# Fides: A System for Verifiable Computation Using Smart Contracts

Mahmudun Nabi(✉), Sepideh Avizheh, and Reihaneh Safavi-Naini

University of Calgary, Calgary, AB, Canada
{mahmudun.nabi1,sepideh.avizheh1,rei}@ucalgary.ca

**Abstract.** Verifiable computation allows a resource-constrained client to outsource their computation to powerful servers, and efficiently verify their received results. Cryptographic verifiable computation systems, despite their elegant designs, have limited application in practice because of the computational cost and difficulty of correct and flexible implementation of complex cryptographic systems. An attractive approach to verifiably compute general functions is to use more than one server to compute the same function, and decide the computation result based on the submitted results of all servers. In this paper, we propose a system for delegation of computation to two cloud servers using a smart contract (SC), that guarantees correct computation results as long as at least one of the two servers is honest. Our work adapts the Refereed Delegation of Computation (RDoC) model of Canetti, Riva and Rothblum (ACM CCS'11) to the SC setting. This was first considered by Avizheh et al. (ACM CCSW'19) who showed that the direct employment of RDoC in the smart contract setting will be insecure because of the *copy attack*, where a server copies the result of the other server, and becomes possible due to the transparency of SC. However, the implementation of their protocol was left as future work. Our work is a new SC-aided RDoC design with proved security that significantly reduces the computation of the smart contract. Additionally, it provides security against the misbehaviours of the client and an implementation of the design over the Ethereum blockchain. The proposed system, which is called *Fides*, is the first implementation of SC-aided RDoC. We discuss the challenges of this implementation and our design decisions, and present the cost analysis of the system for an example computation. We also propose extensions of our work and directions for future research.

**Keywords:** Refereed delegation of computation · blockchain · smart contract · verifiable computation · copy attack

## 1 Introduction

Resource-constrained enterprises and individuals increasingly outsource their computation and data intensive applications to cloud services such as Amazon's EC2 and Microsoft's Azure. Outsourcing computation however raises the need of the outsourcer to be able to efficiently verify the received computation result.


S. Matsuo et al. (Eds.): FC 2022 Workshops, LNCS 13412, pp. 448–480, 2023.
https://doi.org/10.1007/978-3-031-32415-4_29

Verifying outsourced computation has been studied with two types of adversaries: (i) *rational adversary model* where the adversary has well-defined utility [6,30,33,36], and (ii) *malicious adversary* where the adversary can arbitrarily deviate from the protocol. Verifiable computation (VC) systems with security against malicious adversary have been designed both for general computation [1,10,13,19], and specific computations [16,25,28,46]. These protocols despite their elegant designs have limited use in real applications because of the computation and storage cost, as well as implementation challenges of cryptographic systems [23].

*Verifying Outsourced Computation by Replication.* A natural approach to verifying computation is to replicate the computation on multiple servers and accept the received results if they all match. Replication of computation for correctness guarantee has been studied in rational [6,30,33,36] and malicious adversary model [7,10,11], where in the former the adversary is assumed to have a known utility they want to maximize while the latter considers an adversary that can deviate arbitrarily from the computation. We consider malicious adversary model.

In [10], Canetti, Rothbulm and Riva (CRR) built on the general *refereed computation model* of Feige and Kilian [17], and proposed Refereed Delegation of Computation (RDoC) that allows an *honest client (referee)* to use only two servers, and obtain guaranteed correct result *if at least* one of the two servers is honest. This is the minimum amount of redundancy. In all these works the client is honest and the goal is to ensure correctness when the servers misbehave.

*Smart Contract for Verifiable Computation.* A very attractive approach to implementing an outsourcing service with provable guarantee is to implement CRR using a smart contract as the referee. The smart contract will also distribute the funds and ensure correct payments to the honest cloud, and will remove the trust assumption on the client in CRR which could be a significant limitation in practice. Avizheh et al. [4] considered such a system and noted that despite proved security of CRR, its direct application of CRR with a smart contract will be insecure because of a new attack called *copy attack* that arises because of the transparent nature of the smart contract, and allows a malicious server to simply copy the submitted result of the honest server. Copy attack disincentivizes honest behaviour and undermines the underlying principle of replicated computation that is the need for two independent computations. Avizheh et al. also proposed a modification of CRR, called scCRR, that has security against copy attack, and noting the challenges of implementing the scCRR (and CRR) protocol in the smart contract setting, left it as future work.

### 1.1 Our Work

We design and implement a computation outsourcing service that is managed and run by a smart contract, and is based on the adaptation of CRR to the smart contract setting. We propose a new modified version of CRR, that we call cmCRR, or *committed CRR*, that reduces the computation of the smart contract

compared to scCRR, and implement *Fides*, a verifiable computation service, that uses cmCRR as the underlying protocol over Ethereum blockchain.

**Two-Server SC-Aided Verifiable Computation Service.** We consider three types of parties: a client who wants to outsource their computation, servers who perform the computation, and the smart contract (SC) that is a trusted mediator for transferring funds, and plays the role of the referee in cmCRR.

In cmCRR, the client sends a function $f()$ and the commitment to the input $x$ to the SC, and $f()$ and the opening of the commitment to the two servers (that have been chosen by the client). The servers verify the opening of the committed value, perform the computation, and submit their commitment to the computation result, and the root of a Merkle tree that is constructed on their computation trace and incorporates the server's public key in the last step. Once the SC receives both commitments, the servers open their commitments. If the results match, the computation is complete and correct because at least one server is honest. If they do not match, a *referee protocol* is run by the SC. Our referee protocol is similar to the corresponding protocol in CRR, with an additional Merkle proof that is required to be generated by the servers at each round of binary search and is used to prevent the copy attack. The protocol uses a sequence of challenge-response to determine the first different computation step in the reported computation traces. The SC then performs a single step of the computation to determine the honest server. We prove security against copy attack and correctness of the computation result (i.e. SC's final decision) by effectively reducing them to the collision-resistance property of the hash function that is used in the Merkle tree, and the security of the commitment scheme. To achieve this result however we require the servers to participate in the full protocol, and submit their opening of the commitments. This is realized by requiring collateral from servers who participate in the computation (similar to putting a bid).

**Implementation.** Fides implements cmCRR over Ethereum blockchain. The main challenge of implementation is efficiently transferring the required information for the single step computation to the SC. The theoretical model of CRR uses a Turing Machine (TM) as the computation model, and defines computation trace as a sequence of *reduced computation state*, consisting TM's current state, current head position and value on that position, and Merkle root constructed on the tape. We use EVM (Ethereum Virtual Machine) to model the computation and Solidity language to express the function $f(x)$. In Sect. 4.1 we describe the EVM configuration equivalent to TM configuration, and define the reduced computation state as EVM machine state consisting the values of stack, memory and storage. The Merkle tree commitment is constructed over the sequence of reduced computation states by each server. The referee protocol determines a single computation step that must be transferred to the SC in case of dispute. We do this by sending the single step data as an executable code (in the form of bytecode) to SC, that is executed as a newly deployed smart contract on Ethereum.

Similar to CRR, cmCRR and Fides require the computation to be deterministic. We note that randomized functions can be made deterministic using a

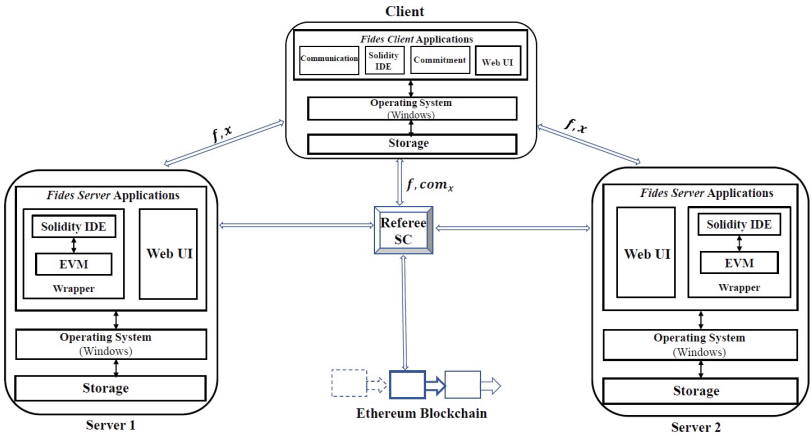


**Fig. 1.** *Fides*. Client's communication with Server is done through secure channel. SC communications with client and servers happens over public channels. Messages sent and received by SC, and the result of SC functions, are posted on the ledger and are publicly visible. *WebUI* (i.e., a local html file) is developed for each Fides server and the Fides client that runs JavaScript code to load data from its storage and uses *Web3.js* library for interacting with the SC.

pseudorandom generator. We use Ganache framework [18] to simulate a public Ethereum blockchain and Solidity language to write smart contracts.

**Fides Implementation and Experiments.** Figure 1 shows the overview of *Fides* with two servers, consisting of the following modules.

(i) *Fides client* that processes the function $f()$ for submitting to the SC and sending to the servers, and communicates with the servers and the SC;

(ii) *Fides servers* receive the EVM executable from the client, run it in their EVM *that is not connected to the Ethereum network*, and send the result to the SC. We develop a wrapper program that generates state files for each computation step together with the final result and stores them on the servers. This allows the servers to provide the required information for a single step computation to the SC without the need for any extra processing (see Sect. 4.1 for details).

(iii) *Referee* is a smart contract (SC) that runs on Ethereum blockchain.

As a concrete computation we consider multiplication of two matrices. We measure the execution cost of each server in terms of the *running time*, and the financial cost of executing the referee smart contract on the Ethereum network. Our results are reported in Sect. 4.3. For honest executions, the main cost is the program execution time of the servers, and the financial cost of verifying the results by the SC which is small in our system. For non-matching results, financial cost grows with the size of the computation steps and rounds of binary search stage. This cost can be charged to the malicious server to disincentivize dishonest behaviour.

### 1.2 Related Work

Verifiable delegation of computation with a single server has been studied based on cryptographic proof systems [12,14,21,35,38–40,42,43] which are based on interactive proofs ( [5,22,34]) and argument systems ( [8,27,31,32]). Please see Appendix A for more details on these schemes.

Canetti et al. [10,11] introduced refereed delegation of computation (RDoC) with two servers that is based on the *refereed game model of computation* [17] where a computationally bounded honest referee must decide on the honesty of one of the two computationally unbounded players. Canetti et al. protocol, CRR, guarantees correctness of the results if at least one of the provers is honest. In all above systems the client and the verifier are the same entity and is trusted.

*Using smart contract* for managing outsourced computation through refereed game model has been considered in systems such as TrueBit [41], HB [24], Plasma [37] and Arbitrum [29]. All these works use multiple servers to run the computation off-chain and use a smart contract to work as a referee. Additionally, they consider rational adversary model and do not provide provable correctness guarantee for the outsourced computation. In contrast, in [4] authors adapted CRR to the smart contract setting for malicious adversaries, identified copy attack and proposed scCRR with two servers that uses additional challenge and response in the CRR protocol to achieve correctness and soundness. In cmCRR we reduce interaction with, and computation of, the SC.

