





Trustworthy Distributed Certification of Program Execution

Alex Wolf , Marco Edoardo Palma , Pasquale Salza , and Harald C. Gall , *Member, IEEE*

Abstract—Verifying the execution of a program is complicated and often limited by the inability to validate the code's correctness. It is a crucial aspect of scientific research, where it is needed to ensure the reproducibility and validity of experimental results. Similarly, in customer software testing, it is difficult for customers to verify that their specific program version was tested or executed at all. Existing state-of-the-art solutions, such as hardware-based approaches, constraint solvers, and verifiable computation systems, do not provide definitive proof of execution, which hinders reliable testing and analysis of program results. In this paper, we propose an innovative approach that combines a prototype programming language called Mona with a certification protocol OCCP to enable the distributed and decentralized re-execution of program segments. Our protocol allows for certification of program segments in a distributed, immutable, and trustworthy system without the need for naive re-execution, resulting in significant improvements in terms of time and computational resources used. We also explore the use of blockchain technology to manage the protocol workflow following other approaches in this space. Our approach offers a promising solution to the challenges of program execution verification and opens up opportunities for further research and development in this area. Our findings demonstrate the efficiency of our approach in reducing the number of program executions by up to 20-fold, while maintaining resilience against various malicious attacks compared to existing state-of-the-art methods, thus improving the efficiency of certifying program executions. Additionally, our approach handles up to 40% malicious workers effectively, showcasing resilience in detecting and mitigating malicious behavior. In the EQUIVALENTREGISTERSATTACK scenario, it successfully identifies divergent executions even when register values and results appear identical. Moreover, our findings highlight improvements in time and gas efficiency for longer-running problems (scaled with a multiplier of 1,000) compared to baseline methods. Specifically, adopting an informed step size reduces execution time by up to 43-fold and gas costs by up to 12-fold compared to the baseline. Similarly, the informed step size approach reduces execution time by up to 6-fold and gas costs by up to 26-fold compared to a non-informed variation using a step size of 1,000.

Index Terms—Program execution certification, distributed computation, blockchain.

Received 21 February 2024; revised 5 February 2025; accepted 7 February 2025. Date of publication 13 February 2025; date of current version 18 April 2025. This work was supported by Swiss National Science Foundation (SNSF) under Grant SNSF204632. Recommended for acceptance by Á. Beszédés. (Corresponding author: Alex Wolf.)

The authors are with the University of Zurich, CH-8006 Zurich, Switzerland (e-mail: wolf@ifi.uzh.ch; marcoepalma@ifi.uzh.ch; salza@ifi.uzh.ch; gall@ifi.uzh.ch).

Digital Object Identifier 10.1109/TSE.2025.3541810

I. INTRODUCTION

VERIFYING that a program execution produced the correct output given specific inputs is a fundamental challenge in software verification [1], [2]. This task involves confirming that the intended code was executed having the same input, code, and output. However, various risks arise in this confirmation process: the code could be modified by malicious actors, the inputs altered, the correct output might have been generated by a different algorithm than intended, or random errors could occur due to varying configurations. Unlike verifiable computing [3], which focuses on providing proofs for results, verifying complete program executions introduces distinct challenges (e.g. code modifications that do not reflect in the final result) due to the difficulty of ensuring the integrity of all these aspects across the entire execution path. Current state-of-the-art solutions (such as Parno et al. [3], Teutsch et al. [4], and Ben-Sasson et al. [5]) fail to guarantee the authenticity of the execution comprehensively, leaving a gap in reliable verification methods.

This problem spans multiple domains where trust in program executions is critical. In research, reproducibility is essential for validating findings, yet replicating complex processes is often time-consuming and resource-intensive [6], [7]. Despite reproducibility being a core principle of scientific integrity, achieving it consistently remains challenging due to various challenges, such as the significant effort, cost, and resources required, the lack of standardized methods and tools, and the necessity for a thorough understanding of the underlying code [8], [9]. Similarly, in industrial settings, verifying program executions is crucial, whether for confirming that software tests were genuinely conducted or for ensuring that program outputs are trustworthy.

There are several approaches (such as Parno et al. [3], Teutsch et al. [4]) that aim at verifying program execution. A traditional one involves naively re-executing the entire program and carefully scrutinizing each step to confirm that the result matches the anticipated output [2]. However, this method is infeasible for large and complex programs that could potentially run for days or require significant computational resources. Additionally, it cannot differentiate between programs that produce the same output but follow different execution paths, making it unable to verify the intended execution occurred. Moreover, when multiple stakeholders need to verify a program's execution, repeatedly re-running the process becomes infeasible. For instance, during software testing in a customer project, stakeholders, i.e. clients, typically lack reliable methods to confirm that tests were

run on the correct version of the program, forcing them either to acquire significant technical expertise or to trust the developers blindly [10]. Long-running processes, such as batch processing tasks, further discourage re-execution due to time and resource demands [8].

Similarly, in the industrial setting, consider a scenario where a software company delivers a critical application to a client, claiming that it has passed all required tests. The client must verify that the tests were conducted on the correct code version, with accurate inputs, and valid results. However, there are risks that the test results could be from an outdated or modified version of the software, that incorrect inputs were used, or that results were falsified. Both examples lack a method to authenticate the program execution in a feasible way.

Previous work has suggested various approaches to tackle such challenges. For instance, hardware-based approaches, such as Trusted Execution Environments (TEEs), offer tamper-resistant processing environments running on a separated kernel, that ensure the authenticity of executed code, the integrity of runtime states and memory [11], [12], [13], [14]. TEEs provide a solid foundation for secure execution and can be effectively combined with other approaches. However, such systems require the availability of specialized hardware, which can be a significant barrier to their widespread adoption [15]. Even with hardware attestation mechanisms, there is no guarantee that malicious actors would not alter results post-execution [15], [16]. Additionally, TEEs are vulnerable to side-channel and cross-layer attacks, further complicating their reliability [17]. In practice, using TEEs requires fully trusting a single, centralized instance to execute the code correctly, demanding complete confidence in that entity. In the previously mentioned industry scenario, while TEEs could provide a controlled environment for running tests, they fall short of fully addressing verification concerns unless the environment is entirely trusted and safeguarded against both malicious tampering and accidental errors.

Software-based approaches, such as constraint solvers [3], do not require specialized hardware and have the theoretical capability to analyze every line of code in a given constraint. Nonetheless, the verification process would require either a profound understanding of the internals of the code or blind trust in a third party that provides the constraints. Additionally, when dealing with recursive functions or large programs, the number of constraints required would increase exponentially, leading to significant computational and practical challenges [18], [1]. In the industry scenario, verification through constraints would require detailed constraints for each code statement to ensure that the correct code, inputs, and intermediate results were processed accurately. Therefore, the feasibility and ease-of-use of such systems remain questionable at best.

Other software-based approaches, such as verifiable computation systems, face similar challenges as constraint solvers. Verifiers would require deep knowledge of the code to generate and verify the necessary assertions to produce a proof. Additionally, writing assertions depends on an understanding of the code and domain, which requires significant prior knowledge,

time, and effort [19]. Alternatively, one could generate random assertions based on the execution of sample inputs, but those might not be non-trivial assertions.

A potential solution would be to combine the naive approach of re-executing the full program with a trustworthy and immutable environment to persist the result [4]. By doing so, the need for multiple re-executions by every interested party would be eliminated. However, re-executing the full program significantly reduces the usability of such a system due to increased time and computational demands, as a single re-execution could be malicious. As such, multiple re-executions are necessary for the system to achieve trustworthiness.

By considering all the above mentioned limitations, in this paper, we present an innovative approach that combines a prototype programming language, MONA, with a certification protocol. Our prototype language facilitates the segmentation of programs into smaller, more manageable components through our novel Halt and Resume (H&R) approach. When combined with our certification protocol On-Chain Certification Protocol (OCCP), which allows these segments to be certified by re-executing them in a distributed and trustworthy manner—without resorting to naive re-execution—the protocol takes advantage of the immutability and decentralized nature of the underlying blockchain to ensure the reliable certification of program segments. This design enhances robustness by mitigating several potential attack vectors and fortifying the system against malicious activity. Moreover, it can reduce the number of re-executed expressions compared to a naive re-execution of the entire program and can detect executions of different programs that produce identical register values (Equivalent Register Attack (ERA)). Our approach focuses on functional aspects, such as the authenticity of the execution given the code, memory states, and the final result of the computation, leaving non-functional properties like performance or resource consumption out of scope. In line with other approaches in this space, we packaged our protocol utilizing a blockchain setup [4], specifically POLYGON as a layer 2 blockchain, to manage the protocol workflow. Hence, this results in a distributed, immutable, and trustworthy system [20], [21].

We evaluate our approach by comparing the number of re-executed expressions (as it remains unaffected by parallelization) with a naive baseline approach on six popular benchmark problems. Thus, we evaluate the feasibility of our approach focusing on robustness and effectiveness.

To assess the effectiveness and robustness of our approach, we conducted experiments to answer three key research questions: For RQ1 – Program Segmentation, our experiments confirmed that the Mona Interpreter (MI) prototype consistently records and replays program executions across various step sizes and benchmarks. This allows our prototype language to segment an execution into traces and replay any given trace to reproduce the original outcome. We observed a trade-off between trust and performance, where smaller step sizes enhanced trust but required more storage and computational resources, while larger step sizes improved performance at the expense of reduced trust.

For RQ2 – Certification Protocol, we evaluated the effectiveness of OCCP and found it capable of handling malicious scenarios with up to 40% malicious workers, reliably certifying tasks or rejecting them as needed. Additionally, our results demonstrate that our approach can reduce the number of executed expressions by as much as 20 times.

For RQ3 – Informed step size, our experiments show that an *informed* step size reduces both time and gas costs. Specifically, it reduces time by up to 43-fold and gas costs by up to 12-fold compared to the baseline after a scaling multiplier of 1,000, with time savings already observed at a scaling multiplier of 100. However, these reductions do not apply to gas costs for smaller scaling multipliers.

When compared to a non-informed step size variation (see RQ2), we observed up to a 10-fold improvement in time requirements for a step size of 1,000, and gas costs were reduced by up to 6-fold.

Overall, our findings demonstrate that the proposed approach reduces re-execution requirements, enables time and gas cost savings through the use of an informed step size, and exhibits robustness against various malicious attacks, achieving a zero error rate compared to the baseline.

To summarize, the main contributions of this paper are:

- a prototype programming language called MONA, which enables distributed and decentralized re-execution of program segments;
- a certification protocol OCCP that allows for certification of program segments of sequential and deterministic programs in a distributed, immutable, and trustworthy system without the need for naive re-execution;
- an implementation of our protocol using POLYGON as a layer 2 blockchain technology to manage the protocol workflow.

The implementation, benchmark datasets, and results are available in the replication package [22] and published at the address <https://github.com/Lochindaal/occpReplicationPackage>. The MONA language [23] is publicly available at the address <https://github.com/MEPalma/Mona/>.

The rest of the paper is structured as follows. In Section II, we provide a detailed description of our prototype language. Section III presents our proposed on-chain certification protocol. Sections IV and V showcase the experiments, threats to validity, and results. In Section VI, we survey the related work. Finally, we conclude in Section VII with a summary and future work.

II. MONA INTERPRETER

The system introduced in this paper utilizes a unique interpreter, built on top of ANTLR4¹, to verify the reproducibility of specific segments of previous program executions, without the need to re-run the entire program from the beginning. In more formal terms, when a program \mathcal{P} is executed, it produces a sequence of expressions denoted as $\{e_0, e_1, \dots, e_n\}$ along with their corresponding memory states $\{m_0, m_1, \dots, m_n\}$. In this system, the new interpreter is used to verify that evaluating \mathcal{P}

on a specific memory state m , which is obtained after evaluating some t number of expressions, for another p expressions, leads to the memory state m_{t+p} . Importantly, this verification process is conducted without having to re-execute the entire program from e_0 to e_{t+p} . This means that verifying m_{t+p} from m_t involves evaluating just p expressions instead of $t + p$ expressions. The novel Halt and Resume (H&R) mechanism is what provides this functionality, which is also directly responsible for enhancing the efficiency of the proposed system. This efficiency boost enables the system to certify program execution within a distributed setting by re-executing the program just once, overall, a crucial advancement that will be discussed in greater detail later in this paper, including its exploitation for the purpose of verifiability.

In this section, we introduce the Mona Interpreter (MI) as a fully functional interpreter for the MONA programming language. The section also discusses the innovative Halt and Resume (H&R) mechanism integrated into the MI, providing insights into its inner logic. Additionally, an overview of the language features of the MONA language and its assumptions are presented.

Assumptions: Our current implementation of MONA operates under the following assumptions:

- the evaluated application operates sequentially,
- it is deterministic and contains no external API calls with non-determinism.

While MONA is Turing-complete, the current implementation lacks certain features typically found in more mature languages. For example, we currently do not support multi-threading or constructs for iterating using the “in” syntax commonly seen in for-loops. Although for-loops are not directly available, any program requiring them can still be implemented using while-loops, which are fully supported.

A. Mona Language

MONA is a *C-style*, dynamically typed, Turing-complete programming language interpreted by the MI. It offers support for the primitive types: *character* (e.g., 'h'), *string* (as mutable lists of characters e.g., "HelloWorld"), *integer* (e.g., 17), *floating* (e.g., 1.24), and *boolean* (True and False). Moreover, it offers native support for mutable *list* types (e.g., [1, 'a', []]), and related functionalities for list extensions (e.g., [1, 2] + [3]), access (e.g., [1, 2, 3][0]), pythonic slicing (e.g., [1, 2, 3][0:2], [1, 2, 3][1:], [1, 2, 3][:2]), and item assignment (e.g., [1, 2, 3][0] = 0). Any primitive or list values, as well as expressions, can be bounded to a variable identifier through variable declarations (e.g., var identifier = 44;).

