Efficient Deterministic Execution of Smart Contracts

ENIS CEYHUN ALP, CRISTINA BĂSESCU, PASINDU TENNAGE, NOÉMIEN KOCHER, GAYLOR BOSSON, and BRYAN FORD, Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland

1 INTRODUCTION

Smart contracts are user-defined programs that are automatically executed by a blockchain network. Many blockchain nodes redundantly execute the same smart contact code in order to check each others' work, and execute the consensus algorithm of the underlying blockchain to agree on the result. In effect, the blockchain network guarantees the correctness and integrity of smart contract execution without relying on a single trusted entity. In this way, smart contracts enable mutually-distrustful parties to transact with each other in a fully-decentralized manner.

One of the main properties of smart contracts is determinism. Execution of a smart contract has to produce the same result across all blockchain nodes so that they can reach consensus. The largest smart contact platform Ethereum enforces deterministic smart contracts at both language- and execution-level. Ethereum smart contracts are written in high-level domain-specific languages (*e.g.*, Solidity, Vyper) and are executed on a specialized virtual machine with a restricted instruction set called the Ethereum Virtual Machine (EVM).

Although EVM and its high-level languages address the determinism challenge, they also have shortcomings. First, EVM suffers from poor performance. Second, EVM and its languages do not support floating-point arithmetic, because IEEE floating-point arithmetic is not strictly deterministic. Finally, since EVM languages are still immature, they lack standard libraries and development and debugging tools. This not only increases the burden on the programmers and slow down the development process but also causes bugs in smart contracts that can potentially lead to security vulnerabilities.

In this paper, we explore the design space for a deterministic and high-performance smart contract sandbox that can support programs written in general-purpose programming languages. We consider four alternative designs based on at what level the sandbox enforces determinism: the most generic - system level (§4), the most specific - language level (§5, §7.4), and an intermediate level - any virtual/software-based instruction architecture, *e.g.*, EVM, WASM, JVM, LLVM IR (§6, §7.1, §7.2, §7.3). We consider the following challenges for the different design spaces. The sandbox has to: (1) be deterministic, (2) achieve smart contract execution efficiency comparable to native execution of standard code, (3) support smart contracts written in standard general-purpose languages, such as C/C++, Go, Java, and (4) reliably detect and halt a faulty/malicious program that does not terminate and clog the smart contract network.

The remainder of the document is organized as follows. Section §2 gives background on blockchains, smarts contracts, alongside the threat model. Section §3 describes the goals of our system, together with the challenges of fulfilling the goals. Sections §4, §5, §6, §7 address these challenges, each exploring different levels to enforce determinism. Next, Section §8 provides implementation details of our approaches, and Section §9 measures the performance of the different approaches through a set of benchmarks. Finally, we present our conclusions and ongoing work in Section §10.

The author order is alphabetical.

The work was conducted while Gaylor Bosson was affiliated with EPFL. In the meantime, his affiliation changed to Taurus Group SA, Switzerland.

2 BACKGROUND

Smart contracts are user-defined programs typically managed by a blockchain. We first give background on what blockchains are, who runs them, and their threat model. Second, we focus on smart contract execution and their threat model.

2.1 Blockchains and Smart Contracts

A blockchain is a distributed system run by nodes connected through a network such as the internet. These nodes build and maintain the blockchain, which is a data structure containing an ordered collection of blocks. The blockchain has two important properties: (1) It is append-only, meaning that blocks can be added but not deleted, and (2) The blocks themselves are read-only, meaning that the contents of a block cannot be changed once the block is part of the blockchain. The blocks can contain code, *i.e.*, smart contracts, or data, *e.g.*, smart contract inputs and outputs.

Users interact with the blockchain by submitting transactions. Transactions can simply store data on the blockchain, for example. But transactions can also store code, and call smart contracts with input data stored in the blockchain.

The nodes running the blockchain need to agree, *i.e.*, reach consensus, on the order of transaction execution. In the case of smart contracts, this entails two steps. First, the nodes need to agree on having received the same smart contract code to be stored at the same block index. Second, if the transaction is a call, the nodes execute that call on the smart contract and need to agree on the execution output.

2.2 Threat model

In a blockchain platform that enables smart contract execution, there are two types of actors that could be malicious: (1) Blockchain nodes that execute smart contracts and (2) Users who write and deploy the smart contracts. Regarding blockchain nodes, neither users, nor nodes, trust any particular node or central authority. Instead, users and nodes trust the consensus results that the blockchain nodes run collectively. For example, in proof-of-work blockchains, the assumption is that more than 50% of the computational capacity is in the control of honest nodes. In blockchains using classical Byzantine fault tolerant consensus algorithms, the assumption is that at most 1/3 of the nodes exhibit malicious behavior, including arbitrarily deviating from the protocol and being silent. The explorations we develop inherit the underlying blockchain's trust model for smart contract execution.

The blockchain nodes do not trust the users. Specifically, malicious users might write rogue smart contracts that: (1) Attempt to escape their execution environments and sniff information, execute arbitrary code, or (2) Break consensus by triggering non-deterministic or platform-dependent execution behaviour such that different (innocent) blockchain nodes arrive at different, conflicting results from executing the same code, or (3) Run indefinitely or longer than "expected". To address the first issue, nodes typically execute smart contracts in sandboxed environments, *e.g.*, Ethereum nodes run the Ethereum virtual machine (EVM). To address the second issue, the sandboxed environment, *e.g.*, EVM, guarantees deterministic execution of smart contracts. To address the third issue, there is typically a limit to the number of instructions that can be executed. Ethereum, for example, uses the concept of gas to price instructions, and nodes stop the execution if the call runs out of gas.

Users might, willingly or not, write non-deterministic code that outputs different results on different nodes. For example, a user might use non-deterministic instructions on different CPU architectures. Since the nodes are decentralized (*i.e.*,

not under the control of a single entity) they can have different CPU architectures. This might prevent nodes to agree on the execution result, which harms the users. The cause could be an honest mistake, which hurts the user. But, it could also be a purposeful denial of service attack: because it prevents consensus on the result, which would be part of a block, it also prevents consensus for all the other transactions in the block, decreasing the overall blockchain throughput.

Common sources of non-determinism. We enumerate some of the many common sources of non-determinism in classical general-purpose execution environments: threads and other parallel programming constructs; operations with unspecified order of operand evaluation; data structures such as maps or sets with unspecified traversal order; architecturally-unspecified instruction behaviour such as undefined condition code outputs; floating-point NaN propagation; the least-significant output bits of transcendental floating-point operations.

We illustrate two strategies in current blockchains that prevent non-deterministic execution. First, nodes should ensure that they all use the same input for the execution, for example by using input data already on the blockchain. Second, to avoid execution non-determinism, platforms such as Ethereum provide a restrictive programming language: Solidity does not support floating-point arithmetic, for example, and EVM supports only a restricted instruction set. With restrictive languages, the tradeoff to enforcing determinism is performance penalty.