**Organization.** Section 2 reviews preliminaries, Sect. 3 describes our system. Section 3.2 gives security and privacy analysis of our system, and Sect. 4 describes our implementation. Section 5 concludes the paper.

## 2 Preliminaries

The building blocks of our system are, a family of *collision-resistant hash functions* $H$, a commitment scheme, and a Merkle tree that is used as a commitment. Detailed definitions of these primitives are in Appendix B. We also give a brief definition of the commitment scheme, and Merkle trees below.

A commitment scheme consists of two algorithms $Commit()$ and $Open()$. $Commit()$ is a probabilistic algorithm that takes an input value $x \in \{0,1\}^*$ and outputs a public commitment $com$ and a private opening $d$. $Open()$ takes the commitment $com$, the private opening value $d$, and the input value $x$, and outputs 0 or 1, which indicates an invalid or a valid commitment, respectively. We require that the commitment scheme satisfies two properties, *hiding* and *binding*.

*Merkle Tree and Merkle Proof.* A Merkle tree is a binary tree that is constructed using a *collision-resistant* hash function where the leaves are the hash values of data elements $d_1, \cdots, d_n$. Every internal node of the tree is the hash of the concatenation of its two child nodes. The root of the Merkle tree is denoted by $r = MH_{root}(D)$, where $D = \{d_1, \cdots, d_n\}$. Given the root $r$ of a Merkle tree, to prove that an element $d_i$ is included in the tree, a *Merkle proof*, $p_i = MH_{proof}(D,\ d_i)$, is used, which consists of the hash values along the path from

$H(d_i)$ to the root. The $VerifyMHProof(r,\ i,\ d_i,\ p_i) = 0/1$ function verifies whether $d_i$ corresponds to an element in position $i$ of a Merkle tree with root $r$ using the proof $p_i$.

CRR uses *Turing Machine (TM) representation of computation* and stores the state of computation by a *configuration of TM* which is an ordered triple $(t,\ s,\ h) \in \Sigma^* \times \mathcal{K} \times N$ where $t$ denotes the tape content, $s$ denotes the machine's current state, and $h$ denotes the position of the head on the tape, where $0 \leq h < |t|$. Given an initial description of the TM, and a configuration $C_i$, the next configuration $C_i$ can be easily determined. See Appendix B for details.

*CRR* [10] uses a *referee protocol* to determine the honest server. The protocol is an interactive challenge-response protocol that determines the first single step computation of the servers that do not match, and is performed by the honest referee to determine the correct server. More details are in Appendix C.1.

*Ethereum* [9] is a public distributed ledger system that enables users to develop smart contracts and decentralized applications (DApps). A smart contract is a computer program that is executed by each Ethereum node on their EVM. The computation cost of EVM is measured by *gas* which is the unit cost of executing an operation on the Ethereum, and translated into *Ether*, the Ethereum currency.

## 3 Fides System Design

In this section, we give the security goals, the protocol description and the security analysis of cmCRR system.

**Definition 1** [10]**.** *$(S_1, S_2 \cdots S_n, R)$ is an $\epsilon$-RDoC with $n$ servers and referee $R$ for a function $f$ if the following holds:*

- *For any input $x$ and for any $i \in [1, n]$, if $S_i$ is honest, then for any $(S_1^*, S_2^* \cdots S_{i-1}^*, S_{i+1}^*, \cdots S_n^*)$ the output of $R$ is $f(x)$ with probability at least $1 - \epsilon$.*
- *The complexity of the client is at most quasi-linear in $|x|$ and the complexity of the (honest) servers is polynomial in the complexity of evaluating $f()$ (for example if the complexity of evaluating $f()$ is $O(|x|^2)$ then the complexity of the servers are polynomial in that, e.g. $O(|x|^2)$.)*

We design cmCRR protocol that outputs correct result, or $\perp$ assuming at least one honest server exists (this is done by identifying a deviating server, including the copying server). cmCRR protocol satisfies the following security properties.

- *Correctness.* Client always receives the correct result if the honest server runs the computation (if the malicious server aborts at the initialization stage, $\perp$ is returned to the client).
- *Soundness.* The malicious server will be detected and the correct computation result will not be affected.

Soundness includes copy detection which is important to ensure security of the replicated computation that requires two independent results, as well as incentivizing the honest behaviour.

### 3.1 cmCRR Protocol

We propose cmCRR, an SC-RDoC scheme that uses an smart contract as a referee which interacts with a client $C$, two servers $S_1$ and $S_2$ and ensures correctness and detection of misbehaviors. We assume, there is a secure channel between each server and the client. We assume servers have reliable computation environments and they publish their results after running the computation. Each entity in the system has a public key $Pk_i$ (or pseudonym which can be obtained by a hash function on $Pk_i$ and taking the first $\kappa$ bytes where $\kappa$ is a security parameter) and it is recognized based on that. We do not require a public key infrastructure and the public keys that are used for communicating with the underlying blockchain system suffices.

**Details.** Algorithm 1 shows the simplified protocol description. We describe each stage below:

**Initialization.** Client $C$ transforms the function $f()$ to its Turing machine description which we denote by $M$, $M = TMencode(f())$. $C$ also computes a commitment to the input $x \in \{0,\ 1\}^{\kappa}$, denoted by $com_x$, and sends $(com_x,\ M)$ to SC, and a message to servers $S_1$ and $S_2$ to notifies them to register. Then, servers register by sending their public keys $Pk_i$ to SC which will be stored there. When both servers are registered, $C$ opens its commitment to the input by sending $(x,\ d_x)$ to both servers through the secure channel. $d_x$ is the opening value for the commitment. Both servers verify that the opening of commitment is correct $(Open(com_x,\ d_x,\ x) = 1)$. If any server finds a conflict, then they abort. Otherwise, they accept the computation job by sending an accept message to SC. When SC receives the accept message from both servers it notifies the servers to start the computation.

**Computation.** Both servers perform the computation and obtain the result $y_i$. To reveal the computation result, servers follow two steps to prevent copy attack for both computation result and binary search stage. In the first step, they make a commitment to $y_i$, $(com_{y_i},\ d_{y_i}) = Commit(y_i||Pk_i)$ and construct a Merkle tree on the hashed reduced configuration arrays of their TMs, $\mathcal{C}_i$, and their public key $Pk_i$. Let's represent the root of this Merkle tree as $MH_{root}(\mathcal{C}_i||Pk_i)$. Each server sends the tuple $(com_{y_i},\ N_i,\ MH_{root}(\mathcal{C}_i||Pk_i))$ to SC, where $N_i$ is the length of the reduced configuration array. Then, SC asks both servers to reveal their commitments. Each server will reveal $(y_i,\ d_{y_i})$ which enables SC to check whether its a valid opening under each server's public key. SC checks if $y_i$ received from both servers match with each other or not.
(a) If both $Open(com_{y_i},\ d_{y_i},\ y_i) = 1$ for $i \in \{1, 2\}$ and $y_1 = y_2 = y$, SC sends $y$ to the client as the computation result. (b) If $Open(com_{y_i},\ d_{y_i},\ y_i)$ is equal to 0 for one server (and 1 for the other server) but $y_1 = y_2 = y$, SC identifies the server who has revealed an incorrect commitment as copying server. SC sends $y$ to the client. (c) If $Open(com_{y_i},\ d_{y_i},\ y_i)$ is equal to 0 for one server (and 1 for the other server) and $y_1 \neq y_2$, SC identifies the server who has revealed an incorrect commitment as malicious and the other server as honest. Then, it sends the result of the honest server to the client. (d) If $Open(com_{y_i},\ d_{y_i},\ y_i) = 1$ for $i \in \{1, 2\}$ but $y_1 \neq y_2$, they go to the dispute resolution stage which is as follows:

**Algorithm 1.** *cmCRR Protocol*: Client $C$ has function $f$ and input $x$, servers $S_1$ and $S_2$ receive $f$ and $x$, and compute $f(x)$ and return output $y_1$ and $y_2$ to referee contract $SC$.

**Input:** Computation function $f()$ and input $x$
**Output:** Computation result $y$
**Initialization: Computation initialization and server registration**
1: $C \rightarrow SC$: Initialize $(com_x,\ M)$ ▷ $M$: Turing machine description of $f()$ ▷ $com_x$: commitment to the input $x$
2: $S_i \rightarrow SC$: Register $(Pk_i)$ $(\forall\ i \in \{1,\ 2\})$
3: $C \Rightarrow S_i$: Open $(com_x,\ d_x,\ x)$, $\forall i \in \{1,2\}$ ▷ $d_x$: commitment opening value ▷ $\Rightarrow$: secure channel
4: $S_i \rightarrow \mathcal{SC}$ : *accept*, $\forall i \in \{1,2\}$
**Computation: Execution and result comparison**
5: $S_i \rightarrow \mathcal{SC}$: receive-Commitments $(com_{y_i},\ N_i,\ MH_{root}(\mathcal{C}_i||Pk_i))$, $\forall i \in \{1,2\}$
6: $S_i \rightarrow SC$: receive-Results $(y_i,\ d_{y_i})$, $\forall i \in \{1,2\}$ ▷ $y_i$, $d_{y_i}$, $N_i$: result of computation, opening key, length of $\mathcal{C}_i$
7: $SC$: compare $y_i$ and verify commitment to $y_i$
- If (opening valid) & $(y_1 = y_2) \rightarrow$ no attack
- Else if (opening invalid) & $(y_1 = y_2) \rightarrow$ copy attack
- Else if (opening invalid) & $(y_1 \neq y_2) \rightarrow$ incorrect opening
- Else if (opening valid) & $(y_1 \neq y_2) \rightarrow$ go to Interaction stage

**Dispute resolution: Binary search and single-step execution**
– **Binary search:**
8: While $n_b \neq n_g + 1$ do:
$SC \rightarrow S_i$: get-Reduced-config$(j)$ $\forall i \in \{1,2\}$
$S_i \rightarrow SC$: receive-Reduced-config$(H^i_{rc_j},\ p^i_{rc_j})$, $\forall i \in \{1,2\}$ ▷ $H^i_{rc_j}$, $p^i_{rc_j}$: Hashed reduced configuration from server $i$, MProof for $H^i_{rc_j}$
SC acts as below:
- If (both hashes match) & (valid MProof) $\rightarrow$ update $n_g$
- Else if (both hashes match) & (invalid MProof) $\rightarrow$ copy attack
- Else if (hashes mismatch) $\rightarrow$ update $n_b$

– **Single-step execution:**
9: $S_1 \rightarrow \mathcal{SC}$:receive-Single-step-data $(rc_{n_g},\ rc_{n_b})$
SC performs one step execution and identifies the correct result and the malicious server

**Dispute Resolution.** SC takes the smallest reduced configuration length among $N_i$, where $i = \{1, 2\}$, as $n_b$ (i.e. the mismatching point), also lets $n_g = 1$ (i.e. the matching point), and specifies the index of the computation step that both parties should reveal their reduced configuration through bisection, $j = n_g + \frac{n_b - n_g}{2}$. Each server $S_i$ will send the reduced configuration $H(rc^i_j)$ for computation step $j$ together with a proof $p^i_{rc_j}$ that shows $H(rc^i_j)$ is in the Merkle tree having root $MH_{root}(\mathcal{C}_i||Pk_i)$ (this prevents copying during the interaction). SC checks if the hashed reduced configurations, $H(rc^i_j)$, of both servers match or not. (a) If they match it verifies whether the Merkle proof $p^i_{rc_j}$ is valid. If both Merkle proof are valid there is no copy attack, then SC updates $n_g$ (let it be the new matching point) and repeats the bisection again. (b) Else, if one

Merkle proof is invalid the copy attack is identified and SC sends the result of the honest server to the client. (c) Otherwise, if the reduced configurations do not match, SC updates $n_b$ (the new non-matching point) and computes the next computation step $j$ that servers have to reveal their reduced configuration through bisection again.