The language exposes functionality for the definition of boolean and mathematical expressions respectively (`==`, `<=`, `<`, `>`, `>=`) and (`+`, `-`, `*`, `/`). Boolean expressions can be nested in *C-style if – else if – else* blocks for the computation of conditional logic. MONA supports function declarations with variable input arguments and return values. Function invocations are considered expressions and recursive function calls are supported. Finally, MONA offers support for loops in the form of *C-style while* expressions (e.g., while (func() > min) {print("HelloWorld!")}).

¹<https://www.antlr.org> v4.12.0

B. Halt and Resume

Support for the Halt and Resume (H&R) strategy is integrated throughout various phases of a MONA program's lifecycle. This lifecycle encompasses several stages, beginning with the initial parsing phase, referred to as *Parse-time* followed by the execution of the program, during which the evolution of the memory state is recorded in *Record-time* and concluding with partial replay during *Replay-time*. During *Parse-time* the MI's parser undertakes custom logic to construct the program's AST while incorporating essential annotations to facilitate H&R support. *Record-time* signifies the phase in which the interpreter evaluates the program. Here, after processing a user-defined number of program expressions, execution is halted, and a representation of the memory state at that point is persisted. In the final phase of the program's lifecycle, known as *Replay-time* the MI is capable of loading a specific memory state recorded during *Record-time*. It then resumes program evaluation for a specified number of expressions, subsequently halting and presenting the resulting memory state.

This section presents the Halt and Resume (H&R) strategy and how it seamlessly integrates into the *Parse-time*, *Record-time*, and *Replay-time* stages of a MONA program's lifecycle.

Parsing Mona Programs: During *Parse-time*, the MI's parser employs specialized algorithms to construct the program's AST while simultaneously adding vital annotations essential for supporting the H&R strategy. This annotation process establishes a precise *one-to-one* relationship between program expressions and a unique set of positive integers, known as Sequence Ids (seqids). Each program expression is assigned a specific seqid, and the assignment is carried out in such a way that sorting the expressions based on their corresponding seqid values naturally aligns with the order in which these expressions are evaluated. The MI achieves this binding through a bottom-up parsing approach, systematically assigning seqid values to expressions from an ascending integer counter.

The seqid annotation seamlessly integrates into the AST, irrespective of parsing strategy (bottom-up or top-down). While the choice of bottom-up parsing was for implementation convenience, it does not limit the applicability of the seqid annotation.

Fig. 1 presents a simplified representation of the original AST derived from the MONA program in Listing 1, showcasing the outcomes of preprocessing. This program encompasses a function declaration and a function call designed to print the content of a string literal to the standard output. Notably, each expression within the program is annotated with a unique seqid value, as displayed in the bottom right corner of each grammar rule node in Fig. 1.

The significance of these seqid values becomes apparent when we consider the execution logic within the code block labeled `strlst`. The evaluation of the function in question hinges on the systematic evaluation of its various subcomponents, strictly adhering to the order dictated by their respective Sequence Id (seqid) values. For instance, in the function `strlst` (labeled as 9), the evaluation initiates with the function's parameters (`Params`, seqid 0). Subsequently, it proceeds to evaluate the

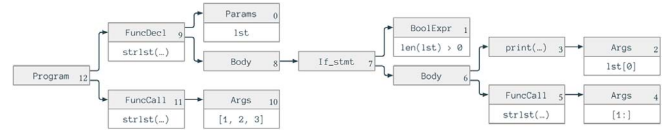


Fig. 1. The Abstract Syntax Tree (AST) of the code in Listing 1.

Listing 1. An example of code in MONA language.

```
decl strlst(lst) {
  if (lenof(lst) > 0) {
    print(lst[0]);
    strlst(lst[1:]);
  }
}
strlst([1, 2, 3]);
```

function's body (`Body`, seqid 8). However, this body is evaluated only after its child `if` expression (`If_stmt`, seqid 7) has been processed. In turn, the `If_stmt` itself is evaluated following the evaluation of its Boolean expression (`BoolExpr`, seqid 1), and subsequently, its own body (`Body`, seqid 6), before ultimately returning a result. Expanding this logical sequence to the two expressions within the body leads to the following order of evaluations, where “evaluated” means that the evaluation has either returned or terminated: `Params` (seqid 0), `BoolExpr` (seqid 1), `Args` (seqid 2), `print` (seqid 3), `Args` (seqid 4), `FuncCall` (seqid 5), `Body` (seqid 6), `If_stmt` (seqid 7), `Body` (seqid 8), `FuncDecl` (seqid 9).

This initial step is crucial for enabling the subsequent *Resume* logic, which intelligently prunes the execution of previously evaluated expressions when loading a specific memory state, contributing to the efficiency of the H&R strategy.

Recording Mona Programs: During the program execution, the MI introduces additional operations compared to conventional interpreters. These operations are designed to modify the memory state in a manner that allows future *Replay* workflows to resume evaluations from the precise expression where the execution was halted, all while bypassing the re-execution of previously processed expressions. The primary objective of this stage in the program's lifecycle is to ensure that the resumption logic operates seamlessly without awareness that a portion of the evaluation tree is being pruned as already executed. To achieve this, the *Record* stage introduces a novel and essential component to the environment's value access and update logic, known as the *memvar*.

The memory state utilized by the MI can be referred to as the tuple (S, M, C, i) . In this tuple S can be considered as a traditional program stack, hence a Last-In-First-Out (LIFO) data structure that serves in exchanging inputs and output between expressions in the same program scope. M is the program's memory, in which the program can read and write key-value pairs, in accordance with program scopes and access policies. C is instead the list of the last executed expression seqids for each open program scope. Finally, i is the integer index value determining which position in C the program is currently evaluating.

The information contained in C is crucial for *Resume* workflows, as it informs the MI about to what depth the evaluation tree was evaluated for each program scope. In more practical terms, during evaluation, after each program expression has been evaluated and returns, its seqid is set as the seqid in C for the current scope index i . For instance, let us consider the program illustrated in Listing 1. At the outset of evaluation, the values of C and i are set to $[-1]$ and 0 respectively, indicating that the program's main scope has not yet executed any expressions. As the program begins, it encounters the first function call, `strlst([1, 2, 3])`. Initially, it evaluates the argument expression, `Args([1, 2, 3])`. Upon completion, this update causes C to become $[10]$. Furthermore, this function call introduces a new scope, where `strlst` is evaluated. Consequently, the values of C and i change to $[10, -1]$ and 1 respectively, signifying that the program is now operating within scope 1, where no expressions have been executed. The `If_stmt` in the function's body causes another scope addition whenever the evaluation unfolds in its body. As the boolean expression `BoolExpr` evaluates to `True` in the current scope, C and i are updated by `If_stmt` to $[10, 1, -1]$ and 2 before evaluating its body. Continuing with the example, as the evaluation of `strlst` proceeds and computes the first print expression, C and i become $[10, 1, 3]$ and 2 respectively. As the recursive call to `strlst` is executed, it updates C and i to $[10, 1, 4, -1]$ and 3, signaling another scope addition. When scopes are eventually closed, as in the case of a return from a call to `strlst`, the last open scope is removed from C , and i is decremented by one step. Consequently, the program concludes with C and i holding the values $[12]$ and 0 respectively, reflecting the scope changes and expression execution throughout the program's evaluation.

While using C and i to keep track of the evaluated tree depth helps the *Resume* logic determine when to prune evaluation, it is insufficient for handling scenarios where the unfolding of evaluation depends on execution data. For instance, *If Blocks* contain multiple expression bodies, of which only one should be executed based on the boolean expression of each *if/else if* conditions. Similar limitations are found in other *C-Style* constructs like loops, where the body is executed based on loop conditions and update expressions. To address such cases, we introduce the concept of *memvars*. This novel strategy connects the program's evaluation components with the program's environment and caches associations between evaluation components and stack values. This ensures that the evaluation always receives the expected value during both execution and resumption. During execution, when a component needs to retrieve the value of a branch decision variable, it invokes the *memvar* to read the stack. To do it, the component provides its seqid and a unique identifier representing the name of the variable being read. This identifier is most relevant when a component accesses multiple such values in its logic thereby ensuring multiple *memvar* do not overwrite each other. The *memvar* uses these values to search in M within the current evaluation scope for the cached output of the variable. If available, it is simply returned to the component; otherwise, the value is fetched from S , stored in M , and then returned. As a result,

during evaluation, *memvar* continually caches values from S into M . However, during *Resume*, these values are retrieved from the cache rather than from S when pruning. This strategy ensures that when resuming, the evaluation unfolds in the correct branch of the sub-program without requiring explicit support within the evaluation components. Importantly, this strategy does not lead to memory leaks in M . When a scope is fully evaluated and closed, it is not only reflected in C but also in M , where all data-related values from the just-closed scope are removed. The *memvar* only retains values that certain evaluation nodes may have used within the currently open evaluation scopes.

While the MI handles the seqid update logic and *memvar* strategies during program execution, it also takes responsibility for creating periodic snapshots of the execution memory. The MI creates periodic snapshots of the execution memory as part of its program evaluation process. To achieve this, it maintains a counter that keeps track of the number of expressions executed. When this counter reaches a user-defined threshold, the evaluation is paused, and the current memory state is saved to disk. The threshold is denoted as the *Step* value, which infers the number of expressions evaluated between *Halt* events. It is worth noting that program start and end snapshots are exceptions and are always recorded, regardless of the expression count. Moreover, unlike the relatively coarse-grained node separation seen in the AST in Fig. 1, the MI produces a significantly more fine-grained node separation. This means that the program can be halted in a much larger number of scenarios, including during the evaluation of expressions.

Replaying Mona Programs: In the final phase, known as *Replay-time*, the MI can be directed to load a specific memory state recorded during the *Record-time* phase. It then proceeds to *resume* program evaluation from the saved memory state, continuing for a specified number of expressions. During this *Resuming* action, the MI traverses the AST but prunes previously evaluated subtrees, utilizing seqids to determine which parts of the evaluation tree should be skipped.

Thanks to the work performed by the MI during *Record-time*, the *Resume* logic presents no significant deviations from traditional evaluation logic. This means that the MI evaluates the program without being aware that it is resuming from a non-clean memory state. Each evaluation node follows a logic wherein, if the value of C at i is greater than its own *seqid*, it will not execute its inner logic but will simply return, effectively pruning its subtree of evaluation. This logic is valid because if the environment's active seqid is greater than that of a node, it indicates that this node has already been evaluated. Simultaneously, *memvar* ensures that when a critical branching value is being read, it is always available.

As a result, the *Resume* logic can reconstruct the evaluation tree by loading the provided memory state into memory and resetting i to 0. The previously executed evaluation tree is pruned, and execution resumes whenever the environment's active seqid is not greater than the current node being evaluated.

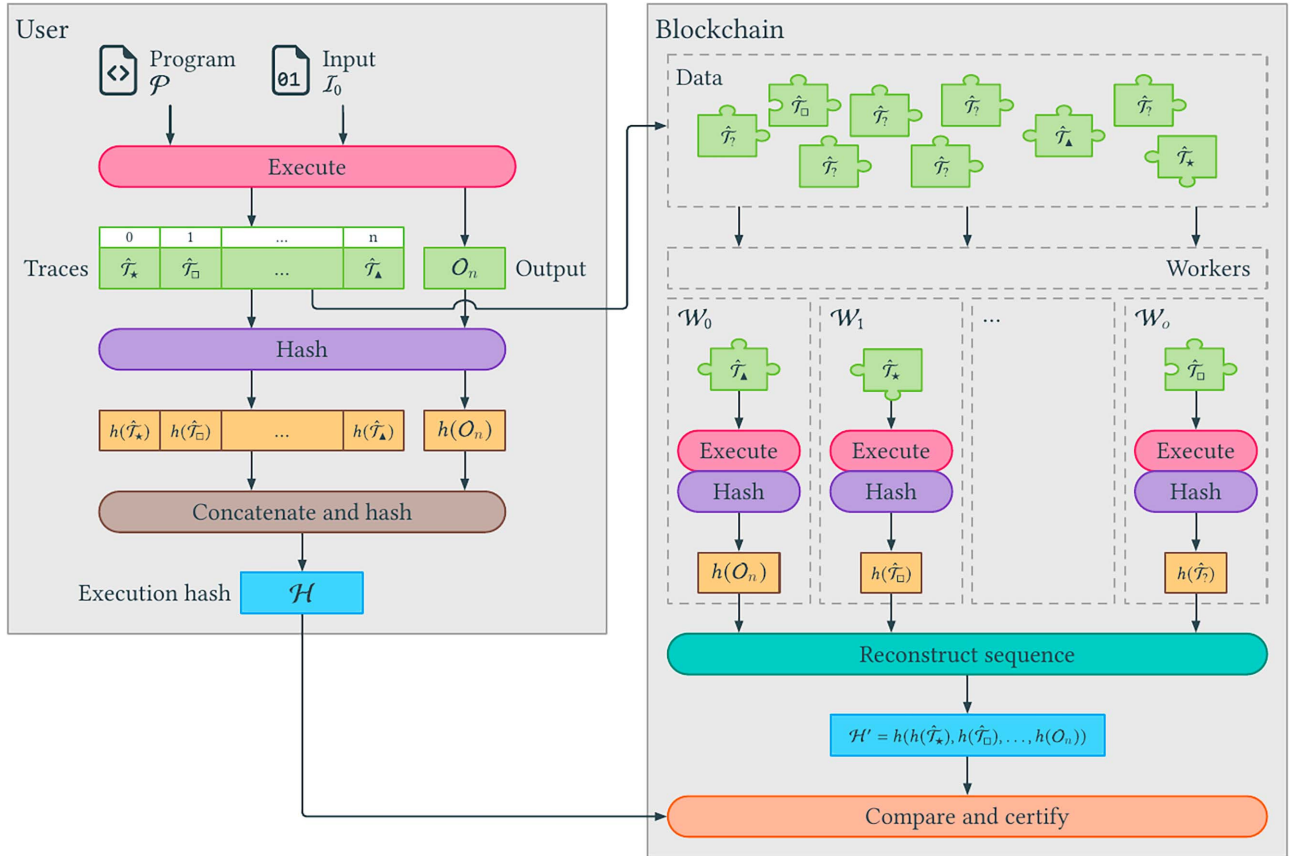


Fig. 2. An overview of the interaction between the user operations and protocol on the blockchain.

