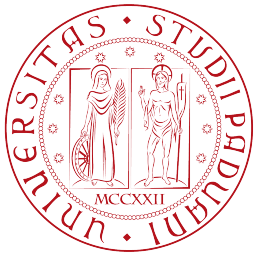

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in
Ingegneria Informatica

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

Relatore:
Prof. Luca Boldrin

Laureando:
Gabriele Miotti
2000165

Anno Accademico 2022/2023

Abstract

How can a distributed network agree on the execution of an arbitrary code in a trust-free way? Currently there are plenty of solutions to achieve that. Every solution differs by the network's structure and protocols but the common ground is the presence of a bytecode that is able to run arbitrary code in a deterministic manner. This paper will describe the basic characteristics of a Platform Agnostic Bytecode and how is used in Polkadot, a distributed network that 'aims to provide a scalable and interoperable framework for multiple chains with pooled security' [14].

Contents

1 PAB **1**

1.1 Definition 1

1.2 Execution 1

1.3 Key features 2

1.4 Current usage 3

1.5 PAB in blockchains 3

2 Wasm **7**

2.1 Definitions 7

2.2 Specifications 8

2.3 Execution 10

2.3.1 Execution Types 10

2.3.2 Embedders 12

2.4 Security guarantee 12

2.4.1 Linear Memory 13

2.4.2 Communication in a sandboxed environment 13

2.4.3 Wasmtime Security guarantee 14

3 Polkadot **15**

3.1 What's Polkadot? 15

3.2 Polkadot's protocol 16

3.2.1 State Transition Function 17

3.3 Wasm in Polkadot 17

3.3.1 PVF 18

3.3.2 PoV 19

3.3.3 SmartContracts 20

3.4 The crucial Role of WASM in Polkadot 20

4 Rivals **21**
4.1 EVM 21
4.2 eBPF 21
4.2.1 What is eBPF 21
4.2.2 Solana eBPF 22
4.3 RISC-V 23
4.3.1 RISC-V for SmartConctract 23

5 Conclusions **25**

Chapter 1

PAB

1.1 Definition

A Platform-Agnostic Bytecode (PAB), from now on will be defined as a bytecode that follows those two main principles:

- Turing Completeness
- Support for tooling that makes it executable on every machine

A bytecode like this ideally is designed to be executed on a virtual machine that follows general patterns. 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 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 Virtual Machine (VM) able to run arbitrary PAB code interpreting it.

1.3 Key features

Every bytecode, ideally, can become a PAB if tools to make it runnable to different machine exist. There are however some metrics to define which one is better than others; example of metrics are:

Hardware Independence

A bytecode if tightly related to specific hardware can't become a PAB because translations to different hardware would be impossible. A weakly coupled bytecode though can become PAB if the coupling to a specific hardware is negligible and requires some minor adaptations to run on different hardware is required. A bytecode completely hardware independent, that does not make any specific assumption on the hardware on which will be running on, is the better one.

Sandboxing

The machine used to execute the PAB is defined as *embedder*. The embedder will execute arbitrary code, possibly malicious. A sandboxed environment is the typical solution to overcome any security problem. The execution in a different environment makes almost impossible to compromise the embedder from the PAB code. Implementing a proper sandboxed environment is embedder dependent but a PAB can be more or less suitable for this feature.

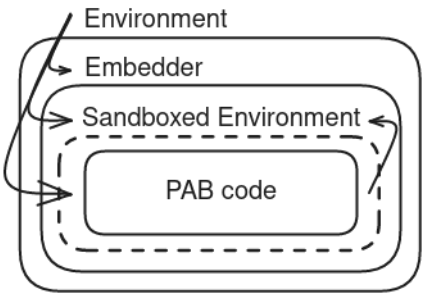

Figure 1.1: Sandboxing graphic example

Efficiency

The efficiency of a PAB has several facets, it could refer to:

- the efficiency of compiling High-Level Language to the PAB
- the efficiency of the execution of the PAB. In this case it could refer to the compilation to the final bytecode and the subsequent execution, the interpretation or more complex solutions

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

Tool Simplicity
The easiness of compiling a High-Level language and executing the PAB is very important to make it usable by every one.

