Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Fair payments for verifiable cloud services using smart contracts

Mallikarjun Reddy Dorsala[a,b,*], V.N. Sastry[a], Sudhakar Chapram[b]

[a] *Center for Mobile Banking, Institute for Development and Research in Banking Technology(IDRBT), Hyderabad, India*
[b] *Department of Computer Science and Engineering, National Institute of Technology, Warangal, India*

ARTICLE INFO

Article history:
Received 12 July 2019
Revised 9 November 2019
Accepted 31 December 2019
Available online 2 January 2020

Keywords:
Blockchain
Outsourcing
Smart contracts
Verifiable computation
Incentivized outsourced computation
Cryptocurrency

ABSTRACT

With the advent of cloud computing, outsourcing a computation has become a common practice. A client, to have greater confidence in computations performed by the cloud, should be able to verify the correctness of the results returned. Two approaches are followed to verify the correctness of the results: (1) Proof-based verifiable computing and (2) Replication-based Verifiable Computing. The first approach uses cryptography techniques, whereas the second approach follows game-theoretic methods. Although both approaches are almost perfect solutions to verify the results, they do not discuss fairness in verifiable computing.

In this paper, we consider the problem of fairness in verifiable cloud computing, which means that the cloud gets the payment from a client for an outsourced computation if and only if the client receives the correct output of the computation. Most of the existing solutions rely on a trusted third-party to realise fairness. Motivated by the recent advances in blockchain technology and smart contracts, we propose protocols for different verifiable computing techniques using smart contracts. We first present a smart-contract based solution for fair proof-based verifiable computation and show that the overhead of using smart-contract is almost negligible when both client and cloud are honest. Then we show two different solutions depending upon the number of clouds hired to achieve fairness in replication-based verifiable computing. We have designed our protocols in the Blockchain model of cryptography and proved their security under universally composable theory. We also show the financial and transactional cost analysis of proposed contracts by implementing them in solidity and running them on Ethereum Blockchain.

1. Introduction

Outsourcing computation is one of the most widely studied problems in computer science. Projects like SETI@Home (Anderson et al., 2002) amasses vast distributed computing power by outsourcing the computations to millions of smaller clients. However, if a monetized reward is attached to the output of outsourced computations, then the smaller clients may not act honestly and may return bogus results without actually performing the computation thus by conserving their computational resources. On the other end, a resource-constrained client may outsource a resource-intensive task to large computational systems like a cloud. In today's world, a client and cloud/cloud service provider(CSP) should get into a pay-per-use agreement before actually using cloud services(the client has to subscribe to the cloud service provider by paying an amount to him). These pay-per/pay-and-use agreements guarantee pay to the cloud and output to the client. Any violations or disputes from the agreement are negotiated through a trusted entity like the judiciary. Resolving through judiciary is a time-consuming process. In existing cloud computing scenarios, the cloud is trusted to perform the computation honestly and return the correct output of the computation. However, if a monetized reward is attached to the correct computation of the task, the cloud as being rational will always try to minimize its cost and may not perform the correct computation. In this scenario, the client has to verify the correctness of the output computed by the cloud. The client can verify the output by following one of the two verifiable computation techniques, (1) Proof-based verifiable computation and (2) Replication-based verifiable computation. In the first technique, the cloud has to send the proof of correctness of computation along with the output. The client will verify the proof and accept the output if and only if the proof is correct. In the second technique, the client will outsource the same task to multiple clouds and obtain the correct output by comparing all the returned outputs.

On the other hand, the client may behave mischievously and may not pay the cloud after receiving the output or may abruptly

[*] Corresponding author at: Center for Mobile Banking, Institute for Development and Research in Banking Technology(IDRBT), Hyderabad, India.
E-mail address: arjun753016@gmail.com (M.R. Dorsala).

https://doi.org/10.1016/j.cose.2019.101712
0167-4048/© 2020 Elsevier Ltd. All rights reserved.

exit the protocol to minimize cost. So, there is a need for a fair incentivization model to which both client and cloud will adhere. As both client and cloud do not trust each other and are always rational incentivization has to happen through a trusted party. However, hiring a trusted party is costly and finding an ideal trusted party which will behave honestly at all times is difficult. The recent progress in Blockchain technology allows a public Blockchain network to emulate the properties of a Trusted party. The public Blockchain network is trusted for the immutability of data it possesses, the correctness of the code(smart contract) execution in its environment and its availability. In this paper, we present an incentivization model on both the verifiable computation techniques by emulating the trusted party through a smart contract running on a public Blockchain network.

1.1. Related work

There are many models proposed for verifying the correctness of the output of a function. Some of the works(collectively known as proof-based verifiable computing techniques) are interactive proof systems (Goldwasser et al., 1989; Lund et al., 1990), probabilistic proof systems(PCP's) (Goldreich, 1999; Sudan, 2009), argument systems (Brassard et al., 1988), PCP with cryptographic commitments(Kilian, 1992) and non-interactive protocols for general-purpose computations (Chung et al., 2010; Gennaro et al., 2010). Unfortunately, most of the these works are of theoretical interest and suffer from practicality, but of late using above works some practical models like Pepper(Setty et al., 2012b), Ginger(Setty et al., 2012a), Zaatar(Setty et al., 2013), Pinocchio(Parno et al., 2013), TinyRam(Ben-Sasson et al., 2014), Geppetto(Costello et al., 2015), and Xjsnark(Kosba et al., 2018) are proposed. Even considering modern-day computational capabilities, these models have high prover's overhead, and the verifier's costs per instance are also too high. Most of these models fail to run experiments on a realistic scale. Even though if all these models are highly practical, they can only verify the correctness of the output. In summary, these models are helpful to detect cheating cloud but fail to provide fair incentivized outsourced verifiable computation.

Another line of work focusing on fair verifiable computations requires a trusted party which acts as a mediator between the client and the cloud. The client and the cloud deposits some amount of payment with the trusted party; the cloud performs the computation outsourced by the client and submits the output of the computation to the trusted party. The trusted party verifies the correctness of the output, if it satisfies with the correctness, the cloud will get the pay otherwise pay is refunded to the client, and the cloud's deposit is forfeited. An alternative model is that the client itself will verify the correctness and prove to the trusted party that the cloud is cheating so that the cloud will lose its deposit. These type of models are built on top of the previously mentioned works, but again, as discussed earlier, they will suffer from high prover's overhead and high costs per instances for the verifier. Another approach is to have multiple clouds and use incentive-compatible Byzantine fault tolerance (BFT) (Li et al., 2006) to achieve correctness and also to detect rational clouds. Nevertheless, the BFT model assumes that at least two-thirds of the clouds are honest and performs the computation honestly. This assumption does not hold in real-world scenarios where every cloud is rational and want to maximize its utility.

Another line of work uses the concept of ringers, first proposed in Golle and Mironov (2001). Carbunar and Tripunitara (2010, 2012) uses bank as a third party for payment transactions between a client and a cloud. They use the concept of secret sharing and ringers as proof of correctness of the output. However, their method is complicated, requires many rounds of interactions, does not impose fines on cheating cloud and is inefficient for real and practical applications. The authors in Chen et al. (2012) also uses the concept of ringers, but the main difference is that they do not use secret sharing methods for payment token generation and verification. However, this scheme also does not impose fines on a cheating cloud.