The bisection procedure and challenge-responses are continued as described above until they reach to $n_b = n_g + 1$. At this stage, SC asks server $S_1$ to reveal the reduced configurations for computation steps $n_g$ and $n_b$, i.e. $rc_{n_g}$ and $rc_{n_b}$, in plain. Then SC runs *VerifyReducedStep* algorithm (please see Appendix C) which identifies the honest server based on the single-step computation.

### 3.2 Security and Privacy Analysis

In the following we show the that cmCRR protocol ensures the security requirements mentioned in Sect. 3.

*Copy Detection.* A dishonest server can try to do the copy attack in two situations- (i) *copy the final result in computation stage* and (ii) *copy during the binary search stage* which will be detected by the SC.

**Theorem 1.** *Let $H$ be a collision resistant hash function that is used to construct the Merkle hash tree on the array of reduced configurations, $\mathcal{C}$, commit() be a hiding and binding commitment scheme and $Pk_i$ be the public key of server $i$ which uniquely identifies Server $i$. Then cmCRR protocol provides protection against copy attack of honest cloud for (i) the computation stage where the results are published and (ii) the binary search stage where the challenge-response is run.*

*Proof.* (Sketch) First consider the computation stage, where each server reveals a commitment to their result and their public key, $commit(y_i||Pk_i)$, in one round and then they open their commitments in the next round. Consider a copy attack where server $S_1$ has published the $com_1 = commit(y_1||Pk_1)$ and server $S_2$ has copied the result. To successfully copy the result, $S_2$ has to either (i) find $y_1$ and make a commitment to $y_1$ using its own public key $Pk_2$, or (ii) it has to set $com_2 = com_1$ and later opens the commitment $com_2$ to $y_1||Pk_2$ where $Pk_2 \neq Pk_1$. Therefore, the success probability of such an attacker is bounded by the maximum probability of breaking either the hiding (case i) or binding (case ii) property of the commitment scheme.

Second consider that server $S_2$ wants to run a copy attack in the binary search stage. Let's assume server $S_1$ has revealed $(com_{y_1},\ N,\ root_1)$ after performing the computation and $S_2$ reveals $(com_{y_2},\ N,\ root_2)$ where $y_1 \neq y_2$ and $root_1 = MH_{root}(\mathcal{C}_1||Pk_1)$. W.l.o.g we set $N_2 = N_1 = N$. Now consider that $S_1$ reveals $(H(rc_j^1),\ p^1_{rc_j})$ at the computation step $j$. If $S_2$ copies $H(rc_j^1)$, then it has to provide a proof $p^2_{rc_j}$ such that it shows $H(rc_j^1)$ is in the Merkle tree with root $root_2$ where $root_2$ includes $Pk_2$, i.e. $root_2 = MH_{root}(\mathcal{C}_2||Pk_2)$. Let $S_1$ generate a proof $p^1_{rc_j} = (h^1_1, h^1_2, ..., h^1_{log(N)})$. If $VerifyMHProof(root_1,\ H(rc_j^1),\ Pk_1,\ p^1_{rc_j}) = 1$, $S_2$ can copy some part of this proof. However it needs to generate a proof, $p^2_{rc_j} = (h^2_1, h^2_2, \cdots, h^2_{log(N)})$ that matches with $root_2$. Because for $S_2$ the Merkle

proof is valid if it includes $Pk_2$, $S_2$ can copy all the nodes in the path to root $root_1$ except the node before root which is the sibling of $Pk_1$ (i.e. $h^1_{log(N)}$). If the proof revealed by $S_2$ is verified without being detected as malicious, then there should be either:

(a) The value $h^2_{log(N)}$ such that it is different from $h^1_{log(N)}$ but $H(h^2_{log(N)}||Pk_2) = H(h^1_{log(N)}||Pk_1) = root_1$, where $root_2 = root_1$ and $S_2$ has copied $root_1$, or
(b) The value $H(h^2_{log(N)}||Pk_2) = root_2$, where $root_2$ was guessed by $S_2$ initially.

In case (a), $S_2$ successfully breaks the collision resistance property of the hash function and in case (b) $S_2$ breaks the pre-image resistance property of the hash function. Therefore, the success probability of $S_2$ is bounded by the maximum of the two which is negligible. □

Using Theorem 1 above and Theorem 1 in [10], we have the following:

**Corollary 1.** *Assuming the hash function is collision resistant, the commitment scheme is hiding and binding, and $Pk_i$ uniquely identifies server $S_i$ then cmCRR protocol is a computationally sound $\epsilon$-RDoC with two servers for any function computable in polynomial time.*

**Discussion.** cmCRR scheme also ensures *input privacy* when both parties are honest and *security against malicious client* who may try to send different inputs to different servers (please see Appendix D for the discussion on this and the list of attacks that are prevented in our scheme).

## 4 Implementation of Fides

We have performed a proof-of-concept implementation of cmCRR system that includes a client who delegates a computation to two servers and uses SC as a referee. We call it "*Fides*" that works directly with EVM instructions following the Ethereum execution model. In the following sections we provide our implementation details of Fides and the obtained evaluation results.

### 4.1 Challenges and Design Choices

A significant challenge of implementing the cmCRR protocol is specifying a single computation step in an Ethereum Virtual Machine (EVM), and delegating execution of that step to the referee smart contract such that it produces the same result as the honest server's EVM. Details are given below.

***(i) Single-step execution on Ethereum network***: On Ethereum, each node runs the code inside an Ethereum Virtual Machine (EVM) that understands EVM bytecode only [45]. If the delegated program is written in languages such as C/C++ then to execute the single step of computation, the instruction in C/C++ must be compiled into an equivalent EVM instruction to be

run on the Ethereum blockchain. To achieve this conversion (C/C++ code to EVM bytecode), a software (e.g., a compiler) is required that will work as a bridge between the C/C++ code and the referee contract that is on the blockchain. This requirement makes the single-step execution on Ethereum challenging because an instruction of a C/C++ and its corresponding execution state (e.g., in 8086 architecture) needs to be transformed in to an equivalent EVM execution state. Moreover, it brings another security issue, that is, we need to prove to the referee contract that the transformation is done correctly (which is done off-chain by the servers).

*(ii)* ***Deterministic execution of delegated program***: CRR [10] and scCRR [4] assume that each execution of the delegated program should be deterministic (i.e., each execution of the program for a fixed input should produce the same result). However, there exists some instructions in the high-level language (e.g., C/C++, Solidity etc.) that are non-deterministic by definition (i.e., they can fetch data from outside). If such instructions are included in the delegated program then execution of the same code can produce different/wrong output having non-identical intermediate step results. Therefore, the presence of non-deterministic instructions will make it challenging for the referee contract to identify the correct result (in case of a dispute).

We have made the following design choices for the implementation of *Fides*:

- Since the referee contract needs to run the single-step computation on Ethereum, we restrict ourselves to *Solidity* language for writing the delegated program whose compiled binaries are equivalent to EVM bytecode. We also require each server to run the computation bytecode inside their private EVM which is not part of the Ethereum network. In this way, the required single step data can be sent directly to SC without the need for any conversion.
- To ensure deterministic program execution, we assume that the input to the delegated program is hard-coded (i.e., input is part of the program itself). Also, if $f()$ is a random function, the client can hard-code an initial random seed in the delegated program and use a pseudorandom number generator within the program to generate the required randomness. This can guarantee that both servers will generate and use the same randomness. Additionally, we note that the program should not use any instructions that requires the program to interact with outside of the current (and local) EVM execution environment. In Appendix E.1 we provide the categories of available EVM instructions in Ethereum and identify the instructions from these categories that are non-deterministic by definition (e.g., `EXTCODECOPY`, `DELEGATECALL`, `TIMESTAMP` etc.) and should not be used in the delegated program.

**Adaptation of the Protocol for Ethereum Execution Model.** The EVM is a stack-based architecture. Its execution environment consists of a virtual ROM (where the program code is loaded) and three locations to store/access data during execution: *stack*, *memory*, and *storage* [45]. We consider the TM equivalent of an EVM configuration to be as follows:

- The computation is represented as *bytecode* that is the machine code to be executed inside EVM. Each step of the protocol is an execution of a single EVM instruction.
- The EVM machine *state* $\mu$ that is equivalent of a TM state includes: (i) current contents of the stack ($\mu_s$) and memory ($\mu_m$), (ii) current values of the program counter ($pc$), and (iii) the available gas ($g$).
- Finally, the TM working *tape* is replaced with the contents of the storage of EVM. Before the execution of the program, the storage and the gas values are initialized with the user defined values and others are set to 0.

Suppose, $\delta$ is the EVM state-progression function (i.e., transition function). Then $\delta\ (\mu, I_\mu) = \mu'$ represents that the machine moves from state $\mu$ to the next state $\mu'$ by executing the instruction $I_\mu$. In other words, the machine state moves from $\mu$ to $\mu'$ in one step. Here, $I_\mu$ is the instruction to be executed in state $\mu$.

We define a *reduced-configuration* ($rc$) to be the tuple $[pc, (\mu_s, \mu_m), S]$, where $S$ represents the EVM storage contents (i.e., TM tape contents). Our protocol requires that, given a reduced configuration, the referee smart contract should be able get the next reduced configuration by simulating single EVM instruction of the computation on Ethereum. Based on the above discussion, given a reduced-configuration $rc_j$ for state $j$, the referee contract generates the next reduced-configuration $rc_{j+1}$ by executing a single EVM instruction $I_j$ as follows: $rc_{j+1} = \delta(rc_j, I_j)$.

Therefore, given two consecutive reduced configurations, $rc_j$ and $rc_{j+1}$, one can efficiently verify this claim by doing the following checks:

(1) Get the next reduced configuration value $v'$ by executing the EVM instruction at step $j$ on the data provided in $rc_j$ (in our implementation we do this by running a new copy of EVM constructed using the instruction $I_j$ and the data of $rc_j$, by the referee contract. Details of this technique is given in Sect. 4.1).
(2) Verify $v' = v_2$, where $v_2$ $(= S^{(j+1)})$ is the claimed next state output by the server.
If the above checks pass, this server is considered as *honest* and its result is accepted as correct. Otherwise, it is considered as *malicious* (and result of the other server is accepted as correct with the assumption that at least one of the servers is honest).