Support as Compilation Target
Writing bytecode by hand 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 already widely used. A few examples are:

- The Java Virtual Machine (JVM) is one of the first that made the portability of the code one of the main concern of the language
- Linux brought eBPF [12], explained in one of the following chapters, into the kernel, enabling arbitrary programs to be executed in a privileged context (OS level)
- LLVM IR is the LLVM [2] 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.
- WebAssembly [8] is a safe, portable, low-level code format designed for efficient execution and compact representation.

1.5 PAB in blockchains

Blockchains are distributed systems that need to agree on the execution of arbitrary code performed on different machines. While it is not the purpose of this work to provide a detailed explanation of blockchains, it is essential for our purposes to understand that the goal of each machine is to produce blocks that are

connected with hashes to the previous blocks, so creating a block chain. Machines that take part in the block production are also called nodes, and they are part of a trust-less network. This means that nodes can't simply trust other nodes and rely on the blocks produced by them, on the contrary they must verify in some way the correctness of those blocks by themselves.

Let's simplify and define the block production algorithm as a function that takes an arbitrary input and produces and arbitrary output. Input and output are then bundled together to produce a block (little caveat, though, is that each input depends also on all the previous produced blocks).

If Block A is the first block, it contains A.input and A.output (A.input is a special input that does not depend on the previous block because A is the genesis block). A node then produces block B on top of block A while another node produces block C on top of A. At this point the new blocks can be arbitrarily added on top of either A, B, or C. A new node that just joins the network must decide where to append a new block, Figure 1.2. This question is extremely complex and outside our scope. We will however provide an answer to a part of the question: on top of which block the node must not attach the new block.

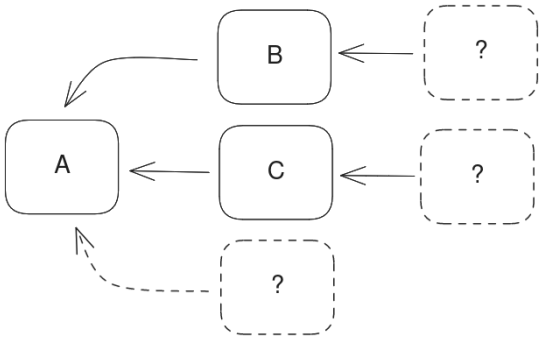

Figure 1.2: Block Production Example

PABs are the answer to this question. The main point of having input and output contained in the blocks, inside a trust-less network, is to let the nodes re-validate blocks to make sure other nodes behaved correctly.

The function that gives the output must return the same result regardless of the node and the node's architecture to make everyone agree on the correctness of new blocks. This property is called 'execution determinism'.

The new node in the network will validate all the blocks in the chain. It will start from the block A, re-compute A.output starting from A.input, make sure the block is correct (i.e., compare the computed A.output with the A.output contained in the block). The new node will then repeat the process for each block

up in the chain until the leafs. Every invalid block will be discarded as a possible base on which to append a new block.

PABs are the most suitable solution to implement this validation. PABs can encode any type of logic (first principle of a PAB) and can be executed on every machine (second principle). PABs can also fully address execution determinism, as we will describe in the following chapters.

Chapter 2

Wasm

2.1 Definitions

WebAssembly, shortened to Wasm, is a binary instruction format for a stack-based virtual machine [8]. It is a platform-agnostic binary format, meaning that it will run the same exact instructions across whatever machine it operates on. [5]

'asm.js' was Wasm precursor, but browser vendors like Mozilla, Google, Microsoft, and Apple focused on the Wasm design. The main goal was to create a binary format with some mandatory features: compact, support for streaming compilation and sanboxed execution.

Wasm is currently a compilation target for a lot of high level language. This allows many different languages to enter the web-world, in both client and server sides, but also in completely different applications.

Examples are plugins, imagine an application able to execute custom script developed by the user, if those scripts are in Wasm then the High-Level language used to develop those plugins is not constrained to a single one but to all the languages that are able to compile to Wasm.