Replication-based verifiable computation techniques (Belenkiy et al., 2008; Canetti et al., 2011; Küpçü, 2017) are proposed as a solution to the shortcomings of proof-based verifiable computation techniques. Belenkiy et al. (2008) ensure the cloud runs the correct computation by requesting the output of all the intermediate steps of the computation. However, this information is large, so only the hash of output of the all intermediate steps is requested from the cloud. Molnar (2000) proposed to use the hash of intermediate steps of a computation as a proof of correctness of the computation. This concept is known as the inner state hash(ISH). Advantage of using ISH is of two-fold, first, suppose the client chooses m clouds and asks them to compute the same job then their resultant hash strings can be compared easily, and second, even the slightest deviation from the correct computation can be detected. The authors also describe cloud utility for being honest or malicious and also show that penalties are an efficient way to make rational clouds work honestly and necessary for resilience against malicious clouds. The main disadvantage of their scheme is that the clouds should trust the client for fair payment after receiving the output of the computation. This scheme also lacks experimental evaluation and does not discuss overheads for the client and clouds. Canetti et al. (2011) designed a generic protocol in which a client outsources a computation to a minimum of two clouds to achieve verifiability. The client searches for the inconsistencies between the intermediate states of the clouds' computations. If any inconsistencies are detected, then a cheating cloud is found after a round of delegation with both the clouds. However, their protocol assumes at least one cloud is honest and incurs an overhead of 10–20 times higher than the plain execution. The scheme in Belenkiy et al. (2008) was extended by Küpçü (2017) by adding bounty given to an honest cloud which helps in the detection of cheating clouds. Their final construction shows both experimental and theoretical overheads on the cloud for using ISH as a proof of correctness of the output. They use a trusted bank for facilitating payments of rewards and fines. Our work mostly follows their approach, but we replace the trusted bank with a smart contract running on a public Blockchain network.

Recent advances in Blockchain networks motivated researches to work towards achieving fairness in incentivizing outsourced computation. Kumaresan and Bentov (2014) presented a Bitcoin (Nakamoto, 2008) based fair verifiable computing. However, his method involves using proof-based verifiable computing, which is not feasible with current Bitcoin architecture. Later Luu et al. (2015) designed a $\in$-consensus protocol and argued that the stability of a blockchain indeed depends on the size of the verifying work. They show the amount of work an honest miner can do is at most $\in W_{blk}$ per block. Nevertheless, the verifying function in proof-based verifiable computing or the ringers evaluation will take more than $\in W_{blk}$ work. Their results are the motivation for us to use multiple clouds and inner state hashes to verify the correctness of the output. Ethereum (Wood, 2014), with its rich Turing-complete instruction set, offers development of smart contracts in its network. Juels et al. (2016) have shown smart contracts for criminal activities whereas Dong et al. (2017) have shown smart contracts for verifiable computing. In Dong et al. (2017), the authors create a prisoner's dilemma between two clouds and show bounties are the way to discourage clouds collusion. However, their protocol involves the client to verify the results obtained from the clouds, who in the real world may behave rationally. They also discuss only a rational cloud; there might be a malicious cloud who on purpose want the client to accept the incorrect result or do extra work. Zhang et al. (2018a); Zhang et al. (2018b) had proposed a fair payment for outsourcing services in cloud computing. Their solution is more like a Bitcoin-based solution, whereas we propose a smart-contract based solution for fair payments. More recently, the authors in Dorsala et al. (2018) designed ideal functionalities in UC framework (Canetti, 2001) for fair payments for verifiable computations. The authors presented the realization of their ideal functionality in Bitcoin, which is not a Turing complete language and does not support the complex operations of proof-based verifiable computing. Also, the authors did not consider replication-based verifiable computation and the usage of smart contracts. However, in this paper, we propose fair protocols for both proof-based and replication-based verifiable computations using smart contracts.

1.2. Contributions

As we have seen pure cryptographic solutions are inefficient and pure game-theoretic solutions have unfair trust assumptions, we propose a new smart contract-based approach where the client gets the correct results, and the overall overhead is minimal even though there are a large number of lazy workers.

- We design a new Incentivized model for proof-based verifiable computation and show that the cost of running a smart contract is negligible when both Delegator and Worker are honest.
- We design a new fair incentivized model for replication-based verifiable computation for a Two-worker case and a Multi-worker case. We obtain honest computation from the worker by imposing monetized penalties and our protocols guarantee to pay for the honest workers.
- We show smart contracts($\mathcal{SC}$) are an efficient way to send the reward to honest workers and also to penalize malicious workers. By using $\mathcal{SC}$, we are emulating a Bank or any other trusted party for payments between client and cloud.
- We used the Blockchain model of cryptography to design our protocols and show our protocols are secure under $\mathcal{G}(\cdot)$-hybrid model of the universally composable theory.
- We have implemented our smart contracts in Solidity using Truffle framework and the financial and transactional cost of the proposed contracts are presented.

2. Preliminaries and notations

2.1. Verifiable computation

A resource constraint delegator ($\mathcal{D}$) outsources a computation to a worker($\mathcal{W}$), who get paid in return for delivering the correct output of the computation. The output returned by the worker is verified by the delegator or by a third-party. The work performed to verify the output must be lesser than the work required to compute the output. The delegator accepts the output of the computation if and only if its correctness is verified.

2.2. Proof-based verifiable computation

In proof-based verifiable computation schemes, the correctness of the computation is given as a valid NIZK(Non-Interactive Zero Knowledge) proof. We present the state of the art public verifiable computation scheme (taken from Parno et al., 2013)

Definition 1. A public verifiable computation scheme consists of a set of three polynomial-time algorithms.

- $Keygen(F, 1^{\lambda}) \rightarrow (ek_F, vk_F)$ A randomized key generation algorithm takes the function F to be outsourced and a security parameter λ; It outputs a public evalution key ek_F and a public verification key vk_F.
- $Compute(ek_F, x) \rightarrow (y, \pi_y)$. The deterministic worker algorithm uses ek_F and input x. It outputs $F(x) \rightarrow y$ and a proof π_y of y's correctness.
- $Verify_{vk_F}(x, (y, \pi_y)) \rightarrow \{0, 1\}$ Given the verification key vk_F, x, y and π_y the deterministic verification algorithm outputs 1 if $F(x) = y$ and 0 otherwise.

2.3. Replication-based verifiable computation

Instead of outsourcing the computation to a single worker and obtaining the proof of correctness from him, in replication-based verifiable computation, the same computation is outsourced to multiple workers, and correctness of the output is evaluated by comparing results obtained from multiple workers.

Definition 2. A replication based verifiable computation involving a delegator and multiple workers is as follows.

- $REP.Outsource(F)$: The delegator outsources F on input x.
- $REP.Intent$: The workers shows intent for the computation of F. Let $\{\mathcal{W}_1, \dots, \mathcal{W}_n\}$ be the set of workers who has shown intent to compute $F(x)$.
- $REP.Compute(F, x) \rightarrow y_i$. Each $\mathcal{W}_i \in \{\mathcal{W}_1, \dots, \mathcal{W}_n\}$ computes $F(x)$ and outputs $y_i = F(x)$.
- $REP.Verify(y_1, \dots, y_n) \rightarrow \{0, 1\}$. Every worker's output is compared against another worker, output 1 if and only if all the outputs are equal, else output 0.

