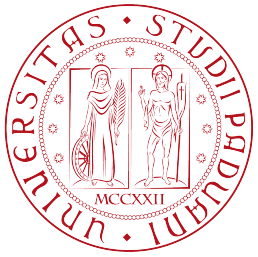

Università degli Studi di Padova

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'(polkadot-overview paper).

Contents

0.1 Platform-Agnostic Bytecode 1
0.1.1 Definition 1
0.1.2 Execution 1
0.1.3 Key features 1
0.1.4 Current usage 3
0.1.5 PAB in blockchains 3
0.2 WASM 5
0.2.1 Definitions 5
0.2.2 Specifications 6
0.2.3 Execution 7
0.2.4 Security guarantee 10
0.3 Polkadot 13
0.3.1 What's Polkadot? 13
0.3.2 Polkadot's protocol 13
0.3.3 WASM in Polkadot 15
0.3.4 The crucial Role of WASM in Polkadot 17
0.4 WASM Rivals 19
0.4.1 EVM 19
0.4.2 eBPF 19
0.4.3 RISC-V 20

0.1 Platform-Agnostic Bytecode

0.1.1 Definition

A Platform-Agnostic Bytecode (PAB) is 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.

0.1.2 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.

0.1.3 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 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 the relations require only a little overhead for the execution on different hardware.

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. Implemening a proper sandboxed environment is embedder dependent but a PAB can be more or less suitable for this feature.

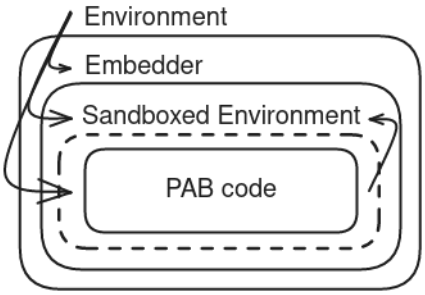

Figure 1: Sandboxing graphic example

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 (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.

0.1.4 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

  into the kernel, enabling arbitrary programs to be executed in a privileged context (OS level)
- 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. [**?**]
- WebAssembly is a safe, portable, low-level code format designed for efficient execution and compact representation. [**?**]

0.1.5 PAB in blockchains

Blockchains are distributed systems that needs to agree on the execution of arbitrary code performed on different machines. The code execution must conclude to the same result, regardless of the machine the code is running on. What has just been described is called 'deterministic execution'. PABs are the solution for both problems.

0.2 WASM

0.2.1 Definitions

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

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

'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 complication 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, applications are able to accept, execute and make the wasm code to interact with the environment, allowing different High-Level languages to work together having wasm as minimum common divisor.

Wasm main design goals, as specified in the wasm specification introduction [**?**], 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
- **Language-independent** There's no strong influence by other binary formats, languages or programming models
- **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 ëxecutorör ëmbedderẗhave the same meaning.

0.2.2 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 [**?**].

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

0.2.3 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. Wasm can be executed in different ways. The main used in blockchains 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 articulated test suite to check the correctness of the embedder. [?]

The common divisor for the first three types of executions is the transpilation of a stack-based bytecode to a register 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.

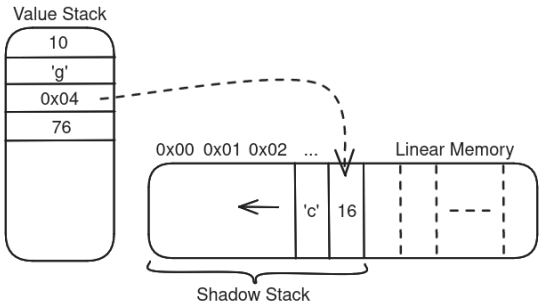

Figure 2: Value and Shadow Stack

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

AOT

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

JIT

JIT is a dynamic compilation

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 that given an input calls function A or B. The JIT will then understand this structure and compile the entry point and only one function between A or B based on the initial input.

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 final bytecode not caring too much about adding optimizations.

Wasmtime is specialized in this type of executions and it makes it really efficient while keeping everything secure.

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 methods here the objective is not to create efficient final code but to create the final bytecode as fast as possible.

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

Interpret

wasmi-spec

Interpretation is the easiest way to execute wasm, which 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. [**?**]

Currently the first wasm bytecode pass produce another stack based bytecode, called WASMI IR, and then this bytecode is interpreted by a Virtual Machine. Even with this transpilation it is only 5 time slower then the compilation to native bytecode of the architecture.

0.2.4 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.

Linear Memory

(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. [**?**]

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 ponear Memory 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.

