem is that a completely isolation makes wasm useless, so there's a way to communicate with the environment or also have access to it, but those features are extremely limited and designed to be secure.

2.3.1 Linear Memory

(linear-memory)

From WebAssemply you have direct access to raw bytes, but where are allocated those bytes? Wasm uses a MemoryObject provided by the embedder to describe the only accessible memory, beside the stack. (linear-memory)

Wasm does not have pointer types, values in the linear memory ara accessed as a vector, where the first index of the memory is 0.

Wasm, for security reason that will be explained in next chapters, works in a 32-bit address space, this makes usable only 4GiB of memory. Being the position of the Linear Memory memory unknown to the wasm blob every load or store the the memory is made passing through the embedder that will also do bounds checks to make sure the address is inside the wasm Linear Memory.

This level of control makes impossible to have memory leak in the environment during the wasm execution because there is a completely memory isolation. (linear-memory)

2.3.2 Stack

stack more or lesse expained

How the stack is managed?

substrate -¿ wasm-instrument -¿ stack injection to make it deterministic (shadow stack at the beginning)

2.3.3 Communication in a sandboxed environment

We just described how wasm provides no ambient access to the computing environment in which code is executed (wams-core), thanks to a mix of wasm design choice and embedder implementation. But how works then the interaction with the environment?

Every interaction can be done by a set of functions provided by the embedder and imported in the Wasm module(wasm-core), those functions are called **Host Functions**. Host functions allow wasm code to access to resources, operating system calls or any other type of computation offered by the embedder.

2.4 Execution

General description over wasm execution
(didn't find a lot of resources on this general topic)

2.4.1 AOT

explenation + tool description (wasmtime spec)

2.4.2 JIT

explenation + tool description (wasmtime spec?)

2.4.3 Interpret

explenation + tool description (wasmi spec)

2.4.4 SPC

explenation + tool description (wasmer spec)

3 Polkadot

3.1 What's Polkadot?

Polkadot is an heterogeneous multi-chain system.

Polkadot provides the bedrock "relay-chain" upon which a large number of validatable, globally-coherent dynamic data-structures may be hosted side-by-side (polkadot whitepaper)

Polkadot aims to provide a scalable and interoperable framework for multiple chains with pooled security that is achieved by the collection of components described in this paper (polkadot overview paper)

Briefly: Polkadot utilises a central chain called the relay chain which communicates with multiple heterogeneous and independent sharded chains called parachains (portmanteau of parallel chains). The relay chain is responsible for

providing shared security for all parachains, as well as trust-free interchain transactability between parachains. In other words, the issues that Polkadot aims to address are those discussed above: interoperability, scalablility, and weaker security due to splitting the security power. (polkadot overview)

3.2 Polkadot's protocol

1. Each parachain:
   (a) collectors run full relay chain node to keep up the latest state
   (b) build new block on top of this latest state and submit blocks to the parachain's validator
   (c) parachian's validator produce the new relay chain block candidate
2. validator block producing behaviour ...
3. substrotocol to ensure data sharding
4. managing of messagging between parachians
5. validators submit votes to resolve forks and have a single head

(from polkadot overview)
Explanation of the STATE TRANSITION FUNCTION
'Like any transaction-based transition system, Polkadot state changes via an executing ordered set of instructions, known as extrinsics' (from polkadot overview)

Polkadot relay chain is divided in: 1. Runtime -¿ contain the state transition logic, compiled into WASM and stored as part of the state (under well known keys), in this way the state transition logic can be upgraded 2. Runtime environment / Client -¿ contian all the remaining blockchain relted stuff

QUESTION -¿ technically this section could require 100 pages, is it ok to explain here ONLY the relevant part to understand the next section?

3.3 WASM in Polkadot

'Polkadot's state is changed by executing an ordered set of instructions(extrinsics)'
'For easy upgradability this Runtime is presented as a Wasm blob'
(from polkadot spec)

3.3.1 Usage a WAMS in Polkadot

STF Talking about SUBSTRATE and how facilate the construction of the client and runtime

PVF talking about cumulus, PVF and PoV

SmartContracts talking about pallet-contracts that enable you to uplaod smart contracts onchain

SPREE ???

4 WASM Rivals

4.1 EVM

Explain what is EVM Bytecode and how is used in Ethereum

4.2 eBPF

Explanation over eBPF, how Solana is using it and then how and why is not a good fit for polkadot (https://forum.polkadot.network/t/ebpf-contracts-hackathon/1084)

4.3 RISC-V

What is RISC-V, why seems to be a good fit for polkadot (https://forum.polkadot.network/t/exploring-alternatives-to-wasm-for-smart-contracts/2434)