2.4. Fair verifiable computation/fair incentivized outsourced computation

Definition 3. A fair verifiable computation between two parties $\mathcal{D}$ and $\mathcal{W}$ must provide the following guarantee:

- **Fast verification**: The work performed to verify the correctness of output of F is less than the work performed to compute F.
- **Pay to learn output**: $\mathcal{W}$ obtains pay from $\mathcal{D}$ if and only if $\mathcal{D}$ received the correct output of the computation from $\mathcal{W}$.

2.5. Blockchain

The Blockchain ($\mathcal{BC}$) was first introduced by Satoshi Nakamoto in Bitcoin (Nakamoto, 2008). $\mathcal{BC}$ is maintained by a set of decentralized peers(miners) running a secure consensus protocol to agree on a common global state. Every peer maintains a ledger (blockchain) which stores complete transaction history of the network. A Block b in a blockchain is of the form $b = \langle s, t, ctr \rangle$ where $s \in \{0,1\}^k, t \in \{0,1\}^*, ctr \in \mathbb{N}$ satisfying

$(H(ctr, G(s, t)) < T) \wedge (ctr \leq q)$ where $H(\cdot)$ and $G(\cdot)$ are cryptographic hash functions which outputs in length of k bits, $T \in \mathbb{N}$ is known as block difficulty level set by the consensus algorithm and $q \in \mathbb{N}$.[1] A blockchain is a sequence of blocks. Let $b' = \langle s', t', ctr' \rangle$, be the right most block in the chain. $\mathcal{BC}$ is extended to a longer chain by adding a new block $b = \langle s, t, ctr \rangle$ to b' such that it satisfies $s = H(ctr', G(s', t'))$. The security of the blockchain is maintained by the consensus protocol. The consensus protocol makes the miners compete to generate a new block periodically. The miners are rewarded for mining new blocks in the form of cryptocurrency native to that $\mathcal{BC}$. While generating a new block, the miners verify all the transactions going to be added in a block, and also miners check the validity of a new block generated by other miners before adding to their local blockchain. These two steps ensure fairness and correctness of the execution of transactions. These verification steps make the $\mathcal{BC}$ trusted for correctness and availability. We use

[1] implementation dependent.

these properties to achieve fair verifiable computation without a trusted third party between a delegator and a worker.

2.6. Smart contract

A smart contract ($\mathcal{SC}$) is a piece of code stored on a Blockchain and executed by the mining nodes of the Blockchain. Smart contract can hold many contractual clauses between two mutually distrusted parties. $\mathcal{SC}$ is executed by miners and similar to transactions, its execution correctness is also guaranteed by miners running the consensus protocol. Assuming the underlying consensus algorithm of a Blockchain is secure, the $\mathcal{SC}$ can be though of a program executed by a trusted global machine that will faithfully execute every instruction (Dong et al., 2017).

Ethereum (Wood, 2014) is a major Blockchain network supporting smart contracts. A $\mathcal{SC}$ in Ethereum is a piece of code having its own contract address, balance and state. The state of the contract is changed by the execution of its code. A smart contract is written as a set of functions. The execution of a function in $\mathcal{SC}$ is initiated by sending a transaction to its contract address. As the scripting language in Ethereum is Turing-complete language, the smart contracts can be written for a wide variety of applications. But to discourage developers from writing scripts which takes long verification time, Ethereum introduces the concept of gas. The scripts are compiled into Ethereum opcodes and each opcode costs predefined gas.[2] This amount of gas consumed by a transaction is converted into the Ether and charged from the transaction initiator's Ethereum account and paid to the miner in the form of transaction fee.

2.7. Blockchain model of cryptography

The Blockchain model of cryptography, which is first introduced in Kosba et al. (2016) and later adopted in Juels et al. (2016), is a formal framework for specifying and reasoning about the security of the protocols. We also adopt the same model to describe our protocols.

- **Timer.** All parties are aware of time which progress in rounds. At the beginning of each round, the contract's timer function is executed. The $\mathcal{SC}$ can also query $\mathcal{BC}$ for the current time denoted by variable τ.
- **Pseudonymity.** A party can obtain any number of pseudonyms to communicate with a smart contract. Contract wrapper($\mathcal{G}$) in Juels et al. (2016) generates pseudonyms on the request of any party.
- **Availability.** We assume that the blockchain is always available to be queried by any party.
- **Correctness**. We also assume that all the computations computed on a $\mathcal{BC}$ are correct.
- **Currency**. All the monetary variables are prefixed with \$ sign. $ledger[\mathcal{P}]$ denotes the party $\mathcal{P}$'s balance in native cryptocurrency of a Blockchain.
- **Variable scope and functions**. A $\mathcal{SC}$ is written as a set of functions, and each function is invoked with an corresponding message type. We assume all the variable in a $\mathcal{SC}$ are globally scoped.

2.7.1. Wrappers and programs

Wrappers contain a set of common features that are applicable for all the ideal and contract functionalities. We use the same wrappers described in Juels et al. (2016) and Kosba et al. (2016). We define our protocols in $\mathcal{G}(Contract)$-hybrid model, where $\mathcal{G}(\cdot)$ is a contract wrapper which models many concepts of the decentralized Blockchains like Bitcoin and Ethereum. The Contract program is the user-defined portion of the contract, i.e., a Contract program contains the business logic, whereas the $\mathcal{G}(\cdot)$ contains the operational semantics. Both are combined to model a real-world smart contract executing on top of a decentralized Blockchain system. Similarly, we have ideal functionality wrappers $\mathcal{F}(\cdot)$ to be used in combination with Ideal programs. All the wrappers are discussed in Juels et al. (2016).

Table 1
Notations.

Notations	Meaning
$\mathcal{D}$	Delegator/Client
$\mathcal{W}$	Worker/Server/Cloud/Cloud Service Provider
$\mathcal{BC}$	Blockchain
$\mathcal{TP}$	Trusted party
$\mathcal{A}, \mathcal{S}$	Real world adversary and Ideal world adversary
blockchain	Distributed ledger
$\mathcal{SC}$	Smart Contract
F	Outsourced function/algorithm/task
x	Function/algorithm/task input
y	Function/algorithm/task output computed by a worker
ish	Inner state hash computed by worker
r	Reward for correct computation of $F(x)$
τ, τ_i, τ_c, τ_a, τ_r, τ_{end}	Timing parameters, τ is current time
\$c, \$d, \$r, \$b	Monetized parameters representing deposit of delegator,deposit of worker,reward, and bounty respectively(in units of native cryptocurrency of the blockchain network in which the smart contract is running).

2.8. Notations

We present the notations used in this paper in Table 1.

3. Fair proof-based incentivized outsourced computation(PBIOC) using smart contract

In this section, we discuss fair incentivization of proof-based verifiable computation. As discussed earlier, a public verifiable computation scheme consists of two parties, a Delegator and a Worker. The delegator runs *Keygen* algorithm, and the worker runs *Compute* algorithm. The *Verify* algorithm is executed by a delegator or by a trusted third party. There are three possible approaches to incentivize a public verifiable computation.

Case 1: A contract is signed between $\mathcal{D}$ and $\mathcal{W}$, such that $\mathcal{D}$ runs *Verify* algorithm and pays $\mathcal{W}$ for using its services, if and only if *Verify* algorithm returns 1. An honest $\mathcal{W}$ receives pay only if $\mathcal{D}$ is honest. In this case, $\mathcal{W}$ has to put trust on $\mathcal{D}$ for honest payment.

