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 real machine's bytecode. 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 your 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. 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 for this feature.

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 tools 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:

JVM

LOL

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. llvm

WASM

WebAssembly is a safe, portable, low-level code format designed for efficient execution and compact representation wasm-spec

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 execution must arrive at the same result, regardless of the machine the code is running on. What has just been described will be called ëxecution determinismf̈rom now on.

2 WASM

2.1 Definitions

(wasm-core) (wasm-polkadot) (wasm-spec)

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 contraintless, 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

TODO

Exports

TODO

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 Execution

(wasmtime)

Wasm specifications can be perfect making everything unbreakable but at the end everything depends on the embedder's implementation, if it is not secure then wasm execution itself is not. Wasm can be executed in different ways, the main one used in the blockchain world are: Ahead Of Time Complication (AOT), Just In Time Compilation (JIT), Single Pass Compilation and Interpretation.

Every type has its own advantages but also requires different tricks to make everything secure, one important thing provided by wasm is an intensive test suite to check the correctness of the embedder, all the tests are maintained here: .

The common divisor for the first three types of execution is the transpilation of a stack-based bytecode to a register one, this means that the compiler tries to elide every access to the main stack used by wasm allocating everything needed in the registers. It's impossible to completely avoid the interaction with the stack, this means that at the end the final bytecode will use a stack, in some cases the stack of the embedder. (This is could be explained more in depth but requires a lot of time to make sure everything is correct)

There is though an important difference between value stack and shadow stack ... (it's worth to explain?)

2.3.1 AOT

AOT is the standard compilation, all the code is compiled and later executed. Wasmtime is a wasm embedder, it is a stand alone wasm environment but it could be also used as library to create a wasm environment in your bigger application. Wasmtime offer offers this feature, it accept wasm in text or binary format and compiles it to some architecture's bytecode.

2.3.2 JIT

JIT is a dynamic compilation where the bytecode si compiled only if it needed, the compiler first need to create an intermediate representation to being able to compile the different parts only if the execution requires to. A really simple example to make is: we have a program that given an input calls function A or B, the JIT then will understand this structure and compile the entry point and only one function between A or B based on the initial input.

JIT is really efficient because the huge work is already been done by the first phase of compilation where the High-Level language is compiled into wasm, now the JIT has only to compensate all the differences between bytecose, but they are both really low level and this makes this process really more efficient then the first one.

Wasmtime is specialized in this type of execution and it makes it really efficient keeping every secure.

2.3.3 SP

A Single Pass compiler is a restriction of AOT compiler, the complexity of the compilation must be O(n) so the wasm bytecode will be scanned through only once. Like every other compilation method here the trade off si not to create efficient final code but to create the final bytecode as fast as possible, why not optimizing should be worth? Because wasm is already compiled from some higher level language and the compiler probabily alredy did a lot of optimizations.

Wasmer is wasm embedder with a lot of features, in particular they implemented a single pass compiler for all the most important architecures.

2.3.4 Interpret

wasmi-spec

Interpretation is easier to think way to execute wasm, it becomes like any other interpreted language executed by a specialized Virtual Machine. There are multiple ways to interpret code but we will focus on one of the most efficient wasm interpreter, wasmi.

Wasmi is an efficient WebAssembly interpreter with low-overhead and support for embedded environment such as WebAssembly itself (wasmi-spec).

Currenly the first wasm bytecode pass produce another stack based bytecode, called WASMI IR, and then this bytecode is interpreted by the Virtual Machine, even with this transpilation it is only 5 time slower then the compilation to native bytecode of the architecture. (Resource to be found)

2.4 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.

2.4.1 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)

2.4.2 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.

2.4.3 Wasmtime Security guarantee

wasmtime-book

Wasmtime is widely used in different environmets and one of those is all the polkadot ecosystem, precisely wasmtime is used inside substrate. Substrate is a framework to develop blockchains that with some tweaks can become parachains, polkadot itself is a substrate based chain.

Wasmtime main goals is to execute untrusted code in a safe manner.(wasmtime-book)

Some features that makes executing wasm by wasmtime so secure are just inherith by wasm specifications, some examples are: the callstack is inaccessible, pointers are compiled to offsets into linear memory, there's no undefined behavior and every interaction with the outside world is done through imported and exported functions(wasmtime-book).

Wasmtime to those features adds a lot of mitigations to limit issues:

- Linear memories by default are preceded with a 2GB guard region
- Wasmtime will zero memory used by a WebAssembly instance after it's finished.
- Wasmtime uses explicit checks to determine if a WebAssembly function should be considered to stack overflow (this is really a deep concepts about how wasmt time manage the wasm's stacks, not sure if it's worth to keep)
- The implementation language of wasmtime, Rust, helps catch mistakes when writing Wasmtime itself at compile time

3 Polkadot

3.1 What's Polkadot?

polkadot-whitepaper polkadot-overview

Polkadot is an heterogeneous multi-chain system (polkadot-whitepaper), the aim is to provide a scalable and interoperable framework for multiple chains with shared security(polkadot-overview).

At the center of the entire system there is the 'relay chain', responsible for providing shared security to the other chains that are part of the system. Those chains are called 'parachains', they can be heterogeneous and independent between eachothers; the central point (relay chain) is making possible a trust-free interchain transactability and the pooled security. (polkadot-overview)

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)

In the Figure 1 you can see in the middle the relay chain that connects multiple parachains.