### 4.2 Fides System Overview

***Actors***. There are three main actors in the *Fides* system.

(i) The **client** is the outsourcer of the computation who writes the delegated program. To communicate over Ethereum network the client must have an Ethereum address.
(ii) The **server** is the entity that executes the outsourced computation and returns a result. In Fides we require two servers ($S_1$ and $S_2$). Each server executes the computation inside an EVM that is isolated from the Ethereum

network. To participate in the protocol each server must have an Ethereum address.

(iii) A **referee** is a *smart contract* on the Ethereum blockchain. This contract enforces the verifiable computation algorithm. It also manages the interactions between the client and the servers.

Details of the functionalities of the Fides actors are described in Sect. 4.2.

**Functions in *Fides*.** Functionalities in Fides can be divided into three main parts: (1) *FidesClient* (i.e., represents the functionalities of the client), (2) *FidesServer* (i.e., represents the server functionalities) and (3) *FidesReferee*. The steps of the cmCRR protocol is presented in Algorithm 7 in Appendix E.5.

**(1)** ***FidesClient*.** The client writes the source code of the delegated program, *program.sol*, in Solidity and compiles the program into *bytecode* (also called *EVM code*. We denote this as $B_{f()}$. Then the client generates commitment $com_x$ to the *input* $x$ and initializes the computation process by calling the *initialize()* function of the referee contract with the necessary arguments and sends program to each of the servers (see Sect. 3.1 for details). Algorithm 2 in Appendix E.2 shows the complete pseudo-code of client's functions in cmCRR protocol.

**(2)** ***FidesServer*.** Each server interested in solving the problem needs to register for the computation by calling the *Register ()* function of the referee smart contract. Once registered, it receives the program from the client, and checks the commitments and inclusion of any non-deterministic EVM instructions to ensure determinism of the execution. If the checks pass, the server executes the computation inside its private EVM and sends its result to the referee contract. In case of inconsistency, during each round of the binary search phase, each server $S_i$ will send the hashed reduced configuration $H_{rc_j} = H(rc_j)$ for a requested index $j$ by the referee together with a proof $p_{rc_j}$. See Sect. 3.1 for details of computation and dispute resolution process.

After the binary search stage is completed (when $n_b = n_g + 1$), the referee contract simulates single-step of the computation. It asks one of the servers to reveal the reduced configuration $rc_{n_g}$ and $rc_{n_b}$ in plain. To allow the referee to execute the single-step of the computation, the reduced configuration data must be sent as a deployable EVM code. For example to construct the deployable code for step $n_g$ it does the following:

*(a)* Include additional EVM instructions (as bytecode) that will copy the stack, memory and storage values from $rc_{n_g}$ into the Ethereum's EVM stack, memory, and storage, respectively.

*(b)* Append the bytecode of the instruction to be executed at step $n_g$ ($I_{n_g}$) with the bytecode from step (a). We denote this bytecode as $B_{n_g}$.

*(c)* Prepend the bytecode ($B_{n_g}$) with an *initialization* code $c_{n_g}$ that will enable the referee to initialize a new EVM.

The above steps can be performed using the EVM-tool [15]. Finally, the server will send the deployable bytecode ($c_{n_g} B_{n_g}$) to the referee

contract for single-step execution. Server's functions in cmCRR protocol are shown in Algorithm 3 in Appendix E.3.

**(3)** ***FidesReferee.*** The referee functionalities in our system are written as a smart contract using the Solidity language and deployed on local Ethereum network. Functions of the referee contract are shown in Algorithms 4, 5 and 6 (for lack of space we divided the functions of the referee contract into 3 algorithms. All the functions in these three algorithms reside inside a single contract. See Appendix E.4 for more details). The referee smart contract performs the following functions, (i) registers two servers interested in solving the computation problem, instructs them to perform the computation and receives their returned commitments to the results, (ii) compares the results of the two servers, (iii) verifies the commitments from the servers, and (iv) in case of result mismatch, finds the step of the computation of the two servers' results that do not match by performing a binary search and then identifies the correct result by executing the disputed computation step itself on Ethereum. Single step execution by the referee contract is implemented as below.

*Single-Step Execution.* In *Fides*, each step of the protocol is an execution of a single EVM instruction. Once the referee contract finds the the step where the servers disagree, it asks $S_1$ to reveal the values for two consecutive reduced states, $rc_{n_g}$ and $rc_{n_b}$ (s.t. $n_b = n_g + 1$). In response, it receives the deployable bytecode $c_{n_g}B'_{n_g}$ of the reduced configuration $rc_{n_g}$ along with its claimed output $v_{n_b}$ for $rc_{n_b}$. Upon receiving $S_1$'s response, referee contract will do the following (pseudo-code is given in Algorithm 6):

*(i)* It will check whether the instruction $I_{n_g}$ at step $n_g$ is a valid instruction or not, by checking it in the published program bytecode ($B_{f()}$).

*(ii)* If the check passes, then it will deploy the state bytecode $c_{n_g}B_{n_g}$ by calling the *Deploy()* function in the referee contract that takes $c_{n_g}B_{n_g}$ as the function argument, and returns an address for the newly deployed contract. We call this *disp-state contract* (disputed state contract) that holds the runtime bytecode for the configuration $rc_{n_g}$ (provided by the $S_1$).

*(iii)* The referee contract then executes the code in the *disp-state contract* by using *(bool status, bytes memory data) = disp_state_address.call ()* and receives the execution result, denoted as $v'$. If $v' = v_{n_b}$, then the output of $S_1$ (and so its overall computation) will be accepted as correct. Otherwise, the referee contract identifies $S_1$ as the "malicious" server, and accepts the results of server $S_2$ as correct (since at least one server is assumed to be honest).

Note that, execution of *disp-state contract* will fail if it run out-of-gas or an invalid code is provided by the server. If any of these cases happen, it will be considered as a malicious behaviour. In order to prevent execution failure due to out-of-gas problem, we require the client to specify the maximum amount of gas that might be required to perform the single step execution by the referee

contract (during the initialization of the computation process). At the end, once the referee contract finds the correct result (as well as the honest server), the honest server will be paid by the client (through the referee contract) based on the amount of gas used for performing the computation.

### 4.3 Evaluation

To show practicality of our verification protocol, we conducted a number of experiments with *Fides* prototype. The experiments are designed to evaluate the execution complexity (cost) of the server in terms of *running time* and the referee contract in terms of *gas*. We implemented the smart contracts in *Solidity* v.0.5.17. The experiments are performed using *Ganache* [18] - a framework to simulate the EVM on local computer. For development of the smart contracts we used Remix IDE [26]. For our example computation we used matrix multiplication. That is the client writes the delegated program *matrix_multiplication.sol* (= $f()$) in Solidity and the input to this program is a $n$ x $n$ matrix. We evaluated performance of *Fides* client and server programs on Windows 10 with a 3.60 GHz Intel Core i7 CPU and 8 GB RAM.

**Cost Analysis.** We measured the following costs:

*(i) Program execution cost.* Our goal is to measure the *execution cost (in terms of running time) of the delegated program by a server*. Our matrix multiplication algorithm has run time $\mathcal{O}(n^3)$ for $n \times n$ matrix. The program was executed 3 times for each of the matrix sizes with different inputs and the average time is reported in Table 1.

*(ii) Financial cost of verification.* For verification, *our goal is to measure the financial cost that indicates the actual monetary cost of executing the referee contract on the Ethereum network*. We use *gas* to measure the computation complexity of the smart contract execution and then convert it into the monetary value in US dollar. We considered two scenarios: (i) servers send same results, and (ii) servers send different results.
For the first scenario, the execution cost of the referee contract for verification is limited to comparing the returned results ($y_i$) and verifying the commitments $com_{y_i}$. We used *Keccak256* hash function for the commitments. For the second scenario, the referee initiates the *dispute resolution* protocol (called *MCId-malicious cloud identification*). The execution cost in this case will depend on (i) the number of rounds during the binary-search step and (ii) the length of the Merkle proof that is sent along with the hashed reduced configuration in each round. The maximum number of rounds that can be executed is logarithmic in the number of computation steps $N$ and the maximum length of a Merkle proof for reduced configuration is also logarithmic in the size of reduced configuration array $\mathcal{C}$.

In Ethereum, the amount of gas that can be used per block (i.e., the *block capacity*) is bounded by a limit. This limit is called *block gas limit*. The *block gas limit* in Ethereum is set approximately 29.9 million[1] gas by the miners.

[1] According to etherscan.io/chart/gaslimit on November,18, 2021.

**Table 1.** Cost analysis of *Fides*. The left side of the Table shows the program execution cost (in *time*) inside EVM by each server for different matrix sizes. The right side of the Table outlines the transaction execution cost (in *gas*) of calling referee contract's functions for matrix size $n = 10$. The number of rounds in the MCId is $\lceil log_2(N) \rceil = 10$ where the number of computation steps is $N = 780$. The total cost is converted into US$s that represents only the total cost of executing the referee contract on Ethereum and excludes any transaction fee.

| **Prog. Execution cost** | | **Financial cost of verification** | | |
|---|---|---|---|---|
| **Matrix Size ($n$)** | **Time (sec.)** | **Phases** | **Gas Used** | |
| 10 | 5.06 | Initialize | 115,135 | |
| 20 | 34.40 | Register | 62,400 | |
| 30 | 111.67 | receiveResultCommitments | 109,349 | |
| 40 | 317.67 | receiveResults | 87,275 | |
| 50 | 603 | Compare | *Scenario 1 (results match)* | *Scenario 2 (results mismatch)* |
| | | verifyCommitments | 27,583 | 27,583 |
| | | MCId (dispute resolution) | - | 689,556 |
| | | **Total (Gas)** | 401,742 | 1,091,298 |
| | | **Cost (in ETH)**[a] | 0.0601 | 0.1633 |
| | | **Cost (in USD)**[b] | 240.29 | 652.73 |

[a] According to https://etherscan.io/chart/gasprice on November 18, 2021
[b] According to https://etherscan.io/chart/etherprice on November 18, 2021.

Based on the values in Table 1, the referee contract functions (*Initialize*, *Register*, *receiveResultCommitments*, and *receiveResults*) cost around 0.21% to 0.38% of block capacity.

To evaluate the cost of the *MCId* function, we executed this experiment with 3 different disagreement points. For each of these 3 executions we ran the experiment with one honest server and one dishonest server that cheats on one of the three disagreement points. The disagreement points were chosen- one at the beginning (at step $N/4$), one near the middle (at step $N/2$), and one at step $3N/4$ of the computation, where $N$ is the total number of computation steps. For matrix size 10, we had $N = 780$ and for all these 3 executions we had same number of rounds ($\lceil log_2(N) \rceil = 10$) and the same size of the single-step data ($\approx 1.2$ kB). Note that, in general the size of the single-step data depends on the size of the computation state data at the location of disagreement point, which varies between several hundred bytes to few Kilobytes for this example program. Table 1 shows that the MCId function execution cost is approximately 2.3% of the block capacity. This suggests that the cost increase due to the higher number of rounds will remain acceptable for the current block sizes in Ethereum.

