# VM Matters: A Comparison of WASM VMs and EVMs in the Performance of Blockchain Smart Contracts

YIXUAN ZHANG, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education and School of Computer Science, Peking University, Beijing, China
SHUYU ZHENG, Peking University, Beijing, China
HAOYU WANG, Huazhong University of Science and Technology, Wuhan, China
LEI WU, Zhejiang University, Hangzhou, China
GANG HUANG, School of Computer Science, Peking University and National Key Laboratory of Data Space Technology and System, Beijing, China
XUANZHE LIU, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education and School of Computer Science, Peking University, Beijing, China

Beyond an emerging popular web applications runtime supported in almost all commodity browsers, WebAssembly (WASM) is further regarded to be the next-generation execution environment for blockchain-based applications. Indeed, many popular blockchain platforms such as EOSIO and NEAR have adopted WASM-based execution engines. Most recently, WASM has been favored by Ethereum, the largest smart contract platform, to replace the state-of-the-art EVM. However, whether and how well current WASM outperforms EVM on blockchain clients is still unknown. This article conducts the first measurement study to understand the performance on WASM VMs and EVM for executing smart contracts for blockchain-based applications. To our surprise, the current WASM VM does not provide expected satisfactory performance. The overhead introduced by WASM is really non-trivial. Our results shed the light on challenges when deploying WASM in practice, and provide insightful implications for improvement space.

CCS Concepts: • **General and reference** → **Measurement**; • **Software and its engineering** → **Software development process management;**

Additional Key Words and Phrases: WebAssembly, blockchain, and smart contract

**ACM Reference Format:**
Yixuan Zhang, Shuyu Zheng, Haoyu Wang, Lei Wu, Gang Huang, and Xuanzhe Liu. 2024. VM Matters: A Comparison of WASM VMs and EVMs in the Performance of Blockchain Smart Contracts. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 9, 2, Article 5 (March 2024), 24 pages. https://doi.org/10.1145/3641103

This work was supported by the National Natural Science Foundation of China (grant No.62325201), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), the Knowledge Innovation Program of Wuhan-Basic Research, and Center for Data Space Technology and System, Peking University.
Authors' addresses: Y. Zhang and X. Liu, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education and School of Computer Science, Peking University, No. 5 Yiheyuan Road, Beijing, China, 100091; e-mails: {zhangyixuan.6290, liuxuanzhe}@pku.edu.cn; S. Zheng, Peking University, No. 5 Yiheyuan Road, Beijing, China, 100091; e-mail: zhengshuyu@pku.edu.cn; H. Wang (Corresponding author), Huazhong University of Science and Technology, No.1037 Luoyu Road, Wuhan, Hubei, China, 430074; e-mail: haoyuwang@hust.edu.cn; L. Wu, Zhejiang University, No. 866 Yuhangtang Road, Hangzhou, Zhejiang, China, 310058; e-mail: lei_wu@zju.edu.cn; G. Huang, School of Computer Science, Peking University and National Key Laboratory Data Space Technology and System, No. 5 Yiheyuan Road, Beijing, China, 100091; e-mail: hg@pku.edu.cn.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2376-3639/2024/03-ART5
https://doi.org/10.1145/3641103

## 1 INTRODUCTION

Blockchain, as a trustable distributed ledger, has been proliferating rapidly over the last few years. Bitcoin system [49] has demonstrated that it is feasible to use the Internet to construct a decentralized value-transfer system that can be shared across the world and virtually free to use. Due to the performance and scalability issues, however, it is difficult for Bitcoin to support large applications. Thus, Ethereum [8] was proposed to allow users to develop smart contracts and create DApps (i.e., Decentralized Applications) that support advanced functionalities. Smart contract is designed as a computer protocol that allows users to digitally negotiate an agreement in a convenient and secure way.

The expansion of blockchain networks pushes forward higher requirements for scalability, security, and efficiency. Dinh et al. [39] found that *consensus protocol*, *transaction signing*, and *execution engine* are the bottlenecks of Ethereum. Unlike the transaction signing,[1] the other two are fundamental issues that are effective under any circumstances. Not surprisingly, the current consensus protocol (i.e., proof-of-work [18]) is the main bottleneck. To address this issue, the Ethereum community has proposed to replace it with a new alternative (i.e., proof-of-stake [17]). By that time, the execution engine will become the main bottleneck of the entire system. Actually, in the consortium blockchains, the execution engine of smart contract has already been the major bottleneck.

The execution engine in the current Ethereum platform is the so-called **Ethereum Virtual Machine** (**EVM**). Ethereum smart contracts are currently written in the Solidity/Vyper and compiled down into EVM bytecode which is executed by every Ethereum node. As the smart contracts become more functional, the need to speed up EVM execution has become more urgent. However, the tool and language support for writing smart contracts and compiling to EVM bytecode are not well-equipped for real-world applications [2].

The developers of blockchain platforms turned their eyes to **WebAssembly** (**WASM**) [28]. WASM is a new, low-level, binary format, and runtime for the Web applications, and has now been implemented in all major web browsers such as Chrome, Firefox, and so on [28, 40, 43]. The WASM bytecode is designed to serve as a compilation target and has been supported by a wide range of modern programming languages [4].

The Ethereum community has placed **Ethereum flavored WebAssembly** (**eWASM**) on the Ethereum 2.0 roadmap as a replacement to the first generation of EVM. The idea seems quite promising. First, WASM is supported by a rich variety of programming languages, blockchain developers can benefit from the WASM ecosystem, rather than relying on Solidity/Vyper only. Secondly, more transactions can be put into a block if eWASM executes faster [2], which may improve the scalability of Ethereum network. Till now, major Ethereum clients such as Go-ethereum [11] and Openethereum [15] have implemented experimental WASM supports. It is worth noting that, Ethereum is not the first to use **WASM virtual machine** (**WASM VM**) as the contract execution engine. Many popular blockchain platforms, such as EOSIO [6] and NEAR [14], are using execution engines based on WASM. Some other ones, like Tron [22] and Perlin [13], are also on the way to develop their own WASM VMs.

While WASM is generally expected to improve the performance of blockchain execution engines [2], there has been no clear evidence to support this statement. *Whether WASM bytecode really outperforms EVM bytecode on blockchain clients is still unknown.* This is non-trivial in practice as the performance of VM can definitely impact the scalability of the whole blockchain applications. To the best of our knowledge, neither the research community nor the industry has investigated the potential performance improvement introduced by WASM VMs in a detailed manner.

[1] It just affects a concrete implementation [39].

**This Work.** In this article, we take the first step to measure the impact of smart contract engines on the performance of smart contract execution. To be specific, we focus on two types of VM engines, *EVM engines,* and *WASM engines*, corresponding to the execution of EVM bytecode and WASM bytecode, respectively. Our study covers almost all the popular EVM engines and WASM engines (see Section 3), including two most widely used client-based EVM engines (`Go-Ethereum` and `Openethereum`), and 10 state-of-the-art WASM engines (including three engines specialized for blockchain, i.e., EOS VM engine, and WASM-version engines provided by `Go-Ethereum` and `Openethereum`). To pinpoint the root cause leading to the performance bottleneck, we provide a comprehensive study that includes both *intra-bytecode* engine comparison study and *inter-bytecode* engine comparison study (see Section 4). For the intra-bytecode engine study, we compare the contract execution performance *within EVM engines* and *within WASM engines,* respectively, which can provide insights on the performance overhead (bottleneck) caused in the design of some specific engines. For the inter-bytecode engine study, we compare the contract execution performance *across the VM engines that support different types of bytecode.* Specifically, we compare the EVM-version and WASM-version engines of `Go-Ethereum` and `Openethereum`, as they are by far the most popular blockchain clients that support these two types of engines. Considering that no benchmarks are available in the research community, we further make an effort to design a set of benchmarks with both WASM bytecode and EVM bytecode formats for evaluation, covering the most representative operations in smart contracts. Among many interesting or even surprising results and observations, the following are prominent:

— *The support for WASM on Ethereum-based blockchain clients is far from satisfactory.* We observe that a number of emerging issues: (1) existing WASM support is unstable and much effort is needed to adapt WASM engines; (2) it is almost impossible to implement eWASM smart contracts in a general form as the coding convention varies among clients; and (3) few languages are supported by existing tool chains, leading to the cumbersome and error-prone development process.
— *For the two most popular Ethereum EVM engines, the performance of contract execution on Geth is better than that on Openethereum for most cases in our benchmarks.* We notice that the difference in opcode-level implementation of EVM engines can significantly impact the overall performance.
— *The performance of contract execution across standalone WASM engines differs greatly, even up to three orders of magnitude.* While WASM bytecode is generally expected to achieve excellent performance, the implementation of WASM engines can greatly impact (even impede) the performance.
— *The native supported data type has a substantial impact on the performance of WASM contract execution.* As the native data type of WASM is 32/64 bit, it can introduce additional overhead during the execution of 256-bit contracts (i.e., the native data type of EVM bytecode).
— *While standalone WASM engines are faster than EVM engines, the eWASM VMs are slower than EVMs for all of the 256-bit benchmarks and most of the 64-bit benchmarks on Geth and Openethereum clients.* The overhead is mainly introduced by gas metering before the execution of WASM bytecode and the inefficient context switch of **Ethereum Environment Interface** (**EEI**) methods.

To the best of our knowledge, this is the first measurement study to date that investigates the performance of smart contract execution across a number of EVM and WASM engines. Our major observations, however, contradict to the general expectation that WASM engines will improve the contract execution performance. The overhead introduced by gas metering and WASM VM embedding overwhelms the performance gain of WASM, thus, the WASM engines run slower than

EVM engines on most of the benchmarks in blockchain clients Geth and Openethereum. There is a need for optimizing the execution process of WASM on the blockchain client and improving the interface design between WASM and the blockchain environment to reduce additional overhead. Our efforts highlight the practical challenges of applying WASM engines in executing smart contracts and promote better operational practice. As an additional contribution, we have made the code and benchmarks available for public access [31].

## 2 BACKGROUND

We start by briefly presenting key concepts, including smart contract, and EVM/WASM/eWASM bytecode.