2.3 Executing a smart contract

We now describe in more detail the process of running a smart contract, which consists of the following:

(1) Obtaining the smart contract

(2) Obtaining the inputs for an execution of the smart contract

(3) Running the smart contract with the given input

(4) Agreeing on the result

(5) Persisting the result of the execution, which can later be retrieved by the user

Obtaining the smart contract code. Nodes are guaranteed that every node in the system receives the same code for a given smart contract. In other words, a malicious user cannot trick different nodes to run different code for the same smart contract. Because nodes do not trust users, but do trust a majority of the nodes in the network, they obtain the smart contract code through the blockchain instead of directly from the users. The user sends the smart contract to one or more nodes, which then broadcasts it to the other nodes. Using a consensus protocol such as Byzcoin [9], nodes agree on whether they received the same code for the same smart contract; the smart contract may be identified by name. In Byzcoin, a contract that is agreed upon by a majority of nodes has a collective signature that cannot be forged. When a node observes a smart contract with a correct collective signature, it trusts that the code corresponds to the given smart contract.

Obtaining the input. Similar to obtaining the smart contact code, nodes use the blockchain to obtain the current state of a smart contract. Nodes also receive input from users for every execution of a smart contract in the form of a cryptographically signed transaction. Nodes can use this signature to check that the user has the rights to execute the smart contract, for example. In Solidity, for example, a developer can restrict both read access to the contract's state by other contracts and who can call the functions/change the state of the contract.

One thing to note here is that, for efficiency reasons, several input transactions are usually batched and bundled in one block. Otherwise, the overhead of reaching consensus is too large to support high transaction throughput.

Running the smart contract and agreeing on the result. Even if the nodes obtain the correct code and inputs for a smart contract, that does not guarantee that the code itself is safe to execute or that it executes deterministically and produces the same output for all nodes. Nodes typically run such untrusted code in a sandboxed environment. Agreeing on the execution result has two components: (1) The information to reach consensus on, *i.e.*, the smart contract execution result, and (2) The origins of this information, *i.e.*, the different nodes that execute the contract. Preventing non-determinism is the focus of our work, which we describe in the next section.

3 OVERVIEW

This section defines our goals, namely ensuring safe and deterministic execution of smart contracts, and enabling efficient general-purpose computation programming in ubiquitous languages. Then, we review the system workflow. Finally, we explain the challenges in reaching these goals, regarding the level of enforcing determinism, preventing infinite executions, and providing floating point support.

3.1 Motivation and Goals

Non-deterministic executions of smart contracts create security vulnerabilities because they break consensus. When nodes reach different execution outputs because of non-determinism, some of the (innocent) nodes might think that some other of the (innocent) nodes are malicious, because each subset is producing blocks that look "invalid" from the perspective of the other subset. This might raise (false) alarms at the very least. In BFT-type consensus systems, it might stop progress entirely in the case when neither disagreeing subset has a 2/3 threshold supermajority of consensus nodes; In proof-of-work blockchains it can lead to accidental forks of the blockchain, which can persist permanently unless manually fixed (i.e., re-merged).

In fact, "accidental hard forks" due to bugs resulting from divergent behavior have happened before. In Nov 2020, a user intentionally triggered a bug in Geth, an Ethereum client. The bug caused a DoS attack, producing a 30-block minority chain [6].

We argued that deterministic execution of smart contracts is necessary, but existing approaches to deterministic smart contracts incur a high performance cost. Solidity, for example, restricts developers to an extent that becomes counterproductive. Developers struggle to code without floating point type support. Also, developers can only use 256-bit types, which is unnatural given the multitude of types in other languages. Further restrictions come from Solidity's runtime, which is the Ethereum Virtual Machine (EVM). Because EVM restricts the instruction set to ensure determinism, the implementation of some cryptographic functions, such as zero-knowledge proofs, results in prohibitively high gas costs. This means that such contracts are very rare in practice.

Goals. Motivated by the above, our work has the following goals:

- **Deterministic execution.** Enforce deterministic execution of smart contracts. The executor nodes load and run contracts using our sandbox designs, which ensures deterministic execution where possible, or issues alerts of potentially non-deterministic executions.
- **Safe execution.** Nodes safely run untrusted code, without fearing infinite executions or privilege escalation.

- **Versatility.** Enable developers to write smart contracts in ubiquitous languages, such as C/C++, Java and Go.
- **Efficient general-purpose computation.** Support ubiquitous types such as floating point, previously unsupported in smart contracts, and makes possible advanced cryptographic functions that are too costly in Ethereum.

Non-goals. Some determinism features, though potentially interesting in the long term, are out-of-scope for this paper. One such non-goal is handling parallel programming (*e.g.*, threads). Another non-goal is providing support for mathematical (not cryptographic) randomness generation, so that all nodes executing `rand` obtain the same random number.

3.2 System workflow

Our work explores designs that ensures deterministic execution of smart contracts. We provide a description of the general workflow applicable to each design at a high level. One or more developers write a smart contract in a language of their choice, sign the contract to attest its provenance, and then upload the smart contract to the blockchain. The blockchain simply stores the smart contract as code. When a user submits a transaction that triggers a smart contract execution, there are three execution steps that take place. First, the blockchain nodes agree on the execution input and store it on the blockchain. Next, each executor node loads the smart contract in the sandbox, then loads from the blockchain the input and the smart contract state for stateful contracts, and runs the contract. Finally, the executor nodes agree on the result and store the result and the smart contract state on the blockchain.

3.3 Challenges

Fulfilling the goals in the system workflow above poses several challenges, which we describe next.

Level at which to enforce determinism. The first challenge regards the level of enforcing determinism. We explore three approaches in our work: (1) The most general approach is to enforce determinism at the *system level* [7, 12]. This approach has the advantage of ensuring deterministic native execution for all languages that the sandbox supports, for example by ensuring a deterministic multi-threaded environment. But, the disadvantage is that system-level approaches typically have high entry and exit costs for the execution. (2) The least generic approach is *language-level* determinism, whereby one analyzes the API of each supported language and blacklists non-deterministic features such as multi-threading, collection types, etc. This approach seems to be lightweight in terms of static analysis latency, but requires a per-language blacklist, which is tedious. (3) The third approach is an intermediary level between the previous two - any *virtual/software-based instruction architecture* (*e.g.*, EVM, WASM, JVM, LLVM IR) that does not correspond to any "native" hardware-supported instruction architecture (like x86 or ARM) but is also much lower-level than programming language constructs (*e.g.*,, instruction-based rather statement/expression-based). This approach promises a better generality than language-level approaches, at a lower cost than system-level approaches.

Limiting execution cost. The second challenge is to secure the smart contract network against malicious or buggy smart contracts that execute infinitely, or have finite but too long/expensive executions. If there is no mechanism in place to limit the execution duration or resource usage of a smart contract, such an execution can cause a denial-of-service (DoS) attack by using entire network's resources. Ethereum solves this problem by charging a fee for the execution of every operation in a smart contract. More specifically, there is a *gas cost* associated with each EVM opcode. To execute a transaction, the user has to pay enough gas to cover the cost of executing opcodes as part of the transaction. If the

execution of a transaction requires more gas than the *gas limit*, which is a system parameter, then the execution is halted and its changes are reverted. Similarly, we need to instrument the code in order to count executed instructions, however, the challenge is that the instrumentation introduces execution time overheads.

