Electronic Communications of the EASST
Volume 85 Year 2025

deRSE25 - Selected Contributions of the 5th Conference for Research Software Engineering in Germany

Edited by: René Caspart, Florian Goth, Oliver Karras, Jan Linxweiler, Florian Thiery, Joachim Wuttke

Reproducible scientific simulations using ideas from distributed ledger technology

Ashwin Kumar Karnad

DOI: 10.14279/eceasst.v85.2687

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Reproducible scientific simulations using ideas from distributed ledger technology

Ashwin Kumar Karnad [1]

[1] a.karnad@fz-juelich.de
Forschungszentrum Jülich

Abstract: Ensuring the reproducibility of scientific simulations is a persistent challenge, despite current best practices like version control and containerization. Factors such as floating-point arithmetic variations, hardware differences, and concurrency issues often prevent bit-for-bit replication of results. This paper investigates the techniques that distributed ledger technologies employ to achieve deterministic computations and application of these techniques to enhance the reproducibility, trustworthiness and verifiability of scientific simulations. We explore two primary approaches: executing simulations directly "on-chain" for complete transparency and deterministic replay, and performing computations "off-chain" while anchoring their integrity to a blockchain via cryptographic proofs, such as Zero-Knowledge Proofs (ZKPs) and Merkle trees.

Keywords: Reproducibility, Scientific Simulation, Distributed Ledger Technology, Blockchain, Deterministic Execution, Zero-Knowledge Proofs

1 Introduction

The reproducibility of scientific simulations remains a significant hurdle in research. While current best practices, such as version-controlled code, meticulous dependency tracking, detailed hardware specifications, and the use of containerization technologies like Docker, have improved the situation, they do not guarantee true bit-for-bit reproducibility. Docker, for instance, relies on the host system's kernel, meaning that updates or changes to the kernel can render previously working simulations inoperable. This is particularly problematic for high-performance computing (HPC) applications, which often have tight couplings with specific kernel modules and system headers.

Furthermore, inherent non-determinism in many computational methods, such as Monte Carlo simulations, poses a challenge. These methods may not produce bit-identical results even when executed with the same code on the same hardware due to factors like floating-point arithmetic nuances, concurrency, and differences in low-level hardware execution [AH24].

We call bit-for-bit reproducibility the ability to re-run a simulation with the same input and obtain exactly the same output, down to the last bit, across different runs which may be on different hardware or software environments [AFH14]. Bitwise reproducibility is particularly crucial for software testing and debugging. Fundamentally, computational testing relies on comparing results of computations either against known reference results or against outputs from alternative, more reliable implementations [khi]. This makes bitwise reproducibility essential for systematic

debugging and validation of scientific simulation codes. Lack of bitwise reproducibility makes it hard to reliably distinguish between a bug in the code and a nondeterministic outcome.

Bit-for-bit reproducibility is also crucial in simulating complex and highly non-linear systems. This is particularly true in fields like climate [TRJ07], astrophysics [KKK+18], and molecular dynamics [WBZC20], where small numerical differences can lead to vastly different outcomes due to the chaotic nature of the systems being modelled.

Bitwise reproducibility is often denoted as "unrealistic" in scientific computing circles, given the practical challenges in achieving it [khi]. There has been some literature exploring strategies for achieving bitwise reproducibility in the context of HPC calculations [AFH14, SMR24], but they are often limited to specific applications.

Distributed ledger technologies such as blockchains require bitwise reproducible computation in order to function. They are thus designed to ensure deterministic execution and verifiable state transitions across a distributed network. This is achieved through strict constraints, including the deliberate omission of floating-point arithmetic in many blockchain virtual machines. Previous efforts to integrate blockchain technology into scientific research have primarily focused on using blockchain for its immutable store of data, essentially as a data management and reputation tracking platform [FDJB17, WWL+21]. This paper takes a different approach by exploring blockchain's solutions to the computational reproducibility problem. We investigate the potential of leveraging concepts and ideas from the blockchain domain to enhance the credibility and reproducibility of scientific simulations. We propose that by adapting techniques used in blockchain systems to ensure deterministic execution and verifiable state transitions, we can address some of the persistent challenges in scientific reproducibility. This includes examining both "on-chain" execution of simulations, where computations are performed directly on a blockchain, and "off-chain" execution with on-chain verification, where simulations run on conventional hardware but their integrity and results are anchored to a blockchain using cryptographic proofs. The core idea is to create a transparent and auditable trail for simulations, thereby increasing trust and verifiability in scientific findings.

This exploration intends to be a conversation starter, aiming to introduce these concepts and foster discussion and collaboration on integrating blockchain-inspired ideas into scientific workflows.

2 The Problem of Reproducibility in Science

The challenge of reproducing scientific experiments is a well-documented issue [Bak16]. A survey published in Nature highlighted that a significant majority of researchers have experienced failures in reproducing experiments, both their own and those of others [Bak16]. This crisis affects the credibility and progress of science. Specific to computational science, the three issues that often arise are

- **Reproducibility:** inability to re-run the calculations and obtain the same results.
- **Authenticity:** discrepancies between the data analysed and the data published.
- **Provenance:** a missing link between the entity performing the analysis and the entity publishing the results.

2.1 Current State of the Art

To combat reproducibility issues, the scientific community has adopted several key practices that form the foundation of modern computational reproducibility [PB21, Ram13]. Version control systems, particularly Git, have become ubiquitous in scientific computing, providing researchers with the ability to track changes in source code meticulously and revert to or inspect previous versions of their simulation software.