### 2.1 Smart Contract

A smart contract is designed as a computer protocol that allows users to digitally negotiate an agreement in a convenient and secure way. In the context of blockchain, a smart contract stands for complex applications having digital assets being directly controlled by a piece of code [8]. Ethereum acts as the first blockchain system that supports developing smart contract. In Ethereum, smart contracts are typically written in higher level languages, e.g., Solidity and Vyper, and then compiled to EVM bytecode. The EVM bytecode will be executed in a VM embedded inside of the Ethereum execution environment. Note that smart contracts in some blockchains do not necessarily follow the EVM bytecode format. As previously stated, EOSIO, as the first blockchain that accepts **Delegated Proof-of-Stake** (**DPoS**) consensus protocol, adopts C++ as the official language in developing smart contracts. In particular, the source code of smart contract is first compiled into WASM bytecode. Upon invocation, the bytecode will be executed in EOSIO's WASM VM. A number of popular blockchains including Ethereum and Tron, have claimed to support WASM bytecode in their next generation VM engines.

### 2.2 Virtual Machine

VMs (Virtual Machines) play a crucial role in executing smart contracts on blockchain for several reasons [8]. First, VMs provide a sandboxed execution environment that protects the code and state of the contract from malicious attacks and unauthorized access. This isolation ensures the security and integrity of the contract. Second, VMs abstract the underlying hardware and operating system, enabling consistent execution of smart contracts across different nodes in the blockchain network. This determinism is essential for achieving consensus and maintaining the reliability of the blockchain. Third, VMs offer a standardized execution environment, allowing smart contracts to be deployed and executed seamlessly on different blockchain platforms. This portability facilitates interoperability, enabling contracts written for one blockchain to be executed on another compatible VM with minimal modifications. Lastly, VMs support high-level programming languages, making it easier for developers to write smart contracts in familiar languages rather than low-level bytecode. This simplifies the development process and attracts a broader range of developers to participate in the blockchain ecosystem. In summary, VMs provide a secure and consistent execution environment, enable portability and interoperability, and support high-level languages, all contributing to the successful execution of smart contracts on the blockchain.

However, there are limitations and requirements associated with VMs in the context of blockchain. Firstly, VMs introduce an additional layer of abstraction, which can impact the performance of smart contract execution. It is crucial to optimize the execution speed of VMs to minimize overhead and maximize efficiency. This becomes especially crucial as blockchain networks aim for scalability and high transaction throughput. Secondly, since many blockchains utilize gas

as a measure of computational resources consumed by smart contracts, VMs must support gas metering mechanisms. These mechanisms ensure fair resource allocation and prevent abuse. The design of VMs should consider gas limitations and effectively manage gas usage during contract execution. Thirdly, VMs should be designed with backward compatibility in mind. This means that older versions of smart contracts should still be executable even as the VM evolves. By addressing these limitations and fulfilling the requirements, VMs can effectively support the execution of smart contracts in a secure, consistent, and portable manner. This, in turn, contributes to the growth and advancement of blockchain technology.

### 2.3 EVM/WASM/eWASM Bytecode

We next introduce the similarities and differences in the design and implementation of EVM, WASM, and eWASM bytecode.

*2.3.1 EVM Bytecode.* The Ethereum smart contract is compiled to a low-level, stack-based bytecode language called *EVM bytecode*. An EVM bytecode consists of a series of operations. There are three types of space to store data, including stack, memory and long-term storage. The stack and the memory will be reset after computation ends, while the data in storage persists for the long term. The EVM has many high-level operations, such as `SSTORE` (writes a 256-bit integer to storage), `CREATE` (creates a child contract) and `CALL`(calls a method in another contract), `SHA3` (compute Keccak-256 hash). The native data type of EVM is 256-bit integer. EVM supports down to 8-bit integer calculation, but computations smaller than 256-bit must be converted to a 256-bit format before the EVM can process them. To ensure determinism, EVM does not fully support floating point. Floating point can be declared in smart contracts, but cannot be assigned to or be used in the calculation.

*2.3.2 WASM.* WASM is a binary instruction format for a stack-based virtual machine that features near-native execution performance. To ensure security, WASM bytecode is executed within a sandboxed environment isolated from the host runtime that enforces the security policies for permissions [28]. A growing number of programming languages [4, 51] including C/C++/C#/Java/Go/Lua, can be compiled to WASM bytecode. The execution of WASM bytecode is done within the embedding environment on a virtual machine. The WASM abstract machine includes a stack recording operand values and control constructs, and an abstract store containing the global state. WASM manipulates value of the 4 basic value types: integers and floating-point data with both 32-bit and 64-bit support, respectively.

*2.3.3 eWASM.* eWASM [10] stands for Ethereum flavored WebAssembly. It is a restricted subset of WASM to be used for contracts in Ethereum. Since WASM does not support high-level opcodes, interactions with the Ethereum environment are defined through EEI implemented by WASM execution engines. eWASM restricts non-deterministic behavior such as floating point, because every node in the blockchain has to run with complete accuracy. The metering of computation cost in eWASM is similar to that of EVM. The WASM execution engine sums up the costs on the execution of each operation. The eWASM project also defines an EVM transcompiler to achieve backwards compatibility with the EVM bytecode on the current blockchain.

## 3 STUDY DESIGN

In this section, we present the details of our measurement study.

### 3.1 Research Questions

We seek to focus on the following **research questions** (**RQs**):

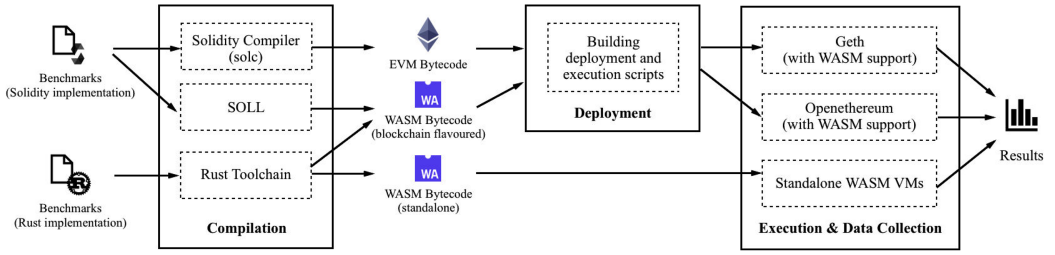


Fig. 1. An overview of our measurement study.

RQ1 **WASM supports of Blockchain Clients.** Considering that major Ethereum clients have claimed to support WASM in order to embrace EVM 2.0, it is thus interesting to investigate *how well do existing Ethereum clients support WASM bytecode?*

RQ2 **A Comparison of EVM Engines.** As there are several clients that support the execution of EVM bytecode, we are wondering *are there any performance gaps of running smart contracts among them?* It can provide insights on analyzing the performance overhead caused by the design of specific engines.

RQ3 **A Comparison of WASM Engines.** Similar to RQ1, there are a number of WASM engines available, including both client-based blockchain engines and standalone engines. We seek to explore *to what extent does the performance gap exist among the WASM engines?*

RQ4 **WASM Engines vs. EVM Engines** It is widely acknowledged that WASM bytecode has the advantage of execution performance. Thus, it is interesting to investigate *whether WASM bytecode outperforms EVM bytecode on existing blockchain clients? If so, how large is the performance gap can we expect? If not, what are the major reasons?*

### 3.2 Overview

Figure 1 shows an overview of our measurement study. First, we need to create a comprehensive benchmark that provides both EVM bytecode and its equivalent WASM bytecode to enforce fair comparison. To achieve this purpose, for a smart contract written in Solidity, we take advantage of Solidity Compiler solc (v0.5.4) to compile it to EVM bytecode. We further utilize SOLL [19], an emerging compiler for generating eWASM bytecode from Solidity. However, the eWASM bytecode cannot work directly on Openethereum and other general WASM engines.The WASM support on Openethereum is now usable but unstable as well.Geth does not support WASM natively. Thus, for a given solidity smart contract, we have to implement an equivalent contract using WASM-supported high level languages, and then compile it to WASM bytecode, to evaluate its execution performance on Openethereum and other standalone WASM engines. We manually transform the entry point, and export functions and library functions in WASM-supported high level languages to meet the Openethereum WASM coding convention. In this article, we make effort to implement the corresponding benchmark samples in Rust, and take advantage of Rust toolchain (nightly-2018-11-12) to get the WASM bytecode. Note that, all the manually generated samples were double checked by the first two authors of this article to ensure they were correctly implemented and are equivalent to the original smart contracts, which can generate identical results based on extensive test cases. To the best of our knowledge, this is the only feasible way for us to create comprehensive benchmarks for enabling comparison.

Next, we build the deployment and execution scripts for the compiled EVM bytecode or eWASM bytecode. The deployment scripts are executed once to deploy the smart contracts on the blockchain. The execution scripts call the deployed smart contract. For constant functions in the smart contracts, the execution scripts call them without creating a transaction on the blockchain.

Non-constant functions in the smart contracts will be invoked by transactions and then committed to the blockchain after mining. Constant functions and non-constant functions are different types of functions in smart contracts. Constant functions do not modify the state of the smart contracts. They are read-only functions that retrieve data from the contract's storage or perform calculations based on the existing data. Constant functions are used to query information from the contract without making any changes. They are executed without creating a transaction on the blockchain. Non-constant functions are transactional functions. These functions can modify the state of the smart contracts. They perform actions such as updating values, transferring funds, or interacting with other contracts. Non-constant functions require a transaction to be created and included in a block through mining. The changes made by non-constant functions are recorded on the blockchain and become permanently visible to all participants. The scripts are running on a console connected to an Ethereum private chain through an IPC pipe.

After that, we measure the execution time of EVM bytecode and WASM bytecode by instrumenting the blockchain clients. Specifically, we insert a time measuring code snippet before and after the contract execution loop on Geth and Openethereum. Besides, we also instrument the standalone WASM VMs before the decoding phase and after the execution phase. Thus, the time-consuming impact of network latency, consensus protocols, and scalability has yet to be excluded from our experiment. Our work's scope is to analyze the performance of the VMs of executing smart contracts rather than network latency, consensus protocols, and scalability. In the decoding phase of standalone WASM VMs, the WASM binaries are parsed and decoded to extract the necessary information for execution. The decoder analyzes the structure of the binaries, including sections, function bodies, and instructions. It translates the encoded binaries into an internal representation that the VM can understand and process during execution. The decoding phase typically involves reading and interpreting the binaries, identifying instructions and operands, and resolving necessary references. In the execution phase, the standalone WASM VMs load and manipulate the data, perform instructions, control program flow, and interact with the runtime environment as specified in the binaries' instructions. Finally, we conduct the execution experiments on Geth, Openethereum, and standalone WASM VMs and collect the execution time.