In summary, the results in Table 1 demonstrate that cmCRR requires reasonable program execution time by the servers and the (financial) cost of verification

on Ethereum network is minimal. In addition, the client must pay the execution cost for *scenario 1* and the system can penalize a malicious server by asking to pay the cost difference between *scenario 1* and *scenario 2* (that increases with the size of the computation and the number of rounds in binary search phase) in order to disincentivize the malicious behaviour.

*Comparison with Other Works.* Performance of *Fides* is not comparable to the related works because majority of these works [35,39,40,42] use single server to do the computation and the other work that uses multiple server for computation is [10], which uses a different execution environment compared to Fides. On the other hand, existing works in the smart contract setting such as [24,29,37,41] consider a different model of adversary (e.g., rational adversary) which are not comparable to Fides (that considers malicious adversary) in terms of security they achieve and also lack performance measurements.

## 5 Concluding Remarks

SC-RDoC offers an attractive model for verifiable computation, that minimizes trust on the client and servers by delegating the referee task to a smart contract. We proposed cmCRR protocol which protects against copy attack and misbehavior of client, and proved its security. We discussed challenges of implementing the single step computation by the referee contract, and showed how our prototype, Fides, addresses these challenges. Our work can be extended to multiple server SC-RDoC. Interesting open questions are to provide a formal security model that can capture copy attack, to hide the single step of the computation from the SC (privacy against an observer), and to allow servers to perform their computation in a commonly used architecture such as X86 while using the smart contract for verifying a single computation step.

## A Related Works: Single Server Outsourcing Computation Schemes

Interactive proof system where a prover proves correctness of a statement to a verifier is a fundamental problem in theoretical computer science [5,22,34]. The basic proof systems assume computationally unbounded prover and efficient verifier. *Argument systems* [8,31,32] assume computationally bounded provers. Probabilistically checkable proofs (PCPs) [3] assume the prover is unbounded and the proof can be probabilistically verified by a (computationally bounded) verifier by querying a bounded number of bits in the proof. PCPs have been used to construct efficient argument systems [27,31].

Verifiable delegation of computation (aka verifiable outsourced computation) considers a weak client that delegates the computation of a function $f$ on an input $x$ to a powerful server, and can verify the correctness of the result with significantly less resources than what is required to perform the computation.

Goldwasser, Kalai and Rothblum [21] (GKR) proposed an interactive proof system to prove the correctness of the delegated computations where both the verifier and the prover are efficient. This work was extended by Cormode et al. [14] (CMT). Both GKR and CMT require the computation be represented as a layered arithmetic circuit, and provide security against an unbounded prover. Delegation of general computation using fully homomorphic encryption (FHE) has been proposed in [13,19]. Efficient single server verifiable computation systems have implemented [38–40] using an efficient argument system of Ishai et al. [27]. In these systems, the computation is written as a set of constraints (i.e., a system of equations). Pepper [40] and Ginger [39] use classical linear PCP of Arora et al. [2] and Zaatar [38] uses linear PCP based on quadratic arithmetic programs (QAPs) [20]. Vu et al. [42] proposed Allspice, where a suitable verifiable computation system is chosen among CMT and Zaatar using static analysis. Zaatar uses batching of multiple instances of a single computation to amortize the setup cost. Another concurrent implementation of a verifiable computation system is Pinocchio [35] that uses QAPs [20] for encoding computation. To be practically viable Pinocchio needs key setup whose cost will be amortized over all instances of a computation [12,44].

## B Details of Preliminaries

In this section, we provide an overview of our cryptographic building blocks and the Turing machine model of computation.

*Hash Function.* We use a family of collision-resistant hash functions $H$. Roughly speaking, a hash function $H$ is collision-resistant if there is no polynomial time algorithm that can find a collision. That means, it is computationally infeasible to find two distinct inputs $m_1$ and $m_2$ such that $H(m_1) = H(m_2)$.

*Commitment Schemes.* We require that the commitment scheme satisfies two properties, *hiding* and *binding*. Hiding guarantees that for any two messages $x_0$ and $x_1$, and $(com_0,\ d_0) = Commit(x_0)$ and $(com_1,\ d_1) = Commit(x_1)$, we have that $com_0$ is computationally indistinguishable from $com_1$. The binding property requires that it is computationally hard to find a triple $(com, d, d')$ such that for $x \neq x'$, we have $Open(com,\ d,\ x) = Open(com,\ d',\ x') = 1$.

*Turing Machine Representation of Computation.* A Turing machine $M = (\Sigma,\ \mathcal{K},\ \delta)$ is an abstract model of computation that is specified by a finite alphabet $\Sigma$, a finite set of states $\mathcal{K}$ with a special element $I$ (the starting state), and a transition function $\delta : \mathcal{K} \times \Sigma \rightarrow (\mathcal{K} \cup \{halt,\ yes,\ no\}) \times \Sigma \times \{\leftarrow,\ \rightarrow,\ -\}$. A configuration of a Turing machine is an ordered triple $(t,\ s,\ h) \in \Sigma^* \times \mathcal{K} \times N$ where $t$ denotes the tape content, $s$ denotes the machine's current state, and $h$ denotes the position of the head on the tape, where $0 \leq h < |t|$. For a Turing machine $M$ with configuration $(t,\ s,\ h)$ at any point in time, the next configuration $(t',\ s',\ h')$ at the following point in time is determined as follows. Let $(s',\ v',\ d') = \delta(s,\ v)$. The string $t'$ is obtained from $t$ by changing $v$ (or $t_h$) to $v'$. The new state is $s'$, and the new position $h'$ is equal to $h - 1,\ h + 1,$ or $h$ based on whether $d'$ is $\leftarrow,\ \rightarrow$ or $-$, respectively.

## C Function: VerifyReducedStep()

Consider that the two consecutive reduced configurations claimed by server $S_1$ are: $rc_{n_g} = (s_{n_g},\ h_{n_g},\ v_{n_g},\ p_{n_g})$ and $rc_{n_b} = (s_{n_b},\ h_{n_b},\ v_{n_b},\ p_{n_b})$. First SC checks whether the revealed reduced configurations match the hash values that had received. If one of them does not match it returns the result of $S_2$ to the client. Otherwise, if both match, then it verifies the single step computation. This approach requires that the SC stores the intermediate hashed reduced configurations. To avoid this storage, one can require that $S_1$ also reveals the Merkle proof that shows $rc_{n_g}$ and $rc_{n_b}$ are in the Merkle tree constructed on its array of hashed reduced configurations.

To verify the correctness of computation all the following steps should be completed consecutively (every verification should return True). This indicates that $S_1$ is honest (these steps are checked by $\mathcal{SC}^{\mathcal{L}d,\ \mathcal{H}}$); if any of the checks return false then the claimed consecutive execution is incorrect and server $S_2$ is honest: 1) Verify that the Merkle proofs are correct by checking
$VerifyMHProof(r_{n_g},\ h_{n_g},\ v_{n_g},\ p_{n_g})$ and
$VerifyMHProof(r_{n_b},\ h_{n_b},\ v_{n_b},\ p_{n_b})$. 2) Simulate the Turing machine, starting from $rc_{n_g}$ and obtain the next state $s'$, head position $h'$ and value $v'$ (written as $h_{n_g}$-th character). 3) Verify that $s'$, $h'$ are equal to $s_{n_b}$, $h_{n_b}$. 4) Using $p_{n_g}$ and $v'$, compute
$r' = MHroot(t_{n_g}|_{[h_{n_g}]=v'} = v')$ where $t_{n_g}|_{[h_{n_g}]=v'}$ is the tape of configuration $rc_{n_g}$ except for the value $v'$ in its $h_{n_g}$-th character. For computing $r'$ the property of Merkle hash trees are used. By knowing the Merkle proof for character $h_{n_g}$, i.e. $MH_{proof}(t_{n_g},\ h_{n_g})$, we can compute $MHroot(t')$ where $t'$ equals to $t_{n_g}$ except that the character in $h_{n_g}$ is $v'$. First, we can compute $d_{h_{n_g}} = H(v')$ which is the data (hash value) stored in position $h_{n_g}$ of the Merkle tree and by then using $MH_{proof}(t_{n_g},\ h_{n_g})$ we compute the new hash values along the path to $d_{h_{n_g}}$ (this can be done in logarithmic time in the length of $t$, without knowing the full tape). 5) Verify that the hash of the root in $p_{n_b}$ equals to $r'$.

### C.1 CRR Protocol

In CRR the *reduced configuration* for a computation step $j$ is given by $rc_j = (s_j, h_j, v_j, r_j)$, where $s_j$ and $h_j$ represent current *state* of TM and the *head* position, respectively, $v_j$ represents the value on the tape at the TM's head position, *tape[head]*, and $r_j = MH_{root}(t_j)$ is the *root* of the Merkle tree on the tape $t_j$. The reduced configuration $rc_j$ captures the computation state $s_j$.
The CRR protocol works as follows:

i) The client requests the two servers to execute the function $f$ for the input $x$. Server $i, i \in \{1, 2\}$ returns its result $y_i = f(x)$ for $i \in \{1, 2\}$ to the client after executing the function along with the number of steps ($n$) of the execution.
ii) The result is accepted as *correct* if the two results match.
iii) Otherwise, the client initiates a challenge-response protocol to detect the malicious server. The protocol has two functions: (i) *binary-search* and (ii) *verify-reduced-step*.

- The *binary-search* function takes the minimum number of steps, $n_b$, that is taken by one of the servers to compute $f(x)$, as input, and returns the index $n_g$ of the last reduced configuration matched for the two servers.
- The *verify-reduced-step* function takes two consecutive reduced configurations, $rc_{n_g} = (s_{n_g}, h_{n_g}, v_{n_g}, r_{n_g})$ and $rc_{n_b} = (s_{n_b}, h_{n_b}, v_{n_b}, r_{n_b})$ where $n_b = n_g + 1$, and the proof of consistency $p_{n_g} = MH_{proof}(t_{n_g}, h_{n_g})$ for the tree index $h_{n_g}$ from one of the servers (say $S_1$), where $t_{n_g}$ is the tape at computation step $n_g$.

Then it calls $VerifyMHProof(r_{n_g}, h_{n_g}, v_{n_g}, p_{n_g})$ to recompute the Merkel root for the configuration $rc_{n_g}$, and performs the computation from $rc_{n_g}$ till $rc_{n_b}$ and verifies the Merkel root for $rc_{n_b}$ (invoking $VerifyMHProof(r_{n_b}, h_{n_b}, v_{n_b}, p_{n_b})$).

If both the outcomes (of $VerifyMHProof()$) match, the server ($S_1$) is considered as honest and its result is accepted. Otherwise, the other server ($S_2$) is honest and its result is accepted.

CRR protocol provides following guarantees. (1) If any server cheats, the client can successfully detect the cheating assuming collision-resistant hash function used for Merkle tree construction and obtain a publicly verifiable proof of this fact. (2) The client can efficiently determine the correct result as long as there exists a single honest server.

## D Attacks Prevented in cmCRR Protocol

Table 2 shows the summary of some of the important attacks that are considered against cmCRR protocol and how the protocol protect against them.