Complementing version control, researchers have increasingly focused on dependency management as a critical aspect of reproducible simulations. By explicitly listing and managing software the exact version of dependencies—including libraries, compilers, and runtime environments—scientists can help ensure that simulations are executed in consistent computational environments [AH24]. Tools such as Conda [GCK+18], Spack, and language-specific package managers like pip for Python or npm for JavaScript have become standard practice [The25], allowing researchers to specify exact versions of dependencies and create reproducible software environments.

The documentation of hardware specifications on which the simulations were run also helps in reproducing the results, particularly for performance-sensitive or hardware-dependent computations. Researchers record detailed information about CPU architectures, GPU models, memory configurations, and interconnect technologies, recognizing that these factors can significantly influence computational outcomes and performance characteristics.

Perhaps most significantly, containerization technologies such as Docker and Singularity / Apptainer have revolutionized the reproducibility landscape by attempting to provide one-click simulation setups [CBC25]. These platforms package applications along with their dependencies and necessary configurations into portable containers, theoretically enabling researchers to recreate identical computational environments across different systems and time periods. In order to establish authenticity, research data and the analysis source code are often stored in repositories like Zenodo, Figshare, or institutional repositories [AH24, BNV25]. These platforms provide persistent identifiers (DOIs) and metadata to ensure that the data can be reliably cited and accessed. The problem of provenance is not easily addressed by current practices. Version control systems come close by associating authors to source code changes. Yet they do not inherently provide a clear link between the entity performing the analysis and the entity publishing the results.

2.2 Reasons for Persistent Non-Determinism

Despite these comprehensive measures, achieving bit-for-bit reproducibility remains elusive due to fundamental challenges inherent in modern computational systems. The most pervasive source of non-determinism stems from floating-point arithmetic and hardware differences [STC+24]. While the IEEE 754 standard for floating-point arithmetic [IEE19] provides a framework for numerical computation, it does not guarantee identical results across different hardware architectures or even different compiler optimizations [AH24]. The order of operations, implementation of fused multiply-add (FMA) instructions, and variations in math library implementations can introduce minute differences that accumulate over time in complex simulations, eventually leading to significantly divergent outcomes.

Modern computational approaches further complicate reproducibility through their reliance on concurrency and parallelism. Contemporary simulations routinely leverage multi-core CPUs and GPUs to achieve acceptable performance, but parallel execution can introduce non-determinism if not carefully managed [AH24]. Shared memory access patterns and thread synchronization issues can lead to race conditions where the final outcome depends on the unpredictable timing of threads or processes, making it nearly impossible to guarantee identical results across different execution environments [AH24].

Environmental factors and external dependencies represent another significant challenge to reproducibility. Many simulations depend on external inputs such as real-time data feeds, system clocks, or network conditions, which are inherently difficult to control or replicate perfectly [IT18]. Even seemingly deterministic simulations can be affected by subtle environmental variations that influence their execution.

At the lowest level, hardware-specific execution details contribute to persistent non-determinism in ways that are often invisible to researchers. Subtle differences in CPU microarchitectures, memory controller behaviors, or even firmware versions can influence execution paths and timing, leading to divergent results [AH24, Hir24]. Additional complicating factors include variations in system time handling, random number generation (when seeds are not explicitly controlled), and network latencies in distributed computing environments.

Blockchain systems, despite operating in inherently non-deterministic distributed environments, have been forced to solve these fundamental reproducibility challenges due to the critical requirement that all network participants must reach identical computational results to maintain consensus. The economic stakes and decentralized nature of blockchain networks have driven the development of deterministic execution mechanisms that guarantee bit-for-bit identical results across diverse hardware and network conditions. This paper investigates how these battle-tested blockchain solutions for deterministic computation might be adapted to address the persistent challenges in scientific simulation reproducibility.

3 Blockchain and its solutions to deterministic computation

3.1 Introduction to Blockchain terminologies

A blockchain can be conceptualized as a decentralized, globally shared database. Or more generally, as a globally shared computer. Key characteristics and terms relevant to our discussion include [DHW23]:

- **State Transitions:** Computations on a blockchain, often referred to as transactions, are essentially state transitions. Each valid transaction modifies the blockchain's current state to a new, agreed-upon state.
- **Distributed Ledger:** The state of this global computer is stored in a distributed ledger, replicated across many network nodes. This redundancy provides fault tolerance and transparency.
- **Consensus Mechanism:** Validators (or miners, depending on the blockchain type) are responsible for executing transactions and agreeing on the new state. Consensus mech-

anisms (e.g., Proof-of-Work, Proof-of-Stake) ensure that all honest nodes in the network eventually converge on a single, consistent version of the ledger.

- **Virtual Machines:** Blockchains often use virtual machines to execute smart contracts. These VMs provide a standardized execution environment that ensures deterministic behaviour across all nodes. Unlike containerization technologies such as Docker, blockchain VMs use specially designed bytecode that eliminates all sources of non-determinism.
- **Smart Contracts:** Smart contracts are programs that run automatically on the blockchain when specific conditions are met. They contain code that defines the rules for state transitions and are executed by the virtual machine. These programs essentially contain the application logic of blockchain systems. For example, a smart contract for a digital currency (tokenized asset) would contain functions that define how currency units can be transferred between users, how new currency is created or destroyed, and how ownership records are maintained.

3.2 The Need for Determinism in Blockchain

Determinism is essential for blockchain systems because their consensus mechanisms require all nodes to process transactions and reach identical results. Any variation—whether from floating-point errors, timing differences, or implementation inconsistencies—would prevent agreement on the ledger state. Non-determinism leads to conflicting outcomes across nodes, causing forks that fracture the shared history and compromise the blockchain's integrity as a trustless consensus system.