Case 2: $\mathcal{D}$ subscribes to $\mathcal{W}$'s service by transferring some pay to it and asking it to run *Compute* algorithm. Here a legal contract may be made between these two parties containing all necessary clauses. $\mathcal{W}$ may or may not adhere to the legal contract. If $\mathcal{W}$ does not adhere to a legal contract and sends an incorrect output to $\mathcal{D}$, the only way for $\mathcal{D}$ to get back his pay is going through the cumbersome legal process. In this case, $\mathcal{D}$ has to put trust on $\mathcal{W}$ for honest computation.

Case 3: Let $\mathcal{D}$ and $\mathcal{W}$, recruit a third party $\mathcal{TP}$. $\mathcal{D}$ sends pay to $\mathcal{TP}$, and asks the $\mathcal{W}$ to run *Compute* algorithm. $\mathcal{W}$ sends the output to $\mathcal{TP}$. Now, $\mathcal{TP}$ runs *Verify* algorithm, if it returns 1, $\mathcal{TP}$ sends pay to $\mathcal{W}$, otherwise, it will send pay back to $\mathcal{D}$. Here $\mathcal{D}$ trust $\mathcal{TP}$ for honest verification and $\mathcal{W}$ trust $\mathcal{TP}$ for honest payment. A special scenario where the

[2] it is a form of transaction fee paid in ether, a native cryptocurrency of Ethereum.

dis-honest $\mathcal{W}$ can also be penalized by asking $\mathcal{W}$ to deposit some pay with $\mathcal{TP}$, before claiming the pay for computation.

In case 3, both $\mathcal{D}$ and $\mathcal{W}$ use a third party and put their trust in it. However, using the services of a $\mathcal{TP}$ comes with a cost and $\mathcal{TP}$ may not guarantee to behave honestly every time. As the decentralized Blockchains like Bitcoin and Ethereum are trusted for correctness and availability, they can emulate the trusted third party functionality. The Blockchains also offer programmability, so the users can create and run small programs known as smart contracts. Now, we show our Proof-based verifiable computation using smart contracts.

3.1. PBIOC contract

The PBIOC is an outsourcing contract signed between $\mathcal{D}$ and $\mathcal{W}$. The high-level idea is if $\mathcal{D}$ and $\mathcal{W}$ behaves honestly, then $\mathcal{D}$ will get the output of $F(x)$ and $\mathcal{W}$ will get the pay for computing $F(x)$. Otherwise, a verifying contract PBIOCV is invoked, and payment is made according to the result returned by PBIOCV contract. As we know a smart contract can hold cryptocurrency with it, $\mathcal{D}$ will deposit a reward \$r, which will be paid to $\mathcal{W}$ for correct and timely computation of $F(x)$. $\mathcal{D}$ and $\mathcal{W}$ are also asked to pay a safety deposit \$$c$ and \$$d$ respectively to ensure honest behavior. If they behave honestly, their deposits are refunded to them. Otherwise, their deposit is forfeited. The clauses in the PBIOC contract are as follows:

(1) $\mathcal{D}$ prepares two contracts *PBIOC* and *PBIOCV* where the *Verify* algorithm from Parno et al. (2013) is modeled as *PBIOCV* contract and is executed only in case of disputes.
(2) The *PBIOC* contract is signed between $\mathcal{D}$ and $\mathcal{W}$.
(3) $\mathcal{D}$ will choose the function $F(\cdot)$, and input x. $\mathcal{W}$ agrees to compute $F(x)$.
(4) $\mathcal{D}$ agrees to pay a reward \$$r$ to $\mathcal{W}$ for the correct computation of $F(x)$. $\mathcal{D}$ deposits the reward \$$r$ and a safety deposit of \$$c$ with the *PBIOC* contract. $\mathcal{D}$ also sends (ek_F, vk_F, x) to $\mathcal{W}$ in off-chain mode.
(5) Both agree on timing parameters $\tau < \tau_i < \tau_c < \tau_a < \tau_{end}$ where τ is the current time.
(6) $\mathcal{W}$ must pay a deposit \$$d$ before τ_i. If $\mathcal{W}$ fails to deposit \$$d$ before τ_i, then the contract is terminated and $\mathcal{D}$'s deposit \$$r$ + \$$c$ is refunded.
(7) $\mathcal{W}$ computes $F(x)$ and sends commitment of the output to the smart contract before τ_c. $\mathcal{W}$ sends (y, π_y) to $\mathcal{D}$ in off-chain mode. If $\mathcal{W}$ fails to deliver the commitment of the output before τ_c then, the deposit \$$d$ made by him and the deposit \$$r$ + \$$c$ made by $\mathcal{D}$ is sent to $\mathcal{D}$, and the contract is terminated.
(8) If $\mathcal{D}$ agrees to the output sent by $\mathcal{W}$ before τ_a then \$$d$+\$$r$ is sent to $\mathcal{W}$ and \$$c$ is sent to $\mathcal{D}$, and the contract is terminated.
(9) If $\tau > \tau_a$ and the contract is not terminated then the PBIOCV contract is invoked.
(10) If *PBIOCV* returns $\mathcal{W}$ as honest before $\tau < \tau_{end}$, then \$$r$+\$$c$+\$$d$ is sent to $\mathcal{W}$. If *PBIOCV* returns $\mathcal{W}$ as lazy before $\tau < \tau_{end}$, then \$$r$+\$$c$+\$$d$ is sent to $\mathcal{D}$ and the contract is terminated.
(11) If $\tau > \tau_{end}$ and the contract is not terminated then \$$r$+\$$c$+\$$d$ is sent to $\mathcal{W}$ and contract is terminated.

3.2. Analysis of PBIOC

In the PBIOC contract, the parties agree on different timing parameters $\tau_i < \tau_c < \tau_a < \tau_{end}$. These timing parameters are required to enforce timely computation and also to avoid locking of the funds if one of the party refuses to move forward. The contract can always query the underlying blockchain for current time.[3] In the contract, clause 6 and clause 7 ensures the delegator to gets his funds if no worker has shown intent or has not sent the result of the computation after showing intent. If the delegator is honest and the worker has sent the correct result of the computation, then clause 8 is executed, and the contract is terminated. Clause 9 ensures the honest worker to get his payment if the delegator is not honest or does not agree to the result sent by him, and Clause 10 ensures the delegator to get his funds back if the worker has not computed the results correctly and also ensures the worker to get his pay if delegator does not agree on correct result. If in any case, the PBIOCV contract fails to return the result before τ_{end}, then clause 11 ensures the worker to get his deposit and pay.

The formal contract of PBIOC is shown in Fig. 1, which is simple and guarantees to pay for the honest worker and also guarantee the correct output for the delegator. The user-side programs required to interact with PBIOC are presented in Fig. 2, and an abstract representation of the PBIOCV contract is presented in Fig. 3. If both $\mathcal{D}$ and $\mathcal{W}$ are honest then the execution cost of PBIOC contract is small and also have the privacy of their data. However, if any one party is dishonest, then PBIOCV contract is invoked. As we have already discussed the *Verify* algorithm from Parno et al. (2013) is modeled as PBIOCV contract, running it is costly, and the privacy of the results is no longer exists. The inherent problem with the proof-based systems is the worker's overhead and the verifier's[4] cost per instance is high. The verification time for the state-of-the-art verifiable computation scheme (Parno et al., 2013) is 9ms and takes 288 bytes of storage. In our protocol, even if we do not consider the delegator overhead in converting the function into a circuit, the verification of the proof(i.e., execution of PBIOCV contract) by the blockchain network will consume a huge amount of time. The verifiers in public blockchain systems can choose the transactions which will go into a block and then into a blockchain. Since the average block generation times are very low, it is not practical for the verifiers to accept transactions which would take a long time. Therefore, a verifier will avoid the verification of the transactions which consumes huge computational resources (Luu et al., 2015). A generic PBIOCV contract can be very complex, costly and even may not be feasible to deploy in current Blockchain networks. In Section 5.1, we show the implementations of some basic PBIOCV contracts designed for specific problems.