III. ON-CHAIN CERTIFICATION PROTOCOL (OCCP)

OCCP combines a certification mechanism with blockchain technology to provide an immutable, decentralized, and trustworthy system for verifying program execution. To enable distributed certification, we leverage the functionality of Halt and Resume (H&R) of MI (see Section II). This feature is essential for splitting program execution into parts or traces that can be independently and in parallel managed by a distributed and trustworthy system. A centralized approach to computing the entire program is unfeasible because it cannot guarantee that every participant will execute their part correctly. With the ability to H&R execution, the interpreter enables the re-execution of all expressions once, as in the initial and original execution of the program, but with the added advantage of parallelization, which reduces the probability of introducing malicious behavior.

The certification of program execution is essentially the problem of connecting each sub-execution of the program to its successor in a distributed manner. We can think of this process as the workers trying to solve a puzzle by finding the correct sequence of sub-executions. To achieve this, the workers in the blockchain provide the output hash of each sub-program. With at least half of the workers in agreement on the output hash, the sequence of the execution memory states can be rebuilt. If it is possible to rebuild the sequence, the sequence is hashed and compared to the one provided by the original executor.

However, if a sequence cannot be rebuilt due to weak consensus among workers regarding trace connections, the corresponding traces are re-evaluated until agreement is reached. Ultimately, the workers either agree that a correct execution sequence has been reconstructed or agree that the given program cannot produce a sequence for the execution traces provided.

In this section, we present OCCP that we have developed for the distributed certification of program execution on blockchain technology. We describe the actors and workflow involved in the protocol, along with how it copes with possible malicious attacks. We also present the implementation details of OCCP using Polygon as layer 2 blockchain technology.

A. Actors and Workflow

A simplified overview of how the approach works is illustrated in Fig. 2. We define a *User* as the entity who wishes to verify the execution of a program. In particular, here we refer to a program \mathcal{P} as an evaluable MONA language derivation. In order to verify an execution of \mathcal{P} , the *User* is expected to execute the Mona Interpreter (MI) in *Record Mode* on \mathcal{P} and produce the set of execution *Traces* \mathcal{T} . This means that the MI is given a fixed and arbitrary number of execution steps after which a *Halt* event is invoked; or in other words the number of expressions between two *Halt* events. Each *Halt* event produces a \mathcal{T} , which is the tuple $(\mathcal{I}_t, s, \mathcal{P}, \mathcal{O}_t)$ where \mathcal{I}_t is the program's memory

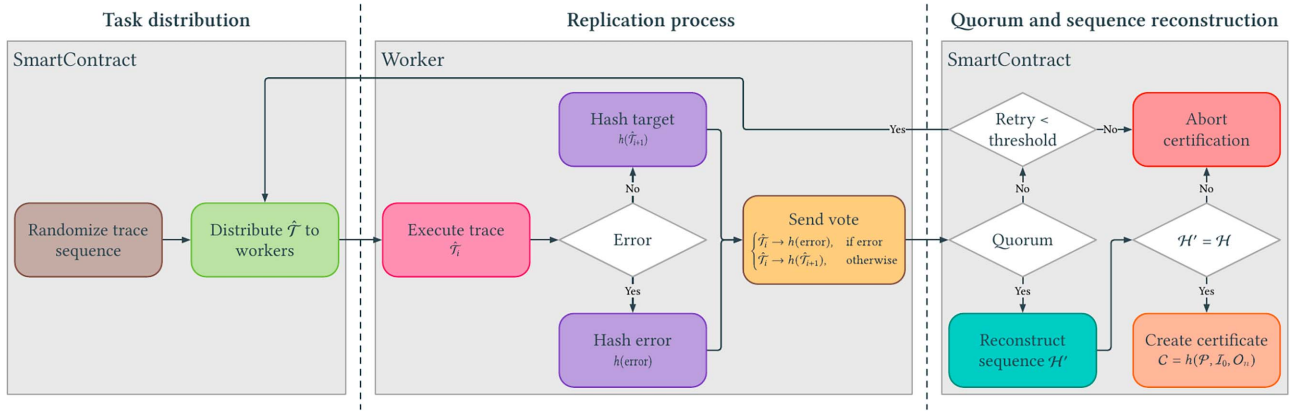


Fig. 3. The detailed flowchart of OCCP.

state at time t , and \mathcal{O}_t is the program's memory obtained after executing P from \mathcal{I}_t for s steps. It is important to note that the recording of the program execution results in an ordered set of $\mathcal{T} : \{\mathcal{T}_i\}$. It is therefore true that $\mathcal{I}_{t+1} \equiv \mathcal{O}_t$. Moreover, the step value s is guaranteed to be constant for all traces but the trace for which \mathcal{O}_t is the program's final memory state, for which s is equal to the remaining number of steps from the penultimate trace.

After the execution recording, the user provides the system with an Execution Hash (\mathcal{H}). It encodes the sequence of memory states encountered during the execution. This is computed by first applying the hashing function h on the sequence $(\mathcal{I}_0, \mathcal{O}_0, \mathcal{O}_1, \dots, \mathcal{O}_n)$ where n is the total number of traces. The resulting hashed values sequence (h_0, h_1, \dots, h_n) is applied again to h to obtain \mathcal{H} . Hence, from \mathcal{T} , the *User* produces $\hat{\mathcal{T}}$, for which $\hat{\mathcal{T}}_i = \mathcal{T}_i \setminus \mathcal{O}_i$ and sends the system a certification request consisting of the tuple $(\hat{\mathcal{T}}, \mathcal{H})$, as illustrated in the *User* section in Fig. 2. Thus, the final state \mathcal{O}_n is not directly passed to the replay phase. The system verifies the execution by delegating to the blockchain's Workers (Ws), the task of computing \mathcal{O}_i for each $\hat{\mathcal{T}}_i$, and obtaining the same \mathcal{H} as the one provided. More specifically, for each trace $\hat{\mathcal{T}}_i$ a worker will compute the resulting memory state \mathcal{O}_i by resuming \mathcal{P} for s expressions from \mathcal{I}_i using the MI.

The following concepts regarding the blockchain side are detailed in the flowchart depicted in Fig. 3. Once a worker obtains \mathcal{O}_i , it replies to the system with the hash of \mathcal{O}_i h_i . This reply represents a *vote*, for which \mathcal{I}_i is the source memory state for the output state described by h_i . Therefore, after the workers voted for each \mathcal{I}_i , the attempts to reconstruct the execution's memory state sequence. Such computation is carried out by the workers until quorum is reached on the sequence value, i.e. of all the responses, the sequence computed by half of the workers plus one, with a simple majority rule implemented to improve decision-making efficiency and responsiveness, while allowing system operators to configure the majority threshold in the smart contract based on specific needs [24], [25], [26], [27]. A W builds such a sequence by first computing to which h_i the workers agree \mathcal{I}_i should map to. We call h_{qi} the output hash with quorum votes for some \mathcal{I}_i . Hence, each \mathcal{I}_i is connected

to some other \mathcal{I}_p for which h_{qi} equals to the hash value of h_p . If a sequence going through each \mathcal{I}_i can be computed, then the system certifies that for this sequence the same hash value \mathcal{H} as the one submitted by the *User* can be computed. Instead, should a complete sequence not be computable, the workers provide the system with the set of traces that do not reach quorum to some output memory state. Thus, the system delegates the computation of such traces to the workers, until a quorum is obtained. Additionally, the system ensures that such traces are not delegated to the workers who worked on the traces previously. Therefore, ensuring that conflicts are resolved by a diverse set of workers. Specifically, each worker is assigned a unique identifier through which we distinguish the workers to ensure this. Every worker is viewed equivalently by the protocol and not further distinguished based on other criteria.

Therefore, we can think of the traces in the form of a puzzle that needs to be solved by the workers. More specifically, the puzzle consists of a very specific directed acyclic graph (DAG), which is structured as a linear chain—a *straight line*. Formally, the DAG is defined as $G = (V, E)$ with N vertices, each representing a trace $\hat{\mathcal{T}}_i$. The goal of the puzzle is to find the correct sequence of connecting the vertices to produce a desired final output \mathcal{O}_n and generating a sequence hash \mathcal{H} that matches the one provided by the *user*. Therefore, each W proposes independently an edge by computing the resulting vertex from the one it got assigned.

Due to obfuscating the target of each trace $\hat{\mathcal{T}}_i$ and ensuring that the correct sequence of traces is unknown to the workers, the only possibility of certifying an execution is through re-execution and matching the newly computed outputs to reconstruct the sequence hash \mathcal{H} as illustrated by Fig. 2.

However, before we reconstruct the sequence, we execute conflict checks. Any vertex that produces a conflict and is therefore not a vertex that has quorum is redistributed to a worker. To formalize this, we first define the following notions.

We have a finite set of vertices, denoted as $V = \{v_1, v_2, \dots, v_N\}$, where each v_i represents a trace. Additionally, we have a set of directed edges, E , where each edge is represented as an ordered pair (v_i, v_j) , indicating a directed edge from trace v_i to trace v_j . These edges represent the voting actions

of workers. Furthermore, the edges in E form a linked list, where each trace is connected to exactly one other trace in the list, with the exception of the last trace, which has no outgoing edge. Formally, for each v_i , there exists at most one v_j such that $(v_i, v_j) \in E$, and for the last trace v_N , there exists no v_j such that $(v_N, v_j) \in E$. While the final output \mathcal{O}_n is excluded from the set of traces \hat{T} , the execution of the final trace by the workers would create a new edge connecting to the final output \mathcal{O}_n , resulting in $|\hat{T}|$ edges. For instance, consider $\hat{T} = \hat{T}_0, \hat{T}_1, \hat{T}_2$, where the correct sequence is $\hat{T}_0 \rightarrow \hat{T}_2 \rightarrow \hat{T}_1 \rightarrow \mathcal{O}_n$. Re-execution of \hat{T}_1 would generate the edge $(\hat{T}_1, \mathcal{O}_n)$, leading to the edge set $E = (\hat{T}_0, \hat{T}_2), (\hat{T}_2, \hat{T}_1), (\hat{T}_1, \mathcal{O}_n)$. Consequently, a valid voting sequence must satisfy $|\hat{T}| = |E|$. In cases of conflicting votes, only edges that achieve quorum are considered valid.

We define a *correct execution* as one that satisfies two conditions: 1) it matches the expressions provided by the user, and 2) it can be reproduced by the workers through the re-execution of traces, resulting in a linear sequence of traces.

Lemma 1: A correct execution results in a DAG represented as a straight line, where $|\text{Traces}| = |\text{Edges}|$.

This formalization clarifies the structure and relationships among the traces, thereby aiding in the analysis and verification of the votes. These checks act as safeguards for Lemma 1, ensuring that before the OCCP reconstructs the sequence, the votes on the traces are evaluated to determine if they lead to a potentially valid sequence.

1) *Check 1: No Vertex is Allowed to Have More Than One Outgoing Connection:* For each vertex v_i representing a trace, there should exist at most one v_j such that (v_i, v_j) represents an outgoing connection: $\forall v_i \in V : |\{(v_i, v_j) \in E\}| \leq 1$. This check ensures that each trace, except the last one, has at most one outgoing connection to the next trace.

2) *Check 2: No Vertex is Allowed to Have More Than One Incoming Connection:* Similarly, for each vertex v_i representing a trace (except the first vertex), there should exist at most one v_j such that (v_j, v_i) represents an incoming connection: $\forall v_i \in V : |\{(v_j, v_i) \in E\}| \leq 1$. This check ensures that each trace (except the first) has, at most, one incoming connection.

3) *Check 3: There Should be no Cycle in the Graph:* To maintain the acyclic nature of the graph, we perform cycle detection. $\forall (v_i, v_j) \in E, |\{v_j \rightsquigarrow v_i\}| < 1$. Where $v_j \rightsquigarrow v_i$ means there exists a directed path from v_j to v_i . This condition ensures that there is no way to return to a vertex v_i starting from its neighbors v_j , i.e., no cycles exist.

Note that quorum is considered for all of the checks. Specifically, if an edge receives multiple conflicting votes from workers, the validation checks will take into account the majority vote or the agreed-upon decision by the quorum. It means that if an edge is voted upon more than once, and the votes conflict, the other conflicting votes will be disregarded in favor of the majority vote. This approach promotes the resilience of the protocol to discrepancies and malicious actors and ensures that decisions regarding edge validity are based on a collective consensus. Therefore, Checks 1, 2, and 3 collectively safeguard

the fulfillment of Lemma 1 and ensure that the resulting graph is indeed a DAG.

4) *Voting and Quorum Example:* To further clarify the voting and quorum mechanism, consider the scenario where four workers, W_0, W_1, W_2, W_3 , are tasked with reconstructing a set of traces $\hat{T}_0, \hat{T}_1, \hat{T}_2$. For simplicity, we omit hashing details and refer directly to the traces. Initially, each worker is randomly assigned a trace: W_0 works on \hat{T}_0 and votes that re-executing this trace leads to \hat{T}_2 ; W_1 votes that \hat{T}_1 results in output O ; and W_2 votes that \hat{T}_2 leads to \hat{T}_1 . This voting yields a trace sequence $\hat{T}_0 \rightarrow \hat{T}_2 \rightarrow \hat{T}_1 \rightarrow O$, where each trace currently has quorum and is ready for verification.