Floating point support. The third challenge is ensuring floating point support. Non-canonical representations of floating point numbers, for example of NaNs, pose a determinism problem when these floats are externalized. The way externalizations take place depends on memory type safety. In languages, IRs or runtimes that ensure type safety, such externalizations only take place through explicit type casts. Intercepting these casts is sufficient to enforce determinism, *e.g.*, externalization of NaNs could be transformed into an externalization of a canonical-form NaN. In environments without type safety, externalization can occur at every memory read or write. Intercepting and checking every memory operation poses a much higher overhead than in type safe environments.

3.4 Lessons learned from our determinism explorations

We provide a summary of the lessons learned through our explorations. We used Unikernels for our exploration of system-level determinism. Our findings indicate that instantiating Unikernels on the fly has prohibitively high startup times, especially when running short smart contracts, but it is practical to deploy at each node long-running Unikernels, one per smart contract, and obtain near-native execution for contracts performing thousands of cryptographic operations. However, the biggest drawback is that Unikernels do not currently enforce determinism at sandbox level.

We built a static checker for Golang to explore language-level determinism. The blacklisting of basic constructs is straight-forward, however, whitelisting libraries can be time consuming, because libraries have many functionalities. In the case of popular libraries with many functionalities, it would be unfortunate to discard the library if some part is non-deterministic. In this case, it might be worth whitelisting just parts of the library. Our Golang static checker has no overhead on the execution, because it performs checks only once, before the contract is run. But, it is language dependent and cannot address unbounded executions.

WASM is our exploration of determinism at virtual/software-based instruction architecture level. WASM is attractive from a generality point of view, because many languages compile to WASM. C-to-WASM transformations have near-native execution speed, but Golang-to-WASM incur 3x overhead. The framework requires a significant effort for development, because some parts are not as mature. Also, it does not ensure type safety, which is likely to incur a high overhead to deterministically externalize NaNs.

The most promising approach seems to be virtual/software-based instruction architecture determinism in JVM/GraalVM, for three reasons. First, JVM ensures type safety, promising a low overhead of deterministic NaN externalization. Second, based on our experience implementing a static checker for Golang, and our initial analysis of the Java API, we are confident in our ongoing development of the Java static checker. Third, our runtime instrumentation of Java programs to prevent infinite executions has a low overhead, running in a near-native time.

Roadmap. The next sections explore the challenges above in the context of smart contracts. Section §4 explores system-level determinism with Unikernels. Section §5 explores language-level determinism in Golang. Section §6 explores determinism in WASM and LLVM IR. Section §7 explores deterministic execution of Java smart contracts within GraalVM, using language-level determinism, JVM instrumentation and ASM bytecode rewriting. The remaining sections discuss the implementation and evaluation of our approaches.

4 SYSTEM-LEVEL DETERMINISM WITH UNIKERNELS

This section explores the design space for system-level deterministic execution of smart contracts. We explore Unikernels as the execution sandbox, because they are small, lightweight, quick, and aim for support for ubiquitous languages. We list our learning about Unikernels determinism support, and about the execution model suitable for smart contracts.

4.1 Determinism support

Unikernels are specialized, lightweight operating systems intended to be used within a virtual machine. Unikernels maintain the safe, sandboxed environment of VMs, but are more lightweight because they are built for a single application. For example, for a smart contract written in a particular language, the Unikernel loads the runtime of that language and the smart contract image. Unikernels are efficient because they provide native execution for several languages.

Unikernels do not currently enforce determinism at sandbox level. System-level or VM-level determinism enforcement techniques such as those used in [7, 12] could in principle be incorporated into a Unikernel execution sandbox, but current Unikernel environments do not have this support. Thus, we make our exploration as a "what if" analysis. A deterministic Unikernel, given the same code and the same inputs, would produce the same execution results, regardless of which machine or architecture executes it. It might support, for example, deterministic multithreading through its scheduler and provide a uniform memory layout for non-type-safe languages. As a potential drawback, deterministic Unikernels might have high entry and exit costs.

Assuming that Unikernels will have a deterministic run mode in the future, we explore efficient ways to launch smart contracts in the next section. We also explore language-level determinism for Golang, which would work with Unikernels when their runtime matures and supports Golang.

4.2 Launching smart contracts

Unikernels are intended for a single process and provide no address space isolation. Thus, it is not safe to run within the same Unikernel several transactions at the same time or several smart contracts. If executed in the same Unikernel, multiple transactions and smart contracts might attempt to read the data or interfere with the execution of other contracts / transactions.

One design question is how to efficiently load and run smart contracts in a Unikernel. We explored three possible approaches. One approach is to take advantage of the low startup time and spawn an execution environment on demand. In this approach, we start a Unikernel by loading the smart contract, pass the input transaction and the contract state from the blockchain, run the transaction, return the result and destroy the Unikernel. The advantage of this approach is flexibility. One simply adapts to the transaction throughput by spawning Unikernels on demand. This approach, however, has two disadvantages. First, although the Unikernel startup time is small, it might be prohibitive for very short smart contract executions. Second, with many incoming transactions that spawn a Unikernel each, a node cannot easily control its resource usage.

A second approach is to statically spawn one Unikernel per existing smart contract. Each node would run one Unikernel per smart contract, and execute transactions as they arrive. This approach has the advantages of no start-up time - transactions simply execute without delay, and it is easier for a node to control its resource usage. However, this

design has a few major disadvantages. First, smart contracts that are rarely called waste resources by being available for execution at all times. If there are many such dormant contracts, nodes waste most of their resources. Second, smart contracts that are very popular may be resource-starved, because the design does not adapt to demand.

A third design is a hybrid approach that attempts to fix the shortcoming of both prior designs. In this approach, each node has a fixed number of Unikernels already running, each Unikernel having the Golang runtime loaded. When a transaction calls a particular smart contract, the next available Unikernel loads that smart contract binary on demand, loads the smart contract state and runs the transaction, and then returns the result and retires the smart contract binary. This hybrid approach combines the advantages of the other two approaches. Running smart contracts has a small start-up time, because one needs to simply load the binary instead of build the entire Unikernel. Also, the nodes can control their resource usage. Finally, it adapts to the workload by loading the necessary smart contract image on demand, possibly on multiple Unikernels if the contract is popular at a given time.

Although the hybrid approach seemed most promising initially, we realized that it is still impractical. Loading the smart contract binary in the hybrid approach takes a similar time to spawning a Unikernel on demand in the dynamic approach. Thus, our design uses a static approach with long-lived Unikernels. Each node instantiates a Unikernel per smart contract, which executes all the transactions of that particular smart contract. Note that optimizations are still possible. For example, instead of running Unikernels forever, a node could implement an "aging" policy to garbage collect old Unikernels that did not recently run transactions. Our implementation relies on the Unikraft [10] toolkit to automate building the Unikernel from an existing smart contract binary.