Wasm main design goals, as specified in the Wasm specification introduction [8], are:

- **Fast** Its design allows to create executors with so less overhead that the execution is almost as 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 from the architecture that will run the code
- **Platform-independent** It can be compiled and executed on all modern architectures, embedded systems or applications (like browsers)
- **Open** There is a simple way to interoperate the with executor/environment

Other important considerations are made on the efficiency and portability. The specification describes Wasm also as: compact, modular, efficient, streamable, and parallelizable.

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

2.2 Specifications

The specification does not make any assumption on the embedder. This makes it completely unconstrainted as far as it implements all the defined instruction set, binary encoding, validation, and execution semantics [8].

Wasm is stack-based. This means that the instruction set is very different from the standard architecture's bytecode that are normally registered-based. Wasm has also a one-to-one text representation other than the normal binary representation, which 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. The most relevant are:

- **Values** Wasm has only four data types: integers and floating points (following IEEE 754 standard) both 32 and 64 bits
- **Instructions** Being a stack based language every instruction works implicitly on a stack but there is a general division between:
  - Simple Instructions, performing basic operations on data
  - Control Flows, 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 handled by wams itself but by the embedder
- **Functions** Being so new, this assembly-like language allows users to work with functions abstracting some of the assembly's complexity
- **Linear Memory** This is where the communication between the code and the environment happens: the linear memory 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 is the logical container of the code. Every Wasm code is made by a single module.
- **Exports** Once the module is instantiated all the defined exports are callable from the embedder, examples of possible exports are functions or global variables.
- **Imports** Wasm can import things from the embedder. The more common examples are the functions provided from the outside that are callable from the Wasm code
- **Embedder** Wasm to be executed needs the embedder. Itsthe main jobs are:
  - loading and initiate a new module
  - provide imports
  - manage exports

Other important concepts explained in the specification are Wasm phases:

- **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-formed conditions to guarantee that the module is meaningful and safe [5]
- **Execution** Execution is made by two sub-phases:
  - 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

Wasm specifications try to be unbreakable but at the end everything depends on the embedder's implementation, if it is not secure then Wasm execution itself is not.

2.3.1 Execution Types

There are two main categories of execution types: compilation and interpretation, those are then made by multiple sub-categories based on the technical details. The main difference is: in compilation the Wasm code is re-compiled to the architecture's bytecode and then natively executed on the machine. The interpretation, instead, uses a Virtual Machine to parse the Wasm code and execute it inside of the machine.

Every type has its own advantages but also requires different tricks to make everything secure. One important thing provided by Wasm is an articulated test suite to check the correctness of the embedder. [9]

The common divisor for every compilation type of execution is the transpilation of a stack-based bytecode to a register one; something similar happens also in the interpretation because the Wasm code is stack-based but generally it needs to be executed on a register-based one.

In Wasm there are multiple stacks:

- the Value Stack, used implicitly by Wasm to store temporary data or passing values to functions
- the Shadow Stack, this is not directly related to Wasm but used by many toolchains in the compilation to Wasm.

Passing values around only by values is not always efficient and in Wasm you don't have access directly to the value stack being only implicitly used by all the common instructions. The compiler uses the Shadow Stack, allocated in the Linear Memory (explained later), to put information in it and pass around pointer to this stack as value in the Value Stack.

Those two stacks are present in the Wasm code but when it needs to be translated to the final bytecode the compiler tries to elide every access to the Value Stack allocating everything needed in the registers. Registers are limited

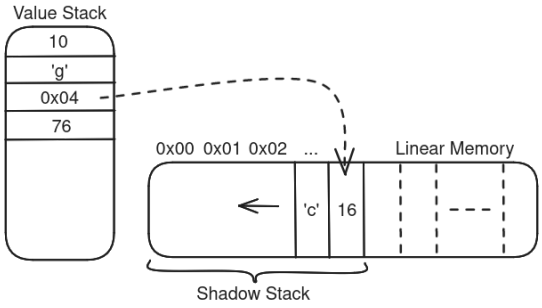

Figure 2.1: Value and Shadow Stack

though, so is impossible to use only them and the native stack of the embedder will be used if needed.