### 3.3 Selected Engines

Our study seeks to cover all the most popular EVM and WASM engines. To be specific, we select two Ethereum clients with WASM support (i.e., **Go-Ethereum** [11] and **Openethereum** [15]) and ten most popular WASM VMs on Github (i.e., **Geth WASM - Hera** [12], **EOS VM** [5], **Life** [13], **SSVM** [21], **wagon** [24], **wabt** [23], **wasm3** [25], **WAMR** [29], and **wasmtime** [27]) to perform comparison.

*3.3.1 Selected EVMs.* Two most popular Ethereum clients with WASM support are selected. **Go-Ethereum** [11] is the official Golang implementation of the Ethereum protocol. It is one of the most popular Ethereum implementations available either as a standalone client called ***Geth***, or as a library. We use the Geth client to deploy and execute the compiled smart contracts. The experiments are performed on Geth version 1.9.1 with Go version 1.14.1.

**Openethereum** [15] is another implementation of Ethereum client. Openethereum was once known as the "Parity Ethereum". In 2019, the ownership of Parity codebase and maintenance was transitioned and Parity was renamed as Openethereum. Openethereum is implemented using Rust programming language and is claimed to be the fastest, lightest, and most secure Ethereum client. We use Openethereum version v3.0.0 with Rust version 1.22.1 in this article.

*3.3.2 Selected WASM VMs.* The selected 10 popular WASM VMs are introduced in the following. These WASM VMs are the most widely used.

**Geth WASM - Hera** [12]. Geth requires EVMC support to execute WASM bytecode. The EVMC is the low-level ABI between VMs and Ethereum clients. On the VM side, it supports classic EVM and eWASM. On the client-side it defines the interface for EVM implementations to access Ethereum environment and state. Hera is the official eWASM virtual machine connector. It is compiled as a shared library and can be dynamically loaded by Geth as its EVMC engine. Hera now supports three WASM backends: wabt (by default) [23], Binaryen [3], and WAVM [30]. Experiments are done using Hera v0.2.6 on a Geth client with EVMC v6.3.0.

**Openethereum WASM - wasmi** [26]. Openethereum currently supports two VM types—EVM and WASM. An execution engine named `wasmi` is integrated as a component in Openethereum, which is a WASM interpreter implemented in Rust. It does not support work-in-progress WASM proposals and features that are not directly supported by the specification to guarantee correctness. In the experiment, we use `wasmi` v0.6.2.

**EOS VM** [5]. EOS VM is designed to meet the needs of EOSIO blockchain. The execution of EOS VM is deterministic and EOS VM uses a software implementation of float point arithmetic to ensure determinism. In the experiment, we use EOS v2.0.5.

**Life** [13]. Life is a secure and fast WASM VM built for decentralized applications, written in Go. In the experiment, we use Life on commit 05c0e0f.

**SSVM** [21]. SSVM is a high performance and enterprise-ready WASM VM for cloud, AI, and Blockchain applications. We use SSVM v0.6.0 (interpreter mode of general wasm runtime) in the experiment.

**wagon** [24]. wagon is a WASM-based Go interpreter. It provides executables and libraries that could be embedded in Jupyter or any Go program. In the experiment, we use wagon v0.4.0.

**wabt** [23]. wabt is a suite of tools for WASM including translation between WASM text format and WASM binary format, decompilation, objdump, stack-based interpreter, and so on. It now supports 11 WASM proposals. In the experiment, we use wabt v1.0.16.

**wasm3** [25]. wasm3 is a high performance WASM interpreter written in C. It claims to have the fastest execution speed and the minimum useful system requirements. wasm3 runs on a wide range of architectures and platforms including PCs, browsers, mobile phones, routers, and self-hosting servers. In the experiment, we use wasm3 v0.4.7.

**WAMR** [29]. **WebAssembly Micro Runtime** (**WAMR**) includes a VM core, an application framework and dynamic management of the WASM applications. WAMR supports WASM interpreter, **ahead of time** (**AoT**) compilation and **Just-in-Time** (**JIT**) compilation. We use the interpreter mode (WAMR-interp) and JIT mode (WAMR-jit) in the experiment. In the experiment, we use WAMR-04-15-2020.

**wasmtime** [27]. wasmtime is a lightweight standalone runtime for WASM. It is built on the optimizing Cranelift code generator to quickly generate high-quality machine code at runtime. In the experiment, we use wasmtime v0.15.0.

### 3.4 Benchmarks

We have compiled two benchmark datasets, including our manually curated benchmark (i.e., implemented in Rust and compiled to WASM bytecode) and the real-world benchmark contracts (i.e., Solidity benchmarks and their eWASM versions transformed by SOLL). Tables 1 and 2 list the details of the benchmark contracts.

*3.4.1 Dataset-1.* It includes 13 different kinds of smart contracts, which can be grouped into four categories according to their functionalities: *simple operation*, *arithmetic*, *block status,* and *hashing*. To better investigate the performance difference of EVM bytecode and WASM bytecode, we adopt an opcode-level measurement by designing group *simple operation*, in which the

Table 1. An Overview of the Artificial Benchmarks (Dataset-1) Used in Our Measurement Study

| | Contract Name | Description | Language | Has 64-bit Counterpart |
|---|---|---|---|---|
| Simple Operation | add | Do ADD operation $x$ times | Solidity and Rust | Y |
| | sub | Do SUB operation $x$ times | | |
| | shr | Do SHR operation $x$ times | | |
| | div | Do DIV operation $x$ times | | |
| | mod | Do MOD operation $x$ times | | |
| | power | Do MUL operation $x$ times | | |
| Arithmetic | fib | Calculate the $x$th term of the Fibonacci sequence | Solidity and Rust | Y |
| | matrix | Do matrix multiplication $x$ times | | |
| | warshall | Find shortest paths in a graph of $x$ nodes using Floyd-Warshall algorithm | | |
| Block Status | builtin | Read block status | Solidity | N |
| Hashing | keccak256 | The Ethereum hashing function, keccak256 | Solidity and Rust | N |
| | blake2b | Cryptographic hash function blake2b, producing a 256-bit hash value, optimized with inline assembly | | |
| | sha1 | Cryptographic hash function SHA-1, producing a 160-bit hash value, optimized with inline assembly | | |

Table 2. An Overview of the **Real-World Benchmarks** (Dataset-2) Used in Our Measurement Study

| Category | Description | #Contracts |
|---|---|---|
| Save Math | Do arithmetic operations with overflow detection | 2 |
| String Operation | Return strings according to the value of input | 4 |
| Save Data | Save data in memory or long-term storage | 5 |
| ERC20Token | Ethereum ERC-20 token smart contracts with basic functionalities such as approval and transfer | 18 |
| Get Context | Read the transaction context such as message sender and receiver | 2 |
| Others | Do nothing or revert | 2 |
| **Total** | | **33** |

benchmarks differ only in key opcodes. The Solidity version of benchmarks in *arithmetic*, *block status* and *hashing* are from the most widely used blockchain test sets: Ethereum Consensus Tests [7] and Blockbench [39]. We exclude the benchmarks that invoke external calls to other contracts and benchmarks that cannot be compiled correctly to eWASM by SOLL compiler. The benchmarks contain only constant functions, so the execution of the benchmarks do not create transactions on the blockchain. As previously stated, we further implement the Rust version of the corresponding benchmarks.

Benchmarks in group *simple operation* execute a simple operation $x$ times, where $x$ is the input value, originally set to be 1,000,000. Benchmarks in group *arithmetic* do numeric computations more complicated than those in group *simple operation*, such as Fibonacci sequence calculation and matrix multiplication. The input value $x$ is set to be 10,000 for benchmark *fib*, and 100 for benchmark *matrix* and *warshall* due to the stack limit of eWASM execution engines. The benchmark in group *block status* reads the status of current block. This benchmark cannot be executed

in standalone WASM VMs, so it does not have Rust implementation. Benchmarks in group *hashing* compute hash values using different hashing algorithms. The *keccak256* invokes EVM instruction SHA3 to calculate hash values. Benchmark *black2b* and *sha1* implement the hashing algorithm manually and optimize the code with inline assembly [16, 20]. Since EVM and eWASM do not support floating point, the benchmarks are implemented in 256-bit integer and 64-integer. Benchmarks in group *block status* and *hashing* only have 256-bit version because the calculation or the results cannot be fit into 64-bit integers.

*3.4.2 Dataset-2.* We further collect a set of real-world Solidity smart contracts with verified source code from Etherscan [9]. Etherscan is the biggest analytics platform for Ethereum and has been widely used. We take advantage of SOLL to generate their eWASM counterparts. Note that the generated eWASM smart contracts can only be executed on the Ethereum Geth client (as the tool SOLL is designed for generating eWASM contracts specifically), thus benchmarks in Dataset-2 will be only used in our evaluation of RQ2 and RQ4 to measure the performance gap in real-world smart contracts. We have manually checked all the new submitted real-world open source smart contracts in Etherscan within one day during our study. We found that all these contracts could be classified into six categories according to instruction functionality, namely, *Save Math*, *String Operation*, *Save Data*, *ERC20Token*, *Get Context*, and *Others*. There are 2 smart contracts in the *Save Math* category, which do a series of arithmetic operations with overflow detection. The 4 smart contracts in the *String Operation* category return strings according to the value of input. For example, one smart contract[2] takes an unsigned integer and returns a string, for use in handling failures. The smart contracts in category *Save Data* save input data in memory or long-term storage. *ERC-20 token* smart contracts have basic functionalities such as token creation, allowance approval and token transaction. Smart contracts in *Get Context* category read the transaction context such as message sender and receiver. The remaining contracts are classified into category *Others*.

### 3.5 Experiment Setup

The EVM and eWASM contracts are deployed on a 2-node private chain and executed locally because we focus on the performance of VMs. The WASM contracts are executed on the standalone engines directly. To ensure the robustness of experiment results, we run repeated experiments on three servers: one GPU server (Dual-core Intel Xeon CPU processor, two Nvidia Tesla M40 GPU, and 128 GB memory), one CPU server (8 core Intel Xeon CPU processor and 15 GB memory), and one cloud server (16 core Intel Xeon Platinum 8269 processor and 64 GB memory). All three servers are running Ubuntu 16.04. We do the experiment repeatedly for 100 times and calculate the average running time. *The results on the three servers show the same tendency and similar relative value.* We report only the experiment results on the GPU server in the following.