4. Replication-based verifiable computation

Now, we discuss achieving fairness and correctness in verifiable computation by outsourcing the same task to multiple workers. Let $\mathcal{D}$ outsources a computation to multiple workers $\mathcal{W}_1,\ldots,\mathcal{W}_n$. The workers will compute the outsourced computation and return the output. There is a chance that workers may use different algorithms other than the one prescribed by the delegator and yet deliver the correct output with negligible probability. That is there might be an algorithm which gives higher utility than the specified algorithm and yet deliver the correct output. The delegator always wants the workers to compute his prescribed algorithm. For example, the delegator does not outsource "search for an element in a given input set". He will outsource a "particular searching algorithm along with the input". Another example is when the output range of the outsourced algorithm is binary. In this case, the worker can guess the output with a 50% probability without running the algorithm. To prevent workers from using different algo-

[3] Most smart contracts use block number/block timestamp as a timer.
[4] Miners in Blockchain.

```
Contract-PBIOC
Init:     Set $state := Init$, $\$reward := 0$, $\$deposit := \{\}$
Create:   Upon receiving ("create", $E$, $V$, $X$, $\tau_i$, $\tau_c$, $\tau_a$, $\tau_{end}$, $\$r$, $\$c$ ) from $\mathcal{D}$
            assert $state = Init$
            assert $\tau < \tau_i < \tau_c < \tau_a < \tau_{end}$
            assert $ledger[\mathcal{D}] \geq \$r + \$c$
            set $ledger[\mathcal{D}] := ledger[\mathcal{D}] - (\$r + \$c)$
            set $\$reward := \$r$
            set $\$deposit := \$deposit \cup (\$c, \mathcal{D})$
            set $state := Created$
Intent:   Upon receiving ("intent", $\$d$) from $\mathcal{W}$
            assert $\tau < \tau_i$ and $state = Created$
            assert $ledger[\mathcal{W}] \geq \$d$
            set $ledger[\mathcal{W}] := ledger[\mathcal{W}] - \$d$
            set $\$deposit := \$deposit \cup (\$d, \mathcal{W})$
            set $state := Intent$
Commit:   Upon receiving ("commit", $Y$, $P$) from $\mathcal{W}$
            assert $state = Intent$ and $\tau < \tau_c$
            assert $\mathcal{W}$ has deposited $\$d$ previously
            set $state := Committed$
Agree:    Upon receiving ("agree") from $\mathcal{D}$
            assert $state = Committed$ and $\tau < \tau_a$
            set $ledger[\mathcal{W}] := ledger[\mathcal{W}] + \$reward + (\$d, \mathcal{W})$
            set $state := Terminated$
Verify:   Upon receiving ("verify", $ek_F$, $vk_F$, $x$, $y$ , $\pi_y$) from $\mathcal{W}$
            assert $\tau_a < \tau < \tau_{end}$
            assert $state! = Terminated$
            assert $E = H(ek_f)$ and $V = H(vk_f)$ and $X = H(x)$ and $Y = H(y)$ and $P = H(\pi_y)$
            send ($ek_F$, $vk_F$, $x$, $y$, $\pi_y$) to $\mathcal{G}(contract - PBIOCV)$
            set $state := Dispute$
Return:   Upon receiving ("verify return", $Honest$) from $\mathcal{G}(contract - PBIOCV)$
            assert $\tau < \tau_{end}$
            assert $state = Dispute$
            if $Honest = true$
              set $ledger[\mathcal{W}] := ledger[\mathcal{W}] + \$reward + (\$c, \mathcal{D}) + (\$d, \mathcal{W})$
            else
              set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + \$reward + (\$c, \mathcal{D}) + (\$d, \mathcal{W})$
            set $state := Terminated$
Timer:    if $\tau > \tau_{end}$ and $state! = Terminated$
            if $state = Created$
             set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + \$reward + (\$c, \mathcal{D})$
            if $state = Intent||Committed$
             set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + \$reward + (\$c, \mathcal{D}) + (\$d, \mathcal{W})$
            if $state = Dispute$
             set $ledger[\mathcal{W}] := ledger[\mathcal{W}] + \$reward + (\$c, \mathcal{D}) + (\$d, \mathcal{W})$
          set $state := Terminated$
```

Fig. 1. Contract-PBIOC.

rithms, Belenkiy et al. (2008) introduced the concept of the inner state of an algorithm.

Definition 4. Assuming the algorithm is composed of a finite number of atomic operations and each atomic operation takes some state information as input and produces another state information as output. The inner state of an algorithm is defined as the concatenation of all the input and output states of the atomic operations of an algorithm, along with the definition of the algorithm in terms of atomic operations.

Inner State Hash(ISH): An l-bit hash function takes an inner state of an algorithm as input and maps it into an l-bit random string. Even if the algorithm has many number of steps/its output is large, the hash value is always short having a constant length. The probability of producing the same hash value without using

```
Proto-PBIOC
As Delegator $\mathcal{D}$:
  Create:   Up on receiving ("create", $F$, $x$, $\tau_i$, $\tau_c$, $\tau_a$, $\tau_{end}$, $\mathcal{D}$) from environment $\mathcal{E}$.
            run $Keygen(F, 1^\lambda) \rightarrow (ek_F, vk_F)$.
            compute $E = H(ek_F)$, $V = H(vk_F)$, $X = H(x)$.
            Send ("create", $E$, $V$, $X$, $\tau_i$, $\tau_c$, $\tau_a$, $\tau_{end}$, $\$r$, $\$c$) to $\mathcal{G}(Contract - PBIOC)$.
            Send ($ek_F$, $vk_F$, $x$) to $\mathcal{W}$.
  Verify:   Upon receiving ("verify", $y$, $\pi_y$) from $\mathcal{W}$.
            run $Verify_{vk_F}(x, (y, \pi_y))$.
            if $Verify_{vk_F}(x, (y, \pi_y)) \rightarrow 1$ then send ("agree") to $\mathcal{G}(Contract - PBIOC)$.
As Worker $\mathcal{W}$:
  Intent:   Upon receiving ("intent", $\mathcal{W}$) from environment $\mathcal{E}$.
            send ("intent", $\$d$) to $\mathcal{G}(Contract - PBIOC)$.
  Commit::  Upon receiving ("commit") from environment $\mathcal{E}$.
            Assert an intent message was sent earlier
            run $Compute(ek_F, x) \rightarrow (y, \pi_y)$.
            compute $Y = H(y)$ and $P = H(\pi_y)$, where $H$ is a hash function
            send ("commit", $Y$, $P$) to $\mathcal{G}(Contract - PBIOC)$.
            send ("verify", $y$, $\pi_y$) to $\mathcal{D}$.
  Verify:   if $\mathcal{D}$ has not sent message ("agree") to $\mathcal{G}(Contract - PBIOC)$ before $\tau_a$
              send ("verify", $ek_F$, $vk_F$, $x$, $y$, $\pi_y$) to $\mathcal{G}(Contract - PBIOC)$.
```

Fig. 2. User-side programs for PBIOC .

```
Contract-PBIOCV
Verify:  Upon receiving ($ek_F$, $vk_F$, $x$, $y$, $\pi_y$) from $\mathcal{G}(Contract - PBIOC)$.
           run $Verify_{vk_f}(x, (y, \pi_y))$.
           if $Verify_{vk_f}(x, (y, \pi_y)) \rightarrow 1$ set $Honest = true$, else set $Honest = false$.
           Send ("verify return", $Honest$) to $\mathcal{G}(Contract - PBIOC)$.
```

Fig. 3. Contract-PBIOCV.

the algorithm prescribed by the delegator is negligible, i.e., $neg = O(2^{-l})$.