3.3 How Blockchain Achieves Determinism

Blockchains, particularly those supporting smart contracts (like Ethereum) employ several strategies to enforce deterministic execution:

- **No Floating-Point Instructions:** Many blockchain virtual machines (e.g., the Ethereum Virtual Machine i.e EVM) deliberately omit floating-point arithmetic to avoid its inherent non-determinism. Calculations are typically performed using fixed-point or integer arithmetic.
- **Single-Threaded Execution:** Smart contract execution is generally single-threaded within a transaction context, eliminating parallelism and concurrency issues at the level of individual transaction processing.
- **Controlled External Inputs:** Access to external information (oracles, current time, block numbers) is strictly controlled and provided as part of the transaction's deterministic context. For example, the "current time" is typically the timestamp of the block in which the transaction is included, which is agreed upon by consensus.
- **Controlled Randomness:** True random number generation is problematic. Blockchains often use pseudo-random numbers derived from deterministic sources like block hashes,

or require randomness to be provided from an external, trusted source (an oracle) whose input is then fixed for the transaction.

- **Bytecode Standardization and Deterministic VMs:** Smart contracts are compiled into bytecode that runs on a standardized, deterministic virtual machine. Every node running the VM will produce the same output for the same bytecode and input.

Given these stringent measures for achieving determinism, one could leverage blockchain infrastructure for scientific simulations as we see in the next section, at least for certain types of calculations where absolute reproducibility is paramount

4 Adaptations to scientific simulations based on Blockchain ideas

If blockchains already have the infrastructure to ensure deterministic execution and verifiable state transitions, why not use them for scientific simulations? We explore this idea in the section 4.1. Then we discuss its limitations and consider a hybrid approach in section 4.2.

4.1 Running Simulations "On-Chain"

"On-chain" simulations involve executing the entire simulation logic as a smart contract on a blockchain. The primary benefit lies in achieving complete reproducibility, as every computational step becomes a deterministic state transition that is permanently recorded on the immutable ledger. This mechanism ensures that anyone can re-execute the simulation or verify its past execution to obtain exactly the same results and intermediate states.

Equally important is the establishment of provable provenance and authorship, through blockchain's inherent transaction recording system. Both the deployment of the simulation code (implemented as a smart contract) and the initiation of simulation runs are recorded as blockchain transactions, creating an unalterable audit trail. This provides a verifiable and cryptographically secured link between the simulation, its original author, and the entity reporting the results, addressing the challenge where the connection between analysis and publication can be difficult to establish definitively.

A typical workflow for a researcher interested in conducting an on-chain simulation might look like this:

1. **Develop Smart Contract:** The simulation logic is coded in a smart contract language (e.g., Solidity for Ethereum).

2. **Deploy Contract:** The compiled smart contract is deployed to the blockchain, creating an instance of the simulation.

3. **Execute Simulation:** Users interact with the smart contract by sending transactions that trigger simulation steps or provide input parameters.

4. **Emit Logs/Events:** The smart contract emits events or logs for each significant state transition or result. These logs are stored on the blockchain and can be queried.

5. **Verify Results:** Anyone can inspect the transaction history, the emitted logs, and the final state of the smart contract to verify the simulation's outcome. One can also re-run the block that contains the transaction to reproduce the results exactly.

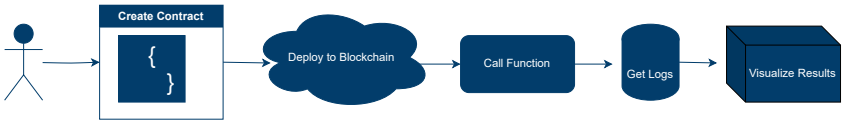

Figure 1: Workflow for running scientific simulations on-chain. The process begins with a researcher creating a smart contract containing the simulation logic. This contract is then deployed to the blockchain network, making it permanently accessible and verifiable. Once deployed, the simulation is executed by calling functions within the smart contract, with each computational step being recorded as a blockchain transaction. The smart contract emits logs and events that capture intermediate results and state transitions, creating an immutable audit trail. Finally, researchers can visualize and analyze the results by querying these logged events from the blockchain, ensuring complete transparency and reproducibility of the simulation process.

An example of this approach can be seen in the example repository [Kar25] accompanying the paper, with simple simulations like Population growth or that of a Bell state simulation implemented on the Ethereum blockchain.

The repository employs a three-tier architecture consistent with established Ethereum development patterns. Each simulation comprises an Ethereum smart contract written in Solidity (stored in the `src/` directory), comprehensive test suites implemented using the Forge framework (in the `test/` directory), and deployment scripts (in the `script/` directory) that demonstrate complete blockchain-based execution workflows. The repository also includes continuous integration for automated format checking and testing.

The core model addresses the reproducibility aspect by implementing state transition logging, where each computational step is emitted as events along with corresponding inputs. This enables complete verification of simulation accuracy through deterministic reconstruction of computational pathways off-chain.

Two representative examples from this repository illustrate this framework:

Population Growth Simulation: The `PopulationGrowth.sol` contract implements a straightforward iterative algorithm executing directly within the Ethereum Virtual Machine. The contract models discrete-time population dynamics using the formula $P_{t+1} = P_t + \frac{P_t \times r}{100}$, where P_t represents the population at time t and r is the growth rate percentage. Each iteration emits a `PopulationUpdate` event containing the year and corresponding population value, creating an immutable computational trace stored on the blockchain. These event logs are then used to make analyses and visualizations off-chain, ensuring that the entire computational history is verifiable and reproducible at any later time.