3.2 Polkadot's protocol

Briefly the Polkadot system consists of a single open collaborative decentralised network (relay chain), that interacts with many other external chains (para chains)(polkadot-overview). For the realy chain

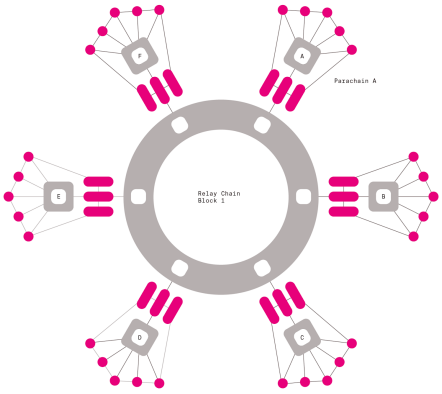

Figure 1: Polkadot's Architecture

the internals of the parachians are not relevant, parachains need only to adhere to a specific interface. The relay chain then ensure that the parachain is trustable as a whole, not single nodes.

The ecosystem expect multiple actors to make the protocol to work, the main are:

- Validator, performs the bulk of the security work
- Nonimator, stakeholder who backs and selects validator candidates
- Collator, collects and submits parachain data to the relay chain

(polkadot-overview)

The protocol is composed by multiple phases and it defines the communication between parties, the main processes are:

1. The 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
2. The realy chain:
   (a) validator associeted to parachians produce the new relay chain block candidate
   (b) validator follow a substrotocol to ensure data sharding
   (c) validators submit votes to resolve forks and have a single head
   (d) managing of messagging between parachians

(from polkadot overview)

3.2.1 State Transition Function

The protocol demonstrate how the main goal is to verify what's happened on parachains, this is described by the parachain logic. One of the few required thing to a parachain is indeed a State

Transition Function, this describe the parachain logic and produce the transition between two states. Like any transaction-based transition system, Polkadot state changes via an executing ordered set of instructions, known as extrinsics(polkadot-overview).

The STF is also present in every validator nodes because it describe the relay chain logic. Generally STFs in parachain or relay chain are wasm blob, currently all STFs in the polkadot ecosystem are written in wasm.

Every parachain or relay chain node can be divided in two parts: the STF (or Runtime) and the Client, the Client implement everything else required to make the protocol to work, from storage management to gossip transactions.

The runtime is compiled into WASM and stored as part of the state, in this way the state transition logic can be upgraded.

3.3 WASM in Polkadot

Substrate is a framework to build block chain, it is separate from Polkadot, with it you're able to build so called Solo Chains, that do not care about the polkadot protocols.

Substrate is the main, and only for now, framework used to build blockchains in the Polkadot ecosystem, even the relay chain is built with substrate. What does substrate is abstract all the complexity of writing client and runtime code, you have already almost everything done at client level and you have the freedom of developing whatever you want in the runtime, the state transition function of the blockchain you are building.

Substrate compiles the runtime to wasm and the client has an embedder, wasmtime, able to run the STF. As long as the chain is a Solo Chain substrate manage everything, it compiles the runtime in wasm and implements all the costum logic to make the client and the runtime communicating through the embedder wasmtime.

The problems occur when you want to be part of the polkadot ecosystem, where there is pooled security and the logic not only must be executed by the nodes that compose the parachain but also by the validators of the relay chain. This is made possible following a protocol made by multiple phases that is built on top of two building blocks: the Proof Validation Function and the Proof of Validity (from polkadot spec).

3.3.1 PVF

Using Substrate you end up with a Substrate-Runtime, it is only a wasm blob that implements the State Transition Function of you're blockchain but to be a parachain you need to provide to the relay chain somethig that differs a bit form the Substrate-Runtime. The protocol requires a Proof of Validity that is composed by the Substrate Runtime and another function called 'validate_block' and the reason why this function is required is related to a constraint of the relay chiain: polkadot does not know anything about all previous state of the parachain.

Everything needed by the Runtime is not present in the validators node but is provided in the block proposed by the collators, that's called: PoVBlock, the structure of the PoV will be explained later, important to know that every information needed in the execution of the PVF indeed here.

The 'validate_block' function is the glue between the parachain'STF and the PoVBlock, it accept the PoVBlock and reconstruct the previous state, applies all the extrinsics using the runtime and then check that the new state is consistent with the one proposed by the collators.

Both relay chain and parachain implements the same HostFunctions for the runtime, this means that the STF of the parachain would accesses the storage thorough the same functions in the relay chain and in the parachain but what is known by the nodes is different.

As was just said the relay chain does not know the previous state of the parachain but the information resides in the PoV. The function 'validate_block' override the host functions to let the STF access the reconstructed internal state instead of going into the relay chain client.

This encapsulation with a substitution of the host functions makes validator able to execute different STFs without knowing anything about the parachain that is validating.

3.3.2 PoV

(cumulus-notes)

The Proof of Validity is made by all the things needed by Runtime Validation, it is mainly composed by:

- Header of the new block
- Transactions included in the block
- Witness Data
- Outgoing messages

The third and the fourth elements in the list are really peculiar to the polkadot protocol.

The witness data is what makes possible the state transition validation withouth knowing the entire state of the parachain by the relay-chain.

The state in polkadot is managed as a Merkle-Patricia Base 16 Trie where the hash of the root define the state proof, the state proof of the previous block is stored in the relay chain and the witness data are:

- Data used in the state transition by the collator
- Alongside their inclusion proof in the previous state

Instead the Outgoing messages are everything that need to be delivered to other parachains, the underneath protocol is XCMP/XCM which enables the parachain communication between parachain or the relay chain itself.

3.4 SmartContracts

SmartContracts are arbitrary code that can be upload on the chain and executed in the states transition.

PEPEEE

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)