## 4 MEASUREMENT STUDY

In this section, we present the measurement results. We discussed about the WASM support of blockchian clients to answer **RQ1**. And we present the experimental results of the performance measurement, including a comparison between EVM engines, a comparison between WASM engines, and a comparison between WASM engines and EVM engines, to answer **RQ2**, **RQ3,** and **RQ4**, respectively.

### 4.1 WASM Supports of Blockchain Clients

We compare three widely used clients, including *Go-Ethereum*, *OpenEthereum*, and *EOSIO*. The first two are Ethereum-based, while the last one is not but it supports WASM VM as its native

[2]SmartWalletRevertReasonHelperV3

execu- tion engine. Here, we mainly compare differences of these clients from three perspectives: *WASM Support*, *Coding Convention,* and *Development Tool Support*.

*4.1.1 WASM Support.* **Geth** does not support WASM natively. The WASM support on Geth is now usable but unstable, and developers are still working on several EEIs in progress. As to **OpenEthereum**, it supports EVM and WASM from version 1.9.5 and later, with EVM being the default. The WASM support on Openethereum is now usable but unstable as well. For example, in the deployment and execution process of WASM contracts, developers might still encounter several unexplained exceptions [19]. As a comparison, EOSIO supports WASM VM as its native execution engine, and it provides stable WASM support for production usage.

**How to activate WASM VMs.** To enable the WASM support in Geth, developers have to compile the **Geth** client with EVMC support first, and then launch Geth with eWASM's official engine Hera attached as its EVMC engine. Hera leverages various WASM backends and serves as the connector between WASM VM and the Geth client. To activate WASM VM in **Openethereum**, several parameters should be specified in the chain specification file. Third, **EOSIO** supports WASM VM as its native execution engine, so there is no need to activate WASM VM.

*4.1.2 Coding Convention.* We next investigate the coding standard across clients.

**Entry point and exports.** The WASM smart contracts to be deployed and executed in blockchains must follow a specific format, for example, a given function should be exported as the contract entry point. **Geth** requires a contract to have exactly two exported symbols: the contract entry point main function with no parameters and no result value, and memory, the shared memory space for the EEI to write into. In **Openethereum**, it requires a function named call exported as the contract entry point. An optional exported function deploy will be executed on contract deployment to set up the initial storage of this contract. As to **EOSIO**, it exports an action as the basic element of communication between smart contracts, so the entry point is defined by instancing template action_wrapper with the the action name and action method as parameters.

**Library.** Specific libraries need to be included when develop- ing WASM smart contracts for Openethereum and EOSIO. These libraries provide APIs for chain-dependent smart contract functionalities, provide templates for data types, and delegate the mem- ory allocation. Since the above functionalities are provided by the eWASM runtime on Geth, there is no need to include specific library when developing smart contracts for Geth.

*4.1.3 Development Tool Support.* **Geth** does not provide official WASM toolchain for developers, while **Openethereum** provides an official tutorial that uses Rust nightly-2018-11-12 as the toolchain. As the official toolchain of **EOSIO**, EOSIO.CDT (i.e., Contract Development Toolkit) provides a set of tools and libraries to facilitate smart contract development for the EOSIO platform. After compiling the source code to WASM bytecode, Geth and Openethereum require a binary packer to transform WASM into a restricted format that can be used in blockchain context. All three clients have their own official testnets.

**Answer to RQ1:** *The support for WASM on Ethereum clients is far from ideal. First, the WASM support on Ethereum clients is experimental and not stable. A number of configuration efforts are required to activate the WASM-verison engines for both Geth and OpenEthereum. Second, the WASM coding convention varies across Ethereum blockchain clients, which makes it impossible to implement WASM smart contracts in a general form, i.e., devel- opers have to write and compile smart contracts separately for different clients. Third, few languages are supported by existing tool chains, and the contract development process is cumbersome and error-prone.*

Table 3. Execution Time Ratio of Openethereum to Geth

| Benchmark | Time ratio (256-bit) | Time ratio (64-bit) | 64-bit speedup |
|---|---|---|---|
| add | 2.121 | 2.018 | 1.05 |
| sub | 2.030 | 2.126 | 0.95 |
| shr | 2.256 | 2.117 | 1.07 |
| div | 2.192 | 2.121 | 1.03 |
| mod | 2.098 | 2.100 | 1.00 |
| power | 2.024 | 2.174 | 0.93 |
| fib | 3.592 | 3.872 | 0.93 |
| matrix | 2.016 | 2.760 | 0.73 |
| warshall | 1.234 | 1.046 | 1.18 |
| builtin | 2.420 | NA | NA |
| keccak256 | 1.960 | NA | NA |
| blake2b | **0.606** | NA | NA |
| sha1 | **0.857** | NA | NA |

*NA* indicates the 64-bit version is not available.

### 4.2 A Comparison of EVM Engines

To measure the performance gap between EVM engines of different blockchain clients, we first deploy the 13 kinds of benchmark contracts (Dataset-1) on Geth and Openethereum, to investigate what kinds of operations lead to the major difference. Then we deploy the 33 real-world smart contracts (Dataset-2) on Geth and Openethereum to further investigate the performance gap between EVMs in actual use. We gather the execution time of each client executing the benchmark contracts and calculate the average over 100 runs. After that, we calculate the `time ratio`, i.e., `T(Openethereum) / T(Geth)` for both 256-bit and 64-bit versions.

*4.2.1 Overall Performance Comparison.* Table 3 presents the overall results on Dataset-1. Note that `NA` means the corresponding benchmark does not have a 64-bit version. In general, Geth EVM engine runs faster than Openethereum EVM engine for most of the cases. The `time ratio` of Openethereum to Geth is between 1 and 4 in group *simple operation*, *arithmetic* and *block status*, indicating that Geth EVM engine runs 1 to 4 times faster than Openethereum EVM engine, regardless of the integer type (64-bit and 256-bit) of the smart contract. The `time ratio` in group *hashing* is smaller than other groups. When the source code is optimized manually with inline assembly, the performance gap between the two EVMs is significantly reduced. The time ratio is 0.606 and 0.857 on benchmark *blake2b* and *sha1*, respectively.

To explore why Openethereum runs slower, we manually analyzed the source code of Geth EVM and Openethereum EVM. We found that the different implementation of opcode might be the leading reason. As shown in Figure 2, Geth and Openethereum implement the same opcodes in quite different ways. For example, the opcode `ADD` (Figure 2(a)) is implemented with a simple *Add* operation in Geth. However, Openethereum will further check if an arithmetic overflow occurs. The EVM yellow paper does not explicitly specify the need for overflow checking in the ADD opcode [1], so it is reasonable for Geth to omit this step. However, compared to Geth, Openethereum conducts arithmetic overflow checks for increased safety. Figure 2(b) shows another case: the opcode `EQ` is implemented by an if-else condition, and a 64-bit result is pushed directly into the stack. While in Openethereum, a function named *bool_to_u256* is called to transform a boolean into a 256-bit integer result. Figure 2(c) shows the different implementations of opcode `DIV`. Geth implements the opcode `DIV` with simple division operation. Openethereum

```
41  x, y := stack.pop(), stack.peek()
42  math.U256(y.Add(x, y))
43  interpreter.intPool.put(x)
44  return nil, nil
```

(1) Geth

```
899  let a = self.stack.pop_back();
900  let b = self.stack.pop_back();
901  self.stack.push(a.overflowing_add(b).0);
```

(2) Openethereum

**(a) Opcode implementation of ADD**

```
242  x, y := stack.pop(), stack.peek()
243  if x.Cmp(y) == 0 {
244      y.SetUint64(1)
245  } else {
246      y.SetUint64(0)
247  }
248  interpreter.intPool.put(x)
249  return nil, nil
```

(1) Geth

```
1012  let a = self.stack.pop_back();
1013  let b = self.stack.pop_back();
1014  self.stack.push(Self::bool_to_u256(a == b));
```

(2) Openethereum

**(b) Opcode implementation of EQ**

```
65  x, y := stack.pop(), stack.peek()
66  if y.Sign() != 0 {
67      math.U256(y.Div(x, y))
68  } else {
69      y.SetUint64(0)
70  }
71  interpreter.intPool.put(x)
72  return nil, nil
```

(1) Geth

```
914  let a = self.stack.pop_back();
915  let b = self.stack.pop_back();
916  self.stack.push(if !b.is_zero() {
917      match b {
918          ONE => a,
919          TWO => a >> 1,
920          TWO_POW_5 => a >> 5,
921          TWO_POW_8 => a >> 8,
922          TWO_POW_16 => a >> 16,
923          TWO_POW_24 => a >> 24,
924          TWO_POW_64 => a >> 64,
925          TWO_POW_96 => a >> 96,
926          TWO_POW_224 => a >> 224,
927          TWO_POW_248 => a >> 248,
928          _ => a / b,
929      }
930  } else {
931      U256::zero()
932  });
```

(2) Openethereum

**(c) Opcode implementation of DIV**

Fig. 2. Examples of opcode implementation in Geth and Openethereum.

leverages the shift right operation to optimize the execution with a fallback to the primitive division operation. However, if the divisor does not match the given value, the calculation will go through 10 more checks, leading to a worse performance. Thus, it is apparent that the opcode-level implementation of EVM engines would significantly impact the overall performance of smart contract execution. Compared to Openethereum, Geth has a simpler implementation of opcodes and omits some check steps in the source code. Therefore, Geth performs better in the benchmarks.