Now, consider an alternative voting sequence where W_0 votes that \hat{T}_0 leads to \hat{T}_1 , while the other workers vote as before. In this case, \hat{T}_1 receives two incoming edges, violating *Check 2* of Lemma 1. Consequently, the OCCP reassigns \hat{T}_0 and \hat{T}_2 to other workers. Worker W_3 re-executes \hat{T}_0 and votes that it leads to \hat{T}_2 , while W_1 re-executes and votes that \hat{T}_2 leads to \hat{T}_1 . This results in quorum with two votes for $\hat{T}_2 \rightarrow \hat{T}_1$. However, this still leaves two outgoing votes from \hat{T}_0 , violating *Check 1*. Therefore, \hat{T}_0 must be re-executed. Achieving quorum for all traces necessitates a worker voting for $\hat{T}_0 \rightarrow \hat{T}_2$.

B. Malicious Attacks Protection

The proposed OCCP protocol is designed to mitigate the impact of malicious actors who may try to undermine the system's trustworthiness. Here are some scenarios in which the protocol can protect against malicious behavior.

1) *Malicious Workers Intentionally Return Incorrect Memory States:* In this scenario, workers may try to tamper with the memory states they return to the system. The protocol can detect such behavior by comparing the sequence hash \mathcal{H} to the constructed one. If they do not match, the system can reassign the task to another worker, and it will be ignored if quorum is reached on another correct result (see also LAZYWORKER case in Section IV).

2) *Malicious Users Submit an Incorrect \mathcal{H} :* In this scenario, a user may try to submit an incorrect \mathcal{H} to trick the system into certifying an incorrect execution. However, the protocol can detect this behavior because the sequence of memory states is included in the \mathcal{H} . If the sequence does not match the one computed by the workers, the system will not certify the execution (see also MALICIOUSUSER case in Section IV).

3) *Equivalent Register Attack (ERA):* In this scenario, a malicious user may try to submit a different program than the one that was executed. This scenario showcases that programs producing the same register values and results at each step can be differentiated by the system. Furthermore, this scenario identifies a gap in existing approaches that produce proofs based on these register values [5], [28], [29]. Hence, it is possible to provide a different implementation that results in the same register states and outcome, rendering such systems unable to provide verification of executions with certainty and ease-of-use.

To further illustrate this case consider two research teams implementations: 1) the first one using a recursive approach, 2) while the other one reads a pre-defined sequence from an array. While both algorithms produce the same register states and outputs they differ in many other aspects (e.g. code, execution trace). Therefore, generating proofs using register states and outputs would not guarantee that a specific algorithm was used to compute said outputs.

The protocol can detect such behavior through the combination of re-executed traces by the workers that vote on the correct path and the fact that MONA records the flow of an application through an AST traversal mechanism. Additionally, MONA records further values (e.g., memory, registers, IO) that make the execution unique. Furthermore, the sequence hash \mathcal{H} includes those further values and would lead to a different hash. Thus, replacing any trace or part of the program would be noticed by the workers and the reconstructed sequence hash would be distinguishable from the originally committed one.

4) *Malicious Workers Collude to Manipulate the Outcome:* In this scenario, a group of workers may collude to manipulate the outcome of the quorum voting process by intentionally returning incorrect memory states. However, the protocol can detect such behavior by relying on the nature of blockchain networks, i.e., we rely on the fact that it is highly unlikely that all the workers are colluding. Therefore, quorum on a sequence must be reached and additionally the sequence hash H must be reproduced for a certificate to be issued.

5) *Malicious Workers Collude With the User to Manipulate the Outcome:* In this scenario, similarly to the colluding workers case, the workers may collude with the User to manipulate the outcome of the quorum voting process. Analogous to the previous case, we rely on the nature of the blockchain network to deal with this issue.

6) *False Negatives and Positives:* Considering the different possible malicious attacks we discuss the impact and likelihood of false negatives and positives.

A false negative occurs when malicious workers produce a non-conflicting trace sequence that aligns with a hash \mathcal{H} different from the intended one. The probability of generating such a valid non-conflicting sequence across the large number of possible graph configurations is low and hinges on achieving a malicious majority vote. The protocol also requires a consistent quorum across traces to finalize certification, making it difficult for adversaries to succeed.

To illustrate the probability of producing a non-conflicting sequence when a majority vote of colluding malicious workers is achieved, consider the following. Let n denote the number of distributed traces $|\hat{T}|$. The number of possible connected graphs using n traces is $2^{\binom{n}{2}}$, while the number of valid non-conflicting sequences is $n!$. The probability of randomly guessing one of these non-conflicting sequences from all possible configurations is $p_{nonConflicting} = \frac{n!}{2^{\binom{n}{2}}}$. However, even if this sequence is found, the success of malicious actors still depends on reaching a majority vote.

Additionally, for a false negative to occur, malicious workers must consistently generate non-conflicting trace sequences across rounds until a threshold t is met. The probability that malicious workers achieve quorum for all traces \hat{T} in the first round, assuming each trace is controlled by a malicious worker, is $p^{|\hat{T}|}$, where p represents the likelihood of a worker being malicious. As the number of traces increases (with smaller step sizes), this probability decreases.

In scenarios where malicious workers collaborate effectively, assuming they operate without internal conflicts and can communicate their intended trace sequences, they must control all assigned traces to avoid conflicts. As workers are excluded from revoting on the same trace, the group with a larger number of members is more likely to dominate the outcome. The probability of malicious workers controlling all traces depends on the number of traces, which is influenced by the step size. A smaller step size increases the number of traces, reducing the chance that a majority of malicious workers are assigned all of them. Conversely, a larger step size decreases the number of traces, increasing the likelihood of malicious control. For instance, if the number of distributed traces is smaller than the total population of workers, the likelihood of malicious workers receiving all traces increases. Overall, a false negative would occur in scenarios where malicious workers consistently produce a non-conflicting and incorrect sequence hash, reaching quorum and generating a reconstructed sequence hash \mathcal{H}' that deviates from the original \mathcal{H} once the threshold t is met.

False positives arise when malicious workers either guess a sequence resulting in a hash collision for SHA-256 or collaborate as a group with a malicious user. In the latter scenario, given a majority, the conditions mirror those faced by non-malicious workers cooperating with an honest user.

The complexities arising from the voting mechanism— such as exclusions and varying communication strategies— warrant more detailed analysis in future research.

Our approach focuses on preventing false positives, as these have a higher impact as this would entail a falsely certified execution. On the other hand, a false negative necessitates a renewed re-execution.

C. Implementation on Polygon Layer 2 Blockchain

We have implemented a *Smart Contract* (382 LOC) that utilizes the OCCP to manage and store all traces \hat{T} for each program \mathcal{P} uploaded. The *Smart Contract* was implemented by using the *Solidity* language². However, in order to improve the efficiency and scalability of our protocol, we propose an adaptation to a layer 2 POLYGON chain. By leveraging the benefits of POLYGON, we can significantly reduce gas fees and increase the transaction throughput for our *Smart Contract*. Additionally, leveraging POLYGON as a layer 2 chain, we can store the final certificate permanently on the underlying ETHEREUM layer 1 chain once the main operations of the protocol have been completed. This approach ensures that the certificate is permanently stored on a more secure and widely adopted blockchain

²<https://soliditylang.org/>: v0.8.2

network. However, it is important to note that this choice of using POLYGON as a layer 2 chain is mainly for the purpose of producing a working prototype, and further optimization and exploration with different chains is left to future work.

To implement our adaptation, we deployed a local POLYGON chain using POLYGON EDGE³. We used HARDHAT⁴, a library that enables compilation, testing, and deployment of smart contracts, to compile and deploy ours. Additionally, we deployed a local AMAZON S3 instance using LOCALSTACK⁵ to store the large amount of trace files. While the use of S3 as a storage solution for the traces may pose a potential security risk, it is important to note that it only serves as a temporary storage solution. The security of the protocol is ensured through the transmission of the hash provided by the user on the blockchain, and the storage of the final certificate, which is also stored on-chain. However, for future optimization, the storage system can be replaced with a more decentralized and secure solution, such as Interplanetary File System (IPFS). This would provide a fully distributed on-chain system, ensuring the highest level of security and immutability for the stored data. Nonetheless, this is left as an optimization for future work on the system.

Furthermore, we use a state-of-practice hash function, specifically *SHA-256*, for the hashing throughout the application. In order to do so we use the *hashlib* library provided by PYTHON.

Finally, each of the *workers* is computing the replay of the individual traces in a PYTHON client using our MI and the WEB3.PY library⁶ to interact with the smart contract.

IV. EXPERIMENTS

To test the feasibility of the proposed programming language (see Section II) and the on-chain protocol (see Section III), we organized the experiments into two parts. In the context of our study, we formulated the following research questions:

RQ1 *Can program executions be segmented into a collection of traces that can be re-executed to reproduce the original result and what impact does this have?*

We evaluate the feasibility of the proposed programming language interpreter MI described in Section II by assessing its ability to produce correct and consistent output results when re-executed from a specific snapshot. Additionally, we measure the performance overhead associated with using the proposed programming language, providing further insight into the feasibility of the approach. By addressing these questions, we aim to provide a comprehensive assessment of the proposed programming language in terms of its ability to produce correct and consistent results, as well as its performance and scalability.

RQ2 *What impact does our proposed approach have on reducing the number of executed expressions, and how does it fare in terms of robustness against malicious acts?*

³<https://github.com/0xPolygon/polygon-edge: v0.7.3-beta1>

⁴<https://github.com/NomicFoundation/hardhat: v2.13.0>

⁵<https://github.com/localstack: v1.4.1.dev>

⁶<https://github.com/ethereum/web3.py: v6.0.0>

TABLE I
USED SYMBOLS

Symbol	Description
S	LIFO stack
M	Program's memory
C	Sequence IDs of last executed expressions for open scopes
i	Index for the current position in C being evaluated
\mathcal{I}	Program input
O	Program output or a specific trace
\mathcal{P}	Evaluable Mona language derivation
\mathcal{T}	Set of traces
$\hat{\mathcal{T}}$	Set of traces without outputs
$\mathcal{T}_i, \hat{\mathcal{T}}_i$	A specific trace with or without outputs
\mathcal{W}	Workers
h	Hashing function
\mathcal{H}	Execution hash
h_n	Hash of input (I) or output (O)
V	Vertices
E	Edges
v_i	A specific vertex

To assess the feasibility and efficiency of our proposed approach, we conduct an analysis to measure the number of executed expressions across various scenarios. These scenarios include ideal conditions (HAPPY), where the system operates as intended, as well as scenarios deliberately designed to mimic malicious acts. By quantifying the executed expressions in these different contexts, our goal is to evaluate the efficiency and resilience of our approach. Furthermore, all scenarios will be compared to the baseline of re-executing the full program multiple times, also referred to as naive re-execution. We aim to provide a comprehensive evaluation of the proposed on-chain protocol and its ability to securely execute programs in a decentralized environment.

RQ3 *How does an informed step size affect the performance of longer-running benchmark problems?*

This research question investigates how an informed step size influences gas costs, certification time, and executed expressions for longer-running benchmark problems. Additionally, we aim to understand the conditions when the overheads are outweighed by the performance gains, offering a deeper understanding of its benefits. Investigating this relationship can provide insights into choosing an appropriate step size and understanding the trade-offs between gas costs, certification time, and computational efficiency. The informed step size is defined as the total number of expressions divided by the number of available workers, ensuring that each worker is assigned at least one trace. This method aims to optimize resource utilization and reduce protocol-related costs. To evaluate this, we simulate extended benchmark scenarios by applying scaling multipliers to the existing benchmark problems. These results are then compared against two alternatives: the non-informed step size approach outlined in RQ2 and the baseline method described in RQ2.

A. RQ1 – Program Segmentation

To answer this question, we split the program's execution into a series of traces and re-execute them in order to reproduce the

original result. Each trace must be capable of reproducing the next state of the program and, when played in sequence, should ultimately produce the same result as running the program without splitting. In other words, each trace \mathcal{T}_i must be connected to the next \mathcal{T}_{i+1} until the final trace is reached in order to ensure that the program's execution is correctly reproduced.

To evaluate the feasibility and effectiveness of the proposed trace-based certification approach for program execution, we defined the following benchmark problems:

- **FIBONACCI**: We compute fibonacci, which serves as a poignant example of recursive algorithms. Additionally, it highlights the importance of optimizing recursive algorithms to avoid inefficiencies (iterative $O(n)$ vs. recursive $O(2^n)$). We compute fibonacci of 17, resulting in 69757 executed expressions.
- **FIBONACCIITERATIVE**: A variant implementation of FIBONACCI that uses an iterative approach. In this case we use 98 as input resulting in 99934 executed expression.
- **MERGESORT**: We evaluate merge sort on a vector of size 142 using a worst-case scenario ($O(n \log n)$), yielding 99856 executed expressions.
- **MATRIXMULTIPLICATION**: We perform matrix multiplication on two matrices of dimensions 11×11 to produce matrix $C = A * B$, resulting in 86781 executed expressions. This benchmark follows established conventions [3], [30], [31], [32] and illustrates a time complexity of $O(n^3)$.
- **SHORTESTPATHFIRST**: We compute the shortest path using the Floyd-Warshall ($O(n^3)$) algorithm on a 13×13 matrix, resulting in 99619 executed expressions. which is used for network routing and matrix inversion, making it a common benchmark in verifiable computation schemes [3].
- **LANCZOS**: We employ the classic Lanczos resampling [33] algorithm, adopted by various approaches [30], [31], [32], to generate a low-resolution image from a high-resolution image. With a varying time complexity between $O(n)$ and $O(n^2)$. Using a 5×5 pixel image as input yields 76128 executed expressions.

First, these problems are well-known and widely used in the field of computer science and programming. This ensures that our results can be compared with existing solutions and evaluated against established benchmarks. Second, problems have varying levels of complexity and computational requirements, allowing us to evaluate the effectiveness of the proposed trace-based certification approach across a range of problem types and sizes. By testing our approach on problems with varying computational requirements, we can gain insights into its scalability and effectiveness across a range of problem types.