Bell state Simulation: The `QuantumEntanglement.sol` contract demonstrates a simulation of Bell state creation. To address the floating-point arithmetic limitations of blockchain virtual machines, the implementation utilizes fixed-point arithmetic with a scaling factor of 1000, allowing deterministic representation of quantum amplitudes within Solidity's integer

constraints. The simulation applies sequential quantum gate operations, including Hadamard gate implementation $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ and CNOT gate operations for entanglement creation, with all state vector transformations stored on-chain. This approach addresses the fundamental challenge of floating-point determinism by ensuring bit-for-bit reproducibility across different execution environments through deterministic fixed-point representations.

4.1.1 Limitations of On-Chain Simulations

Despite the benefits, running complex scientific simulations directly on-chain faces significant limitations:

- **Lack of Mathematical libraries:** Blockchain virtual machines often have limited built-in mathematical capabilities. Operations like exponentiation, trigonometric functions, matrix operations, and especially floating-point arithmetic are typically not supported natively or are very inefficient to implement. Random number generation is also constrained.
- **Computational and Storage Costs:** Every computational step and every piece of data stored on a public blockchain incurs a monetary cost (gas fees in Ethereum). Complex simulations with many operations or large state sizes can become prohibitively expensive. Our simple example of Bell state simulation on Ethereum costs around 0.00005 ETH (around 0.13 USD at the time of writing) for a single run.
- **Speed and Scalability:** Blockchain transaction processing is inherently slow compared to traditional computing environments due to the need for consensus. It primarily focuses on security and integrity rather than raw computational speed. This limits the complexity and duration of simulations that can be practically run on-chain. Large-scale simulations would be infeasible.
- **Development Complexity:** Writing efficient and correct smart contracts requires specialized knowledge of blockchain programming languages and paradigms, which can be a barrier for scientists accustomed to traditional programming environments.

4.2 Running Simulations “Off-Chain” with On-Chain Verification

The limitations of on-chain calculations have led to the development of hybrid approaches that allow simulations to be run off-chain while still leveraging the blockchain for verification and integrity checks. In blockchain applications, this approach is commonly used by web3 applications (decentralized applications that interact with blockchain networks) that require complex calculations, such as market making and trading. This approach combines the computational efficiency of traditional hardware with the trust and transparency of blockchain. This often involves Zero-Knowledge Proofs (ZKPs) or other cryptographic commitment schemes.

In contrast to the on-chain execution model, the general workflow for off-chain simulations is:

1. **Off-Chain Execution:** The simulation runs on a powerful off-chain system (e.g., a researcher’s workstation, an HPC cluster).

2. **Generate Proofs:** During or after the execution, the off-chain system generates a cryptographic proof (e.g., a ZKP) attesting to the correctness of the computation according to the specified simulation code and inputs. This proof might also commit to the final results or key intermediate states.

3. **On-Chain Verification:** The compact proof and a commitment to the results are submitted to a smart contract on the blockchain.

4. **Verify Proof:** The smart contract verifies the cryptographic proof. If the proof is valid, the contract records the results as verified, linking them to the off-chain computation.

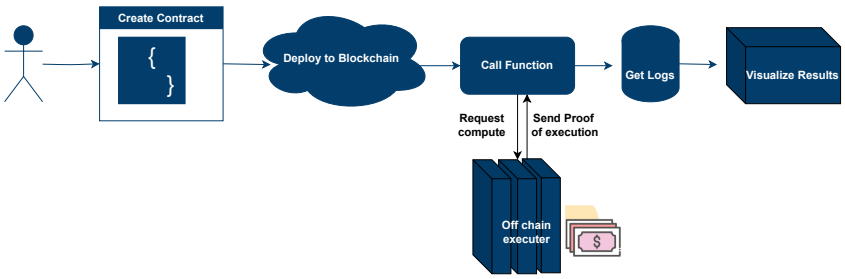

Figure 2: Workflow for running scientific simulations off-chain with on-chain verification. The process begins with the researcher running the computationally intensive simulation on powerful off-chain systems (such as workstations or HPC clusters), where the expensive computational work is performed (denoted by the money symbol). During or after execution, cryptographic proofs are generated that attest to the correctness of the computation according to the specified simulation code and inputs. These compact proofs, along with commitments to the results, are then submitted to a smart contract on the blockchain for verification. The blockchain validates the cryptographic proof without re-executing the original computation, providing verifiable guarantees of correctness while maintaining computational efficiency.

4.2.1 RISC Zero Approach

Zero-knowledge proofs (ZKPs) are cryptographic tools that let one party (prover) prove to another (verifier) that a statement is true without revealing any extra information [GMR89]. They are used for privacy-preserving transactions, secure voting, identity checks, and scaling blockchains [CAD25]. For scientific simulations, ZKPs could be used to prove that a program ran correctly with specific inputs while keeping intermediate steps and sensitive data secret. One such solution that currently exists is the RISC Zero framework [BG+23]. RISC Zero provides a general-purpose zero-knowledge virtual machine (zkVM) that leverages ZKPs for computational verification. This allows arbitrary code (written in languages like Rust or C++) to be executed off-chain, and a ZKP (specifically, a STARK - Scalable Transparent ARgument of Knowledge) is generated that proves the correct execution of that code. It implements a software emulator that runs the RISC-V instruction set and generates cryptographic proofs that verify correct execution independent of the underlying hardware [ris, BG+23].