*4.2.2 64-bit vs. 256-bit.* The column 4 in Table 3 compares the `time ratio` of 256-bit contract to that of 64-bit version, i.e., `T(256-bit) / T(64-bit)`. Obviously, smart contracts with different data types have shown no obvious performance gaps when running on the EVM engines. *The main reason is that the native data type of EVM is 256-bit integer, and it supports down to 8-bit integer calculation. Integers shorter than 256-bit will be padded to 256-bit simply with leading zeros before EVM engines perform calculation on the integers. Thus, switching the data type does not introduce any obvious impact to the execution performance of smart contracts on EVM engines.*

*4.2.3 Real-World Smart Contracts.* Figure 3 shows the execution time ratio of EVM engines (`T(Openethereum) / T(Geth)`) in executing real-world smart contracts. The `time ratio`

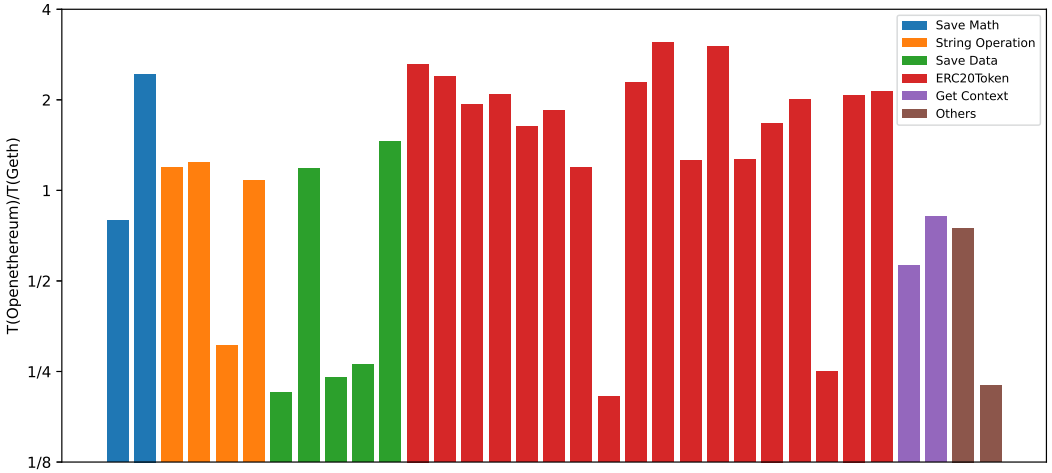


Fig. 3. A Comparison of EVM execution of real-world smart contracts on Geth and Openethereum.

of Openethereum to Geth is between 0.207 and 3.102, and both the maximum and minimum ratios are in category *ERC20Token*. Geth EVM engine runs faster than Openethereum EVM engine for 22 of the 33 real-world smart contracts. The reason why Geth performs better on the 22 benchmarks is consistent with what was mentioned in Section 4.2.1. Geth has a more straightforward implementation of opcodes and omits some check steps in the source code. Therefore, Geth performs better in the benchmarks. We further investigate the smart contracts where Openethereum outperform Geth, and find that these contracts have more non-constant functions than others. Calling non-constant functions will invoke transactions, and the execution time of transaction calls on Openethereum is lower than Geth.

**Answer to RQ2:** *The performance of contract execution on Geth is better than that on Openethereum regardless of contract types and environment. Our investigation suggests that the opcode-level implementation of EVM engines would greatly impact the overall performance of smart contract execution. Moreover, 64-bit and 256-bit smart contracts have shown no obvious performance gap during their execution on EVM engines.*

### 4.3 A Comparison of WASM Engines

To compare the performance of different WASM engines, we compile the Rust version of 12 benchmarks (in group *simple operation*, *arithmetic*, and *hashing*) into WASM bytecode and execute the bytecode on 11 types of standalone WASM VMs: `WAVM` (one backend of Geth-Hera), `wasmi`, `EOS VM`, `Life`, `SSVM`, `wagon`, `wabt`, `wasm3`, `WAMR-interp`, `WAMR-jit`, and `wasmtime`. Note that, we use two different modes of `WAMR` (`WAMR-interp` and `WAMR-jit`) for evaluation. We gather the execution time of executing each benchmark on each standalone WASM engine, and calculate the average of 1,000 runs.

*4.3.1 Overall Performance Comparison.* Table 4 shows the overall result. Surprisingly, the performance across standalone WASM engines differs greatly. *For different benchmarks, there is up to three orders of magnitude difference between the performance of the best engine and the worst engine.* For example, it takes over 21,855 ms for the `EOS VM` to execute the *matrix* benchmark, while it only takes roughly 170 ms for `WAVM` and 156 ms for `wasmtime`. In general, `wasm3` runs the

Table 4. Execution Time (ms) of Standalone WASM VMs

| Benchmark | | WAVM | wasmi | EOS VM | Life | SSVM | wagon | wabt | wasm3 | WAMR -interp | WAMR -jit | wasm -time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| add | 256-bit | 9.89 | 1.28 | 15.63 | 2.88 | 1.01 | 1.39 | 0.91 | 0.07 | 0.53 | 0.53 | 0.52 |
| | 64-bit | 8.28 | 0.44 | 1.65 | 1.39 | 0.44 | 0.44 | 0.75 | 0.02 | 0.44 | 0.43 | 0.35 |
| sub | 256-bit | 10.58 | 82.66 | 704.23 | 111.96 | 55.59 | 64.13 | 18.08 | 4.08 | 10.31 | 10.26 | 1.26 |
| | 64-bit | 8.47 | 0.46 | 1.39 | 1.42 | 0.45 | 0.46 | 0.79 | 0.03 | 0.46 | 0.45 | 0.35 |
| shr | 256-bit | 46.62 | 2606.04 | 13654.66 | 2598.15 | 1653.20 | 2216.84 | 805.28 | 199.77 | 512.54 | 518.38 | 104.02 |
| | 64-bit | 8.54 | 0.46 | 0.83 | 1.41 | 0.44 | 0.44 | 0.76 | 0.02 | 0.44 | 0.43 | 0.34 |
| div | 256-bit | 80.37 | 70.88 | 504.33 | 67.78 | 49.63 | 64.12 | 21.45 | 3.04 | 12.73 | 12.99 | 33.69 |
| | 64-bit | 19.31 | 1.03 | 1.66 | 3.20 | 1.00 | 1.06 | 1.71 | 0.07 | 1.01 | 1.04 | 0.74 |
| mod | 256-bit | 283.88 | 15915.78 | 96174.68 | 21889.12 | 11065.11 | 18697.32 | 5067.36 | 990.62 | 3540.97 | 3252.94 | 404.51 |
| | 64-bit | 8.44 | 0.57 | 0.84 | 1.41 | 0.45 | 0.44 | 0.76 | 0.02 | 0.43 | 0.42 | 0.34 |
| power | 256-bit | 8.46 | 0.47 | 1.72 | 1.44 | 0.45 | 0.43 | 0.74 | 0.02 | 0.45 | 0.43 | 0.36 |
| | 64-bit | 19.18 | 1.05 | 0.80 | 3.19 | 1.02 | 1.09 | 1.71 | 0.06 | 1.03 | 1.01 | 0.75 |
| fib | 256-bit | 207.11 | 513.32 | 1784.59 | 598.37 | 300.93 | 504.27 | 202.76 | 47.31 | 114.96 | 151.88 | 84.75 |
| | 64-bit | 8.38 | 0.45 | 0.81 | 1.41 | 0.43 | 0.44 | 0.76 | 0.02 | 0.44 | 0.45 | 0.35 |
| matrix | 256-bit | 170.45 | 3504.55 | 21855.51 | 3788.64 | 2400.23 | 3284.86 | 1035.07 | 414.12 | 849.47 | 827.52 | 156.02 |
| | 64-bit | 38.42 | 44.21 | 203.55 | 42.93 | 29.36 | 15.88 | 13.39 | 2.06 | 7.45 | 7.68 | 11.96 |
| warshall | 256-bit | 217.15 | 15.90 | 72.35 | 23.54 | 10.84 | 46.14 | 6.51 | 0.73 | 3.52 | 3.53 | 126.88 |
| | 64-bit | 28.82 | 1.87 | 4.71 | 2.85 | 0.90 | 5.33 | 1.08 | 0.06 | 0.52 | 0.51 | 11.83 |
| keccak256 | - | 10.99 | 0.92 | 1.21 | 1.49 | 0.48 | 0.92 | 0.77 | 0.03 | 0.44 | 0.43 | 0.96 |
| blake2b | - | 133.70 | 1.79 | 9.87 | 8.01 | 0.82 | 21.28 | 1.21 | 0.22 | 0.47 | 0.46 | 21.79 |
| sha1 | - | 92.89 | 1.51 | 6.31 | 6.10 | 0.67 | 15.17 | 1.13 | 0.17 | 0.46 | 0.45 | 18.65 |

fastest among all WASM VMs, with execution time less than 1 ms for most of the cases. However, `WAVM`, the engine used in the WASM-version client of Geth, achieves the worst result for all of the 64-bit benchmarks. For some specific 256-bit versions of benchmarks including *shr*, *div*, *mod*, *fib*, and *matrix*, `EOS VM` takes the longest execution time. The execution time of the remaining VMs is between the execution times of the above three engines. And the execution time of `WAMR-interp` and `WAMR-JIT` are very close. It is because the advantage of WAMR JIT cannot be highlighted in these cases. `WAMR-interp` (interpreter mode) and `WAMR-JIT` (Just-in-Time compilation mode) are two execution modes in `WAMR`. The benchmarks used in Table 4 are simple operation, arithmetic, block status, and hashing. These computations do not include WASM binaries that trigger `WAMR` JIT optimization operations, such as heavy loops, so the advantage of WAMR JIT's one-time compilation for multiple runs cannot be highlighted in these scenarios. We speculate that the performance gap is introduced by the diverse implementations of engines. *This result suggests that, although there are a number of WASM engines available, not all of them can be used to achieve excellent performance, especially when adopting WASM to the blockchain environment.*

*4.3.2 64-bit vs. 256-bit.* From a vertical analysis (the two rows for each benchmark contract), we can measure the impact of integer types on the performance of WASM bytecode execution. *Obviously, the integer types in the smart contracts will significantly affect the performance of WASM bytecode execution, across all the engines we studied.* In general, 256-bit benchmarks run slower than their 64-bit counterparts. Most of the WASM engines gain more than 2× speedup with their 64-bit versions. For the most representative benchmarks including *matrix*, *fib* and *mod*, the execution of the 64-bit smart contracts can achieve great speedup for all the engines. For example, it takes about 1,785 ms to execute the 256-bit version of the *fib* contract on the EOS VM, whereas it takes less than 1 ms to execute the 64-bit version. It can be explained that, WASM bytecode and EVM bytecode support different types of integers. WASM uses 32/64-bit bytecode but EVM supports 256-bit bytecode. For the 256-bit smart contract, it must be converted before the WASM engine can execute them, which would introduce overhead.