For each problem, we conducted a preliminary investigation to select the largest input that would lead to the evaluation of no more than 100,000 expressions. Thus, allowing us to compare the results fairly and judge the validity of each problem based on a definitive result.

To evaluate the impact of different snapshot intervals on the performance of the trace-based certification approach, we compared the output of running the program without snapshots

to replaying the snapshots at different segmentation steps (1, 10, 100, 1,000 and 10,000). We also averaged the execution time over 30 runs to provide a clearer view of how the overhead of the snapshots impacted the runtime.

B. RQ2 – Certification Protocol

To examine the viability of our proposed on-chain protocol for program certification, we leverage the blockchain as a trusted, immutable, and distributed system. We assess the protocol's efficacy by executing all program traces in an unordered manner, with each worker independently determining the result of a given trace (without its output) \hat{T} .

The protocol is engineered to certify tasks only when every provided result can be combined into a chain of traces that, when hashed, produce the hash of the target sequence. This allows us to gauge the effectiveness of the protocol by verifying whether a certificate was produced or not. Furthermore, we assess the efficiency of the protocol by measuring the number of executed expressions and compare it against the baseline, providing a comprehensive evaluation of its performance.

Additionally, we record the gas costs [34] of certifying executions on the chosen blockchain system to gain insights into the viability of running our approach on the blockchain.

For the baseline comparison, we used the MONA language along with a straightforward smart contract. Although alternative programming languages may offer faster execution, they currently lack the halt-and-resume functionality provided by our approach. Comparing various languages (e.g., C versus Java) would inherently result in varying execution speeds, thus, our focus is on developing a reliable certification mechanism rather than focusing on execution speed. Our focus is on the number of executed expressions, as this remains constant regardless of parallel execution. In the baseline setup, each worker re-executes the entire program and votes on the correctness of the output, with certification requiring a majority vote, similar to our proposed method. If certification fails, re-execution is performed up to three times. Each baseline experiment was repeated 30 times under conditions identical to our approach to ensure a consistent comparison.

For the assessment of our proposed on-chain protocol, we opted for the same benchmark problems as for RQ1. Additionally, we have defined four distinct scenarios for program evaluation, namely:

- **HAPPY** case, where no malicious actors participate;
- **LAZYWORKER** case, where one or more of the workers produces an incorrect result;
- **MALICIOUSUSER** case, where the user attempts to certify an erroneous execution.
- **EQUIVALENTREGISTERSATTACK** case, where another program with equivalent register values is submitted to exploit a different execution maliciously.

We conducted 30 runs for each of these cases to ensure statistical relevance. However, we only generated an EQUIVALENTREGISTERSATTACK example for FIBONACCI to showcase the protocol's capability to handle such scenarios. Additionally, we measured the executed expressions in each

scenario to evaluate the protocol's performance. Based on the outcomes of preliminary experiments and feasibility analysis, we selected step sizes of 100 and 1,000 for protocol evaluation. These step sizes strike a balance between accuracy and performance overhead, enabling us to gain insights into the scalability of our proposed approach. Larger step sizes decrease the number of generated traces and computational overhead for each worker but lead to a coarser approximation of the original program execution. Conversely, smaller step sizes offer a more precise approximation of the program execution but increase the number of generated traces and the computational overhead for each worker. For all experiments, we employed 20 workers in parallel to retrieve tasks from the smart contract and replay the assigned traces. Additionally, we utilized 3 workers to verify the proposed results and assess them for conflicts.

1) *Simulation Cases Description:* In the HAPPY scenario, we assume that none of the participating actors act maliciously and measure the efficiency of our approach by comparing the number of executed expressions against the baseline, providing insights into the performance impact of trace-based program certification on a blockchain-based platform. Additionally, we measure the time required for certification to gain insights into the overhead produced by different step sizes and indirectly its impact on the blockchain infrastructure overhead.

In the LAZYWORKER scenario, we introduce up to 40% malicious workers who randomly select one of the other available traces and votes for it instead of replaying the assigned trace. Our approach should still be able to produce a certificate and detect the conflicts introduced by the lazy workers. However, we expect the number of executed expressions for this scenario to be higher than the HAPPY scenario due to the additional computational overhead required to detect and resolve the conflicts.

In the MALICIOUSUSER scenario, we assume that the user produces all the required snapshots to replay the program but intentionally provides a different result than the actual output that is supposed to be certified. For example, the user may provide `fibonacci(17) = 5` instead of the correct result `fibonacci(17) = 1597`. We anticipate the number of executed expressions for this scenario to be similar to the HAPPY scenario since the introduced mismatch is simple and does not require significant additional computation to detect and resolve.

In the EQUIVALENTREGISTERSATTACK scenario, the user furnishes an alternate implementation of the algorithm, generating identical register values as the original version. More precisely, we present an iterative implementation for FIBONACCI in lieu of the recursive approach. We expect the protocol to discern this variation and terminate the certification process. Additionally, we anticipate the number of executed expressions for this scenario to closely resemble those of the MALICIOUSUSER scenario.

C. RQ3 – Informed Step Size

To investigate this question, we assess the impact of using an informed step size on tasks with increased computational demands. Benchmark problems are scaled using scaling multipliers 1, 10, 100, and 1,000, which proportionally increase the

TABLE II
RQ3 SCALED BENCHMARK PROBLEMS

Program	Exec. Exprs			
	1	10	100	1,000
FIBONACCI	69,757	697,552	6,975,502	69,755,002
FIBONACCI ITERATIVE	99,934	999,304	9,993,004	99,930,004
LANCZOS	76,128	761,226	7,612,206	76,122,006
MATRIX MULTIPLICATION	86,781	866,550	8,664,240	86,641,140
MERGESORT	99,856	997,219	9,970,849	99,707,149
SHORTEST PATHFIRST	99,619	996,163	9,961,603	99,616,003

workload by raising the number of executed expressions. However, this scaling is approximate because additional expressions are not always generated when defining functions or referencing existing variables. Details of these variations are provided in Table II.

The informed step size is calculated as $\frac{\text{expressions}}{|W|}$, where $|W|$ is the number of workers, ensuring each worker processes at least one trace. To evaluate its effect, we compare the performance of the informed step size against the naive re-execution baseline from RQ2, examining differences in time, gas usage, and executed expressions. The results from multiplier 1 are also compared to the outcomes of RQ2 without an informed step size to isolate its direct impact.

For these experiments, the parameters from RQ2 are reused with modifications: the step size is adjusted, and the number of reruns is reduced to three (from 30) to accommodate time and resource constraints. Although the informed step size is not necessarily the most efficient solution, this analysis provides useful insights into its trade-offs and informs future optimization approaches.

D. Execution Setup

To ensure the consistency of the experiments, we ran all experiments on the same machine with the same specifications, i.e., Xeon Gold 6126 with 32 vCPUs and 256 GB of RAM. The virtual machine was hosted on a cloud computing platform with dedicated resources, ensuring that there were no performance fluctuations due to shared hardware or resource contention. Additionally, the machine was running Ubuntu 22.04 LTS, and all experiments were conducted using PYTHON v3.10.6.

V. RESULTS

This section presents the results obtained from the experiments conducted in Section IV. For each research question, we provide detailed insights on the proposed approach and the corresponding validation method used for the experiments. In comparing observations, the Kruskal-Wallis test revealed overall group differences, and post hoc Mann-Whitney U rank tests confirmed pairwise distinctions, all with p-values ≤ 0.05 . Significance at $\alpha = 0.05$ level was established. Effect sizes were computed using the Rank-Biserial Correlation coefficient (r),

TABLE III
RQ1 RESULTS – AVERAGE RUNTIME IN SECONDS FOR 30 ITERATIONS

Step Size	Avg. Time (s)			Space Requirements	
	Execution	Recording	Replay	Num. Snapshots	Avg. Size (KB)
FIBONACCIITERATIVE					
1	0.223	148.750	227.446	99,935	4.008
10	0.223	14.895	22.661	9,995	4.008
100	0.223	1.720	2.401	1,001	4.010
1,000	0.223	0.391	0.442	101	3.988
10,000	0.223	0.256	0.265	11	3.910
FIBONACCI					
1	0.180	60.519	127.933	69,758	3.667
10	0.180	6.280	12.703	6,977	3.667
100	0.180	0.819	1.444	699	3.657
1,000	0.180	0.257	0.313	71	3.597
10,000	0.180	0.197	0.206	8	2.882
MERGESORT					
1	0.255	545.916	592.353	99,857	21.221
10	0.255	54.520	56.779	9,987	21.220
100	0.255	5.459	5.429	1,000	21.215
1,000	0.255	0.791	0.762	101	20.836
10,000	0.255	0.305	0.310	11	10.623
MATRIXMULTIPLICATION					
1	0.224	327.370	402.173	86,782	14.121
10	0.224	32.667	38.833	8,680	14.118
100	0.224	3.298	3.777	869	14.103
1,000	0.224	0.549	0.571	88	13.851
10,000	0.224	0.268	0.276	10	11.976
SHORTESTPATHFIRST					
1	0.265	943.317	808.596	99,620	20.394
10	0.265	94.804	81.469	9,963	20.394
100	0.265	9.647	8.200	998	20.368
1,000	0.265	1.102	0.837	101	20.143
10,000	0.265	0.361	0.342	11	17.955
LANCZOS					
1	0.219	594.447	588.258	76,129	14.715
10	0.219	59.865	57.526	7,614	14.714
100	0.219	6.146	5.888	763	14.699
1,000	0.219	0.804	0.762	78	14.449
10,000	0.219	0.281	0.277	9	12.585

yielding an r value of 1, signifying a substantial and statistically significant difference between the samples' distributions.

A. RQ1 – Program Segmentation

Our experiments indicate that the Mona Interpreter (MI) effectively records and replays accurate results across all step sizes. To verify that each trace \mathcal{T}_i produces the correct output \mathcal{O}_i we ran all the traces \mathcal{T} in sequence and compared the produced output to the oracle, i.e., the hash of the next trace \mathcal{T}_{i+1} . Additionally, our results suggest that the runtime for execution, recording, and replay decreases as the step size increases for all programs. However, a trade-off exists between trust (low step size) and performance (high step size). While a step size of one provides accurate and trustworthy results, it is impractical due to its longer runtime. Conversely, a higher step size improves performance but may affect the trustworthiness of the results, as shown in Table III. Although full trust can only be achieved with a step size of one, further research and experiments are required to determine the actual impact on performance and trust. Notably, the step size also affects space requirements: the memory needed per snapshot, on average, corresponds to the memory usage of the program under evaluation until the subsequent snapshot. In our experiments, the average space

requirement per snapshot ranged from 4 to 21 KB as shown in Table III. However, a decrease in step size leads to a higher number of snapshots, thereby increasing the total memory demand. This indicates that the step size is directly related to the performance and space consumption. Further research is necessary to thoroughly assess the balance between performance, trustworthiness, and memory usage.

Our prototype language was able to reproduce accurate results for the given benchmark problems, as shown in Table III. However, some of the benchmark problems exhibited unexpected results, as the replay time was higher than the recording time. This deviation is the result of the large tail recursion promoted by these programs, which caused every replay to rebuild large proportionally deep evaluation trees. Further research is necessary to identify potential optimizations for the MI and to investigate how these optimizations can improve the replay time of programs with large tail recursion.

RQ1 – In summary: The experiments demonstrated that the MI prototype is effective in accurately recording and replaying program executions for all step sizes and benchmark problems. The trade-off between trust and performance was observed, with lower step sizes providing higher trust but lower performance, and higher step sizes offering better performance at the cost of lower trust. Additionally, the experiments revealed that lower step sizes result in increased space requirements. This reflects a direct relationship among step size, performance, and space requirements, underscoring the intricate balance that must be managed between these factors.

B. RQ2 – Certification Protocol

The experiments have demonstrated the feasibility of running the OCCP on a Layer 2 blockchain, specifically POLYGON, as presented in Table IV.

When comparing the trade-off between step sizes of 100 and 1,000 in the LAZYWORKER scenario, there is an increase in the number of executed expressions in the latter, and conversely, a decrease in the required certification time. This is due to the fact that the increased step size reduces the communication overhead on the blockchain platform but requires more expressions for any given trace to be re-executed.