The cmCRR protocol satisfies the following additional properties:

*Input Privacy.* Our scheme also provides input privacy with respect to smart contract when the servers are honest. The reason is that the servers receive the input from the client through a secure communication channel and only a commitment to the input is published to SC. As the commitment scheme is hiding the leakage to SC is negligible. Additionally, if both servers are honest they output the same result which is accepted by SC and returned to the client.

*Security Against Malicious Client.* A malicious client may try to send different inputs to the servers. However, we assume that client has to first commit to the input in the SC and later opens it. If client tries to open the commitment in different ways it will be detected with non-negligible probability.

**Table 2.** Summary of considered attacks and our proposed solutions.

| Attacks | Our Proposed Solutions |
|---|---|
| **(Attacks by the malicious client)** | |
| Sending different/wrong computation/input to the servers | Client should publish the computation and commitment to the input on the Blockchain. |
| Designing probabilistic computation | Server should check the computation for the set of instructions that are probabilistic. |
| **(Attacks by the server(s))** | |
| Sending wrong result to SC | Using MCId protocol of CRR and executing single-step on Ethereum network. |
| Copy final result | Using commit and reveal approach for computation result and using a public key to construct the Merkle root $MH_{root}(C_i \| Pk_i)$. |
| Copy during binary search | By hashing the reduced configuration, and using a Merkle proof for the reduced configuration. |
| Sending incorrect single-step information<br>(i) incorrect instruction<br><br>(ii) out-of-gas for bad code | Requires followings:<br>(i) check by the referee contract for the validity of the instruction $I_{n_g}$ in the published program bytecode.<br><br>(ii) the client specifies in SC the maximum amount of gas that might require for singles-step execution and referee checks whether the gas sent by the server is less or equal to this amount. |
| False complain against the client's commitments | Abort the computation process and re-outsource. |

# E Fides

## E.1 EVM Instructions

The EVM is a stack-based machine with a predefined set of instructions [45]. Inside an EVM, a smart contract is simply a series of EVM instructions, which are executed sequentially. The execution environment of EVM consists of a virtual ROM (where the program code is loaded) and three locations to store/access data during execution: stack, memory, and storage. In [45] all EVM instructions are divided into the following categories (Table 3):

**Table 3.** Categories of EVM instructions. (See Appendix H.2 of [45] for the complete list of EVM instructions for each of the above categories.)

| Categories | Example Instructions |
|---|---|
| Arithmetic/comparison/bitwise logic operations | `ADD`, `SUB`, `MUL`, `DIV`, `GT`, `LT`, `AND`, `OR` |
| Stack, Memory, Storage and Flow operations | `PUSH`, `POP`, `DUP`, `SWAP`, `MLOAD`, `MSTORE`, `SLOAD`, `SSTORE`, `JUMP` |
| Hashing, logging and stop operations | `SHA3`, `LOG`, `STOP` |
| Environmental information operations | `ADDRESS`, `BALANCE`, `CALLDATACOPY`, `CODECOPY`, `GASPRICE`, `EXTCODESIZE`, `EXTCODECOPY` |
| Block information operations | `BLOCKHASH`, `TIMESTAMP`, `NUMBER`, `DIFFICULTY`, `GASLIMIT` |
| System operations | `CREATE`, `CALL`, `CALLCODE`, `RETURN`, `REVERT`, `DELEGATECALL`, `SELFDESTRUCT` |

Based on the above categories we require that the program will not include following instructions (to prevent a malicious client from constructing a probabilistic computation):

- Environmental information operations such as `EXTCODESIZE` and `EXTCODECOPY` that can get an external account's size and code, respectively, during the execution.
- All the block information operations that can get external blockchain related data which can be used to make the program randomized during execution.
- System operations such as `CREATE`, `CALL`, `DELEGATECALL` that are used to create a new smart contract account or message-call to an account with an external account's code.

Since these instructions involve creating/calling codes/data that are outside of the current program execution environment of EVM, therefore, these instructions should not be used in the delegated program. If any such instruction is identified, the server will not execute the delegated program and report it to the smart contract.

### E.2 Fides Client Functions

The client functionalities in Fides are shown in Algorithm 2.

### E.3 Fides Server Functions

Algorithm 3 shows Fides server functionalities that include: *(i) Registration.* Each participating server will install an Ethereum client (e.g., `geth`) that connects it to the Ethereum network. This client allows the server to work as an Ethereum node having its own EVM. Each server interested in solving the problem will then register for the computation by calling the *Register ()* function of the referee contract and sends its public key $Pk$ (that is used for generating the Ethereum address of the server) as the function argument. Once both servers are registered, the referee contract will update the state of the computation to "Compute" from "Init" and then the servers will receive the computation related information for execution from the client.
*(ii) Check commitments and deterministic computation.* After registration, each server receives the computation information from the client and checks its validity by verifying the committed values on the blockchain. The servers will verify that the $com_x$ match to what they have received. If any server finds a conflict, then it will aborts. In addition, each server will also check whether the program includes any EVM instructions that produce non-deterministic output.
*(iii) EVM execution of computation and send result.* After the completion of the above steps, the server will perform the computation inside its EVM. Note that, EVM computation is private and server's EVM is not part of the Ethereum network. The computation bytecode will be sent as the input to the EVM. During the execution, the bytecode is split up into its bytes and each byte is either an EVM instruction or data. Once the execution in EVM is complete, the final computation result (along with the intermediate computation results) will be stored in the storage of EVM. Then the server will obtain the $root = (H(MH_{root}(\mathcal{C}_i) \mid\mid H(Pk_i))$ constructed on hashed reduced configuration array

**Algorithm 2.** Client functions in cmCRR protocol.

**Input:** Solidity program $f()$ and input $x$
**function** MAIN($f()$, $x$) **returns** ($result$)
    /* Step 1: Program compilation and generating commitments */
    Compile $f()$ using *solc* compiler and get the program bytecode $B_{f()}$.
    Get ($com_x$, $d_x$) = GENERATE-COMMITMENT ($x$);

    /* Step 2: Get Referee contract's address and publish computation data to blockchain. */
    referee = web3.eth.contract(address="0x< $referee\_contract\_address$ >");
    PUBLISH-TO-BLOCKCHAIN ($B_{f()}$, $com_x$, $referee$);

    /* Step 3: Send computation data to servers */
    Connect to servers;
    **for** `server in connections` **do**
        `server.send(`$x$`,` $B_{f()}$`,` $d_x$`);`  ▷ $d_x$: commitment key
    /* Step 4: Get computation result from blockchain */
    Get $result$ = GET-COMPUTATION-RESULT($referee$);
    return $result$;

**function** GENERATE-COMMITMENT($message$) **returns** ($com$,$key$)
    Get $key = getrandbits(256)$;
    Set $msg = message$;
    Get $com$ = **keccak256** ($key$ + $msg$);  ▷ $com$: commitment value
    return ($com$,$key$);

**function** PUBLISH-TO-BLOCKCHAIN($program\_bytecode$, $com_x$, $referee$)
        ▷ Send transaction that invokes referee contract's Initialize() function
    referee.functions.Initialize($program\_bytecode$, $com_x$).transact();

**function** GET-COMPUTATION-RESULT($referee$) **returns** ($final\_result$)
        ▷ Get the *final result* by calling referee contract's getResult() function
    Get $final\_result$ = referee.functions.getResult().call();
    return $final\_result$;

$\mathcal{C}_i = \{H(rc_1), H(rc_2), \ldots, H(rc_{N_i})\}$ and the hash of public key $Pk_i$ of server $S_i$ for $i \in \{1, 2\}$. Here, $N_i$ is the total number of reduced configurations of server $S_i$. The server will also generate a commitment to its result $y_i$, $com_{y_i} = Commit(y_i) = H\ (d_{y_i}\ ||\ y_i\ ||\ Pk_i)$, where $H$ is a collision-resistant hash function. Finally, the server will send ($com_{y_i}$, $root$, $N_i$) to the referee contract. Then, when the server receives the request from the referee for revealing the commitments, it will reveal ($y_i$, $d_{y_i}$) pair which enables the referee to check whether its a valid opening.

Note that, at a time an EVM can process a single call with the provided bytecode (i.e., only executing one computation at a time). Therefore, if a server wants to run two computations received from two different clients, the EVM needs to be

initialized (i.e., set the stack, memory, and program counter to zero) for each new computation separately depending on the order of the computation request received from the client. However, the storage of the EVM will store the data of all the computations in the address assigned for each computation.
*(iv) Send reduced configurations.* In case of inconsistency, each server $S_i$ will send the hashed reduced configuration $H_{rc_j} = H(rc_j)$ for a requested index $j$ by the referee together with a proof $p_{rc_j}$ that shows $rc_j$ is in the Merkle tree having root $(H(MH_{root}(\mathcal{C}_i) \mid\mid H(Pk_i))$.
*(v) Send single-step data.* After the binary search stage is completed (when $n_b = n_g + 1$), the referee contract simulates single-step of the computation. It asks one of the server to reveal the reduced configuration $rc_{n_g}$ and $rc_{n_b}$ in plain. To simulate the single step, the referee contract will create a new EVM and present the received data as a new smart contract that will be run in the new initialized EVM (more details on single-step execution is described in Sect. 4.2). To allow the referee performing this process, the server must send the reduced configuration data as a deployable EVM code. To construct the deployable code the server will do the following to the reduced configuration $n_g$:

*(i)* include additional EVM instructions (as bytecode) that will copy the stack, memory and storage values from $rc_{n_g}$ into the Ethereum's EVM stack, memory, and storage, respectively.
*(ii)* append the bytecode of the instruction to be executed at step $n_g$ $(I_{n_g})$ with the bytecode from step (i). We denote this bytecode as $B_{n_g}$.
*(iii)* Finally, prepend the bytecode $(B_{n_g})$ with an *initialization* code $c_{n_g}$ that will enable the referee to initialize the new EVM.
The above steps can be performed using the EVM-tool [15]. Finally, the server will send the deployable bytecode $(c_{n_g} B_{n_g})$ to the referee contract for single-step execution.

Algorithm 3 in Appendix E.3 shows the pseudo-code of server's functions in cmCRR protocol.

### E.4 Referee Contract Functions

The referee smart contract performs the following functions, (i) compares the results of the two servers, (ii) verifies the commitments from the servers, and (iii) in case of result mismatch, identifies the step of the computation of the two servers' results that do not match by creating and executing a new smart contract account with the single step information (provided by one of the servers). The referee functionalities in our system are written as a smart contract using the Solidity language. The referee contract is published on the Ethereum blockchain and it's execution is performed by the miners in the Ethereum network. Functions of the referee contract are shown in Algorithms 4, 5 and 6 (for lack of space in a single page and discussion purpose we divided the functions of the referee contract into 3 algorithms. All the functions in these three algorithms reside inside a single contract.).