Table 5. A Comparison of eWASM and EVM Bytecode Execution on Geth and Openethereum

| Benchmark | | Geth | | | Openethereum | | |
|---|---|---|---|---|---|---|---|
| | | Overall | Overhead | Net | Overall | Overhead | Net |
| add | 256-bit | 25.07 | 92.8% | 1.81 | 4.76 | 96.0% | 0.19 |
| | 64-bit | 16.86 | 93.0% | 1.18 | 0.17 | 87.8% | 0.02 |
| sub | 256-bit | 23.25 | 93.5% | 1.51 | 4.59 | 96.0% | 0.18 |
| | 64-bit | 21.21 | 93.9% | 1.29 | 0.28 | 87.7% | 0.03 |
| div | 256-bit | 32.52 | 78.3% | 7.06 | 64.21 | 96.3% | 2.38 |
| | 64-bit | 9.99 | 93.1% | 0.69 | 3.35 | 98.4% | 0.05 |
| mod | 256-bit | 109.33 | 92.3% | 8.41 | 111.85 | 96.7% | 3.69 |
| | 64-bit | 14.45 | 93.8% | 0.90 | 2.95 | 98.3% | 0.05 |
| power | 256-bit | 101.09 | 98.8% | 1.21 | 59.65 | 99.9% | 0.06 |
| | 64-bit | 16.73 | 92.8% | 1.20 | 0.12 | 88.0% | 0.01 |
| fib | 256-bit | 138.33 | 95.8% | 5.81 | 171.02 | 97.5% | 4.28 |
| | 64-bit | 9.09 | 93.1% | 0.63 | 2.10 | 98.5% | 0.03 |
| matrix | 256-bit | 3.95 | 33.6% | 2.62 | 85.19 | 97.4% | 2.21 |
| | 64-bit | 4.06 | 30.7% | 2.81 | 13.96 | 99.8% | 0.03 |
| warshall | 256-bit | 5.65 | 61.7% | 2.16 | 20.92 | 94.6% | 1.13 |
| | 64-bit | 4.80 | 58.8% | 1.98 | 9.36 | 95.8% | 0.39 |
| keccak256 | 256-bit | 33.84 | 98.5% | 0.51 | 119.80 | 87.1% | 15.45 |

**Answer to RQ3:** *The performance of smart contract execution across standalone WASM engines differs greatly, even up to three orders of magnitude difference for some contracts. It suggests that not all the popular WASM engines can be used to achieve excellent performance. Furthermore, the native data type has a great impact on the performance of contract execution, as the native data type of WASM is 32/64 bit, it would introduce additional overhead during the execution of 256-bit smart contracts.*

### 4.4 WASM Engines vs. EVM Engines

As previously stated, we compile the benchnmarks in Dataset-1 into both EVM bytecode and eWASM bytecode. Then we deploy the contracts on Geth and Openethereum, and gather the execution time of EVM and WASM engines on the same blockchain client. For real-world smart contract benchmarks (Dataset-2), we gather the execution time of EVM and WASM engines on Geth, since the eWASM smart contracts generated by SOLL can only be executed on the Geth.

*4.4.1 Overall Performance Comparison.* Table 5 (columns 3 and 6) shows the execution time ratio of WASM to EVM (`T(WASM) / T(EVM)`) in executing benchmark contracts with different integer types. A larger ratio indicates that the WASM engine is slower than the EVM engine on the same blockchain client. The statistics show a counter-intuitive result. The eWASM engines are slower than EVM engines for all of the 256-bit benchmarks and most of the 64-bit benchmarks on both Geth and Openethereum clients. When the integer type is 256-bit, on Geth client (see column 3 on Table 5), the time ratio of WASM VM to EVM is between 3.95 (benchmark *matrix*) and 138.33 (benchmark *fib*). The time ratio of benchmark *mod*, *power*, and *fib* is significantly larger than others. It may be because all of these smart contracts involve `MUL` and `MOD` instructions. Benchmark *matrix* and *warshall* have significantly smaller time ratio than other benchmarks. This may be because these two contracts involve multiple `MLOAD` and `MSTORE` instructions related to memory operating.

Table 6. Overall Performance Speedup of 64-Bit Benchmarks to Corresponding 256-Bit Benchmarks

| Benchmark | Geth | Openethereum |
|---|---|---|
| add | 1.49x | 28x |
| sub | 1.10x | 16.39x |
| div | 3.26x | 19.17x |
| mod | 7.57x | 37.92x |
| power | 6.04x | 497.08x |
| fib | 15.22x | 81.44x |
| matrix | 0.97x | 6.10x |
| warshall | 1.18x | 2.24x |

On Openethereum client, the time ratio of WASM VM to EVM is between 4.59 (benchmark *sub*) and 171.02 (benchmark *fib*), which implies that the bad performance of WASM VM exists across different blockchain clients.

*4.4.2 64-bit vs. 256-bit.* The time ratio of WASM VM to EVM of 64-bit benchmarks is significantly lower than their 256-bit counterpart. As shown in Table 6, switching from 256-bit to 64-bit achieves a speedup up to 15× and 497× on Geth and Openethereum, respectively. When executing benchmark *add*, *sub*, and *power* on Openethereum, WASM VM even runs faster than EVM. This is because the native data type of WASM is 32/64-bit, thus computations higher than 64-bit must be converted to a 256-bit format before the WASM engine can process them. The 64-bit contracts do not go through type conversion, which leads to faster execution.

*4.4.3 Understanding the Execution Overhead.* As previously stated in Section 2.3.3, to be executed in the Ethereum environment, eWASM interacts with the blockchain through EEI methods, restricts non-deterministic behavior, and conducts gas metering before the execution of WASM bytecode. Compared with the WASM execution on native engines, the above operations introduce additional overhead.

**Measuring the Overhead.** To measure the execution overhead of WASM VMs in the blockchain environment, we compare the performance of smart contract execution on eWASM engine with the performance on the corresponding standalone WASM engines. We use the difference of the result `T(overhead) = T(eWASM) - T(standalone WASM)` to approximate the execution overhead of eWASM on Ethereum clients. Table 5 (columns 4 and 7) shows the statistics of the results. As shown in Figure 4, on Geth, the execution overhead is far more than the execution time of standalone WASM VM. The execution overhead takes up 84.9% of WASM execution time on average, with the maximum overhead taking up 98.8% of total execution time (benchmark *power*). Figure 5 is shown in log scaled. Openethereum exhibits a similar result to Geth. The execution overhead takes up 95.0% of WASM execution time on average, with the maximum overhead taking up 99.9% of WASM execution time (benchmark *power*).

**Excluding the Overhead.** Table 5 (columns 5 and 8) shows the execution time ratio of WASM to EVM after excluding the overhead, which can be expressed as: `T(standalone WASM) / T(EVM)`. Interestingly, without considering the overhead, for most cases, the performance of eWASM engine is superior to that of WASM engine on Openethereum, and the performance eWASM engine is quite close to that of WASM engine on Geth. According to the design of eWASM, the execution overhead comes from gas metering before the execution of WASM bytecode, the deterministic stack height metering and the EEI methods. This also explains the performance gap between 256-bit

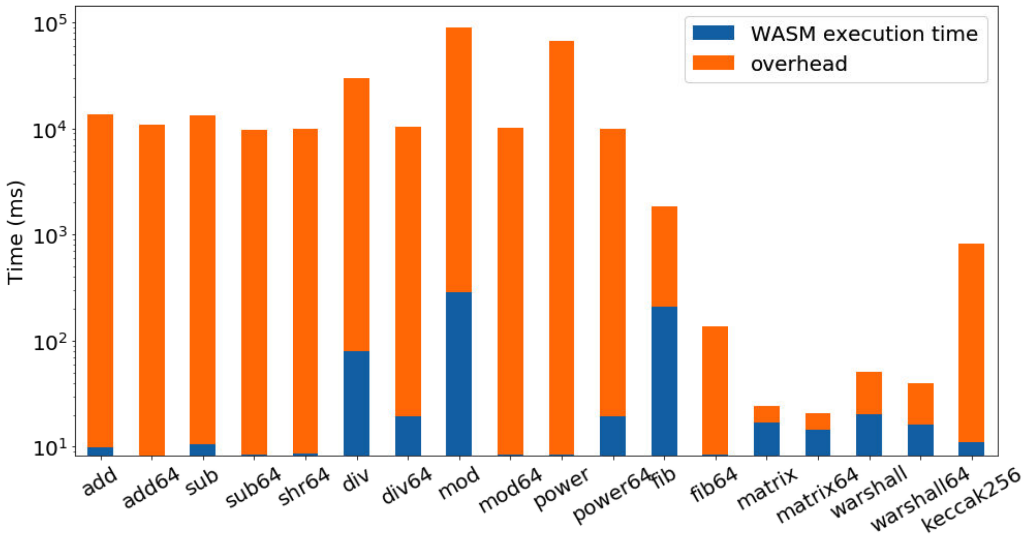


Fig. 4. Execution overhead of WASM VM running on Blockchain clients Geth.

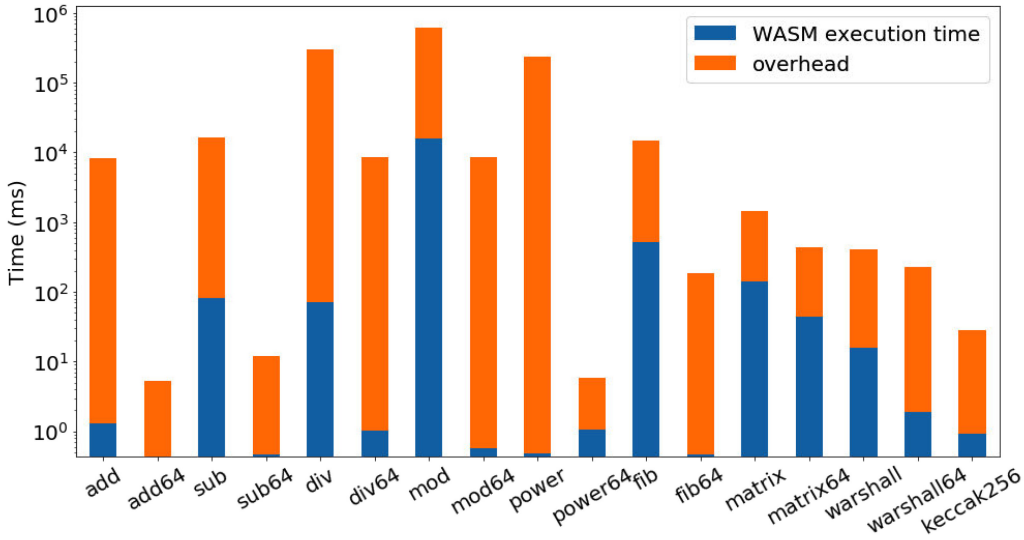


Fig. 5. Execution overhead of WASM VM running on Blockchain clients Openethereum.

contracts and 64-bit contracts. Since the conversion of integer types are done during compilation, the compiled WASM bytecode for 256-bit contracts contain more instructions, so more overhead is introduced.