When comparing the results of the LAZYWORKER scenario experiments, we observe that our approach consistently results in fewer re-executed expression than the baseline of the respective scenario. The increase in executed expressions observed in the LAZYWORKER scenario, compared to the HAPPY scenario, is due to the need to re-execute traces that were assigned to malicious workers. This difference arises because malicious workers cast votes without executing the traces, which requires additional re-execution of the involved traces to resolve the resulting conflicts. On average, a malicious worker incurs an additional overhead of expressions required for executing one trace, whereas full re-execution would double the executed expressions. Furthermore, a higher gas cost is associated with a smaller step size, these costs can be attributed to the higher communication requirements. However, these costs can be controlled through step size adjustments. Specifically, a larger step

TABLE IV
RQ2 RESULTS – AVERAGE RUNTIME, IN SECONDS, AND NUMBER OF EXECUTED EXPRESSIONS, OVER 30 RUNS

Program	Scenario	Avg. Cert. Time (s)			Gas Costs (Mil.)			Exec. Exprs			Error Rates		
		Step Size			Step Size			Step Size			FP/FN		
		BASeLINE	100	1,000	BASeLINE	100	1,000	BASeLINE	100	1,000	BASeLINE	100	1,000
FIBONACCI	EQUIVALENT REGISTERSATTACK	12.716	513.174	73.652	4.202	373.748	39.329	71,360.0	100.0	1,000.0	1.0/0.0	0.0/0.0	0.0/0.0
	HAPPY	15.555	514.896	73.579	3.722	476.797	49.829	1,395,140.0	69,757.0	76,732.7	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 10%	15.387	654.382	135.912	3.616	598.401	69.822	1,255,626.0	72,663.667	95,600.3	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 20%	15.116	702.067	174.426	3.515	617.385	79.706	1,116,112.0	74,730.334	128,793.133	0.0/0.1	0.0/0.0	0.0/0.0
	LAZYWORKER 30%	14.989	823.23	435.668	7.176	617.69	117.174	1,953,196.0	78,107.0	262,824.9	0.0/0.033	0.0/0.0	0.0/0.0
	LAZYWORKER 40%	14.3	1,057.348	1,274.64	7.07	593.071	244.263	1,674,168.0	85,274.833	659,872.834	0.0/0.567	0.0/0.0	0.0/0.0
	MALICIOUSUSER	17.011	515.634	73.639	3.318	471.865	49.089	1,395,140.0	69,757.0	76,732.7	0.0/0.0	0.0/0.0	0.0/0.0
FIBONACCI ITERATIVE	HAPPY	18.515	729.395	87.593	4.12	682.689	69.866	1,998,680.0	99,934.0	99,934.0	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 10%	16.823	954.152	221.272	3.942	866.128	91.811	3,370,950.334	102,170.667	125,100.667	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 20%	16.403	965.92	202.965	3.968	887.046	105.304	3,277,958.6	90,196.133	154,156.334	0.0/0.567	0.0/0.0	0.0/0.0
	LAZYWORKER 30%	15.017	1,066.666	428.06	4.003	814.456	132.508	2,680,154.6	106,184.0	260,056.666	0.0/0.733	0.0/0.0	0.0/0.0
	LAZYWORKER 40%	14.742	1,411.262	1,171.964	3.879	857.327	244.888	2,328,114.467	110,517.4	623,560.533	0.0/0.933	0.0/0.0	0.0/0.0
	MALICIOUSUSER	19.194	730.487	87.741	3.849	675.636	68.894	4,387,217.134	99,934.0	99,934.0	0.0/0.0	0.0/0.0	0.0/0.0
LANCZOS	HAPPY	16.117	581.605	76.26	7.976	521.053	54.467	1,522,560.0	76,128.0	76,128.0	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 10%	16.546	721.855	141.194	7.723	640.154	74.879	1,370,304.0	78,514.667	115,495.466	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 20%	16.674	779.922	176.752	7.804	639.503	83.339	1,218,048.0	80,578.0	129,516.0	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 30%	15.078	938.292	389.54	15.666	622.745	109.196	2,131,584.0	84,032.266	222,675.734	0.0/0.166	0.0/0.0	0.0/0.0
	LAZYWORKER 40%	15.064	1,180.06	1,144.937	15.516	649.454	220.22	1,827,072.0	92,321.333	563,151.2	0.0/0.266	0.0/0.0	0.0/0.0
	MALICIOUSUSER	19.31	579.247	78.147	7.442	515.845	53.64	1,522,560.0	76,128.0	76,128.0	0.0/0.0	0.0/0.0	0.0/0.0
MATRIX MULTIPLICATION	HAPPY	15.807	651.147	90.916	7.037	593.492	61.552	1,735,620.0	86,781.0	86,781.0	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 10%	16.087	779.441	142.687	6.77	804.691	79.721	1,562,058.0	88,994.5	109,533.067	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 20%	16.49	887.788	214.627	6.872	678.446	94.805	1,388,496.0	90,607.667	148,134.033	0.0/0.033	0.0/0.0	0.0/0.0
	LAZYWORKER 30%	15.116	1,084.062	433.089	13.745	723.366	123.869	2,429,868.0	98,074.333	255,178.533	0.0/0.05	0.0/0.0	0.0/0.0
	LAZYWORKER 40%	14.972	1,210.444	2,175.75	13.609	724.327	330.615	2,082,744.0	100,784.367	949,897.0	0.0/0.366	0.0/0.0	0.0/0.0
	MALICIOUSUSER	19.287	650.831	91.947	5.088	587.359	60.662	1,735,620.0	86,781.0	86,781.0	0.0/0.0	0.0/0.0	0.0/0.0
MERGESORT	HAPPY	16.38	736.882	96.897	6.532	682.675	70.084	1,997,120.0	99,856.0	99,856.0	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 10%	17.667	962.512	175.203	6.361	980.638	92.182	1,797,408.0	102,279.333	123,156.0	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 20%	16.524	964.519	195.528	6.378	818.057	106.097	1,597,696.0	103,719.333	155,817.867	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 30%	15.305	1,122.631	453.675	12.794	821.501	138.285	2,795,968.0	105,909.333	276,141.066	0.0/0.1	0.0/0.0	0.0/0.0
	LAZYWORKER 40%	15.022	1,552.131	780.445	12.632	841.294	193.359	2,396,544.0	117,894.534	441,306.933	0.0/0.316	0.0/0.0	0.0/0.0
	MALICIOUSUSER	19.664	739.686	92.292	5.406	675.584	69.112	2,020,419.734	99,856.0	99,856.0	0.0/0.0	0.0/0.0	0.0/0.0
SHORTEST PATHFIRST	HAPPY	18.028	762.622	97.907	8.252	681.703	70.002	1,992,380.0	99,619.0	99,619.0	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 10%	17.472	894.811	168.344	8.182	816.025	90.457	1,793,142.0	101,725.667	119,542.8	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 20%	17.022	953.828	233.937	8.184	830.349	95.614	1,593,904.0	103,682.333	148,934.867	0.0/0.0	0.0/0.0	0.0/0.0
	LAZYWORKER 30%	16.436	1,115.191	711.422	16.456	779.423	156.214	2,789,332.0	106,282.333	355,760.266	0.0/0.083	0.0/0.0	0.0/0.0
	LAZYWORKER 40%	15.032	1,501.588	1,265.742	16.145	823.51	238.196	2,390,856.0	115,730.4	609,861.733	0.0/0.366	0.0/0.0	0.0/0.0
	MALICIOUSUSER	21.084	773.87	99.478	7.071	674.618	69.03	2,042,189.5	99,619.0	99,619.0	0.0/0.0	0.0/0.0	0.0/0.0

size not only improves performance by reducing communication overhead but also helps mitigate gas consumption. Future work should therefore focus on finding a trade-off between smart contract optimization and efficient gas usage.

Our approach reliably handles up to 40% malicious workers, demonstrating its robustness in detecting and addressing malicious behavior. Unlike the baseline approach, which fails to detect subtle code modifications unless they affect the final result, our method consistently identifies such manipulations as demonstrated by the EQUIVALENTREGISTERSATTACK scenario. The integration of our protocol's sequence hash verification and trace-based certification not only addresses the limitations of the naive approach but also enables reliable execution certification by detecting input and output manipulations as well as code modifications, resulting in a more robust solution. However, the increased reliability comes with a trade-off of higher resource costs. Notably, when the percentage of malicious workers reaches 40%, the overhead introduced becomes significant. Despite this, the system remains effective in certifying tasks even in the presence of additional malicious workers. Further research is required to gauge the impact of multiple malicious workers in combination with malicious users (see Section VIII).

When comparing the MALICIOUSUSER results to the HAPPY results regarding executed expressions, we noticed that the executed expressions were the same, indicating that the number

of executed expressions was not impacted by whether the execution was successful or rejected. This lack of difference was anticipated as traces for a full program are provided in both scenarios; however, no quorum is reached in the MALICIOUSUSER scenario. Thus, the workers need to compute each trace individually as in the HAPPY scenario and finally reach the conclusion that the execution does not match the provided \mathcal{H} .

We wish to stress that a rejected execution is not necessarily due to malicious acts, and a false negative has less weight than a successful certification. To restate, our approach has higher resilience to false positives than false negatives. Overall, the results demonstrate that our approach is effective in detecting and resolving conflicts in a distributed computation setting and can achieve reasonable performance and scalability on a Layer 2 blockchain.

The average communication time between the workers and the *Smart Contract* is 6.353 seconds. When analyzing the impact of different scenarios and step sizes, we observe that an increase in the number of traces leads to a corresponding increase in communication overhead. Specifically, in the HAPPY scenario, the average communication time increases 10-fold when the step size is increased from 100 to 1,000. For the LAZYWORKER scenarios, the communication overhead varies due to the additional interactions introduced by conflicts.

The observed increases in communication for LAZYWORKER of 10%, 20%, 30%, and 40% of malicious workers were 8-fold, 5-fold, 3-fold, and 2-fold, respectively. In the baseline approach, the average number of communications is fixed at 20 for HAPPY scenario, while the average communication time between workers and the *Smart Contract* remains consistent with our proposed approach. However, the average overhead introduced by our method, compared to the baseline, is 4-fold for a step size of 1,000 and nearly 44-fold for a step size of 100. This overhead can be reduced by adjusting the step size, which will result in fewer communication events.

RQ2 – In summary: Our experiments have demonstrated that the proposed OCCP is a feasible approach where the overhead of time and resources are dependent on the chosen step size, as observed in RQ1. However, in this initial research, we observe performance degradation due to implementation challenges, such as communication overhead from blockchain interactions, which need to be optimized in future work. In addition, our approach can handle all proposed malicious scenarios and reliably certify tasks or reject them.

C. RQ3 – Informed Step Size

The results show that the informed step size, compared to the non-informed variations (see RQ2), reduces the required certification time by up to 26-fold for the step size of 100 and 10-fold for 1,000. Similarly, gas costs are reduced by up to 44-fold for a step size of 100 and 6-fold for 1,000. As expected, the number of executed expressions for the multiplier 1 remains approximately the same as reported in RQ2.

Compared to the baseline, our approach consistently demonstrated performance gains for benchmarks scaled with a multiplier of 1,000. In this configuration, our method required up to 43-fold less time and 12-fold less gas consumption. However, these performance improvements only became apparent at or beyond the threshold multiplier of 1,000, indicating that performance gains outweigh communication overheads as the program size increases. For smaller problem sizes (multipliers below 1,000), our informed step size approach incurred higher time and gas costs during the certification process, similar to the trends observed in RQ2. Nonetheless, benchmarks scaled with a multiplier of 100 already exhibited notable time savings of up to 9-fold, though gas costs remained higher, with up to a 3-fold increase compared to the baseline. Despite these increases for smaller multipliers, our approach consistently reduced the need for re-execution of expressions across all experimental configurations.

We observe that in the baseline, the required execution time decreases as the number of malicious workers increases. This behavior is consistent with the findings in RQ2, where malicious workers avoid executing the intended program, effectively skipping computational work. The reduction in execution time is more pronounced for larger program sizes and higher proportions of malicious workers, as the skipped workload scales with these factors.

RQ3 – In summary: Our experiments demonstrate that adopting an informed step size improves time requirements and gas costs for longer-running benchmark problems, particularly when scaled by a factor of 1,000. Notably, while the performance gains become most evident after reaching a scale factor of 1,000, some improvements are already observed for time at smaller scales, such as 100. However, at this lower threshold, reductions in gas costs remain negligible compared to the baseline. Expectedly, the informed step size consistently reduces the number of re-executed expressions across all tested scales (similar to RQ2). Compared to the non-informed variation, time requirements are reduced by up to 26-fold for a step size of 100 and 10-fold for 1,000. Similarly, gas costs are reduced by up to 44-fold for a step size of 100 and 6-fold for 1,000.

VI. RELATED WORK

In this section, we provide a brief overview of prior works on verification and computational integrity approaches that relate to our proposed approach.

A. Hardware-Based Verification

Hardware-based verification can offer certain guarantees regarding the authenticity and integrity of an application. However, this approach relies on specialized hardware, such as Intel® Software Guard Extensions (Intel® SGX) [11], Secure Encrypted Virtualization (SEV) [13], or ARM TrustZone (TrustZone) [12], which can be costly and not always available. Additionally, there is still the possibility that malicious actors may alter the results after execution, as highlighted in the challenges of Intel® SGX [15].

B. Constraint Solvers

They translate code into constraints, which can only be solved if all the arguments provide a solution to the equation system of constraints. Systems such as PIPERINE [35], PANTRY [19], and SPICE [36] provide such functionality. However, all the aforementioned systems are limited by one or more of the following characteristics: 1) they require changes to the code to work, 2) have a fixed bound for loops, 3) static size for data structures (e.g., fixed tree depth), and 4) high latency due to waits between batches of verifications. While these limitations may not be problematic for smaller applications or toy examples, they can hinder the performance and scalability of larger applications with considerable dataset sizes. Additionally, the verification process would require either a deep understanding of the code's internals or blind trust in a third party that provides the constraints.