Definition 5. The algorithm used by the worker to complete the assigned work that outputs the correct answer with probability q is known as q-algorithm (Küpçü, 2017). If a worker runs the algorithm prescribed by the delegator then $q = 1$. Similarly, if the worker uses any algorithm other than the one prescribed by the delegator, then $q < 1$.

The cost of running a q-algorithm is $cost(q)$. As $q = 1$ for running the prescribed algorithm, we denote the cost of honest computation as $cost(1)$. The high-level description of our approach is that $\mathcal{D}$ outsources the same computation to multiple workers. The workers compute and return the outputs along with the inner state hash. $\mathcal{D}$ compares all the inner state hashes and the outputs sent by workers. If they all are equal, he accepts one of the output, otherwise re-outsources the algorithm. $\mathcal{D}$ can not be trusted by the workers for honest computation, So $\mathcal{D}$ posts a smart contract on to a distributed ledger initiating it with some pay. Similarly, the workers submit their inner state hash value and output to the smart contract. Now smart contract verifies the outputs; if all are equal, workers will receive the payment for their computation. The interesting point is that even if all the workers submit the same incorrect output, they all get the pay. We categorize the workers into three categories:

Table 2
Utilities in 2-worker case from Belenkiy et al. (2008).

Other/This worker	Diligent	Lazy
Diligent	$u(1) = \$r - \text{cost}(1)$	$u(q) = \$r * q - \$d * (1 - q)\text{-cost}(q)$
Lazy	$u(1) = \$r - cost(1)$	$u(q) = \$r - cost(q)$

- **Honest worker**: An honest worker performs the computation exactly(running the prescribed algorithm) as prescribed by the delegator.
- **Rational worker**: A Rational worker performs the computation exactly as prescribed by the delegator as long as his utility of computing the original algorithm is more than the utility of doing anything else.
- **Malicious worker**: A malicious worker has two objectives in mind (1) Making the delegator to accept the incorrect result or (2) Making the delegator to re-outsource the task.

Let, the workers who use the algorithm as prescribed by the delegator be called as diligent (all honest workers and some rational workers who behave honestly) and who uses a q-algorithm are called as lazy (all rational who behaves maliciously and all malicious). The utilities of the workers in a two-worker case are given in Table 2.

4.1. Fair replication-based verifiable computation using smart contract - two workers case (TWOIOC contract)

The TWOIOC is an outsourcing contract signed between three parties a Delegator ($\mathcal{D}$) and two Workers ($\mathcal{W}_0$, $\mathcal{W}_1$). We follow the prisoner's dilemma like the model presented by Belenkiy et al. (2008) and Küpçü (2017). We assume $F(x)$ is deterministic, and there exists a smart contract TWOIOCV which can compute $F(x)$ and return (y, ish) as output. $\mathcal{D}$ pays a bounty \$$b$ along with reward \$$r$ for an honest worker in case of disputes.

(1) $\mathcal{D}$, $\mathcal{W}_0$ and $\mathcal{W}_1$ agree on a smart contract $\mathcal{SC}$. They also agree on another smart contract TWOIOCV, which is invoked in case of disputes.
(2) $\mathcal{D}$ will choose $F(\cdot)$ and its input x. $\mathcal{D}$ agrees to pay \$$r$ to each worker for the correct computation of $F(x)$. $\mathcal{D}$ also agrees to pay \$$b$ to the honest worker in case of disputes.
(3) All three parties agree on timing parameters $\tau < \tau_i < \tau_c < \tau_{end}$.
(4) Each $\mathcal{W}_i$, $i \in \{0, 1\}$ must pay a deposit of \$$d$ before τ_i. If any $\mathcal{W}_i$ fails to deposit, $\mathcal{SC}$ is terminated and any deposits made are refunded.
(5) Each $\mathcal{W}_i$ computes $F(x)$ and delivers the output to $\mathcal{SC}$ before τ_c. If any W_i fails to deliver output by τ_c set its output as NULL. If both workers fails to send output before τ_c, then send 2*(\$$d$+\$$r$)+\$$b$ to $\mathcal{D}$ and terminate the contract.
(6) At $\tau_c < \tau < \tau_{end}$, $\mathcal{SC}$ compares the outputs delivered by workers for equality. If both the outputs are equal, then $\mathcal{SC}$ sends \$$d$+\$$r$ to each $\mathcal{W}_i$, sends \$$b$ to $\mathcal{D}$ and the contract is terminated. Else TWOIOCV contract is invoked.
(7) TWOIOCV computes $F(x)$ and sends its output to $\mathcal{SC}$ before τ_{end}.
(8) $\mathcal{SC}$ compares the results sent by *TWOIOCV* with the outputs delivered by workers.
    (a) If TWOIOCV and $\mathcal{W}_i$ outputs are same, then send \$$d$+\$$r$+\$$b$ to $\mathcal{W}_i$, send \$$d$+\$$r$ to $\mathcal{D}$ and terminate the contract.
    (b) If TWOIOCV output is not matching with any of the $\mathcal{W}_i$ outputs, then send 2*(\$$d$+\$$r$)+\$$b$ to $\mathcal{D}$ and terminate the contract.
(9) If $\tau > \tau_{end}$ and the contract is not terminated, send \$$d$+\$$r$ to each $\mathcal{W}_i$, \$$b$ to $\mathcal{D}$ and terminate the contract.

4.1.1. Analysis

As in the PBIOC contract, the TWOIOC contract can always query the underlying blockchain for the current timestamp. Unlike in Küpçü (2017) and Belenkiy et al. (2008), we do not assume the delegator acts diligently. In our model, the workers do not trust the delegator for payment, and the delegator does not trust the workers for correct computation. All the three entities will trust the underlying Blockchain for correct computation of the smart contract. Both the workers have to send their deposits before the intent time τ_i. This deposit is returned to workers only if they behave diligently. As a condition of both the workers have to submit their deposit before τ_i, if only one worker has sent the deposit or none of the workers has sent, then the contract is terminated. The workers have to compute and submit their output before τ_c. If both workers fail to submit, then their deposits are sent to delegator. If only one worker has submitted, then TWOIOCV contract is invoked, and payment is made according to the result returned by TWOIOCV. If the workers computed the algorithm diligently and the delegator acts honestly, then the TWOIOCV contract is never invoked. The utilities for the delegator and the workers are given in Table 3. We already defined that the cost of honest computation is $cost(1)$, and the cost of running a q-algorithm is $cost(q)$. Let the cost of running the TWOIOCV contract be $cost(V)$. We assume that the deposit by a worker \$$d \geq cost(V)$. \$$d$ compensates the delegator for the $cost(V)$. If the workers collude and sent the same incorrect output, then their utility will be maximum. If no collusion occurs, then the best strategy for the workers is to act diligently. Colluding of the workers is avoided by offering a bounty for acting diligently such that \$$r - cost(1)$+\$$b >$ \$$r - cost(q)$. This bounty is given to the diligent worker only when the outputs are different. Unfortunately, this bounty is a burden to the delegator as this is an extra payment apart from the reward.