5 LANGUAGE-LEVEL DETERMINISM IN GOLANG

This section explores the design space for language-level determinism in Golang. We provide a static checker that enforces determinism through a combination of blacklisting non-deterministic API language constructs and whitelisting libraries. This approach promises to be lightweight in terms of static analysis latency.

We design a static checker tool that analyzes smart contract source code for potential non-determinism. For basic types, we compiled a list of blacklisted types that can cause non-determinism: channels, float types, and go statements. We blacklisted `float` types, because NaNs with non-canonical representation have non-deterministic results when externalized, for example through casting to other types. We also blacklisted channels `chan type` and go routines `GoStmt`, because they indicate use of multi-threading with goroutines.

Map types in Golang are a special case. Maps are deterministic as a data type, but map *iteration* with `range` has a non-deterministic iteration order. Because of their popularity in code, we did not blacklist maps; instead we issue a warning only for those maps that are iterated.

For libraries, the search space is prohibitively large to blacklist all standard or third-party libraries containing non-deterministic calls. Instead, we opted for whitelisting those libraries that we have vetted to be deterministic. The list of whitelisted packages contains `fmt, bytes, hash, encoding`. As future work, we envision this list increasing as we analyze more packages.

6 WEBASSEMBLY-BASED EXECUTION ENVIRONMENTS

WASM is a low-level binary code format that is designed to serve as a portable compilation target for various programming languages [5, 8]. Although it was originally designed for enabling high-performance web-browser applications, WASM has certain features that makes it a good candidate for a smart contract execution environment. First, WASM aims at providing near-native performance so it can enable high-performance smart contract execution. Second, since WASM is designed to be portable across different browsers and architectures, it limits non-deterministic execution to a small number of well-defined cases. Third, WASM provides a memory-safe sandbox environment that can safely execute untrusted code. Finally, WASM can support a wide range of popular programming languages, such as C/C++, Rust and Go. The promise of WASM is also recognized by Ethereum network as it has already decided to replace EVM with an Ethereum-flavored WASM implementation called eWASM as part of its transition to Ethereum 2.0.

One of the few sources of implementation-dependent behavior that WASM identifies is the NaN payloads. WASM defines a canonical NaN payload and guarantees that if all NaN values that are passed to an instruction are canonical, then the result is also canonical. However, there are a lot of cases where floating-point instructions in WASM produce non-deterministic NaN bit patterns.

WASM recognizes that NaN-based non-determinism can be circumvented by canonicalizing NaNs before their bit patterns can be observed [3, 4]. Canonicalizing a NaN is the process of converting the bit pattern of the NaN into a single standard bit pattern. The problem with this technique is that it can have a significant performance overhead due to WASM's design. WASM's linear memory is an untyped array of bytes and it is possible to interpret any sequence of bytes as any value. This means that the type of a value stored to memory can be reinterpreted as a different type when loaded from memory, which is an externalization of bit patterns. Therefore, we have to canonicalize NaNs before every memory store operation. However, this approach can add a significant overhead to WASM's performance due to the frequency of memory store operations. eWASM solves the floating-point non-determinism problem simply by eliminating floating-point operations entirely from the instruction architecture.

A potential alternative to WASM for building a smart contract sandbox that enforces determinism at IR level is LLVM IR. However, LLVM IR is not intrinsically portable like WASM and it does not by default restrict non-determinism.

In Section 8.3, we discuss our implementation of a simple WASM-based smart contract execution environment and in Section 9.2 we evaluate the performance of our implementation. The preliminary results show that WASM does have the potential to achieve near-native performance for certain programming languages but it is still not stable enough to be efficiently used outside of web-browser applications.

7 DETERMINISTIC JAVA SMART CONTRACTS

In this section, we discuss the design of a JVM-based deterministic sandbox for smart contracts. We explain how our design addresses the challenges of enforcing determinism at both language and virtual instruction architecture level, as described in Section §3.3.

There are several reasons why it is attractive to build a deterministic smart contract execution environment on top of JVM. First, since JVM is a well-researched VM and there are different implementations of it with performance optimizations, it can enable high-performance smart contracts. Second, as JVM enforces both memory and type safety, it is a good fit for executing untrusted code that can potentially be buggy or malicious. Finally, a JVM-based execution

environment can support various popular programming languages that have good standard library and development tool support, which can simplify implementing smart contracts.

In the rest of this section, we first discuss two approaches for instrumenting Java bytecode to create a deterministic execution environment for smart contracts: instrumenting the JVM (§7.1.1) and rewriting bytecode with ASM (§7.1.2). Then, we explain how we use these instrumentation approaches to enforce bounded execution of smart contracts (§7.2). Finally, we discuss how we enforce floating-point arithmetic determinism (§7.3) and deterministic libraries and language constructs (§7.4) .

7.1 Approaches to bytecode instrumentation

7.1.1 ***Instrumenting the JVM.*** The first approach is instrumenting at the JVM level. For this purpose, we use the Java on Truffle (Espresso) framework [1] of GraalVM. Espresso is a Java bytecode interpreter at its core that is turned into a Just-In-Time (JIT) compiler by using Truffle [2], which enables building programming language implementations as interpreters for self-modifying Abstract Syntax Trees (ASTs). Truffle creates an optimized compiler by performing an automatic partial evaluation of the AST interpreter [13].

Instrumenting Espresso is a convenient approach because instrumentation can be done by changing how JVM bytecode interpreter interprets the bytecodes. This is a significantly simpler approach compared to instrumenting a complex JIT compiler, such as Graal. Also, more generally, instrumenting the JVM is desirable because it is a class-agnostic solution that does not require doing per-class work. It also gives you a fine-granular control over whether you want to instrument at the bytecode-, method- or class-level. The disadvantage of instrumenting the JVM is that it requires intercepting the execution and running the instrumentation code for each bytecode, which can degrade the performance of the JVM.

7.1.2 ***Bytecode rewriting.*** Our second approach is based on bytecode rewriting. In this approach we obtain the bytecode corresponding to a target bytecode class file, and augment the bytecode file by inserting instrumentation code. More specifically, in this approach we utilize the GraalVM as a JDK and modify the bytecode class file corresponding to the smart contract.

For bytecode rewriting, we use the ASM framework. ASM provides a framework that provides interfaces to access the internals of bytecode. ASM also provides two APIs: (1). visitor API and (2) a tree API. Visitor API traverses bytecode as a sequence of events, whereas the tree API represents the bytecode as object constructs. Both APIs can be used for bytecode rewriting, as well as to generate new bytecode.

The main advantage of the ASM approach is that, it can directly take the compiled bytecode and do the necessary changes to add instrumentation, without the need of having access to the original source file. This allows the instrumentation to run separately from the underlying compilation, thus respecting the interface boundaries. The main disadvantage of ASM is that it processes bytecode on a per-class basis: it processes one class file at a time. In practical smart contract execution platforms, it's common to observe smart contracts that span multiple class files, thus it's not trivial how to apply the ASM instrumentation in such cases.