C. Software-Based Verification

Sasson et al. introduced ZK-SNARKS, a software-based verification approach for non-interactive zero-knowledge proofs. The approach is based on arithmetic circuits and assertions that do not require a re-execution to verify [28]. Later, Sasson et al. improved their previous approach and introduced ZK-STARKS that introduce transparency for zero-knowledge

TABLE V
RQ3 RESULTS – IMPACT COMPARISON OF LONG-RUNNING EXECUTIONS. AVERAGE RUNTIME, IN SECONDS,
AND NUMBER OF EXECUTED EXPRESSIONS, OVER 3 RUNS

Program	Scenario	Approach	Avg. Cert. Time (s)				Gas Costs (Mil.)				Exec. Exprs (Mil.)			
			Multiplier				Multiplier				Multiplier			
			1	10	100	1000	1	10	100	1000	1	10	100	1000
FIBONACCI	HAPPY	NAIVE	14.941	28.159	161.941	1,476.378	3.869	4.416	7.099	32.552	1.395	13.951	139.51	1,395.1
		OCPP	27.163	30.189	48.118	238.078	15.536	15.591	15.529	15.571	0.07	0.698	6.976	69.755
	LAZYWORKER 10%	NAIVE	15.19	33.202	152.703	8,222.387	3.7	9.596		224.273	1.256	14.021	139.51	2,790.2
		OCPP	45.712	109.518	100.672	412.286	24.951	21.163	22.148	21.554	0.105	1.099	11.242	122.188
	LAZYWORKER 20%	NAIVE	17.744	73.7	614.431	5,955.659	3.634	4.572	8.301	37.236	1.116	11.161	111.608	1,116.08
		OCPP	80.157	172.494	112.192	732.102	31.18	25.077	27.127	32.474	0.14	1.431	17.264	248.444
	LAZYWORKER 30%	NAIVE	17.563	65.545	540.228	5,212.203	3.557	5.035	7.85	35.833	0.977	9.766	97.657	976.57
		OCPP	90.191	151.625	202.538	1,420.791	34.475	30.153	36.69	55.965	0.196	1.821	27.228	473.172
	LAZYWORKER 40%	NAIVE	15.553	57.552	463.555	4,462.853	3.511	4.054	8.309	37.569	0.837	8.371	83.706	837.06
		OCPP	163.163	271.201	388.817	1,870.986	45.516	54.254	60.565	68.729	0.337	4.466	53.444	600.939
	MALICIOUSUSER	NAIVE	16.936	54.312	416.188	4,019.539	3.274	5.165	8.186	36.878	1.395	13.951	139.51	1,395.1
		OCPP	27.12	30.489	48.497	241.328	15.111	15.122	15.061	15.101	0.07	0.698	6.976	69.755
FIBONACCIITERATIVE	HAPPY	NAIVE	18.443	63.167	505.242	4,680.364	3.879	5.708	7.047	19.124	1.999	19.986	199.86	1,998.6
		OCPP	27.398	30.6	53.049	283.662	15.593	15.541	15.572	15.58	0.1	0.999	9.993	99.93
	LAZYWORKER 10%	NAIVE	16.969	33.003	181.573	10,026.82	3.815	5.459	8.615	69.538	1.799	19.953	199.86	1,998.6
		OCPP	61.745	99.833	102.112	505.118	26.535	20.945	22.801	22.177	0.174	1.581	16.838	195.03
	LAZYWORKER 20%	NAIVE	19.089	86.409	755.664	7,395.015	3.849	4.855	6.845	18.709	1.599	15.989	159.888	1,598.88
		OCPP	61.594	105.665	125.739	807.301	31.855	24.64	26.822	31.717	0.235	1.99	24.116	330.602
	LAZYWORKER 30%	NAIVE	18.231	77.559	658.887	6,452.206	3.913	5.377	6.557	18.103	1.399	13.99	139.902	1,399.02
		OCPP	93.541	165.477	192.476	1,377.47	35.913	32.047	34.569	48.167	0.321	3.018	36.908	550.281
	LAZYWORKER 40%	NAIVE	17.557	70.205	568.222	5,550.851	3.912	5.703	6.936	19.013	1.199	11.992	119.916	1,199.16
		OCPP	142.367	292.566	410.1	3,542.724	43.655	46.044	63.349	101.065	0.45	4.995	81.143	1,373.371
	MALICIOUSUSER	NAIVE	18.74	65.075	505.963	4,681.979	3.902	5.699	7.06	19.137	1.999	19.986	199.86	1,998.6
		OCPP	27.219	30.565	54.117	289.367	15.122	15.073	15.108	15.112	0.1	0.999	9.993	99.93
LANCZOS	HAPPY	NAIVE	15.225	29.832	168.942	1,494.42	8.088	8.605	9.927	20.733	1.523	15.225	152.244	1,522.44
		OCPP	28.103	30.248	50.731	242.947	15.543	15.517	15.534	15.545	0.076	0.761	7.612	76.122
	LAZYWORKER 10%	NAIVE	15.166	33.268	159.649	8,633.519	7.808	9.853	13.339	78.675	1.37	15.275	152.244	1,522.44
		OCPP	56.341	94.173	104.051	445.491	25.772	20.861	22.279	22.128	0.125	1.166	13.106	146.789
	LAZYWORKER 20%	NAIVE	19.748	77.709	642.336	6,245.005	8.026	9.332	11.418	23.916	1.218	12.18	121.795	1,217.952
		OCPP	78.167	140.013	126.56	610.978	31.687	25.112	26.326	29.332	0.173	1.6	17.66	228.747
	LAZYWORKER 30%	NAIVE	17.566	69.571	561.562	5,456.849	7.724	9.768	10.957	22.748	1.066	10.657	106.571	1,065.708
		OCPP	120.18	189.148	208.381	1,985.648	34.328	37.461	36.873	70.365	0.216	2.914	31.489	687.255
	LAZYWORKER 40%	NAIVE	17.557	61.533	488.208	4,680.881	7.74	9.186	11.556	23.832	0.914	9.135	91.346	913.464
		OCPP	174.428	308.299	496.166	2,402.201	48.204	51.218	73.566	80.282	0.411	4.518	74.752	821.991
	MALICIOUSUSER	NAIVE	19.333	57.304	435.47	4,041.03	7.399	9.786	11.244	23.329	1.523	15.225	152.244	1,522.44
		OCPP	28.863	30.14	50.095	249.054	15.073	15.047	15.061	15.078	0.076	0.761	7.612	76.122
MATRIXMULTIPLICATION	HAPPY	NAIVE	14.892	31.371	181.337	9,360.574	7.132	7.959	11.731	47.969	1.736	17.331	173.285	1,732.823
		OCPP	28.851	29.997	52.597	268.584	15.541	15.568	15.531	15.623	0.087	0.867	8.664	86.641
	LAZYWORKER 10%	NAIVE	15.182	33.066	168.055	8,552.99	6.87	9.055	16.086	176.006	1.562	17.331	173.285	1,732.823
		OCPP	138.944	121.621	108.981	466.942	24.588	20.746	22.115	21.373	0.128	1.324	14.209	158.409
	LAZYWORKER 20%	NAIVE	19.11	81.091	689.727	6,714.39	7.086	8.429	13.649	54.558	1.388	13.865	138.628	1,386.258
		OCPP	76.496	99.202	137.277	786.072	30.669	26.818	28.136	31.414	0.192	1.922	23.206	287.504
	LAZYWORKER 30%	NAIVE	18.884	73.556	604.229	5,866.896	6.813	8.968	12.94	52.778	1.215	12.132	121.299	1,212.976
		OCPP	96.825	137.232	186.412	1,500.339	36.262	32.892	34.629	52.303	0.279	2.649	32.115	540.063
	LAZYWORKER 40%	NAIVE	17.574	65.538	520.883	5,022.172	6.789	9.382	13.629	54.728	1.041	10.399	103.971	1,039.694
		OCPP	203.252	215.02	383.093	3,472.424	48.207	39.874	62.695	103.02	0.463	3.502	67.985	1,224.239
	MALICIOUSUSER	NAIVE	19.444	61.271	409.616	11,951.525	5.076	7.344	11.566	52.424	1.736	17.331	173.285	1,732.823
		OCPP	28.779	30.245	52.138	273.911	15.075	15.095	15.065	15.153	0.087	0.867	8.664	86.641
MERGESORT	HAPPY	NAIVE	15.109	33.37	210.528	1,870.644	6.646	7.548	12.781	63.966	1.997	19.944	199.417	1,994.143
		OCPP	29.027	32.121	56.457	308.65	15.591	15.571	15.583	15.6	0.1	0.997	9.971	99.707
	LAZYWORKER 10%	NAIVE	15.188	33.062	195.587	10,848.192	6.362	8.434	17.893	271.731	1.797	19.213	199.417	1,994.143
		OCPP	89.576	84.419	110.814	543.501	25.549	22.548	22.441	21.919	0.149	1.539	17.665	188.28
	LAZYWORKER 20%	NAIVE	20.431	91.117	797.028	7,731.012	6.582	7.999	14.752	73.555	1.598	15.956	159.534	1,595.314
		OCPP	65.077	89.545	164.705	762.991	31.662	27.73	28.746	28.903	0.219	2.255	27.852	286.492
	LAZYWORKER 30%	NAIVE	19.59	82.883	700.215	6,756.859	6.284	8.507	14.094	70.257	1.398	13.961	139.592	1,395.9
		OCPP	96.028	125.428	235.559	1,496.706	34.495	37.188	37.97	46.337	0.276	3.743	43.423	535.427
	LAZYWORKER 40%	NAIVE	17.566	73.536	605.565	5,804.2	6.291	8.897	14.835	73.48	1.198	11.967	119.65	1,196.486
		OCPP	167.115	210.084	334.589	2,857.447	44.756	45.045	51.549	78.539	0.48	5.023	59.526	1,007.042
	MALICIOUSUSER	NAIVE	19.16	67.599	548.21	5,050.571	5.283	7.733	13.606	71.242	1.997	19.944	199.417	1,994.143
		OCPP	28.904	32.997	56.712	312.064	15.123	15.1	15.119	15.13	0.1	0.997	9.971	99.707
SHORTESTPATHFIRST	HAPPY	NAIVE	15.89	33.291	200.33	1,764.919	8.421	9.002	10.21	21.002	1.992	19.923	199.232	1,992.32
		OCPP	30.448	32.437	54.457	286.605	15.57	15.596	15.582	15.596	0.1	0.996	9.962	99.616
	LAZYWORKER 10%	NAIVE	16.628	33.204	188.464	10,390.837	8.189	10.076	14.414	88.713	1.793	19.326	199.232	1,992.32
		OCPP	92.283	117.923	121.541	510.942	25.393	22.287	22.053	22.195	0.143	1.546	17.117	189.602
	LAZYWORKER 20%	NAIVE	21.119	90.42	763.731	7,422.313	8.37	9.586	11.837	24.066	1.594	15.939	159.386	1,593.856
		OCPP	55.789	92.639	141.294	714.371	30.874	25.818	27.783	28.804	0.198	2.065	24.821	294.531
	LAZYWORKER 30%	NAIVE	19.58	81.571	670.879	6,510.869	8.054	10.141	11.268	23.06	1.395	13.946	139.462	1,394.624
		OCPP	92.422	118.021	215.833	1,719.338	34.328	36.16	36.906	55.015	0.292	3.697	40.012	664.937
	LAZYWORKER 40%	NAIVE	19.555	71.534	578.225	5,576.213	8.07	10.689	11.882	24.199	1.195	11.954	119.539	1,195.392

program, and in case of a dispute, is challenged and must play a verification game to prove that the solution was correct [4]. However, as discussed earlier, re-executing the full program significantly reduces the usability of such a system due to the increase in computation and time requirements.

VII. CONCLUSION AND FUTURE WORK

We have demonstrated the feasibility of our proposed Mona Interpreter (MI) and On-Chain Certification Protocol (OCCP) for certifying program executions with the help of a layer 2 blockchain. Our experimental results indicate that our MI prototype can accurately reproduce correct results for every step size. However, a trade-off between trust and performance exists, which requires further investigation to determine an ideal balance between the two. On average the step size increase from 100 to 1,000 speeds up the certification process by a factor of 7.371 while only slightly increasing the number of executed expressions in certain scenarios. In-depth experiments and optimizations are needed to mitigate the impact of a lower step size.

Additionally, our proposed OCCP was able to certify correct executions, while outperforming the baseline, which re-executes the full program multiple times, with fewer executed expressions in all proposed scenarios. Furthermore, our proposed approach allows for the detection of incorrect or malicious actions with similar effort as certifying a correct one.

In the future, we plan to investigate the adaptation of the mechanisms devised for MI to other programming languages, providing them with the feature of segmentation and replay. While this paper presents evidence on the feasibility of the approach, a large-scale case study is required to thoroughly examine how well this approach scales to real-world scenarios, considering both the performance impact and the practical challenges of deploying it across diverse, complex environments.

For OCCP, we aim to explore the use of different blockchains and develop an incentive mechanism alongside a reward system that better defines the roles and compensations for participants solving the proposed problem. Specifically, potential incentive mechanisms could include a paid service model, funded either by universities and journals or by users who wish to verify their execution, paying into a smart contract. This smart contract would then fairly distribute monetary remuneration to workers based on their contributions. Additionally, to further ensure reliable and non-malicious participation, we propose a dual-layer approach: 1) integrating a mechanism to detect malicious workers, thereby encouraging trustworthy behavior, and 2) requiring workers to deposit a small stake into the smart contract. Workers would forfeit this deposit in cases of malicious behavior, but non-malicious participants would receive it back, providing both a deterrent against misconduct and an additional incentive to act responsibly.

Future work can explore extending our approach to multi-threading aspects. Adding a sequentially numbered annotation layer, using the seqId strategy, to track the evaluation progress of threads within its assigned code derivation. This would capture a memory snapshot showing all active threads at a given

time. Moreover, future work could focus on enhancing the performance of the MI by exploring alternative Intermediate Representations (IRs) beyond the current AST approach, aiming to achieve faster verification times and support more advanced optimizations. The proposed certification protocol can succeed if the likelihood of reaching a snapshot with output altered by a concurrent side effect remains statistically probable within expected retries. The annotation layer would help manage multiple threads, allowing computation to continue from where each thread left off. If concurrency doesn't affect the snapshot's memory state, the snapshot is verified. Otherwise, the same concurrency conditions must be reproducible through a quorum. Further analysis is needed to define an acceptable quorum and the probability of reaching the same memory state. Future research can also focus on a concurrency-aware snapshotting strategy to limit side effects and maintain consistent outputs.

In order to address non-determinism, we plan to explore the feasibility of using a sampling mechanism for random generators until a confidence threshold is reached. This would introduce an element of uncertainty, making it crucial to carefully analyze the impact on system behavior and overall reliability.

Additionally, future work can focus on implementing an identity mechanism to prevent address spoofing and flooding by malicious workers by employing Decentralized Identity (DID) frameworks, each worker would possess an identity anchored in a public blockchain, where cryptographic signatures ensure authenticity without relying on a centralized authority. This would enable consistent worker recognition, prevent malicious actors from forging multiple identities. Additionally, verifiable credentials could be used to certify a worker's qualifications and past performance while preserving privacy, further mitigating risks of impersonation or malicious behavior. Future iterations of our system could implement these mechanisms as a service to strengthen security.