The main types of execution used in blockchains are: Ahead Of Time Complication (AOT), Just In Time Compilation (JIT), Single Pass Compilation and generally Interpretation

- **AOT** is the standard compilation, all the code is compiled and then executed.

- **JIT** is a dynamic compilation type where the bytecode is compiled only when needed. The compiler needs to create first an intermediate representation, to be later able to compile the different parts only if the execution requires to. A really simple example to make is: we have a program with two entry point, function A and B. The JIT process will cover the understanding of the structure, recognize the two functions and compile only the needed one, either A or B.

  Lots of optimizations are already been done in the first phase of compilation, from the High-Level language to Wasm. In the second phase (runtime compilation) the main goal is to compile only the required parts to the machine-specific bytecode not caring too much about adding optimizations.

- **SPC** is a restriction of AOT Compilation, the complexity of the compilation must be O(n) so the Wasm bytecode will be scanned through only once. Like every other compilation methods here the objective is not to create efficient final code but to create the final bytecode as fast as possible.

- **Interpretation** is the easiest way to execute Wasm, which becomes like any other interpreted language executed by a specialized Virtual Machine.

2.3.2 Embedders

There are multiple embedders able to execute Wasm using techniques not even explained before, we will focus on the embedders used in blockchains, they are Wasmtime, Wasmi and Wasmer.

Those three cover all the techniques just explained:

- **Wasmtime** is a stand alone Wasm environment but it could be also used as library to create a Wasm environment in your bigger application. Wasmtime offers multiple features, it accepts Wasm in text or binary format and you're able to used JIT or AOT types of execution.

- **Wasmi** [10] is an efficient Wasm interpreter with low-overhead and support for embedded environment such as Wasm itself.

  There are multiple ways to interpret code, Wasmi makes a first Wasm bytecode pass to produce another stack-based bytecode, called WASMI IR (Intermediate Representation). Thereafter this representation is interpreted by a Virtual Machine, even with this transpilation it is only 5 times slower then the compilation to the native bytecode of the architecture.

- **Wasmer** has a lot of features, both AOT and JIT but in particular they implemented a single pass compiler for all the most important architectures.

2.4 Security guarantee

Wasm principle aim is to be extremely secure. The specifications describes a lot of ways to achieve that feature, but he security guarantee depends mostly on the execution. WebAssembly is designed to be translated into machine code running directly on the host's hardware, so it can be sent to someone and be executed freely, (e.g., in browsers). We are running Wasm our machines every day, so security is a main concern.

Executing Wasm is potentially vulnerable to side channel attacks on the hardware level /citewasm-core-spec and isolation is the only way to secure the execution. The embedder translates one-by-one every instructions to native instructions on your computer, but nothing is 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 must be a way to communicate with the environment or have access to it, but

those features are extremely limited and designed to be secure.

2.4.1 Linear Memory

From WebAssemply you have direct access to raw bytes, but where are those bytes allocated? Wasm uses a MemoryObject provided by the embedder to describe the only accessible memory, besides the stack: the linear memory. [3]

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

Wasm, for security reasons that will be explained in the next chapters, works in a 32-bit address space. This makes usable only 4GiB of memory. Being the position of Linear Memory unknown to the Wasm blob every load or store to 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.

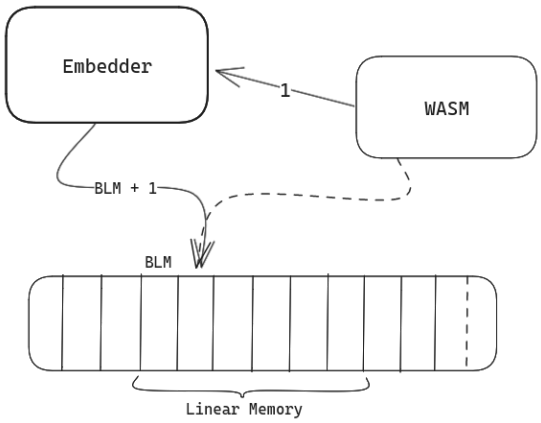

Figure 2.2: BLM: Base Linear Memory Pointer

This level of control makes impossible to have memory leaks in the environment during the Wasm execution because there is a complete memory isolation. [3]

2.4.2 Communication in a sandboxed environment

We just described how Wasm provides no ambient access to the computing environment in which the code is executed [8], thanks to a mix of Wasm design choice and embedder implementation. But how then the interaction with the environment works?