7.2 Enforcing bounded execution

To protect our JVM sandbox from infinite and finite but too long/expensive executions, we count the number of JVM bytecode instructions that are executed during the execution of a Java smart contract. The sandbox internally keeps track of all the Java bytecodes that are executed both directly in the contract code and indirectly through calls to Java libraries. If the count goes above a pre-defined threshold value, we abort the execution of the smart contract. We use the two approaches for bytecode instrumentation that we discuss above to enforce bounded execution.

Instrumenting the JVM. In this approach, we instrument Espresso to intercept an instruction before it is executed and increment the global counter. If the counter value exceeds a threshold value, which can be defined either globally at system level for all contracts or per-contract, then the execution is halted and discarded.

The advantage of this approach is that it does not require any changes to the Java classes that are executed as part of the contract. When we intercept the execution, we simply check for the name of the class that is currently being executed to increment the counter. Since smart contracts in our sandbox environment are placed in a package with the *sandbox.contract* prefix, we check whether the current Java class is part of a package that has this prefix. This approach also simplifies counting the instructions that are executed as part of standard Java libraries. Since smart contracts can only use our shim layer of whitelisted libraries, we can check for the package (or class) prefix (*i.e.*, `sandbox.java.name.class`) to increment the counter for the standard library calls as well.

In our current design, we do not differentiate between instructions, as they all increment the global counter by one. However, in reality, instructions should have different weights based on their computational complexities, similar to how EVM opcodes have different gas costs. We can easily modify our approach to apply different increments to the counter based on the type of bytecode instruction.

Bytecode rewriting with ASM. We implemented the bytecode counting in ASM by adding bytecodes that correspond to incrementing a static integer counter, after each bytecode instruction in the original smart contract bytecode. This naive approach can simply count the number of bytecodes executed at run time. However, this approach has a significant performance overhead due to the addition of count++ operations after each bytecode in the smart contract.

To address this performance overhead, we introduce basic block based bytecode counting. Basic block is a straight line code sequence which has no branches except to the entry and at the end respectively. Basic Block is a set of statements which always execute one after another, in a sequence.

We use the standard three-address code algorithm to identify the basic blocks in a given smart contract. The algorithm has four steps.

- Convert the code into three-address code form.
- A new basic block is begun with the first instruction and instructions are added until a jump or a label is met
- In the absence of jump, control moves further consecutively from one instruction to another.
- Find the leader bytecodes

Following are the rules used for finding leader bytecodes:

- The first three-address instruction of the intermediate code is a leader.
- Instructions which are targets of unconditional or conditional jump/goto statements are leaders.
- Instructions which immediately follow unconditional or conditional jump/goto statements are considered as leaders.

Using this approach, we modify the smart contract bytecode in the following manner.

- Find the instruction that starts a block.
- Let n be the number of instructions in that block.
- To the leader of a block add bytecodes that corresponds to incrementing a static counter

With block based counting the overhead of bytecode counting becomes minimal (as we show in the experiments section) since only four bytecodes (corresponding to count +=n) are additionally executed per block.

7.3 Deterministic floating-point arithmetic

The IEEE-754 standard specifies the arithmetic operations and representation formats for binary and decimal floating-point arithmetic. Even though the results of certain basic arithmetic operations are guaranteed to be the same for all implementations of the standard, there are operations and conversions that are not specified. Therefore, different implementations that follow the standard can obtain different results for the same computation.

In our Java sandbox, we identify and eliminate sources of non-determinism in floating-point arithmetic to safely support it. Java is an ideal starting point for implementing deterministic floating-point arithmetic because, as of Java 17, it restores the always-strict floating-point semantics, which ensures that the same floating-point calculation will produce the same result on every platform.

Although strict floating-point semantics guarantee deterministic floating-point calculations, they do not eliminate all sources of non-determinism. Since the IEEE-754 standard does not specify the exact bit-patterns for NaNs, different platforms can produce different NaN values. Java does have a single canonical NaN value to which it can collapse NaN bit-patterns in certain cases. However, there are also methods in the standard Java library (*e.g.*, `Float.floatToRawIntBits` and `Double.doubleToRawLongBits`) that preserve the bit pattern of NaN values (*i.e.*, no canonicalization). Furthermore, as Java does not distinguish between quiet and signalling NaNs, NaN values can be silently converted to different NaNs.

As we discuss in Section 6, NaN-based non-determinism can be eliminated by canonicalizing NaNs before they can be observed externally. A naive approach would be to canonicalize NaNs after every floating-point operation that produces a floating-point output, but this approach would be very costly. Instead, we make the observation that since Java is a strongly-typed language, a float/double value cannot be assigned to a variable or memory location of a different type without an explicit conversion. Furthermore, the only way different non-canonical NaNs can be externally observed is by converting a float/double to a non-float/double type (*e.g.*, converting to its bit representation via`doubleToRawLongBits`). Therefore, we can rely on Java's type system to canonicalize NaNs only at such conversion points. Limiting NaN canonicalization to a small set of conversion points also means that we do not need to canonicalize any time we store a float/double variable into the memory. This is unlike WASM that requires canonicalizing a float/double before every memory store since it is an implicit conversion to bit representation.

7.4 Enforcing deterministic libraries and language constructs

Similar to other general-purpose programming languages, Java has standard libraries and constructs that are non-deterministic. In our deterministic Java sandbox, we employ two techniques to separately handle the standard libraries and language constructs. Instead of instrumenting the JVM, we use the static checker and library whitelisting techniques that are similar to those described in Section §5. We believe that these techniques are more suitable than instrumentation since the decision of whether a smart contract can be safely executed can be efficiently made before runtime.

First, we discuss non-determinism in Java standard libraries. Some library classes are inherently non-deterministic due to the type of computations they perform. `java.util.Random` and `java.util.Timer` are such classes as they contain random and time-based computations, respectively. Some library classes are not inherently non-deterministic but they contain methods that return non-deterministic results. For example, `java.util.HashSet` and `java.util.HashMap` do not guarantee a deterministic iteration order of their elements.

In our sandbox, we only support a restricted set of whitelisted Java library classes that only contain deterministic operations. We manually compile this list and create a shim layer that interfaces smart contracts to the whitelisted classes. Smart contracts that run in our sandbox can only use the library classes and methods that are accessible via the shim layer. For example, to use the `compare` method of the `Integer` class, a smart contract would import the whitelisted `sandbox.java.lang.Integer`, as opposed to the original `java.lang.Integer`.

Second, we discuss non-determinism due to language constructs. One example is potential non-determinism due to `Object.hashCode` and `Object.toString` methods. Since every class in Java is a descendant of the `Object` class, these methods are automatically inherited by the child class. However, the `Object` class implementations of these methods are not guaranteed to return the same value across different executions. Therefore, a smart contract class should only use these methods if it overrides them in a deterministic way.

We use a static checker in our Java sandbox to analyze the smart contract classes to make sure that `Object.hashCode` and `Object.toString` are used deterministically. The sandbox denies the execution of the smart contract if it fails the check. We can implement two policies in the checker: (1) reject the smart contract if it uses one of these methods and (2) accept the smart contract if it overrides these methods. (2) is more dangerous since the overridden methods can still be non-deterministic but it gives the programmers more flexibility.