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

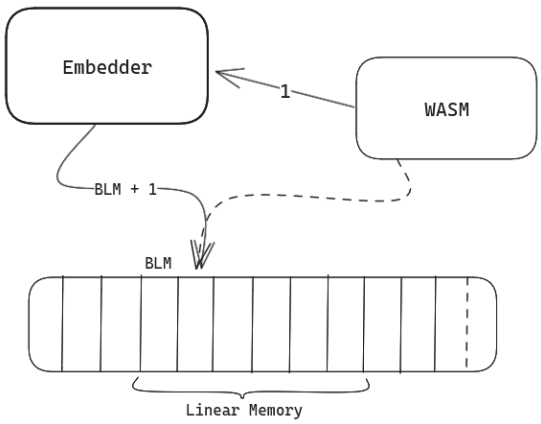

Figure 3: BLM: Base Linear Memory Pointer

Communication in a sandboxed environment

We just described how wasm provides no ambient access to the computing environment in which the code is executed [**?**], 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 [**?**]. 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.

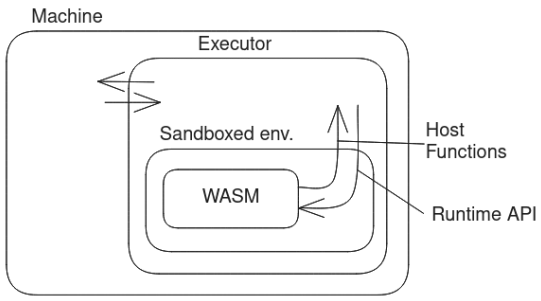

Figure 4: Environment communication

Wasmtime Security guarantee

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

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. [**?**]

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

0.3 Polkadot

0.3.1 What's Polkadot?

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

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. [**?**]

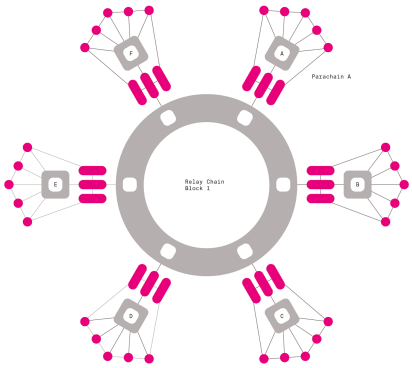

Figure 5: Polkadot's Architecture

Polkadot provides the bedrock relay-chain upon which a large number of validatable, globally-coherent dynamic data-structures may be hosted side-by-side. [**?**]

In the Figure 5 you can see in the middle the relay chain that connects multiple parachains.

0.3.2 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) [**?**]. 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: [**?**]

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 (TODO: maybe explain sharding?)
    - Submit votes to resolve forks and have a single head
    - Manage messages between parachians

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. [**?**]

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 in two parts: the STF (or Runtime) and the Client, it implements everything else required to make the protocol to work, from storage management to transactions gossiping.

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

0.3.3 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 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). [**?**]

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 something that differs a bit form the Substrate-Runtime.

The protocol requires a Parachain Validation Function 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 chain: 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: 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 accept the PoV and reconstruct the previous state, applies all the extrinsics using the parachain-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.

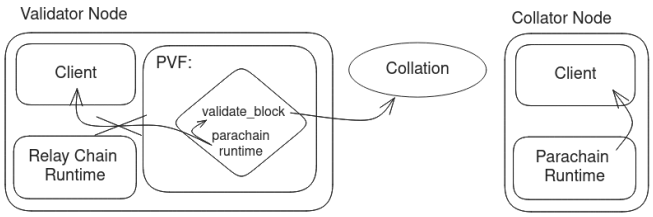

Figure 6: PVF and Collation

From the 6 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 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.

PoV

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 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 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 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.

SmartContracts

SmartContracts are arbitrary code that can be uploaded on chain and be executed in the state transition function. Even the STF is arbitrary code that can be updated but the convenience of SmartContracts are the easiness of create and upload 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, a wasm environment able to 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.

0.3.4 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.

0.4 WASM 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.

0.4.1 EVM

Ethereum Virtual Machine code [?] is one of the first PAB used in blockchain, it does not follow the features 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, follows therefore the same principles of an embedder for wasm with many features tied to the measurement of the computation on-chain, called gas.

0.4.2 eBPF

What is eBPF

eBPF 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 problem 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

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 [**?**] then forked the eBPF backend of LLVM and removed lots of constraint, keeping the finality guarantee by the standard gas metering. [**?**]

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

0.4.3 RISC-V

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

There are several reasons why RISC-V could be a good PAB even if it was designed to be an ISA:

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.

RISC-V for SmartConctract

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

The spec are so easy that creating an interpreter required only one day and the JIT only two days more. Those two executor were then tested against other PAB with different executors, every one executed the same code: a 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 then eBPF are dedicated instructions for host functions (syscall) and the support in rustc and LLVM. One of the main constraint is the number of register used in RISC-V, 32 registers while most architectures use 16, the compiler or interpreter then must properly manage those extra registers.