The following high-level sketch captures the essential ideas without delving into full technical detail, which is beyond the scope of this paper [ris]: The zkVM emulates a RISC-V processor and records an execution trace step-by-step, logging the state of registers, memory, and the program counter at each cycle. For example, executing an ADD instruction produces a trace entry showing how register values change. Each instruction in this trace is then converted into a mathematical equation called an arithmetic circuit, which then gets encoded as a polynomial constraint. For instance, the instruction "ADD R3, R1, R2" at clock cycle t becomes the arithmetic circuit $R3_{t+1} = R1_t + R2_t$ and the following constraint polynomial:

$$C_{\text{ADD}}(X) = R3_{t+1}(X) - \big(R1_t(X) + R2_t(X)\big)$$

over X.

The key insight is that RISC Zero defines a mapping from every instruction in the RISC-V instruction set (e.g., conditional branches, memory loads/stores) to some arithmetic circuit. The entire execution trace becomes a collection of these polynomial equations that must all be satisfied simultaneously. If the program executed correctly, all these equations should equal zero when the correct values are substituted.

To prove correctness without revealing the actual execution data, RISC Zero uses the FRI scheme (Fast Reed-Solomon Interactive Oracle Proofs of Proximity) [BBHR18a] as its polynomial commitment mechanism [ris], which creates cryptographic commitments to polynomials representing the execution trace. The prover commits to polynomials by evaluating them over a large domain and then uses the FRI protocol to prove that these evaluations correspond to low-degree polynomials. This approach allows the verifier (like the STARK protocol in the next step) to check polynomial properties through a series of interactive challenges without requiring the full polynomial data, maintaining privacy while ensuring correctness.

The final step involves the STARK protocol [BBHR18b], which takes these commitments and generates the final proof through a challenge-response process. The system uses public randomness to generate unpredictable test points, and the prover must demonstrate that their hidden polynomials have the correct properties at these points. This involves checking that constraint polynomials equal zero everywhere they should, and that the underlying polynomials have the expected mathematical structure (low degree). The verification process amplifies any errors, making it virtually impossible to create fake proofs.

Finally, the resulting proof, together with the polynomial commitments and public inputs, is submitted to a blockchain smart contract that can verify the proof's validity without re-executing the original computation. For complete technical details, readers should consult the RISC Zero documentation and research papers [ris, BG+23]. This approach allows complex simulations, including those with floating-point arithmetic and extensive libraries, to run off-chain, while still providing strong, verifiable guarantees of correctness on-chain. The down side is that whole code is executed in a VM that emulates RISC V, which is not as efficient as running the code natively on the host machine. Correctness however doesn't imply reproducibility here, as the proof only attests to the fact that the code was run correctly with the given inputs. Inspired by these ideas, we propose a simplified method using Merkle trees, keeping in mind the requirements of reproducibility in the next section.

4.2.2 Utilizing Merkle Trees for State Verification

Inspired by the RISC Zero approach, here we propose an alternative method using Merkle trees, keeping in mind the requirements of scientific simulations. A Merkle tree is a technique that takes in a set of data (in this case, the call stack of a simulation) and produces a single hash (the Merkle root) that represents the fingerprint of the entire dataset [Mer82]. This is done by recursively hashing pairs of data (leaf nodes) until a single hash is obtained. Given the properties of Merkle trees, one can efficiently verify that a specific piece of data (a leaf node) is part of the dataset without needing to reveal the entire dataset. Merkle trees can be used to efficiently verify the integrity of large datasets. Here we use it to verify sequences of state transitions. In order to explain this idea, let us consider the example of a 1D Random Walk simulation. A general simulation can be thought of as a sequence of state transitions, where each transition takes an input state, applies a deterministic function (the simulation logic), and produces an output state. In the case of a 1D Random Walk, the state could be the current position, and the transition function could be a random step left or right based on a coin flip (or some other random choice). Each such state transition can be broken down into smaller state transitions, which can be thought of as atomic operations. An atomic operation is a minimal unit of computation that can be independently verified (See Fig.3).

In the case of the 1D Random Walk, moving from i^{th} position to $(i+1)^{th}$ position is already an atomic operation. But in more complex simulations like Monte Carlo simulations (say of a 1D Ising model), a state transition of moving from one (spin) configuration to another can be broken down into smaller atomic operations, such as flipping a spin, calculating the energy change, and updating the configuration, each of which is individually verifiable. If these atomic operations are formed as function calls in the code, one could then keep a call stack of all these atomic operations. A call stack is traditionally used in computing to track function calls and their execution context, but in our approach, we use it specifically to log only the atomic operations and record the set of variables that each atomic operation affects. This selective logging captures the essential state changes while avoiding the overhead of tracking every computational detail. For example, in the case of a 1D Random Walk simulation, the call stack could look like this:

```
Enter: main()
  Enter: random_walk_step(0, 71876166)
  Exit: random_walk_step(0, 71876166) -> 1
  Enter: random_walk_step(1, 708592740)
  Exit: random_walk_step(1, 708592740) -> 2
  [...]
  Exit: random_walk_step(-7, 1937498036) -> -6
Exit: main() -> 0
```

Thus, each of these entries, along with the source code forms the leaves of a Merkle tree. The Merkle root thus formed would then represent a verifiable fingerprint of the execution path and state (See Fig. 4). By publishing the Merkle root on-chain right after the execution of the simulation, we complete the workflow. One can later prove that a specific state or transition was part of the verified computation by providing a Merkle proof (the path from the leaf to the root). The reproducibility of the simulation is then ensured by the fact that, given the source code,