8 IMPLEMENTATION

8.1 Unikernels

The Unikernel smart contract execution environment is deployed as a TCP server that listens on incoming TCP connections. Additionally, it mounts a shared folder that is used to pass parameters between Dela, which is the blockchain system, and the Unikernel. During the benchmark evaluation, the blockchain system first writes a transaction on a file, and then notifies the Unikernel by opening a TCP connection to it and sending a "command" message. This message tells the Unikernel smart contract to read the transaction and process it. Once the Unikernel smart contract has done its processing, it writes back the result to the shared folder and notifies the blockchain system using the established TCP connection. The blockchain system finally reads back the result in the shared folder and uses the smart contract output to proceed its blockchain operations. This setup and aforementioned flow is illustrated in Figure 1.

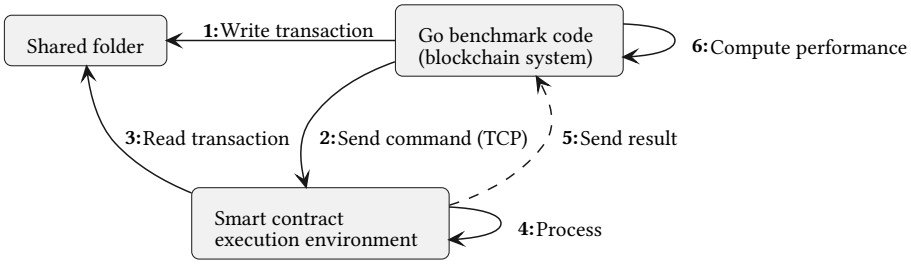

Fig. 1. Setup of the Unikernel benchmark. The go benchmark and the Unikernel use a shared folder to pass the transaction. A TCP connection is used to send notifications between the blockchain system and the Unikernel smart contract execution environment.

The shared folder combined with the TCP connection was the solution chosen among other possibilities due to its flexibility, performance, and known limitations on the networking stack. The rationale for using this combination is out of scope of this document.

Additionally, a real-world blockchain system using the Unikernel smart contract execution environment is deployed. To support the use case, a fully decentralised and blockchain based e-voting system is used, which includes in it the Unikernel support.

Each deployment site needs to deploy a series of components to support the blockchain system. We refer to all deployed components as the deployment package. The deployment package has to be deployed on a specified Unix-based environment. We refer to the machine that hosts the deployment package as the host.

The deployment package is made of two main components: the blockchain node and the Unikernel smart contract execution environment. Those components are deployed via an APT service, which deploys and configures the deployment package. It takes care of installing and configuring the components of the deployment package. The blockchain node component is itself composed of sub-components that provide metrics for monitoring its state, as well as a proxy to handle end-users queries. An operator, managed by EPFL, provides an APT registry to distribute the deployment package. It is also responsible for collecting and monitoring the metrics provided by all blockchain nodes. As such, a host must provide 3 externally accessible services:

- The blockchain node
- The metric service
- The proxy that allows end-users queries

Internally, the blockchain node and the Unikernel smart contract execution environment communicate via the filesystem and an internal TCP connection. The blockchain node also offers a CLI interface for the host operator to perform some manual initial setup steps.

Finally, a client GUI is deployed to offer basic end-users interactions via a Web page. The web page communicates with one of the blockchain nodes via its proxy. The components, their interfaces, and communication schemes are pictured in Figure 2.

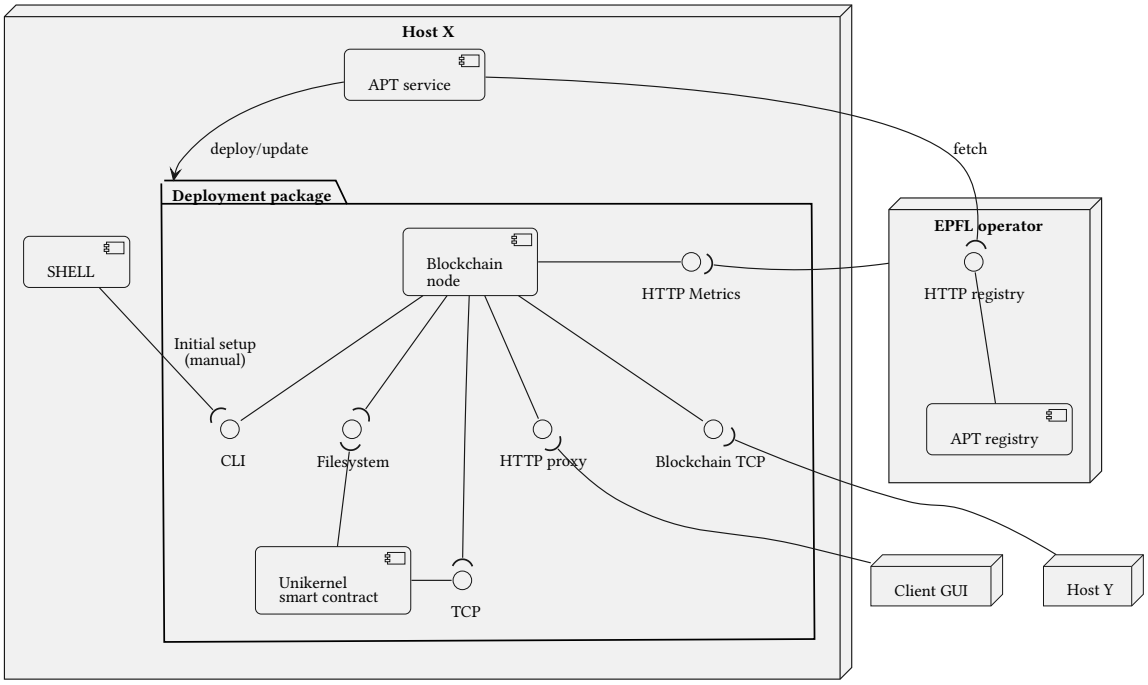

Fig. 2. Component diagram of the deployment setup. Each deployment site must deploy a series of components to support the blockchain system. The set of components needed to be deployed by a site is called the deployment package. Each site deploys on a host a deployment package, which builds up the blockchain system.

8.2 Golang static checker

The Golang static checker analyzes the abstract syntax tree (AST) of the smart contract. Our implementation uses the Golang packages `go/ast`, `go/parser` and `go/token`. The static checker recursively parses all the source files of the smart contract, and issues a warning if the code uses any of the blacklisted types or if it imports any packages besides the whitelisted ones. We blacklisted `float` types, `chan type` and go routines `GoStmt` and map `range` statements. The list of whitelisted packages contains `fmt, bytes, hash, encoding`.