4.1.2. Keeping the secrets

The formal contract for the two workers case is given in Fig. 4. As the communication between the workers and the contract is through a public environment, the lazy worker can listen to the blockchain network and submit the same output as a diligent worker without actually doing the computation. To avoid this attack we used time-based commitments, where the worker has to first submit the commitment of the output before τ_c, and later he needs to reveal his commitment before τ_r(There is a chance that both workers may submit the same commitment, in this case, the contract will reject the submission of the second commitment and asks him to resubmit it. The worker may recompute the commitment with different parameters which are not used earlier, for the sake of simplicity, we have not included this in our protocol). The formal protocol for the delegator and the workers is given in Fig. 5. The workers use the commitment protocol from Pedersen (1991) to generate commitments. These commitments are revealed later. If any worker fails to reveal the commitment, then his deposit is forfeited.

4.1.3. TWOIOC contract states

1. When the delegator successfully calls the create functionality, the TWOIOC contract moves from Init state to Created state.
2. From Created state, the contract will move into one of the two states: Compute or Aborted. The contract moves to Compute state if two workers successfully called the intent functionality.
3. From Created state, the contract is moved to Aborted state if $\tau > \tau_i$. This happens when no worker has shown intent or only one worker has shown intent. Before moving to Aborted state the reward and bounty are refunded to delegator. If any worker has shown intent, his deposit is also refunded.
4. From Compute state, the contract will move into one of the two states: Reveal or Aborted. The contract will move into Reveal

Table 3
Analysis of TWOIOC: 1-Diligent, 0-Lazy, CO-Correct Output, WO-Wrong Output, $cost(V)$ is cost of executing TWOIOCV Contract. Here, we assume TWOIOCV contract is invoked by $\mathcal{D}$ and hence the $cost(V)$ is paid by $\mathcal{D}$. *when both workers return the same incorrect outputs. ** when the workers return different incorrect results.

Enities			Utility				
s.no	$\mathcal{W}_0$	$\mathcal{W}_1$	output	$\mu(\mathcal{D})$	$\mu(\mathcal{W}_0)$	$\mu(\mathcal{W}_1)$	Clause
1	1	1	CO	–	\$$r - cost(1)$	\$$r - cost(1)$	6
2	1	0	CO	\$$d - cost(V)$	\$$r$ + \$$b - cost(1)$	−\$$d - cost(q)$	8.1
3	0	1	CO	\$$d - cost(V)$	−\$$d - cost(q)$	\$$r$+\$$b - cost(1)$	8.1
4*	0	0	WO	–	\$$r - cost(q)$	\$$r - cost(q)$	6
5**	0	0	WO	2*\$$d - cost(V)$	−\$$d - cost(q)$	−\$$d - cost(q)$	8.2

```
Contract-TWOIOC
Init:     Set $state := Init$, $worker := \{\}$, $output := \{\}$, $commitment := \{\}$
Create:   Upon receiving ("create", $F$, $x$, $\tau_i$, $\tau_c$, $\tau_r$, $\tau_{end}$, $\$r$, $\$b$) from $\mathcal{D}$
            assert $state = Init$.
            assert $\tau < \tau_i < \tau_c < \tau_r < \tau_{end}$.
            assert $ledger[\mathcal{D}] \geq 2 * \$r + \$b$.
            set $ledger[\mathcal{D}] := ledger[\mathcal{D}] - (2 * \$r) - \$b$.
            set $state := Created$.
Intent:   Upon receiving ("intent", $\$d$) from worker $\mathcal{W}_i$
            assert $\tau < \tau_i$ and $state = Created$.
            assert $\mathcal{W}_i \notin worker$ and $ledger[\mathcal{W}_i] \geq \$d$
            set $ledger[\mathcal{W}_i] := ledger[\mathcal{W}_i] - \$d$
            set $worker := worker \cup \mathcal{W}_i$.
            if $|worker| = 2$ then set $state := Compute$.
Commit:   Upon receiving ("commit", $cm_{y_i}$,$cm_{ish_i}$) from $\mathcal{W}_i$
            assert $\tau < \tau_c$ and $state = Compute$.
            assert $\mathcal{W}_i \in worker$ and $(\mathcal{W}_i, *, *) \notin commitment$.
            set $commitment := commitment \cup (\mathcal{W}_i, cm_{y_i}, cm_{ish_i})$.
            if $|commitment| = 2$ then set $state := Reveal$.
Reveal:   Upon receiving ("output", $y_i$, $ish_i$, $s_1$, $s_2$) from $\mathcal{W}_i$
            assert $\tau < \tau_r$ and $state = Reveal$
            assert $(\mathcal{W}_i, *, *) \in commitment$ and $(\mathcal{W}_i, *, *) \notin output$.
            assert $cm_{y_i} = y_i P + s_1 Q$ and $cm_{ish_i} = ish_i P + s_2 Q$
            set $output := output \cup (\mathcal{W}_i, y_i, ish_i)$.
            if $|output| = 2$ then set $state := Pay$.
Dispute:  Upon receiving ("resolve", $y_t$, $ish_t$) from TWOIOCV
            assert $\tau > \tau_r$ and $\tau < \tau_{end}$ and $state = Dispute$
            if $y_t = y_0$ and $ish_t = ish_0$
              set $ledger[\mathcal{W}_0] := ledger[\mathcal{W}_0] + \$r + \$d + \$b$ and $ledger[\mathcal{D}] := ledger[\mathcal{D}] + \$d + \$r$
            else if $y_t = y_1$ and $ish_t = ish_1$
              set $ledger[\mathcal{W}_1] := ledger[\mathcal{W}_1] + \$r + \$d + \$b$ and $ledger[\mathcal{D}] := ledger[\mathcal{D}] + \$d + \$r$
            else set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + 2 * (\$r + \$d) + \$b$
            set $state := Terminated$.
Timer:    if $\tau > \tau_r$ and $state = Pay$.
            if $y_0 = y_1$ and $ish_0 = ish_1$ then
              set $ledger[\mathcal{W}_0] := ledger[\mathcal{W}_0] + \$r + \$d$ and $ledger[\mathcal{W}_1] := ledger[\mathcal{W}_1] + \$r + \$d$
              set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + \$b$
              set $state := Terminated$
            else send ("dispute") to TWOIOCV and set $State := Dispute$
          if $\tau > \tau_i$ and $state = Created$
            set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + 2 * \$r + \$b$
            if $|worker| \neq 0$
              set $ledger[\mathcal{W}_i] := ledger[\mathcal{W}_i] + \$d$, $\forall\ \mathcal{W}_i \in workers$
            set $state := Aborted$
          if $\tau > \tau_c$ and $state = Compute$ and $|commitment| \neq 0$
            set $state := Reveal$
          if $\tau > \tau_c$ and $state = Compute$ and $|commitment| = 0$
            set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + 2 * (\$r + \$d) + \$b$
            set $state := Aborted$
          if $\tau > \tau_r$ and $state = Reveal$ and $|output| \neq 0$
            set $state := Pay$
          if $\tau > \tau_r$ and $state = Reveal$ and $|output| = 0$
            set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + 2 * (\$r + \$d) + \$b$
            set $state := Aborted$
          if $\tau > \tau_{end}$ and $state = Pay || Dispute$
            set $ledger[\mathcal{W}_0] := ledger[\mathcal{W}_0] + \$r + \$d$ and $ledger[\mathcal{W}_1] := ledger[\mathcal{W}_1] + \$r + \$d$
            set $ledger[\mathcal{D}] := ledger[\mathcal{D}] + \$b$
            set $state := Terminated$
```