*4.4.4 Real-World Smart Contracts.* Figure 6 shows the execution time ratio of WASM to EVM (`T(WASM) / T(EVM)`) in executing real-world smart contracts on Geth client. The eWASM engines are slower than EVM engines for all of the real-world benchmarks. The time ratio of WASM VM to EVM is between 8.97 (in category *Others*) and 65.84 (in category *ERC20Token*). The time ratio of category *Others* is lower than others. It may be because all of the smart contracts in this category

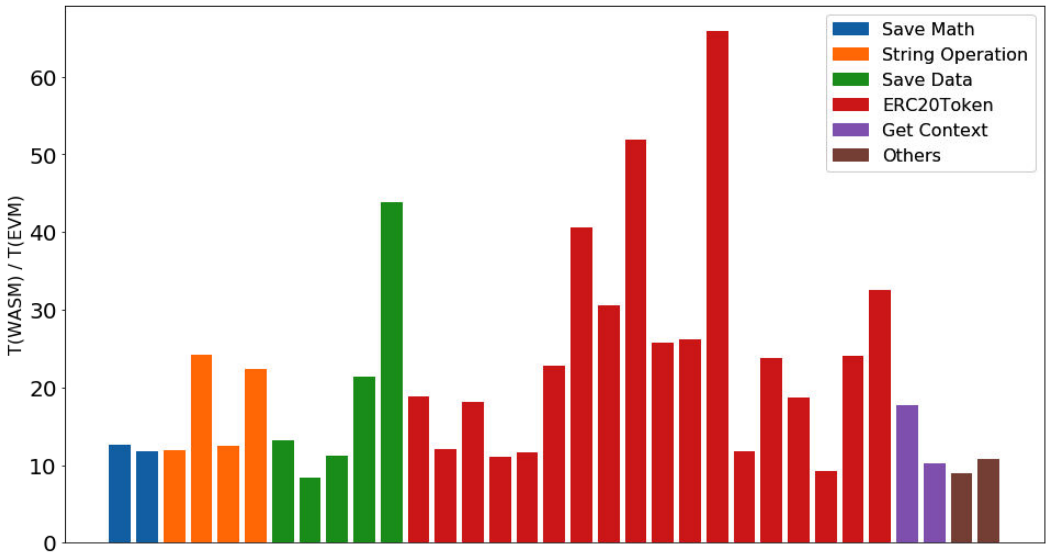


Fig. 6. A Comparison of eWASM and EVM execution of real-world smart contracts on Geth.

are extremely simple (directly return a value or invoke revert operation). The contract in category *ERC20Token* which has the largest time ratio is a rather complex smart contract with 17 transaction calls and no local calls.

**Answer to RQ4:** *While standalone WASM VMs are faster than EVMs, the blockchain flavored WASM VMs are slower than EVMs for all of the 256-bit benchmarks and most of the 64-bit benchmarks on both Geth and Openethereum clients. This is because of the overhead introduced by gas metering before the execution of WASM bytecode and the inefficient context switch of EEI methods. To improve the performance of EVMs, several approaches can be considered. One approach is to minimize the overhead caused by the context switch between the eWASM execution environment and the blockchain environment when using EEI. This can be achieved through efficient memory management, reducing unnecessary data copying, and optimizing the interaction between the two environments. Additionally, enhancing the gas metering mechanism in eWASM can help improve performance. This can involve optimizing the gas calculation algorithm and minimizing the computational cost associated with gas metering.*

## 5 DISCUSSION

### 5.1 Implications

Our findings reveal that the support for WASM on Ethereum-based blockchain clients is far from ideal. Our observations are beneficial to stakeholders in the blockchain community, including blockchain client developers, VM developers, and compiler developers. For blockchain clients, our results can help developers optimize them by reducing additional overhead, including the execution process of WASM on the blockchain clients and the interface design between WASM and the blockchain environment. For VMs, our results can help developers improve the performance of the VMs by removing or adjusting the bottleneck instructions. For compilers, our results can help developers refine the implementation of the compilers by considering the performance gap

between the 256-bit version and the 64-bit one. Furthermore, we implement a relatively comprehensive benchmark set with various integer types and different inputs in Solidity and Rust. We release the benchmark set to the community, and we believe that it can benefit future research in this direction.

### 5.2 Limitations

First, although we tried our best to curate the first dataset that covers diverse benchmarks and real-world smart contracts for evaluation, our benchmark might not be comprehensive enough. Additionally, we excluded popular WASM runtimes, V8, and SpiderMonkey, from our evaluation because our focus was specifically on the impact of WASM and EVM engines on the execution performance of smart contracts. It should be noted that currently, no blockchain platforms utilize V8 or SpiderMonkey to execute smart contracts. This is primarily because V8 and SpiderMonkey are JavaScript engines primarily designed for web browsers. While they are competent and efficient in executing JavaScript code and WASM binaries, they are incompatible with blockchain platforms' specific requirements and design principles. V8 and SpiderMonkey are in-browser VMs and lack standalone support for consensus mechanisms and EEI methods essential in blockchain contexts. Therefore, considering V8 and SpiderMonkey is beyond the scope of our article. Second, there have been other works focused on the time-consuming impact of network latency [45, 57, 60], consensus protocols [36, 44, 50, 59], and scalability [36, 45, 55, 56, 61]. However, these aspects are out of the scope of our work. In this article, we only consider the execution performance of smart contract bytecode itself, while the actual performance of smart contract invocation in the blockchain environment also depends on other factors, such as consensus protocols. And the underlying state database's IO is the key performance factor that could influence OpenEthereum and Geth. We'll consider this factor in the future analysis. In fact, our measurement study is performed on clients that support different types of bytecode, which can faithfully reflect the improvement introduced by bytecode VMs. Thirdly, while our primary focus is on Ethereum-based blockchain clients, our work can be directly applicable to several other blockchain platforms that are compatible with the EVM. The EVM is the pioneering blockchain platform that enables the execution of smart contracts and remains the largest blockchain platform to date. Notable examples of blockchain platforms compatible with EVM for executing smart contracts include BNB Chain [33], Polygon [35], Avalanche [32], Fantom [34], among others. Therefore, our findings and research can be generalized to these platforms. However, it should be noted that blockchain platforms that have not yet adopted WASM-based execution engines, cannot directly benefit from our work. Nevertheless, our research has some degree of generalizability to other platforms in the blockchain ecosystem.

## 6 RELATED WORK

### 6.1 Performance Measurement of WASM

To the best of our knowledge, we do the first study to characterize the impact of WASM and EVM engines on the execution performance of smart contracts. Nevertheless, there is some related work on the general in-browser WASM study. Haas et al. [40] measured the execution time of the PolyBenchC benchmarks running on WASM on V8 and SpiderMonkey normalized to native execution. The results show that WASM is very competitive with native code. Herrera et al. [41] further evaluated the performance of JavaScript and WASM using the Ostrich benchmark suite, and demonstrated significant performance improvements for WASM, in the range of 2× speedups over the same browser's JavaScript engine. They also evaluated the relative performance of portable versus vendor-specific browsers, and the relative performance of server-side versus

client-side WASM. Jangda et al. [42] expanded the benchmark test suite to SPEC CPU suite. They built BROWSIX-WASM to run unmodified WASM-compiled Unix applications directly inside the browser and found that applications compiled to WASM run slower by an average of 45% (Firefox) to 55% (Chrome) compared to native. The root cause of this performance gap is that the instructions produced by WASM have more loads/stores and more branches. Wang et al. [58] investigated how browser engines optimize WASM execution compared to JavaScript. The results showed that JIT optimization in Chrome significantly impacts JavaScript speed but has no discernible effect on WASM speed. These works all focused on the general in-browser WASM execution performance. And to the best of our knowledge, our work is the first to characterize the impact of WASM and EVM engines on the execution performance of smart contracts.

### 6.2 Other WASM Studies

WASM is a promising and newly emerged area. Several studies have been conducted on various aspects of WASM, including WASM application analysis against cryptojacking, dynamic analysis of WASM programs, identification of compiler bugs in WASM, investigation of runtime bugs in WASM, and more. WASM is increasingly popular in recent years. Musch et al. [48] studied the prevalence of WASM in the Alexa Top 1 million websites and found that 1 out of 600 sites use WASM by then. RÃijth et al. [54] found that the majority of in-browser miners utilize WASM for efficient PoW calculation. Although the use of WASM might lead to the problem of in-browser cryptojacking, several detection methods have been proposed. Bian et al. [37] proposed a browser-based defense mechanism against WASM cryptojacking. Lehmann et al. [47] proposed a dynamic analysis tool named Wasabi to meet the demand of analyzing WASM. Chen et al. [38] designed and developed a new concolic fuzzer for uncovering vulnerabilities in WASM smart contracts after tackling several challenging issues. Romano et al. [52] conducted an empirical study on bugs in WASM compilers. They investigated 146 bug reports in Emscripten, focusing on the unique challenges that WASM compilers face in comparison to traditional compilers. Zhang et al. [62] analyzed the WASM runtime bug patterns and the fix strategies and detected bugs in the most popular WASM runtimes. Although there is a wide range of research on WASM, to the best of our knowledge, no existing work has thoroughly examined the impact of WASM and EVM engines on the execution performance of smart contracts. Our work sheds light on the practical application of WASM in blockchain environments.

### 6.3 Performance Measurement of Blockchain

Previous work focused on measuring the performance of the overall Blockchain transitions. A number of studies [39, 46, 53, 63] have measured the performance of blockchain systems from different perspectives. For example, Dinh et al. [39] proposed an evaluation framework for analyzing private blockchains named BLOCKBENCH and conducted an evaluation of Ethereum, Parity, and Hyperledger. They revealed that the main bottlenecks are the consensus protocols in Hyperledger and Ethereum, but transaction signing for Parity. They also revealed that the execution engine of Ethereum and Parity is less efficient than that of Hyperledger. Zheng et al. [63] proposed a performance monitoring framework, providing detailed performance metrics for the users to know the exact performance in different stages of the blockchain. They observed that the blockchain with PoW consensus protocol shows lower overall performance than others. They further analyzed the performance of different smart contracts. Lee et al. [46] studied the graph properties of Ethereum blockchain networks and subnetworks by analyzing the interactions, the traces of transitions, and calls of smart contracts. They found that these blockchain networks are very different from social networks but are similar to the Web. Blockchain networks are small-world and well-connected.