There are two subtleties in the static checker. First, some data types in Golang have multiple ways of declaration, and we need to identify all these to be able to tell the type of a variable. For example, `maps` can be declared just using the keyword `var`, in which case their AST type is `*ast.Ident`. But, maps can also be declared by direct initialization: (i) Either with the keyword `make`, in which case the map variable is the right-hand side of an AST `*ast.CallExpr`; (ii) Or they can be part of a composite expression that simultaneously declares multiple variables, such as `var x, mmmm = 2, make(map[int]int)`, in which case the map variable is part of the right-hand side of an AST `*ast.CompositeLit`. Second, our checker takes a fine-grained approach and issues warnings only for those map variables that are iterated, instead of issuing a generic “range use” warning.

8.3 Smart contracts in WASM

We implement a simple smart contract execution environment in WASM that supports smart contracts written in C and Go [11]. We integrate the WASM-based execution environment with a blockchain framework called Dela. We use the Node.js WASM Interface (WASI) API to establish HTTP communication between WASM and Dela. Our implementation does not include the NaN canonicalization technique described in Section §6.

One of the biggest drawbacks of WASM is due to the fact that it is still a relatively immature system especially when it is used outside of web-browser applications. Due to this reason, we encountered a few difficulties during the implementation phase: (1) As C/C++ treat WASM as a library and Go treats it as an application, the binaries are treated differently based on the language, even after the WASM compilation. Therefore, a lot of the work done for C/C++ could not be used for Go. (2) Emscripten, which is the C/C++ to WASM compiler, requires every necessary library source and header files to be specified. However, building libraries with Emscripten is a tricky process as compilation can fail for unclear reasons when large libraries are imported by the smart contract. This required manual changes to each library. (3) Go to WASM compilation produces very large binaries of several MB, which can be a problem for smart contracts.

8.4 JVM sandbox

We instrument Espresso's source code to count the number of executed instructions. When a class is initialized, we check its name and set a boolean flag if we want to count the instructions of this class. Then, during bytecode execution, we check the flag of the class that is being executed to increment the counter. In our current prototype, we assume that the smart contract does not use any Java libraries so we only set the flag for the smart contract class. Once we port the whitelisted Java libraries to our sandbox, it will be trivial to count their bytecodes as well.

Java ASM. For the implementation of bytecode rewriting, we use Java ASM framework. ASM is a generic Java bytecode manipulation and analysis framework. It is used to modify existing bytecodes directly in binary form. ASM provides bytecode transformations and analysis algorithms which can be extended. ASM has shown to deliver more performance compared to other Java bytecode frameworks.

Java ASM has two main APIs: (1) event based API and the (2) tree based API. Event based API processes a class file by processing different events such as start of a method, invocation of a sub method and so on. In contrast the tree based API creates a tree of nodes in a hierarchical fashion, such that each node corresponds to an element in the program. For example, for the simplest Java program Hello, World, the tree API will have a node for the entire class as the root, a single child of the root as the main method, and one child of the main method node representing the print() call, which will be extended by a node corresponding to the String "Hello, World". For our implementations we used the Tree API.

9 BENCHMARKS / RESULTS

9.1 Performance over different execution environments

We conduct a series of experiments on 5 different smart contract execution environments. Those environments represent common and novel solutions in the blockchain space. We then compare the performance in [ns/op] of an end-to-end smart contract execution environment against an increasing number of operations within the smart contract. We range the number of operations from 1 to 1 million, with an increasing factor of 10.

Results of the performance and comparison of smart contract execution environments are illustrated in Figure 3. It Shows that some smart contract execution environments have an upfront cost, which is amortized with an increasing number of operations. This upfront cost is explained by the communication overhead. This is the case for the Unikernel, Web-assembly, and GraalVM smart contract execution environments. All of which rely on a remote protocol using the networking stack to communicate between the blockchain system and the smart contract execution environment.

In the case of the Unikernel and GraalVM smart contract execution environments, those two eventually outperform the baseline experiment with an increasing number of operations. The baseline experiment is the one using the native smart contract execution environment. The outperformance is mostly explained by the cryptographic library used. In the baseline experiment, a library written in Go has been used, while the Unikernel and GraalVM smart contract execution environments both use the libsodium library. This library benefits from extensive optimization, as well as a native support. The Unikernel smart contract execution environment outperforms the native one at around 3000 cryptographic operations, while the GraalVM does it at around 60. Indeed, the Unikernel smart contract execution environment has a much higher upfront cost, the highest among all execution environments.

In the case of the Ethereum virtual machine (EVM) execution environment, it couldn't perform more than 1000 cryptographic operations, after which its execution required more than the available system memory. This is because EVM has not been primarily designed for intense cryptographic operations.

It must be noted that the Unikernel smart contract execution environment has a known limitation in its networking stack that has the potential of being optimized. It could bring the upfront cost down to the level of the GraalVM smart contract execution environment, if not better. In all smart contract execution environments no extensive optimizations have been applied. A reasonable amount of engineering effort has been invested while developing all the smart contract execution environments.

We conclude that the Unikernel smart contract execution environment positions itself well when a high number of cryptographic operations occurs within the smart contract, should one only focus on the performance. This is not taking into account the additional benefits a Unikernel brings, such as isolation, portability, and security. While performance plays a major role in a blockchain system, it must be carefully balanced among other properties such as, including but not only, the ones mentioned above. This evaluation looks at one particular aspect, but it should always be weighed based on the desired properties of the target blockchain system.

9.2 WASM smart contracts

We compare the execution times of Go and C smart contracts when they are executed natively and as WASM binaries. We use three simple smart contracts: (1) counter incrementing, (2) Ed25519 point addition, and (3) Ed25519 point multiplication. Our results show that the overhead in WASM executions due to the HTTP communication between Dela and the execution environment can be amortized by executing computationally-intensive computations. Our results also show that Go-to-WASM binaries are two or three times slower than the native execution. This is in contrast to the C-to-WASM binaries that are comparable to native execution.

9.3 Instruction counting at JVM level

In this experiment, we measure the overhead of instruction counting at JVM level. To this end, we measure the execution time of a simple smart contract that contains a few basic floating point operations and computes the tenth Fibonacci