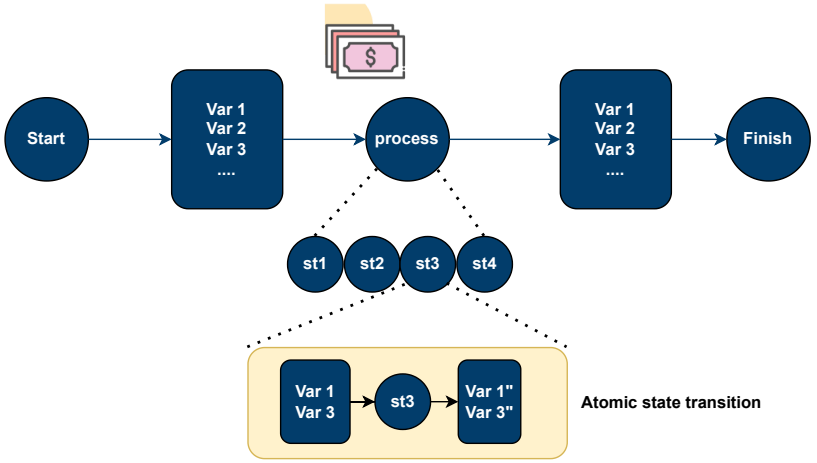

Figure 3: Decomposition of a complex simulation process into verifiable atomic operations. A complete simulation process (top row) transforms an initial state containing multiple variables (`Var 1`, `Var 2`, `Var 3`, etc.) to a final state through a computationally intensive process (denoted by the money symbol indicating high computational cost). The key insight is to break this monolithic process into a sequence of smaller atomic operations (`st1`, `st2`, `st3`, `st4`) that can each be independently verified. Each atomic operation represents a minimal computational step that takes a well-defined input state, applies a specific deterministic transformation, and produces a predictable output state. In this example, for the atomic operation `st3`, variables `Var 1` and `Var 3` get changed throughout the process, while other variables remain unmodified by the atomic operations, demonstrating that not all state variables need to be altered in every step. The call stack used for the Merkle tree construction only uses the variables that change. This decomposition enables fine-grained verification where each individual step can be validated independently, making it possible to verify complex simulations by checking the correctness of their constituent atomic operations rather than the entire monolithic computation.

the input parameters and the call stack (all the leaves of the Merkle tree),one can re-evaluate the simulation as each of the call stack is individually verifiable (as it is an atomic operation whose source is available and by definition is simple to verify independent of the hardware it runs on). Then one could calculate the Merkle root, compare it with the one published on-chain to establish provenance and authenticity of the previously published simulation. In this approach, no verification is done on-chain; only the Merkle root is stored on-chain, which solves the provenance and authenticity problem. The author of the simulation can prove that they indeed did the said simulation with the input and outputs they claimed at the time of publishing the results. The stored Merkle root acts as a cryptographic commitment to this without revealing the entire execution trace on-chain. The reproducibility is still calculated off-chain after the author publishes the source code, input parameters, and call stack by re-evaluating each of the atomic operations in the call stack and comparing the Merkle root with the one published on-chain.

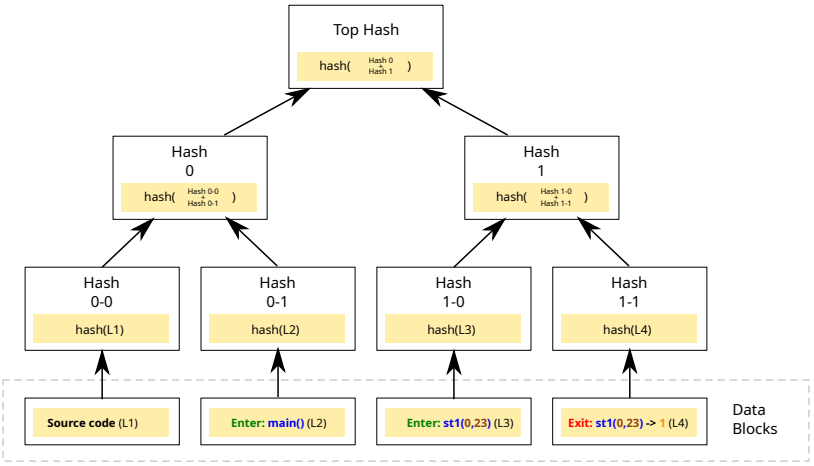

Figure 4: Merkle tree construction from call stack entries. Each leaf node represents the hash of a function call entry (including its inputs and outputs). These leaf nodes (labelled L1 through L4) are then paired and hashed together recursively to form parent nodes until a single hash (the Merkle root) is obtained. This root serves as a compact cryptographic commitment to the entire execution trace. In the diagram, the colours in the leaf nodes of the function call entry indicate different types of information for clarity. Green and red indicate function entry and exit, respectively. Blue indicates the name of the atomic operation (function name). Brown indicates the input parameters, and orange indicates the output of the function call. When publishing results, this root can be stored on-chain, while the full call stack and source code can be stored off-chain. Later, any specific execution step can be verified without revealing the entire execution trace by providing a Merkle proof - the path from that leaf to the root.

In the example repository [Kar25] the complete code for the 1D Random Walk example is

provided (under the `src/1d-RandomWalk` directory). This includes a method to generate the call stack within the CPP code, a code to generate the Merkle root from the call stack, and an example call stack for a sample run.