Every interaction can be done by a set of functions provided by the embedder and imported in the Wasm module [8]. Those functions are called Host Functions

and allow the Wasm code to access to resources, operating system calls or any other types of computation offered by the embedder. Generally the Exports provided by Wasm that are usable and callable from the embedder are called Runtime API.

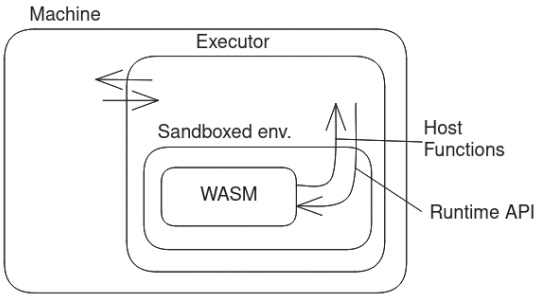

Figure 2.3: Environment communication

2.4.3 Wasmtime Security guarantee

Wasmtime is widely used in different environments and as you will see in the next chapter one of those is the polkadot ecosystem.

Wasmtime main goals is to execute untrusted code in a safe manner. [11]

Some features that makes executing Wasm by Wasmtime secure are just inherited by the 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. [11]

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

- Linear memories by default are preceded with a 2GB guard region
- Wasmtime will zero the 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
- The implementation language of Wasmtime, Rust, helps catch mistakes when writing Wasmtime itself at compile time

Chapter 3

Polkadot

3.1 What's Polkadot?

Polkadot is an heterogeneous multi-chain system [16], the aim is to provide a scalable and interoperable framework for multiple chains with shared security. [14]

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 each others; the central point (relay chain) is making possible a trust-free inter-chain transactability and the pooled security. [14]

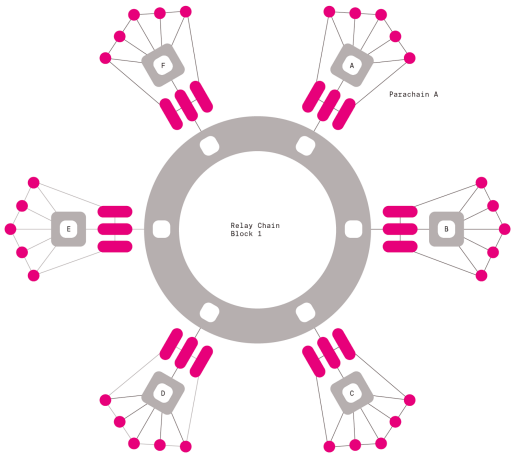

Figure 3.1: Polkadot's Architecture

Polkadot provides the bedrock relay-chain upon which a large number of validatable, globally-coherent dynamic data-structures may be hosted side-by-side. [16] What has just been described are the parachains that are not obligated to be blockchains, they just must respect all the polkadot protocol.

In the Figure 3.1 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) [14]. For the relay chain 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

The protocol is composed by multiple phases and it defines the communication between parties, the main processes are: [14]

1. The parachain's collators:
    - Run full relay chain node to keep up the latest state
    - Build new block on top of the latest state and submit blocks to the parachain's validator
2. The relay chain's validators:
    - The ones associated to parachians produce the new relay chain block candidate
    - Follow a sub-protocol to ensure data sharding, data are mainly parachain blocks and sharding is the process of making the validators collectively and robustly responsible for the availability of these blocks using erasure coding
    - Submit votes to resolve forks and have a single head
    - Manage messages between parachians

3.2.1 State Transition Function

The protocol demonstrate how the main goal is to verify what's happened on parachains, 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. [14]

The STF is also present in every validator node because it describe the relay chain logic. Generally STFs in polkadot are Wasm blob, currently all STFs are written in Wasm, for a parachain would be possible to use different PAB to create the STF but with complex solutions to respect what will be described later.