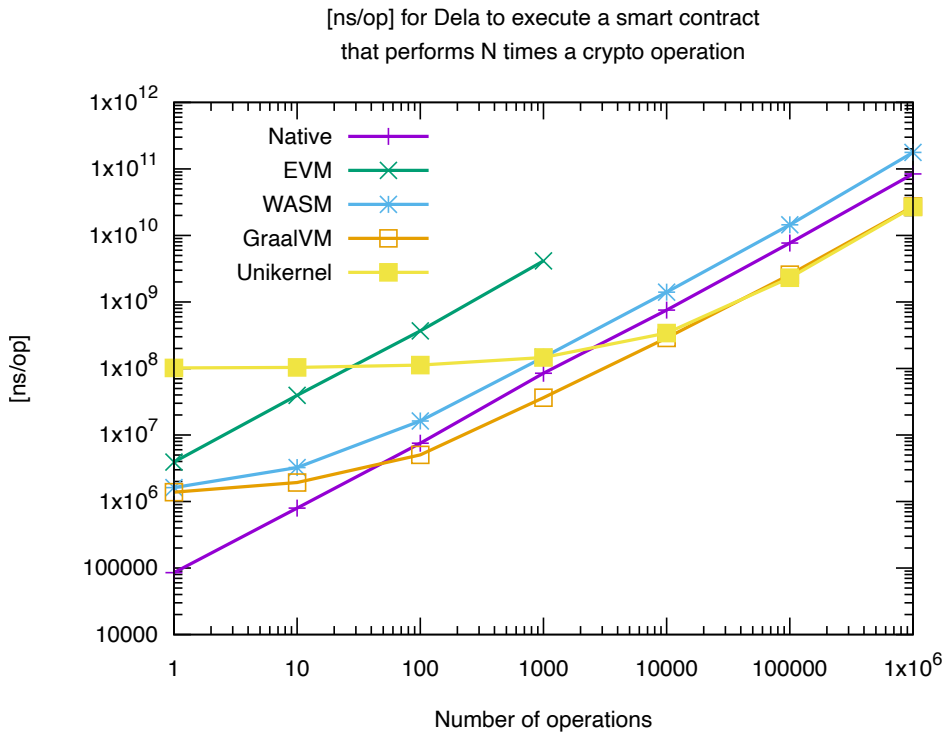

Fig. 3. Comparison on the performance of 5 different smart contract execution environments. The plot displays the performance across an increasing number of operations within the smart contract. The Unikernel smart contract execution environment outperforms the baseline at around 3000 cryptographic operations, and has the highest upfront cost.

number. We carry out the experiment both with and without a warm-up phase where the same contract is executed a number of times before we start recording the execution times. We use the warm-up phase to simulate "popular" contracts that constantly receive transactions and therefore have already been loaded and executed.

We report our results in Table 1. The results are computed over 1 million runs of the smart contract and are in nanoseconds (ns). We use the same number of runs for the warm-up phase. Our results show that it takes 1.21× to 1.64× more time on average to execute a smart contract on the instrumented Espresso JVM with instruction counting compared to the unmodified Espresso JVM.

One thing that stands out in our results is that the maximum execution times are five orders of magnitude larger than the average and 95th percentile values. Since there is a small number of runs with such large values, we hypothesize that these are the "unlucky" runs that coincide with Java's garbage collection. To verify our hypothesis, we conduct the same experiment after disabling garbage collection. To this end, we enable the no-op garbage collector of Java called EpsilonGC. EpsilonGC does not do any garbage collection and therefore does not free any allocated memory. Table 2 shows the results with EpsilonGC enabled. We can see that the maximum execution times are decreased by three orders of magnitude when we disable garbage collection.

Table 1. Execution times of a simple smart contract on the Espresso JVM with and without instrumentation (default garbage collection)

		Espresso JVM	Espresso JVM (Instrumented)
w/ warm-up	Avg (ns)	1387	1685
	95th (ns)	1447	1655
	Max (ns)	149575206	164544234
w/o warm-up	Avg (ns)	1543	2539
	95th (ns)	2246	2636
	Max (ns)	133086212	139021066

Table 2. Execution times of a simple smart contract on the Espresso JVM with and without instrumentation (garbage collection disabled)

		Espresso JVM	Espresso JVM (Instrumented)
w/ warm-up	Avg (ns)	2078	2047
	95th (ns)	4366	4201
	Max (ns)	155289	644346
w/o warm-up	Avg (ns)	2350	2506
	95th (ns)	4581	4869
	Max (ns)	262396	283674

No Counting	Per Instruction Counting	Block Based Counting
4	5	4

Table 3. Execution times of a simple smart contract running on GraalVM with ASM instrumentation. All execution times are in microseconds

The results of our experiments conform with our expectations: instrumenting Espresso for instruction counting is very convenient and simple but it does come with a performance overhead.

9.4 Instruction counting using ASM

In this experiment, we measure the overhead of bytecode counting using the ASM based instrumentation. To this end, we measure the execution time of a simple Java smart contract that recursively adds the numbers from 0 to k, where k is user specified. We conduct the experiments with a warm-up phase where the same smart contract is executed 1,000,000 times before we start recording the execution times. We set k to be 2000 (recursion depth). We verified that both approaches (per instruction counting and block based counting) produce the same bytecode count (20004 for k=2000). The results are the average over 1 million runs of the smart contract.

We report our results in Table 3. The first column corresponds to the execution time without any instrumentation (original smart contract). The second column shows the execution time with per-instruction based counting and the third column shows the execution time with per block counting. All the execution times are in microseconds. We observe that per instruction based counting has a 25% overhead whereas per block counting does not have any overhead.

The results of the ASM experiment confirms our three expectations: (1) Instrumenting ASM for instruction counting is very convenient, (2) naive approach of counting each instruction has a significant overhead and (3) block based instruction counting can count the number of bytecodes without experiencing any additional overhead.

10 CONCLUSION AND ONGOING WORK

In this work, we explored deterministic execution of smart contracts at three levels of enforcing determinism: system level with Unikernels, language level with Golang, and virtual/software-based instruction architecture level with WASM and JVM/GraalVM. Our findings indicate that the Unikernel smart contract execution environment performs close to native speed when a high number of cryptographic operations occurs within the smart contract, should one only focus on the performance. However, Unikernels do not yet ensure deterministic execution. Our Golang static checker has no overhead on the execution, because it performs checks only once, before the contract is run. But, it is language dependent and cannot address unbounded executions. WASM is attractive from a portability point of view and it can perform close to native speed, dependending on language: Go-to-WASM binaries are 2-3 X slower than the native execution, whereas C-to-WASM binaries are comparable to native execution. However, WASM does not ensure type safety, which is likely to incur a high overhead to deterministically externalize NaNs.

The most promising approach seems to be language-level determinism in JVM/GraalVM, complemented with runtime instrumentation, for three reasons. First, our Java smart contracts augmented with runtime instrumentation to prevent infinite executions have a close-to-native performance. Second, based on our experience implementing a static checker for Golang, and our initial analysis of the Java API, we are confident in our ongoing development of the Java static checker. Third, JVM ensures type safety, thus we expect that our ongoing work on deterministic floating point support will have a low overhead of deterministic NaN externalization. Third, based on our experience implementing a static checker for Golang, and our initial analysis of the Java API, we are confident in our ongoing development of the Java static checker.

REFERENCES

[1] Espresso, 2022.

[2] The Truffle Language Implementation Framework, 2022.

[3] WASM Design Rationale, 2022.

[4] Document why NaN bits are not fully deterministic 619, 2022.

[5] WebAssembly, 2022.

[6] Developers Debate Disclosure Protocols After 'Accidental' Ethereum Hard Fork, November 2020. https://www.coindesk.com/tech/2020/11/13/developers-debate-disclosure-protocols-after-accidental-ethereum-hard-fork/.

[7] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM (CACM)*, 2012.

[8] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[9] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[10] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Maxime Sierro. WebAssembly Execution Environment for Dela. 2021.

[12] Weiyi Wu and Bryan Ford. Deterministically deterring timing attacks in deterland. *Timely Results in Operating Systems (TRIOS)*, 2015.

[13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, page 187–204, 2013.