Lastly, a non-local adaptation of IPFS should be analyzed and evaluated in future iterations as well as the use of consistent hashing to improve performance of our approach.

In conclusion, our proposed MI and OCCP hold promise in enhancing the trustworthiness and security of program executions through segmentation and certification, respectively. It encourages further investigation and development in this area by the research community.

VIII. LIMITATIONS AND THREATS TO VALIDITY

A. Malicious Worker Scenarios

Our experiments specifically evaluate the scenario where malicious workers intentionally provides incorrect results. In practice, malicious actors may attempt to interfere with the certification process in other ways, such as by withholding results or intentionally producing conflicting outputs. Consequently, the robustness of our protocol against other types of malicious interference remains uncertain.

B. Collusion Between Malicious Users and Workers

We did not explore scenarios involving collusion between malicious workers and users. If more than 50% of the workers

are malicious, they could potentially disrupt the certification process, leading to failure of the protocol and the underlying blockchain. This requires deeper analysis, which is out of scope for the current work.

C. Local Blockchain Evaluation

The use of a local blockchain platform for our experiments may not fully represent the performance of our protocol in non-local or private blockchain environments. Additionally, the use of a local *Amazon S3* instance for trace data storage might impact protocol performance, potentially affecting the performance of the protocol.

D. Majority Voting Mechanisms

Our protocol relies on majority voting for certification. While it is designed to detect discrepancies between executions, it assumes that the majority of participants are honest. Collusion among malicious workers and malicious users could undermine the voting mechanism's effectiveness. However, this is an intrinsic limitation of majority voting mechanisms.

E. Generalizability to Other Languages and Paradigms

Additionally, we evaluated the feasibility of our proposed programming language on specific use cases (see Section IV-A). Further work is required to extend our work to other programming languages and evaluate real-world applications.

F. Task Distribution Fairness and Integrity

Our current implementation does not address fairness or integrity in task distribution to workers. We mitigate this partially by tracking tasks and worker identification to prevent disputes from being assigned to the same worker. However, more robust mechanisms need to be developed to enhance fairness.

G. Lack of Identity Mechanisms

Although our implementation includes unique identifiers for workers, it currently lacks a robust identity verification mechanism. To address this limitation, future work could integrate DID frameworks, as discussed in prior studies [39], [40], [41], [42]. Specifically, platforms like Hyperledger Indy [43] or uPort [44] could provide a foundation for decentralized and cryptographically verifiable identities. By using DID, workers would be associated with cryptographically secure and verifiable identities, ensuring consistent worker recognition and preventing malicious actors from forging multiple identities.

H. Smart Contract Optimization

We also acknowledge that our implementation of the smart contract may not be optimal, thus leading to higher gas consumption. However, this requires further research into the optimization of smart contracts and is out of scope for this paper.

I. False Positives vs. False Negatives

Finally, it is worth noting that our protocol is designed to be more resilient to false positives (i.e., falsely certifying a task as correct) than false negatives (i.e., failing to certify a correct task). As a result, the protocol may require tasks to be re-executed if they fail to produce a certificate, even if they are correct. This may lead to additional computational overhead and delay in some scenarios.

ACKNOWLEDGMENT

We sincerely thank Dr. Pooja Rani for her invaluable guidance and efforts in helping us revise and improve our paper.

REFERENCES

- [1] C. Sánchez et al., "A survey of challenges for runtime verification from advanced application domains (beyond software)," *Formal Methods Syst. Des.*, vol. 54, pp. 279–335, 2019.
- [2] M. Walfish and A. J. Blumberg, "Verifying computations without reexecuting them," *Commun. ACM*, vol. 58, no. 2, pp. 74–84, Jan. 2015.
- [3] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2013, pp. 238–252.
- [4] J. Teutsch and C. Reitwießner, *A Scalable Verification Solution for Blockchains*, 2023, pp. 377–424. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/9789811278631_0015
- [5] E. Ben-Sasson, I. Bentov, Y. Horeh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptol. ePrint Arch.*, vol. 2018, p. 46, 2018.
- [6] A. L. Beam, A. K. Manrai, and M. Ghassemi, "Challenges to the reproducibility of machine learning models in health care," *JAMA*, vol. 323, no. 4, pp. 305–306, Jan. 2020.
- [7] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguist.*, A. Korhonen, D. Traum, and L. Márquez, Eds. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 3645–3650. [Online]. Available: <https://aclanthology.org/P19-1355>
- [8] R. A. Zwaan, A. Etz, R. E. Lucas, and M. B. Donnellan, "Making replication mainstream," *Behav. Brain Sciences*, vol. 41, p. e120, 2018.
- [9] J. Vitek and T. Kalibera, "Repeatability, reproducibility, and rigor in systems research," in *ACM Int. Conf. Embedded Softw. (EMSOFT)*, Oct. 2011, pp. 33–38.
- [10] D. Srinivasan and R. Gopalaswamy, *Software Testing: Principles and Practices*. Pearson Education India, 2007.
- [11] V. Costan and S. Devadas, "Intel SGX Explained," in *Int. Conf. Financial Cryptography Data Secur. (FC)*, 2016, pp. 17–36.
- [12] S. Pinto and N. Santos, "Demystifying ARM TrustZone: A Comprehensive Survey," *ACM Comput. Surveys*, vol. 51, no. 6, Jan. 2019.
- [13] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," Tech. Rep., 2016.
- [14] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," *IEEE Trustcom/BigDataSE/ISPA*, vol. 1, 2015, pp. 57–64.
- [15] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of SGX and countermeasures: A survey," *ACM Comput. Surveys*, vol. 54, no. 6, Jul. 2021.
- [16] M. Morbitzer, S. Proskurin, M. Radev, M. Dorfhuber, and E. Q. Salas, "SEVerity: Code injection attacks against encrypted virtual machines," in *IEEE Secur. Privacy Workshops (SPW)*, 2021, pp. 444–455.
- [17] P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Trusted execution environments: Properties, applications, and challenges," *IEEE Secur. Privacy*, vol. 18, no. 2, pp. 56–60, Mar. 2020, IEEE Secur. Privacy. [Online]. Available: <https://ieeexplore.ieee.org/document/9041685/?arnumber=9041685>
- [18] T. Vogel, S. Druskat, M. Scheidgen, C. Draxl, and L. Grunske, "Challenges for verifying and validating scientific software in computational materials science," in *IEEE/ACM International Workshop on Software Engineering for Science (SE4Science)*, 2019, pp. 25–32.
- [19] B. Braun, A. Feldman, Z. Ren, S. Setty, A. Blumberg, and M. Walfish, "Verifying computations with state," in *ACM Symp. Operating Syst. Princ. (SOSP)*, 2013, pp. 341–357.

- [20] K. Toyoda, P. T. Mathiopoulos, I. Sasase, and T. Ohtsuki, "A novel blockchain-based product ownership management system (POMS) for anti-counterfeits in the post supply chain," *IEEE Access*, vol. 5, pp. 17465–17477, 2017.
- [21] H. R. Hasan and K. Salah, "Combating deepfake videos using blockchain and smart contracts," *IEEE Access*, vol. 7, pp. 41596–41606, 2019.
- [22] A. Wolf, M. E. Palma, P. Salza, and H. C. Gall. Replication Package. 2023. [Online]. Available: <https://github.com/Lochindaal/occpReplicationPackage/>
- [23] M. E. Palma, A. Wolf, P. Salza, and H. C. Gall. Mona. 2023. [Online]. Available: <https://github.com/MEPalma/Mona/>
- [24] E. Mossel, J. Neeman, and O. Tamuz, "Majority dynamics and aggregation of information in social networks," *Auton. Agents Multi-Agent Syst.*, vol. 28, pp. 408–429, 2014.
- [25] P. Chen and S. Redner, "Majority rule dynamics in finite dimensions," *Phys. Rev. E Statist., Nonlinear, Soft Matter Phys.*, vol. 71, no. 3, 2005, Art. no. 036101.
- [26] A. Mukhopadhyay, R. R. Mazumdar, and R. Roy, "Voter and majority dynamics with biased and stubborn agents," *J. Statist. Phys.*, vol. 181, pp. 1239–1265, 2020. [Online]. Available: <https://api.semanticscholar.org/CorpusID:214713627>
- [27] J. M. Buchanan, "Simple majority voting, game theory, and resource use," *Can. J. Econ. Political Sci.*, vol. 27, no. 3, pp. 337–348, 1961.
- [28] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct Non-Interactive Zero Knowledge for a Von Neumann Architecture," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, 2014, pp. 781–796.
- [29] I. Khaburzaniya, K. Chalkias, K. Lewi, and H. Malvai, "Aggregating and thresholdizing hash-based signatures using STARKs," in *ACM Asia Conf. Comput. Commun. Secur. (ASIACCS)*, 2021.
- [30] J. Zhang, T. Xie, Y. Zhang, and D. Song, "Transparent polynomial delegation and its applications to zero knowledge proof," in *IEEE Symp. Secur. Privacy (SP)*, 2019, pp. 859–876.
- [31] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. X. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Annu. Int. Cryptol. Conf. (CRYPTO)*, 2019, pp. 733–764.
- [32] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zkSNARKs without trusted setup," in *IEEE Symp. Secur. Privacy (SP)*, 2018, pp. 926–943.
- [33] K. Turkowski, "Filters for common resampling tasks," *Graph. Gems*, vol. 94, pp. 147–165, 1990.
- [34] G. Wood et al., "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2017.
- [35] J. Lee, K. Nikitin, and S. T. V. Setty, "Replicated state machines without replicated execution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2020, pp. 119–134.
- [36] S. T. V. Setty, S. G. Angel, T. Gupta, and J. Lee, "Proving the correct execution of concurrent services in zero-knowledge," in *Proc. USENIX Symp. Oper. Syst. Des. Implementation (USENIX OSDI)*, 2–18, pp. 339–356.
- [37] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth, "Succinct non-interactive arguments via linear interactive proofs," *J. Cryptol.*, vol. 35, no. 3, p. 15, Jul. 2022.
- [38] A. Zeiselmaier, B. Steinkopf, U. Gellersdörfer, A. Bogensperger, and F. Matthes, "Analysis and application of verifiable computation techniques in blockchain systems for the energy sector," *Front. Blockchain*, vol. 4, pp. 156–167, 2021.
- [39] R. Soltani, U. T. Nguyen, and A. An, "A new approach to client onboarding using self-sovereign identity and distributed ledger," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Physical Social Comput. (CPSCom) IEEE Smart Data (SmartData)*, 2018, pp. 134–156.
- [40] M. A. Bouras, Q. Lu, F. Zhang, Y. Wan, T. Zhang, and H. Ning, "Distributed ledger technology for ehealth identity privacy: State of the art and future perspective," *Sensors*, vol. 20, no. 2, 2020. [Online]. Available: <https://www.mdpi.com/1424-8220/20/2/483>
- [41] Y. Liu, D. He, M. S. Obaidat, N. Kumar, M. K. Khan, and K.-K. R. Choo, "Blockchain-based identity management systems: A review," *J. Netw. Comput. Appl.*, vol. 166, 2020, Art. no. 102731. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1084804520302058>
- [42] R. Soltani, U. T. Nguyen, and A. An, "A survey of self-sovereign identity ecosystem," *Secur. Communication Networks*, vol. 2021, no. 1, 2021, Art. no. 8873429. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/8873429>
- [43] Hyperledger, "Hyperledger indy: Identity for all." (2024). Accessed: Nov. 24, 2024. [Online]. Available: <https://hyperledger.org/projects/indy>
- [44] D. C. Lundkvist, R. Heck, J. Torstensson, Z. Mitton, and M. Sena, "UPORT: A platform for self-sovereign identity," 2016. [Online]. Available: https://whitepaper.uport.me/uPort_whitepaper_DRAFT20170221.pdf



Alex Wolf received the master's degree in software systems from the University of Zürich, in 2022. He is currently working toward the Ph.D. degree with the Software Evolution and Architecture Lab (s.e.a.l.), University of Zurich, Switzerland. His research focuses on advancing machine learning, software architecture, and engineering. Prior to beginning his Ph.D. studies, he gained industry experience through various software engineering roles, providing a strong foundation in practical, and technical problem-solving. His research interests lie at the intersection of machine learning and software engineering, with a particular focus on practical applications and the integration of blockchain technology. For more information, see wolf@ifi.uzh.ch.



Marco Edoardo Palma is currently working toward the Ph.D. degree with the Software Evolution and Architecture Lab (s.e.a.l.), University of Zurich, Switzerland, and the First-Class Honours degree in computer science with artificial intelligence from the University of Southampton, U.K., in 2021. His research explores the development of artificial intelligence tools and strategies to enhance software engineering processes and tools. Currently, his work centres on the deep abstraction strategy, which automatically compiles algorithms with high space and time complexity into efficient statistical models. For more information, see marcoepalma@ifi.uzh.ch.



Pasquale Salza received the Ph.D. degree in computer science from the University of Salerno, Italy. He is a Senior Research Associate with the Software Evolution and Architecture Lab (s.e.a.l.), University of Zurich, Switzerland. His research interests include software engineering, machine learning, cloud computing, and evolutionary computation. For more information, see salza@ifi.uzh.ch.



Harald C. Gall (Member, IEEE) is a Professor of software engineering and Director of the Software Evolution and Architecture Lab (s.e.a.l.), Department of Informatics, University of Zurich, Switzerland. He held Visiting Positions with Microsoft Research, USA, and University of Washington, Seattle, USA. His research interests include software evolution, software architecture, software quality, and green software engineering. He has worked on developing new ways in which data mining of software repositories and machine learning can contribute to a better understanding and improvement of software development. For more information, see gall@ifi.uzh.ch.