Every parachain or relay chain node can be divided into two parts: the STF (or Runtime) and the Client. The latter one implements everything else required to make the protocol work, from storage management to transaction gossiping.

Generally, the runtime is compiled into Wasm and stored as part of the state, this allows making fork-less upgrades because the code transition happens under consensus in the STF.

3.3 Wasm in Polkadot

Substrate is a framework to build blockchains, it is separate from Polkadot, and with it you're able to build so-called Solo Chains, a blockchain able to run on its own that does 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 abstracting 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 manages everything, it compiles the runtime in wasm and implements all the custom logic to make the client and the runtime communicate 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 are built on top of two building blocks: the Parachain Validation Function (PVF) and the Proof of Validity (PoV). [4]

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 your blockchain but to be a parachain you need to provide to the relay chain something that differs a bit from the Substrate-Runtime.

The protocol requires a Parachain Validation Function that is composed of 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 chain: polkadot does not know anything about the 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: PoV, the structure of the PoV will be explained later, important to know that every information needed in the execution of the PVF is indeed in the PoV.

The 'validate_block' function is the glue between the parachain-runtime and the PoV, it accepts the PoV and reconstructs the previous state, applies all the extrinsics using the parachain-runtime and then checks that the new state is consistent with the one proposed by the collators.

Both relay chain and parachain implement the same HostFunctions for the runtime, this means that the STF of the parachain would access the storage through 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,present in the PoV, instead of going into the relay chain client.

Collation is the term used to represent the outcome produced by the collators that will be used by validators, it is the same as PoV.

From the 3.2 you can see how the parachain-runtime interacts in the same way with the embedder but it is different in the Collator or the Validator, in the first case the embedder is the client itself that knows the previous state instead in the Validator the embedder in the client is tweaked a bit so that the previous

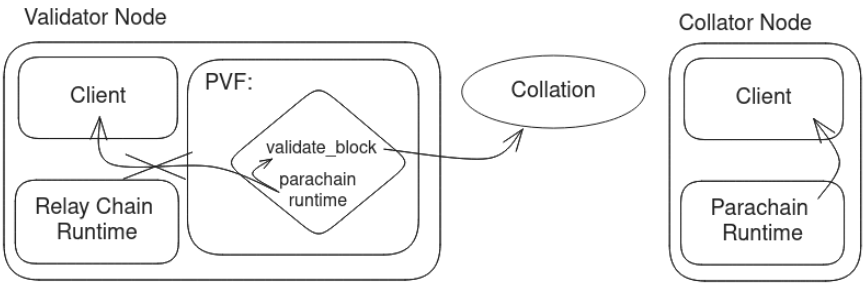

Figure 3.2: PVF and Collation

state is taken from the PoV and not from the Validator.

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

The Proof of Validity is made by all the things needed by Runtime Validation, it is mainly composed by: [1]

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

The witness data is what makes possible the state transition validation without knowing the entire state of the parachain by the relay-chain.

The state in polkadot is represented by a key-value database where each key is unique and there is no assumption on it. The database itself is arranged as a Merkle-Patricia Base 16 Trie that allows to describe the entire state in a single constant state proof. Each block stored in the relay chain contains the state proof of the previous block, and the witness data are then composed by:

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

Instead, the Outgoing messages are everything that needs to be delivered to other parachains, the underneath protocol is XCMP following the XCM format which enables the parachain communication between parachains or with the relay chain.

3.3.3 SmartContracts

SmartContracts are arbitrary codes that can be uploaded on-chain and executed in the state transition function. Even though the STF is arbitrary code that can be updated but the convenience of SmartContracts is the easiness of creating and uploading custom logic on-chain. To make this technology different blockchain came up with different ideas and the one used by Polkadot is to have a recursive embedder.