**Algorithm 3.** Server functions in cmCRR protocol.

**Input:** program bytecode $B_{f()}$, input $x$, commitment keys $key_f$, $key_x$

**function** MAIN($x$, $B_{f()}$, $key_x$)
  /* Step 1: Register for the computation */
  referee.functions.Register($Pk_i$).transact(); ▷ Transaction to referee contract's Register() function
  /* Step 2: Check commitments and program's validity. */
  Get $com_x$ from blockchain;
  **if** open($com_x$,$d_x$, $x$) = 1 **then** continue; **else** break;
  Check $B_{f()}$ for non-deterministic EVM instructions;
  /* Step 3: Execute program in EVM and send result to referee contract. */
  vm = EVM; ▷ Initialize VM environment to run the bytecode
  ($y_i$, $num\_steps$) = vm.execute_bytecode ($B_{f()}$, $x$); ▷ $y_i$: result of computation
  Get ($com_{y_i}$, $d_{y_i}$) = GENERATE-COMMITMENT ($y_i$, $Pk_i$);
  **for** `j = 1` to $num\_steps$ **do**
    Get $H_{rc_j}$ = GENERATE-MERKLE-TREE ($rc_j$);
  Get $root$ = GENERATE-MERKLE-TREE ($hashed\_reduced\_configuration\_array$);
  Compute $root_{S_i} = H$ ($root$ || $H(Pk_i)$); ▷ $Pk_i$: public key of server $S_i$
  referee.functions.receiveResults($com_{y_i}$, $root_{S_i}$, $num\_steps$).transact();
  Reveal ($y_i$, $d_{y_i}$) to referee; ▷ Referee asks the server to reveal its commitment
  /* Step 4: Send hashed reduced configuration during binary-search */
  Get $rc_{index}$ = GET-REDUCED-CONFIG ($index$);
  Get $H_{rc_{index}}$ = GENERATE-MERKLE-TREE ($rc_{index}$);
  Get $p_{rc_{index}}$ = get_MHproof ($index$); ▷ $p_{rc_{index}}$: Merkle Proof for $rc_{index}$
  Send ($H_{rc_{index}}$, $p_{rc_{index}}$) to referee;
  /* Step 5: Send single-step data. */
  Get ($deploy\_Code$, $rc_{n_g}$, $rc_{n_b}$) = GET-SINGLESTEP-DATA ($index$);
  Send ($deploy\_Code$, $rc_{n_g}$, $rc_{n_b}$) to referee;

**function** GET-REDUCED-CONFIG($index$) **returns** ($rc_{index}$)
  Get $v_{index} \leftarrow vm.getStorageAt(index)$;
  Get $s_{index} \leftarrow getVMState(index)$;
  Set $rc_{index} \leftarrow (s_{index}, v_{index})$
  return $rc_{index}$

**function** GET-SINGLESTEP-DATA($n_g$) **returns** ($deploy\_code$, $rc_{n_g}$, $rc_{n_b}$)
  Get $rc_{n_g}$ = GET-REDUCED-CONFIG ($n_g$);
  Get $I_{n_g}$;
  Create $dc_{rc_{n_g}} = DeployCode$ ($rc_{n_g}$, $I_{n_g}$);
  Get $rc_{n_b}$ = GET-REDUCED-CONFIG ($n_b$);
  return ($dc_{rc_{n_g}}$, $rc_{n_g}$, $rc_{n_b}$)

**function** GENERATE-COMMITMENT($message$, $Pk_i$) **returns** ($com$, $key$)
  Get $key = getrandbits(256)$;
  Set $msg = message$;
  Get $com$ = **H** ($key$ || $msg$ || $Pk_i$); ▷ $com$: commitment value
  return ($com$, $key$);

**function** GENERATE-MERKLE-TREE($data\_array$) **returns** ($Mroot$)
  Create MerkleTools Object $mt$ for constructing Merkle tree
  $mt.add\_leaf(data\_array)$; ▷ Adds a value as a leaf
  $mt.make\_tree()$; ▷ Generates the Merkle Tree
  Get $Mroot = mt.get\_merkle\_root()$; ▷ Returns the Merkle root of the tree
  return $Mroot$;

Once the computation is initialized by the client,

- The referee contract registers two servers interested in solving the computation problem, instructs them to perform the computation and receives their returned commitments to the results.
- Then it asks both servers to reveal their commitments and checks the validity of the commitments.
- Next (if commitments are valid), it checks whether the $y_i$ received from both servers match with each other or not.
- If true, the referee selects one of the $y_i$ values as the computation result.
- In case of inconsistency, it executes the binary search function to find the steps where the servers disagree. At each round of the binary search, the referee specifies the index $j$ of the computation step for which both parties should respond with their hashed reduced configuration $H_{rc_j}$ together with a proof $p_{rc_j}$. It then checks whether the hashed reduced configurations of both servers match or not. If they match it verifies whether the Merkle proof $p_{rc_j}$ is valid. If both are valid it updates $n_g$, else it selects the $y_i$ of the server whose Merkle proof is valid as computation result. Otherwise, if the encrypted reduced configurations do not match, it updates $n_b$ and computes the next computation step that servers have to reveal its reduced configuration.
- Once the binary search stage is complete (i.e., two consecutive reduced states $rc_{n_b}$, $rc_{n_g}$ are identified such that $n_b = n_g + 1$), the referee contract will simulate the instruction for step $n_g$ to compute the correct output of the reduced configuration for step $n_b$, compare it to the server's answer and declare the correct result of the computation.

*Single-Step Execution.* Algorithm 6 shows the functions used by referee contract to perform the single-step execution. Here, the *Deploy()* function uses the `CREATE` opcode to deploy the bytecode, and obtain the address of the deployed contract. At the end of successful deployment, the runtime bytecode for the configuration $rc_{n_g}$ (provided by the server) will reside alone at its new contract address.

### E.5 Steps of cmCRRProtocol

The interactions between different parties involved in the cmCRR protocol are shown in steps in Algorithm 7.

### E.6 On-Chain Transactions

As all the communications to and from the smart contract uses the Ethereum network, it introduces delay. We measure the delay in *Fides* when two servers are involved in terms of the number of on-chain transactions that will be sent and received *between the servers and the smart contract* for a given computation. In our implementation, we have four different phases of interactions between the servers and the referee contract, namely - *registration*, *receive commitments*, *receive committed results* and the *malicious cloud identification* (MCId). For the first three phases, each server sends a transaction to the referee contract for registration, for sending the commitments to the results, and for revealing the committed results, respectively. So, in each of these three phases there are *two transactions* involved. In the MCId phase, for the binary-search step up to $2log(N)$ rounds are required where at each round the referee sends the index of the next step that both servers should respond. So, in total, up to $4log(N)$ transactions are required for binary-search. Finally, for the single-step case, one of the servers provides necessary data in 1 transaction. So, at most $4log(N) + 1$ *transactions* are required. Based on this delay analysis, we can also estimate the delay for the multi-server case.

**Algorithm 4.** Functions of the *Referee* smart contract

```
1: function INITIALIZE(com_x, program_bytecode) public
2:     set task = program_bytecode;
3:     com_to_input = com_x;
4:     set index = 0;
5: function REGISTER () public
6:     require (index < 2, "Registration complete");
7:     set Cloud[index] = msg.sender;                          ▷ Storing cloud's address
8:     update index+ = 1;
9:     assert(count < 2);                  ▷ Checking whether two clouds registered or not
10:    emit registered (Cloud[0], Cloud[1]);
11:    set state = Compute;
12: function RECEIVE-RESULT-COMMITMENTS(com_y, MH_root, N) public
13:    assert (msg.sender == Cloud[0] —— msg.sender == Cloud[1]);
14:    if msg.sender == Cloud[0] then
15:        set com_Result[0] = com_y; MRoot[0] = MH_root; Tape_length[0] = N;
16:            ▷ (com_Result[], MRoot[], Tape_length[]: array to store commitment to result,
   Merkle root, length of RC array, respectively)
17:    else
18:        set Result[1] = com_y; MRoot[1] = MH_root; Tape_length[1] = N;
19:    /* if both clouds committed then ask for revealing the result */
20:    emit revealResult ("Reveal result");
21: function RECEIVE-RESULT(y, key_y) public
22:    assert (msg.sender == Cloud[0] —— msg.sender == Cloud[1]);
23:    if msg.sender == Cloud[0] then
24:        set Result[0] = y; com_key[0] = key_y;
25:          ▷ (Result[], com_key[]: array to store result, and commitment key, respectively)
26:    else
27:        set Result[1] = y; com_key[1] = key_y;
28:    /* Once both clouds revealed the results call compare function */
29:    Compare ();
30: function COMPARE () internal
31:    Open1 = verify_commitment(com_Result[0], Result[0], com_key[0]);
32:    Open2 = verify_commitment(com_Result[1], Result[1], com_key[1]);
33:    if (Result[0]==Result[1]) && (Tape_length[0]==Tape_length[1]) then
34:        resultMatch = true;                                          ▷ results match
35:        if (Open1 == true && Open2 == true) then
36:            copyDetected = false;
37:            final_output = Result[0]; resultStatus = "Correct result";
38:        if (Open1 == true && Open2 == false) then
39:            copyDetected = true;
40:            final_output = Result[0]; resultStatus = "Correct result";
41:        if (Open1 == false && Open2 == true) then
42:            copyDetected = true;
43:            final_output = Result[1]; resultStatus = "Correct result";
44:    else
45:        resultMatch = false;                                         ▷ results mismatch
46:        if (Open1 == true && Open2 == false) then
47:            IncorrectOpening = true; final_output = Result[0]; resultStatus = "Correct
   result";
48:        if (Open1 == false && Open2 == true) then
49:            IncorrectOpening = true; final_output = Result[1]; resultStatus = "Correct
   result";
50:        if (Open1 == true && Open2 == true) then
51:            IncorrectOpening = false;
52:        /* Initiate Malicious Cloud Identification Protocol in Algorithm 5 */;
```

**Algorithm 5.** Malicious cloud identification protocol.

Set $n_b = min\ (Tape\_length[0], Tape\_length[1])$;
Set $n_g = binary - search(n_b)$; ▷ index of last matched configuration
**function** BINARY-SEARCH($n_b$) **returns** ($int$)
    Set $idx_s = 0$, $idx_e = n_b$; ▷ $idx_s$=starting index where clouds agree, $idx_e$=last index where clouds disagree
    Set $mid = \frac{idx_e - idx_s}{2} + idx_s$;
    **while** $idx_e > idx_s + 1$ **do**
        PG asks for the Hashed reduced-configurations $H(rc^i_{mid})$ at index $mid$ along with a proof $\rho_{rc^i_{mid}}$ from both server ($i \in \{1, 2\}$);
        **if** $(H(rc^1_j) = H(rc^2_j))$ **then**
            Verify proofs $\rho_{rc^1_{mid}}$ and $\rho_{rc^2_{mid}}$
            **if** (both proofs are valid) **then**
                ▷ mismatch is at the second half $(mid, idx_e]$
                set $idx_s = mid$; update $mid = \frac{idx_e - idx_s}{2} + idx_s$;
            **if** both proofs are invalid **then**
                set $idx_e = mid$; update $mid = \frac{idx_e - idx_s}{2} + idx_s$;
            **if** one of the proofs is invalid **then**
                set *final_output* = Result of the honest cloud; return 0;
    return $idx_s$;
Client asks *Cloud 1* for configurations $rc_{n_g}$ and $rc_{n_b}$ to perform single step of the computation; /* $n_b = n_g + 1$ */
/* Initiate Single-step execution as shown in Algorithm 6