The `main.cpp` implements a simple 1D Random Walk simulation along with the logic to generate a call stack of atomic operations (in this case `random_walk_step` which is shown in Fig 4 as `st1`). Conditional compilation enables detailed logging of each simulation step, capturing function entries and exits along with their inputs and outputs. This information is stored in `function_trace.txt` (with similar output as the example call stack above), which is then used to construct a Merkle tree and compute the Merkle root using the `merkle.cpp` code. The Merkle root is saved in `merkle_root.txt` which can be published on-chain to attest to the integrity of the simulation run. An example output for `function_trace.txt` and `merkle_root.txt` for 100 steps is also provided in the repository. A make target `run` is provided to compile and run the simulation, generate the call stack, and compute the Merkle root in one step.

Performance analysis: The make target `instrumentation_comparison` simulates 1 million steps and compares the run time with and without the call stack generation. Running this benchmark with no compiler optimisation reveals 7-8x overhead for instrumented versions (2 seconds vs 0.24 seconds for uninstrumented execution on a single core of Intel Core Ultra 5 135U and file stored on a PCIe Gen4x4 NVMe SSD), with trace files reaching 80-90MB for one million-iteration simulations. With compiler optimisations (g++ -O3), the overhead reduces to about 4-5x (0.86 seconds vs 0.21 seconds for uninstrumented execution) with the same trace file sizes on the same machine.

5 Conclusion and Future Outlook

The integration of blockchain principles and technologies presents potential opportunities for enhancing the reproducibility and credibility of scientific simulations, though significant challenges remain to be addressed. On-chain simulations provide unparalleled transparency and bit-for-bit reproducibility but are severely constrained by cost, speed, and computational capabilities (e.g., the absence of floating-point arithmetic and multi-threading). Off-chain simulations with on-chain verification, particularly using technologies like Zero-Knowledge Proofs (e.g., RISC Zero) and Merkle trees, present a more practical path forward. They allow complex, computationally intensive simulations to run efficiently off-chain while providing strong cryptographic guarantees of their correctness and provenance on-chain.

Future work could explore:

- **Standardized Frameworks:** Developing user-friendly frameworks and libraries that simplify the integration of ZKPs and Merkle tree commitments into existing scientific simulation codes.
- **Tokenization of Compute and Data:** Exploring models where computational resources for verified simulations could be traded through blockchain-based marketplaces with transparent utilization tracking. This could be in the form of data centers hosting a blockchain, and the native token of this network representing the cost of computation. So an entity that

wishes to utilize these resources could send transactions (jobs in classical terminology) along with the tokens as fees. Larger transactions cost more, and faster computation also costs more. This could create economic incentives by making computational resources more accessible and fairly priced, while rewarding those who contribute computing power. Additionally, access to high-quality verified simulation datasets could also be tokenized as a secondary benefit.

- **Integration with Existing Scientific Platforms:** Building bridges between blockchain-based verification systems and existing platforms for data sharing, publication, and collaboration. This integration would require developing standardized APIs and middleware that connect blockchain verification infrastructure with established scientific repositories like Zenodo, Figshare, and GitHub. For instance, Jupyter notebook extensions could automatically generate Merkle tree proofs for computational cells, while GitHub Actions could create blockchain attestations during repository releases. Integration with manuscript submission systems could enable automatic verification of computational claims during peer review by linking to blockchain-stored Merkle roots or ZKP proofs.

While challenges remain, the pursuit of blockchain-inspired reproducibility has the potential to significantly increase trust and transparency in computational science. This paper serves as an initial exploration and a call for further research and collaboration in this exciting interdisciplinary field.

Acknowledgements: I would like to thank Prof. Kristel Michielsen and her group for their support and productive discussions. Especially discussions with Dr. Fengping Jin, Pim van den Heuvel, Dr. Vrinda Mehta and Andrea Rava were invaluable in shaping this work. I would like to thank Mohamed Ali Chelbi for his input on ZKPs and the determinism aspect of blockchain. I would finally like to thank the JuRSE Travel Grant for funding the travel to the deRSE 25 conference.

Bibliography

[AFH14] A. Arteaga, O. Fuhrer, T. Hoefler. Designing Bit-Reproducible Portable High-Performance Applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. P. 1235–1244. 2014.
doi:10.1109/IPDPS.2014.127

[AH24] B. Antunes, D. R. Hill. Reproducibility, Replicability and Repeatability: A survey of reproducible research with a focus on high performance computing. *Computer Science Review* 53:100655, Aug. 2024.
doi:10.1016/j.cosrev.2024.100655
http://dx.doi.org/10.1016/j.cosrev.2024.100655

[Bak16] M. Baker. 1, 500 scientists lift the lid on reproducibility. *Nature* 533(7604):452–454, May 2016.

doi:10.1038/533452a
http://dx.doi.org/10.1038/533452a

[BBHR18a] E. Ben-Sasson, I. Bentov, Y. Horesh, M. Riabzev. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.
doi:10.4230/LIPICS.ICALP.2018.14
https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.14

[BBHR18b] E. Ben-Sasson, I. Bentov, Y. Horesh, M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018.
https://eprint.iacr.org/2018/046

[BG+23] J. Bruestle, P. Gafni et al. RISC Zero zkVM: scalable, transparent arguments of RISC-V integrity. *Draft. July* 29, 2023.