The Client is the embedder of the STF and the STF becomes also the embedder of the SmartContracs, this is made possible thanks to Wasmi because it can be executed even in wasm itself. To make the execution of arbitrary, possibly malicious code, secure there are different solutions:

- Gas / Fuel measuring
- Storage usage deposit

Where the first one has the aim to limit the number of operations that can be executed in the STF and the second instead is to limit the amount of used storage.

3.4 The crucial Role of WASM in Polkadot

As you may understood from the above sections, the distributed algorithms, the security assumptions, all the cryptography that makes all of this possible would be useless if nodes can't execute in a secure and deterministic manner code that implements all the just described features.

The amount of tools and projects that are using Wasm makes it very suitable for polkadot, an environment with very constrained resources: there is limited space on every block, every state transition must be computed in a limited amount of time and multiple nodes must reach the same results. Wasm has already multiple tools like space and efficiency optimizer that make it even better for polkadot.

An important tools is Binaryen with the wasm-opt optimization phases, largely used in polkadot, that loads WebAssembly and runs Binaryen IR passes on it to make it more space and complexity efficient.

Chapter 4

Rivals

Wasm is not the only PAB used in the blockchain space, other technologies use different solutions involving different protocols and algorithms. Examples are Ethereum with the custom PAB executed by the EVM (Ethereum Virtual Machine) or Solana that used eBPF to implement SmartConctracts, in the following section those solutions and others will be analyzed.

4.1 EVM

Ethereum Virtual Machine code [15] is one of the first PAB used in blockchain, it follows the principles used to describe a perfect PAB, it was created to be a perfect blockchain's bytecode and the Ethereum Virtual Machine is the glue that makes it executable on every machine.

The EVM is the main building block of the Ethereum technology, it executes stack based code and manage all the memory and access to the storage. EVM can be compared to a generic embedder for wasm with many features tied to the measurement of the computation on-chain, called gas.

4.2 eBPF

4.2.1 What is eBPF

eBPF [12] stands for 'Extended Berkeley Packet Filter', what is now eBPF makes the acronym an incorrect representation of it.

Linux brought eBPF into the kernel, enabling sandboxed programs to run inside a privileged context (OS level). For lot of different reasons keeping the

kernel upgraded was a difficult task and eBPF intend to solve this problem.

How can a new feature be developed once and be added to the Linux kernel? Keeping in mind that running arbitrary code developed by whoever in the kernel is absolutely not safe and the same code must be able to run on different architectures?

The operating system guarantee efficiency ans security through a JIT compiler and a verification engine for every eBPF program. To achieve that every kernel contains an eBPF VM able to checks the termination of the program and the security guarantees.

Main points of eBPF to make the verification process possible are:

1. There are no functions in the code, there is only an unique blanket of code
2. Limited control flow
3. Loops need to be statically defined, they are unrolled at compile time
4. The execution can't pass twice on the same code

4.2.2 Solana eBPF

Not every program can compile to eBPF but a distributed systems need to execute arbitrary code and limitation as strict as normal eBPF are too much, Solana [17] then forked the eBPF backend of LLVM and removed lots of constraint, keeping the finality guarantee by the standard gas metering. [13]

Solana also create a new virtual machine for eBPF, , able to check, compile and execute the eBPF code on the blockchain.

eBPF is a perfect PAB for some use cases, as the linux kernel, but it does not seem to be a good fit for blockchains, examples are:

1. limited control flow
2. limited loops
3. 64 bit usage implies lot of checks on the memory access
4. 8 bytes instructions are too long

4.3 RISC-V

RISC-V is a new instruction-set architecture (ISA) [7], even if it is an actual bytecode for a specific hardware and seems to go against what has been described so far as PAB there are running experiments that makes this a valid option as PAB.

There are several reasons why RISC-V could be a good PAB even if it wasn't designed for:

Real ISA suitable for direct native hardware implementation
RISC-V is designed to be executed on real hardware, no assumption is made on the hardware itself, the only required thing is to respect all the constraint specified in the specifications. The ISA is developed following general machine patterns with something completely new but still very related to other machines making really easy the translation.

RISC
The name RISC stands for Reduce Instruction Set Computer, the specifications are then very small and simple making possible the creation of a base executor in really few lines of code.