**Algorithm 6.** Single-step execution by the referee contract.

**function** EXECUTE-SINGLE-STEP($state\_bytecode$, $rc_{n_g}$, $rc_{n_b}$) **returns** ($bool$)
    Verify $B_{n_g}$ is correct ▷ check correctness for EVM instruction
    ($state\_output$, $status$) = VERIFY-REDUCED-STEP ($state\_bytecode$);
    **if** $status = true$ and $state\_output = v_{n_b}$ **then** ▷ $v_{n_b}$: claimed output for step $n_b$
        return $true$;
    **else**
        return $false$;

**function** VERIFY-REDUCED-STEP($state\_bytecode$) **returns** ($state\_output$, $status$)
    Get $desp\_state\_addr$ = Deploy ($state\_bytecode$); ▷ Address of disp-state contract
    Get $(TF, val) = desp\_state\_addr$.call(" "); ▷ Execute disp-state contract
    ▷ $val$: Execution result of disp-state contract
    ▷ $TF$: true/false status of execution
    return ($val$, $TF$);

**function** DEPLOY($state\_bytecode$) **returns** ($desp\_address$)
    Get $addr$ = create(0, add($state\_bytecode$, 0x20), mload($state\_bytecode$));
    return $addr$;

**Algorithm 7.** cmCRR Protocol Steps.

/* **Step 1: Computation initialization and server registration** */
*Client* → *SC*: Initialize ($com_x$, $B_{f()}$) ▷ $B_{f()}$: compiled bytecode of $f()$
$S_i$ → *SC*: Register ($Pk_i$) (for all $i \in \{1, 2\}$)
*SC* → *Client*: (registered, $S_1$, $S_2$)

/* **Step 2: Computation execution by the servers** */
*Client* → $S_i$: Compute ($B_{f()}$, $x$, $d_x$) (for all $i \in \{1, 2\}$)

/* **Step 3: Send commitments and reveal the result to referee** */
$S_i$ → *SC*: receive-Commitments ($com_{y_i}$, $N_i$, $MH_{root}(C_i||Pk_i)$) (for all $i \in \{1, 2\}$)
*SC* → $S_i$: open commitment (for all $i \in \{1, 2\}$)
$S_i$ → *SC*: receive-Results ($y_i$, $d_{y_i}$) (for all $i \in \{1, 2\}$) ▷ $y_i$, $d_{y_i}$: result of computation, opening key

/* **Step 4: Result verification by the referee contract** */
*SC*: compare $y_i$ and verify commitment to $y_i$
    **if** ($y_1 = y_2$) and both commitments are *true*: set *result* = $y_1$ (or $y_2$);
    **else** (one of the servers commitment is $false$) "*copy detected*", set $y_i$ of other server as "correct result"

/* (In case of inconsistency, i.e., $y_1 \neq y_2$) */
*SC*: set $n_g = 1$, $n_b = min(N_1, N_2)$, $j = \frac{n_b - n_g}{2}$
*SC* → $S_i$: get-Reduced-config($j$)
$S_i$ → *SC*: receive-Reduced-config($H^1_{rc_j}$, $p^1_{rc_j}$) (for all $i \in \{1, 2\}$) ▷ $H^i_{rc_j}$: Hashed reduced configuration from server $i$
*SC*: compare $H^i_{rc_j}$ and verify-MHProof ($MH_{root}(C_i||Pk_i)$, $j$, $H^i_{rc_j}$, $p^i_{rc_j}$)
    **if** *true*: update $n_g = j$, $j = \frac{n_b - n_g}{2}$; **else** "*copy detected*"
    **if** $H^1_{rc_j} \neq H^2_{rc_j}$: update $n_b = j$, $j = \frac{n_b - n_g}{2}$
**continue** until $n_b = n_g + 1$

/* (Single-step execution, when $n_b = n_g + 1$) */
*SC* → $S_1$: get-Single-step-data ($n_g$)
$S_1$ → *SC*: receive-Single-step-data ($stateBytecode_{rc_{n_g}}$, $rc_{n_g}$, $rc_{n_b}$)
*SC*: execute-Single-step ()
    check validity of $I_{n_g}$
    **if** *true*: deploy new EVM in Ethereum: *Deploy* ($stateBytecode_{rc_{n_g}}$)
    execute new EVM and get value $v'$
    compare value $v'$ with the claimedOutput $v_{rc_{n_b}}$
    return true/false
    **if** *true*: set *result* = $y_1$; **else** set *result* = $y_2$

## References


1. Applebaum, B., Ishai, Y., Kushilevitz, E.: From secrecy to soundness: efficient verification via secure computation. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 152–163. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14165-2_14
2. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and the hardness of approximation problems. J. ACM (JACM) **45**(3), 501–555 (1998)
3. Arora, S., Safra, S.: Probabilistic checking of proofs: a new characterization of NP. J. ACM (JACM) **45**(1), 70–122 (1998)
4. Avizheh, S., Nabi, M., Safavi-Naini, R., Venkateswarlu K, M.: Verifiable computation using smart contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, pp. 17–28 (2019)
5. Babai, L.: Trading group theory for randomness. In: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing, pp. 421–429. ACM (1985)
6. Belenkiy, M., Chase, M., Erway, C.C., Jannotti, J., Küpçü, A., Lysyanskaya, A.: Incentivizing outsourced computation. In: Proceedings of the 3rd International Workshop on Economics of Networked Systems, pp. 85–90 (2008)
7. Blumberg, A.J., Thaler, J., Vu, V., Walfish, M.: Verifiable computation using multiple provers. IACR Cryptol. ePrint Arch. **2014**, 846 (2014)
8. Brassard, G., Chaum, D., Crépeau, C.: Minimum disclosure proofs of knowledge. J. Comput. Syst. Sci. **37**(2), 156–189 (1988)
9. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. White Paper **3**(37), 21 (2014)
10. Canetti, R., Riva, B., Rothblum, G.N.: Practical delegation of computation using multiple servers. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 445–454 (2011)
11. Canetti, R., Riva, B., Rothblum, G.N.: Refereed delegation of computation. Inf. Comput. **226**, 16–36 (2013)
12. Chen, S., Cheon, J.H., Kim, D., Park, D.: Verifiable computing for approximate computation. Cryptology ePrint Archive, Report 2019/762 (2019). https://eprint.iacr.org/2019/762
13. Chung, K.-M., Kalai, Y., Vadhan, S.: Improved delegation of computation using fully homomorphic encryption. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 483–501. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_26
14. Cormode, G., Mitzenmacher, M., Thaler, J.: Practical verified computation with streaming interactive proofs. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 90–112 (2012)
15. Culture, C.: Evm-tools (2019). https://github.com/CoinCulture/evm-tools
16. Duan, J., Zhou, J., Li, Y.: Secure and verifiable outsourcing of nonnegative matrix factorization (NMF). In: Proceedings of the 4th ACM Workshop on Information Hiding and Multimedia Security, pp. 63–68 (2016)
17. Feige, U., Kilian, J.: Making games short. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 506–516 (1997)
18. Ganache: (2019. https://www.trufflesuite.com/ganache
19. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: outsourcing computation to untrusted workers. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 465–482. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_25

20. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_37
21. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, pp. 113–122 (2008)
22. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM J. Comput. **18**(1), 186–208 (1989)
23. Halevi, S.: Advanced cryptography: promise and challenges. In: ACM Conference on Computer and Communications Security, p. 647 (2018)
24. Harz, D., Boman, M.: The scalability of trustless trust. In: Zohar, A., et al. (eds.) FC 2018. LNCS, vol. 10958, pp. 279–293. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-58820-8_19
25. Hu, C., Alhothaily, A., Alrawais, A., Cheng, X., Sturtivant, C., Liu, H.: A secure and verifiable outsourcing scheme for matrix inverse computation. In: IEEE INFOCOM 2017-IEEE Conference on Computer Communications, pp. 1–9. IEEE (2017)
26. solidity IDE, R.: (2019). https://remix.ethereum.org
27. Ishai, Y., Kushilevitz, E., Ostrovsky, R.: Efficient arguments without short PCPs. In: Twenty-Second Annual IEEE Conference on Computational Complexity (CCC 2007), pp. 278–291. IEEE (2007)
28. Jiang, X., Kim, M., Lauter, K., Song, Y.: Secure outsourced matrix computation and application to neural networks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1209–1222 (2018)
29. Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S.M., Felten, E.W.: Arbitrum: Scalable, private smart contracts. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 1353–1370 (2018)
30. Khouzani, M., Pham, V., Cid, C.: Incentive engineering for outsourced computation in the face of collusion. In: Proceedings of WEIS (2014)
31. Kilian, J.: A note on efficient zero-knowledge proofs and arguments. In: Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing, pp. 723–732 (1992)
32. Kilian, J.: Improved efficient arguments. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 311–324. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-44750-4_25
33. Küpçü, A.: Incentivized outsourced computation resistant to malicious contractors. IEEE Trans. Dependable Secure Comput. **14**(6), 633–649 (2015)
34. Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. J. ACM (JACM) **39**(4), 859–868 (1992)
35. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 238–252. IEEE (2013)
36. Pham, V., Khouzani, M.H.R., Cid, C.: Optimal contracts for outsourced computation. In: Poovendran, R., Saad, W. (eds.) GameSec 2014. LNCS, vol. 8840, pp. 79–98. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12601-2_5
37. Poon, J., Buterin, V.: Plasma: scalable autonomous smart contracts. White paper, pp. 1–47 (2017)
38. Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M.: Resolving the conflict between generality and plausibility in verified computation. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp. 71–84 (2013)

39. Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M.: Taking proof-based verified computation a few steps closer to practicality. In: Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), pp. 253–268 (2012)
40. Setty, S.T., McPherson, R., Blumberg, A.J., Walfish, M.: Making argument systems for outsourced computation practical (sometimes). In: NDSS, vol. 1, p. 17 (2012)
41. Teutsch, J., Reitwießner, C.: A scalable verification solution for blockchains. arXiv preprint arXiv:1908.04756 (2019)
42. Vu, V., Setty, S., Blumberg, A.J., Walfish, M.: A hybrid architecture for interactive verifiable computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 223–237. IEEE (2013)
43. Wahby, R.S., et al.: Full accounting for verifiable outsourcing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2071–2086 (2017)
44. Walfish, M., Blumberg, A.J.: Verifying computations without reexecuting them. Commun. ACM **58**(2), 74–84 (2015)
45. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Paper **151**(2014), 1–32 (2014)
46. Zhang, Y., Blanton, M.: Efficient secure and verifiable outsourcing of matrix multiplications. In: Chow, S.S.M., Camenisch, J., Hui, L.C.K., Yiu, S.M. (eds.) ISC 2014. LNCS, vol. 8783, pp. 158–178. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13257-0_10