## 7 CONCLUSION

In this work, we have presented the first measurement study to date, on the performance of smart contract execution across EVM and WASM VM. To pinpoint the root cause leading to the performance bottleneck, we provide a comprehensive study that includes both intra-bytecode engine comparison (i.e., EVM vs. EVM, WASM vs. WASM), and inter-bytecode engine comparison (i.e., EVM-version engine VS. WASM-version engine of the same blockchain client). Our exploration reveals a number of issues in adopting WASM to blockchain, and our findings can provide insightful implications to the future design, implementation, and optimization of blockchain clients, WASM engines, and compilers.

## REFERENCES


[1] 2016. EVM Yellow Paper. Retrieved from http://gavwood.com/Paper.pdf
[2] 2018. Ethereum 2.0. Retrieved from https://medium.com/rocket-pool/ethereum-2-0-76d0c8a76605
[3] 2021. Binaryen: Compiler Infrastructure and Toolchain Library for WebAssembly. Retrieved from https://github.com/webassembly/binaryen
[4] 2021. A Curated List of Languages that Compile Directly to or have their VMs in WebAssembly. Retrieved from https://github.com/appcypher/awesome-wasm-langs
[5] 2021. EOS VM - A Low-Latency, High Performance and Extensible WebAssembly Engine. Retrieved from https://github.com/EOSIO/eos-vm
[6] 2021. EOSIO. Retrieved from https://eos.io
[7] 2021. Ethereum Consensus Tests. Retrieved from https://github.com/ethereum/tests
[8] 2021. Ethereum Whitepaper. Retrieved from https://ethereum.org/en/whitepaper/
[9] 2021. Etherscan: The Ethereum Blockchain Explorer. Retrieved from https://etherscan.io/
[10] 2021. eWASM Design Overview and Specification. Retrieved from https://github.com/ewasm/design
[11] 2021. Go Ethereum: Official Golang Implementation of the Ethereum Protocol. Retrieved from https://github.com/ethereum/go-ethereum
[12] 2021. Hera: eWASM Virtual Machine Conforming to the EVMC API. Retrieved from https://github.com/ewasm/hera
[13] 2021. Life - A Secure WebAssembly VM Catered for Decentralized Applications. Retrieved from https://github.com/perlin-network/life
[14] 2021. NEAR. Retrieved from https://near.org
[15] 2021. OpenEthereum: The Fast, Light, and Robust Client for Ethereum-Like Networks. Retrieved from https://github.com/openethereum/openethereum
[16] 2021. Project Alchemy: Solidity Implementation of BLAKE2b. Retrieved from https://github.com/ConsenSys/Project-Alchemy
[17] 2021. Proof-of-Stake (PoS) of Ethereum. Retrieved from https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/
[18] 2021. Proof-of-Work (PoW) of Ethereum. Retrieved from https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/
[19] 2021. SOLL, A New Compiler for Generate eWASM from Solidity and Yul. Retrieved from https://github.com/second-state/SOLL
[20] 2021. Solsha1: Pure-Solidity Implementation of the SHA1 Hash Function. Retrieved from https://github.com/ensdomains/solsha1
[21] 2021. SSVM - A High Performance, Extensible, and Hardware Optimized WebAssembly Virtual Machine for Cloud, AI, and Blockchain Applications. Retrieved from https://github.com/second-state/SSVM
[22] 2021. Tron WASM. Retrieved from https://github.com/NikolaLohinski/tron-wasm
[23] 2021. WABT: The WebAssembly Binary Toolkit. Retrieved from https://github.com/WebAssembly/wabt
[24] 2021. Wagon, A WebAssembly-Based Go Interpreter, for Go. Retrieved from https://github.com/go-interpreter/wagon
[25] 2021. Wasm3 - A High Performance WebAssembly Interpreter Written in C. Retrieved from https://github.com/wasm3/wasm3
[26] 2021. Wasmi: WASM Interpreter in Rust. Retrieved from https://github.com/paritytech/wasmi
[27] 2021. Wasmtime: Standalone JIT-Style Runtime for WebAssembly, using Cranelift. Retrieved from https://github.com/bytecodealliance/wasmtime
[28] 2021. WebAssembly. Retrieved from https://webassembly.org/
[29] 2021. WebAssembly Micro Runtime. Retrieved from https://github.com/bytecodealliance/wasm-micro-runtime
[30] 2021. WebAssembly Virtual Machine. Retrieved from https://github.com/WAVM/WAVM

[31] 2023. artifact. Retrieved from https://drive.google.com/file/d/1R5Z5uojHoXAMRoUKG9aoibmXSC5vEMPn/view?usp=sharing
[32] 2023. AVALANCHE. Retrieved from https://www.avax.network/
[33] 2023. BNB Chain. Retrieved from https://www.bnbchain.org/en
[34] 2023. Fantom. Retrieved from https://fantom.foundation/
[35] 2023. polygon. Retrieved from https://polygon.technology/
[36] Salem Alqahtani and Murat Demirbas. 2021. Bottlenecks in blockchain consensus protocols. In *Proceedings of the 2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*. IEEE, 1–8.
[37] Weikang Bian, Wei Meng, and Mingxue Zhang. 2020. MineThrottle: Defending against WASM in-browser cryptojacking. In *Proceedings of the Web Conference 2020*. 3112–3118.
[38] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. 2022. WASAI: Uncovering vulnerabilities in wasm smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 703–715.
[39] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1085–1100.
[40] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.
[41] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. 2018. *WebAssembly and Javascript Challenge: Numerical Program Performance Using Modern Browser Technologies and Devices*. University of McGill, Montreal: QC, Technical Report SABLE-TR-2018-2.
[42] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *Proceedings of the 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 107–120.
[43] Pascal Felber Valerio Schiavoni Jämes Ménétrey, Marcelo Pasin. 2021. TWINE: An embedded trusted runtime for webassembly. In *Proceedings of the 37th IEEE International Conference on Data Engineering*. 205–216.
[44] Manpreet Kaur, Mohammad Zubair Khan, Shikha Gupta, Abdulfattah Noorwali, Chinmay Chakraborty, and Subhendu Kumar Pani. 2021. MBCP: Performance analysis of large scale mainstream blockchain consensus protocols. *IEEE Access* 9 (2021), 80931–80944.
[45] Murat Kuzlu, Manisa Pipattanasomporn, Levent Gurses, and Saifur Rahman. 2019. Performance analysis of a hyperledger fabric blockchain framework: Throughput, latency and scalability. In *Proceedings of the 2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 536–540.
[46] Xi Tong Lee, Arijit Khan, Sourav Sen Gupta, Yu Hann Ong, and Xuan Liu. 2020. Measurements, analyses, and insights on the entire ethereum blockchain network. In *Proceedings of the Web Conference 2020*. 155–166.
[47] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A framework for dynamically analyzing WebAssembly. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1045–1058.
[48] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A study on the prevalence of WebAssembly in the wild. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 23–42.
[49] Satoshi Nakamoto. 2019. *Bitcoin: A Peer-to-peer Electronic Cash System*. Technical Report. Manubot.
[50] Gabriel Antonio F. Rebello, Gustavo F. Camilo, Lucas C. B. Guimarães, Lucas Airam C. de Souza, and Otto Carlos M. B. Duarte. 2022. Security and performance analysis of quorum-based blockchain consensus protocols. In *Proceedings of the 2022 6th Cyber Security in Networking Conference (CSNet)*. IEEE, 1–7.
[51] Micha Reiser and Luc Bläser. 2017. Accelerate javascript applications by cross-compiling to WebAssembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. 10–17.
[52] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2021. An empirical study of bugs in webassembly compilers. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–54.
[53] Sara Rouhani and Ralph Deters. 2017. Performance analysis of ethereum transactions in private blockchain. In *Proceedings of the 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 70–74.
[54] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. 2018. Digging into browser-based crypto mining. In *Proceedings of the Internet Measurement Conference 2018*. 70–76.
[55] Abdurrashid Ibrahim Sanka and Ray C. C. Cheung. 2021. A systematic review of blockchain scalability: Issues, solutions, analysis and future research. *Journal of Network and Computer Applications* 195 (2021), 103232.
[56] Mattias Scherer. 2017. Performance and Scalability of Blockchain Networks and Smart Contracts.

[57] Luming Wan, David Eyers, and Haibo Zhang. 2019. Evaluating the impact of network latency on the safety of blockchain transactions. In *Proceedings of the 2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 194–201.

[58] Weihang Wang. 2021. Empowering web applications with WebAssembly: Are we there yet?. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering*. 1301–1305.

[59] Yang Xiao, Ning Zhang, Wenjing Lou, and Y. Thomas Hou. 2020. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys and Tutorials* 22, 2 (2020), 1432–1465.

[60] Xiaoqiong Xu, Gang Sun, Long Luo, Huilong Cao, Hongfang Yu, and Athanasios V. Vasilakos. 2021. Latency performance modeling and analysis for hyperledger fabric blockchain network. *Information Processing and Management* 58, 1 (2021), 102436.

[61] Di Yang, Chengnian Long, Han Xu, and Shaoliang Peng. 2020. A review on scalability of blockchain. In *Proceedings of the 2020 the 2nd International Conference on Blockchain Technology*. 1–6.

[62] Yixuan Zhang, Shangtong Cao, Haoyu Wang, Zhenpeng Chen, Xiapu Luo, Dongliang Mu, Yun Ma, Gang Huang, and Xuanzhe Liu. 2023. Characterizing and detecting webassembly runtime bugs. *ACM Transactions on Software Engineering and Methodology* (2023).

[63] Peilin Zheng, Zibin Zheng, Xiapu Luo, Xiangping Chen, and Xuanzhe Liu. 2018. A detailed and real-time performance monitoring framework for blockchain systems. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 134–143.

Received 28 January 2023; revised 7 August 2023; accepted 3 January 2024