[BNV25] R. K. Bhardwaj, M. Nazim, M. K. Verma. Research data management and FAIR compliance through popular research data repositories: an exploratory study. *Data Technologies and Applications* 59(3):472–492, Mar. 2025.
doi:10.1108/dta-12-2022-0477
http://dx.doi.org/10.1108/DTA-12-2022-0477

[CAD25] S. Chaliasos, I. Al-Fath, A. Donaldson. Towards Fuzzing Zero-Knowledge Proof Circuits (Short Paper). In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA Companion '25, p. 98–104. ACM, June 2025.
doi:10.1145/3713081.3731718
http://dx.doi.org/10.1145/3713081.3731718

[CBC25] L. Costa, S. Barbosa, J. Cunha. A Framework for Supporting the Reproducibility of Computational Experiments in Multiple Scientific Domains. 2025.
doi:10.48550/ARXIV.2503.07080
https://arxiv.org/abs/2503.07080

[DHW23] W. Deng, T. Huang, H. Wang. A Review of the Key Technology in a Blockchain Building Decentralized Trust Platform. *Mathematics* 11(1), 2023.
doi:10.3390/math11010101
https://www.mdpi.com/2227-7390/11/1/101

[FDJB17] C. Furlanello, M. De Domenico, G. Jurman, N. Bussola. Towards a scientific blockchain framework for reproducible data analysis. 2017.
doi:10.48550/ARXIV.1707.06552
https://arxiv.org/abs/1707.06552

[GCK+18] B. Grüning, J. Chilton, J. Köster, R. Dale, N. Soranzo, M. van den Beek, J. Goecks, R. Backofen, A. Nekrutenko, J. Taylor. Practical Computational Reproducibility in

the Life Sciences. *Cell Systems* 6(6):631–635, June 2018.
doi:10.1016/j.cels.2018.03.014
http://dx.doi.org/10.1016/j.cels.2018.03.014

[GMR89] S. Goldwasser, S. Micali, C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing* 18(1):186–208, 1989.
doi:10.1137/0218012
https://doi.org/10.1137/0218012

[Hir24] C. Hirsch. Microcode update accelerates desktop processors Intel core ultra 200s. Dec 2024.
https://www.heise.de/en/news/Microcode-update-accelerates-desktop-processors-Intel-Core-Ultra-200S-
html

[IEE19] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, p. 1–84, 2019.
doi:10.1109/IEEESTD.2019.8766229

[IT18] P. Ivie, D. Thain. Reproducibility in Scientific Computing. *ACM Computing Surveys* 51(3):1–36, July 2018.
doi:10.1145/3186266
http://dx.doi.org/10.1145/3186266

[Kar25] A. K. Karnad. Simulation on Ethereum. 2025.
doi:10.5281/ZENODO.17205847
https://zenodo.org/doi/10.5281/zenodo.17205847

[khi] Konrad Hinsen's blog — blog.khinsen.net. https://blog.khinsen.net/posts/2015/01/07/Why-bitwise-reproducibility-matters.html. [Accessed 23-09-2025].

[KKK+18] J. A. Kwiecinski, A. Kovacs, A. L. Krause, F. B. Planella, R. A. Van Gorder. Chaotic Dynamics in the Planar Gravitational Many-Body Problem with Rigid Body Rotations. *International Journal of Bifurcation and Chaos* 28(05):1830013, 2018.
doi:10.1142/S0218127418300136
https://doi.org/10.1142/S0218127418300136

[Mer82] R. C. Merkle. Method of providing digital signatures. Jan. 1982.
https://patents.google.com/patent/US4309569

[PB21] A. Peikert, A. M. Brandmaier. A Reproducible Data Analysis Workflow. *Quantitative and Computational Methods in Behavioral Sciences* 1, May 2021.
doi:10.5964/qcmb.3763
http://dx.doi.org/10.5964/qcmb.3763

[Ram13] K. Ram. Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine* 8(1), Feb. 2013.
doi:10.1186/1751-0473-8-7
http://dx.doi.org/10.1186/1751-0473-8-7

[ris] STARK by Hand — RISC Zero Developer Docs — dev.risczero.com. https://dev.risczero.com/proof-system/stark-by-hand. [Accessed 27-09-2025].

[SMR24] B. Siklósi, G. R. Mudalige, I. Z. Reguly. Enabling Bitwise Reproducibility for the Unstructured Computational Motif. *Applied Sciences* 14(2), 2024.
doi:10.3390/app14020639
https://www.mdpi.com/2076-3417/14/2/639

[STC+24] S. Shanmugavelu, M. Taillefumier, C. Culver, O. Hernandez, M. Coletti, A. Sedova. Impacts of floating-point non-associativity on reproducibility for HPC and deep learning applications. 2024.
doi:10.48550/ARXIV.2408.05148
https://arxiv.org/abs/2408.05148

[The25] The Turing Way Community. The Turing Way: A handbook for reproducible, ethical and collaborative research. 2025.
doi:10.5281/ZENODO.15213042
https://zenodo.org/doi/10.5281/zenodo.15213042

[TRJ07] J. Teixeira, C. A. Reynolds, K. Judd. Time Step Sensitivity of Nonlinear Atmospheric Models: Numerical Convergence, Truncation Error Growth, and Ensemble Design. *Journal of the Atmospheric Sciences* 64(1):175–189, Jan. 2007.
doi:10.1175/jas3824.1
http://dx.doi.org/10.1175/JAS3824.1

[WBZC20] S. Wan, A. P. Bhati, S. J. Zasada, P. V. Coveney. Rapid, accurate, precise and reproducible ligand–protein binding free energy prediction. *Interface Focus* 10(6):20200007, Oct. 2020.
doi:10.1098/rsfs.2020.0007
http://dx.doi.org/10.1098/rsfs.2020.0007

[WWL+21] K. Wittek, N. Wittek, J. Lawton, I. Dohndorf, A. Weinert, A. Ionita. A Blockchain-Based Approach to Provenance and Reproducibility in Research Workflows. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. P. 1–6. 2021.
doi:10.1109/ICBC51069.2021.9461139