Completely open ISA
RISC-V is an open standard, this makes possible to everyone to know the behavior and possibly create custom hardware to execute it.

ISA separated into a small base integer ISA
RISC-V is divided into smaller ISA, each one can work alone and can also be composed to achieve different functionalities. One of the most important division is the 32-bit and 64-bit address space division, it makes possible the creation of a sandboxed environment in ad easy and effective way.

4.3.1 RISC-V for SmartConctract

One experiment [6] to port RISC-V in the Polkadot ecosystem as language for SmartConctract is hosted here: polkavm-experiment.

The spec is so easy that creating an interpreter required only one day and the JIT only two days more. Those two executors were then tested against other PABs with different executors, everyone executed the same code: an NES

emulator written in Rust and the interpretation was how close to 60 FPS the code can get.

The results are:

- wasmi: 108ms/frame (9.2 FPS)
- wasmer singlepass: 10.8ms/frame (92 FPS)
- wasmer cranelift: 4.8ms/frame (208 FPS)
- wasmtime: 5.3ms/frame (188 FPS)
- solana_rbpf (interpreted): 6930ms/frame (0.14 FPS)
- solana_rbpf (JIT): 625ms/frame (1.6 FPS)
- RISC-V interpreter: 800ms/frame (1.25 FPS)
- RISC-V JIT: 25ms/frame (40 FPS)

As you can see the simplicity of RISC-V made possible the creation of a JIT compiler able to be faster than lot of competitive compiler already being used in different realities.

An important difference rather than eBPF is dedicated instructions for syscall (used as Host Function in the SmartContract) and the support in rustc and LLVM. One of the main constraints is the number of registers used in RISC-V, 32 registers while most architectures use 16, the compiler or interpreter then must properly manage those extra registers.

Chapter 5

Conclusions

We have seen how PABs are important for a decentralized network, in particular for blockchains. Wasm is the main PAB used in Polkadot but other blockchains are using different solutions because wasm was not invented for this specific purpose and it has some problems and some behavior that needs to be adapted to a decentralized network.

Inventing something from scratch could satisfy every needs of a blockchain but with the cons of creating and managing an entire bytecode only for this nice.

The best approach seems to be something along RISC-V, an Open Standard that can be suitable for multiple environments while also being adaptable and composable. Those characteristics offer the possibilities of modeling the bytecode to almost satisfy every niche and exploit every tool or contribution by other environments that are relying on the same technology.

Bibliography

[1] Cumulus docs. https://github.com/paritytech/cumulus/blob/master/docs/overview.md.

[2] Llvm language reference manual. https://llvm.org/docs/LangRef.html#abstract.

[3] Memory in webassembly. https://hacks.mozilla.org/2017/07/memory-in-webassembly-and-why-its-safer-than-you-think/.

[4] Parachains' protocol overview. https://wiki.polkadot.network/docs/learn-parachains-protocol.

[5] Polkadot wiki - wasm. https://wiki.polkadot.network/docs/learn-wasm.

[6] Risc-v as smartcontracts language. https://forum.polkadot.network/t/exploring-alternatives-to-wasm-for-smart-contracts/2434.

[7] Risc-v spec, volume i. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[8] Wasm github specifications. https://webassembly.github.io/spec/core/.

[9] Wasm test suite. https://github.com/WebAssembly/spec/tree/main/test/core.

[10] Wasmi. https://github.com/paritytech/Wasmi.

[11] Wasmtime book. https://docs.wasmtime.dev/.

[12] What is ebpf. https://ebpf.io/what-is-ebpf/.

[13] What is ebpf. `https://forum.polkadot.network/t/ebpf-contracts-hackathon/1084`.

[14] Jeff Burdges, Alfonso Cevallos, Peter Czaban, Rob Habermeier, Syed Hosseini, Fabio Lama, Handan Kilinc Alper, Ximin Luo, Fatemeh Shirazi, Alistair Stewart, et al. Overview of polkadot and its design considerations. *arXiv preprint arXiv:2005.13456*, 2020.

[15] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[16] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White paper*, 21(2327):4662, 2016.

[17] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.