Fig. 4. Contract-TWOIOC.

state in two cases: (a)if both the workers submit the commitments of the outputs by calling commit functionality before τ_c. (b) If $\tau > \tau_c$ and only one worker has submitted the commitment of the output. In case (b), the second worker's output is set as NULL, and the contract state is changed to Reveal state.

5. From Compute state the contract will move into Aborted state if $\tau > \tau_c$ and no worker has submitted the commitment of the output. Before moving to Aborted state the reward, bounty and the deposits of the workers are sent to delegator.
6. From Reveal state, the contract will move into one of the state: Pay or Aborted. The contract will move into Pay state in two cases: (a) if both workers successfully reveal the commitments before τ_r. (b) if $\tau > \tau_r$ and only one worker has revealed the commitment. In case (b), the second worker's output is set as NULL, and the contract state is changed to Pay state.
7. From Reveal state, the contract will move into Aborted state if $\tau > \tau_r$ and no worker has revealed the commitments. Before moving to Aborted state the reward, bounty and the deposit of the workers are sent to delegator.
8. From Pay state, the contract will move into one of the state: Dispute or Terminated. The contract will move into Terminated state in two cases: (a) if $\tau > \tau_r$ and the equality test of outputs revealed by workers is a success. Before moving into the Terminated state, the workers are paid with reward and their

```
Proto-TWOIOC
Init:            Let $(G, P, Q)$ be the public parameters generated through a
                 trusted setup such that $G$ is an order-$q$ elliptic curve group
                 over $\mathbb{F}_p$, $P$ and $Q$ are random generators of $G$
As Delegator $\mathcal{D}$:
  Create:        Upon receiving ("create", $F$, $x$, $\tau_i$, $\tau_c$, $\tau_r$, $\tau_{end}$, $\mathcal{D}$) from envi-
                 ronment $\mathcal{E}$
                 Send ("create", $F$, $x$, $\tau_i$, $\tau_c$, $\tau_r$, $\tau_{end}$, \$r,\$b) to $\mathcal{G}(Contract - TWOIOC)$
As Worker $\mathcal{W}_i$:
  Intent:        Upon receiving ("intent",$\mathcal{W}_i$) from environment $\mathcal{E}$
                 send ("intent",\$d) to $\mathcal{G}(Contract - TWOIOC)$
  Commit:        Upon receiving ("commit") from environment $\mathcal{E}$
                 assert an intent message was sent earlier
                 run $compute(F, x) \to (y_i, ish_i)$
                 generate two random numbers $s_1 \in_R Z_q$ and $s_2 \in_R Z_q$
                 compute $cm_{y_i} = y_i P + s_1 Q$ and $cm_{ish_i} = ish_i P + s_2 Q$
                 send ("claim", $cm_{y_i}$, $cm_{ish_i}$) to $\mathcal{G}(Contract - TWOIOC)$
  Reveal:        Upon receiving ("reveal") from environment $\mathcal{E}$
                 send (output,$y_i$,$ish_i$,$s_1$,$s_2$) to $\mathcal{G}(Contract - TWOIOC)$
TWOIOCV
  Dispute:       Upon receiving ("dispute", $F$ ,$x$) from TWOIOC
                 run $compute(F, x) \to (y_t, ish_t)$
                 send ("resolve",$y_t$,$ish_t$) to $\mathcal{G}(Contract - TWOIOC)$
```

Fig. 5. User-side programs for TWOIOC.

deposits are refunded, and the bounty is refunded to delegator. (b) if $\tau > \tau_{end}$. Theoretically, this case will not arise due to the deterministic nature of the contract. For the sake of completeness, we included this case in the contract. However, practically this case may arise because the current implementations of Blockchains do not support calling of a contract functionality when a timer expires. Due to this limitation, a separate Pay functionality should be explicitly called by delegator or workers. If all the parties are reluctant to call the Pay functionality, then the contract remains in the Pay state even after $\tau > \tau_{end}$. So, we assume the delegator is responsible for calling Pay functionality, and the workers have already computed, committed and revealed their outputs it is fair to pay the reward and their deposits to them. Before moving into the Terminated state, the workers are paid with reward and also their deposits are refunded. The bounty is refunded to delegator.

9. From Pay state, the contract will move into Dispute state if $\tau > \tau_r$ and the equality test of outputs revealed by workers is failed.
10. From Dispute state, the contract will move into Terminated state in two cases: (a) If the TWOIOCV sends output before $\tau < \tau_{end}$. Before moving into the Terminated state if the output of TWOIOCV matches with any one of the worker, then he is paid with reward along with his deposit and bounty. If the TWOIOCV output doesn't match with any of the workers output, then the reward, bounty and the deposits of the workers are sent to delegator. (b) if $\tau > \tau_{end}$. Theoretically, this case will not arise, but practically this case may arise due to failure[5] in TWOIOCV contract execution. Since the delegator is responsible for contract design and deployment and the workers, have already computed, committed and revealed their outputs it is fair to pay reward and their deposits to them even if their outputs do not match. Before moving into the Terminated state the workers are paid with reward and their deposits. The bounty is refunded to delegator.

[5] Failures like out of gas error, block gas limit exceeded error in Ethereum.

4.2. *Fair replication-based verifiable computation using smart contract - multiple workers case (MULIOC contract)*

We extend the two workers case to a n workers case and try to achieve fair incentivized outsourced computation. If $n = 2$, then this case is similar to a 2-workers case, hence we assume $n > 2$. The high-level overview of the multiple-workers case is that the delegator outsources the job on a public platform(like a public bulletin board) along with the smart contract address. Interested workers show their intent by sending some deposit to the contract. The protocol proceeds in rounds, in each round some k number of workers are randomly hired. The selected workers compute the job and submit their outputs. The contract verifies all the received outputs, and if all the outputs are same, then the contract is terminated by sending appropriate pay to each worker who computed correctly. Otherwise, the contract re-outsources the job to a different set of workers until a correct output is obtained. We assume at least one honest worker is hired per round.

(1) $\mathcal{D}$, $\mathcal{W} = \{\mathcal{W}_1, \ldots, \mathcal{W}_n\}$ agree on a smart contract $\mathcal{SC}$. They also agree on another smart contract MULIOCV, which is invoked in case of disputes.

(2) All parties agree on timing parameters $\tau < \tau_i < \tau_c < \tau_{end}$ and also on k, p where k is the number of workers hired per round and p is the time required per round.
(3) $\mathcal{D}$ will choose $F(\cdot)$ and its input x. $\mathcal{D}$ agrees to pay a minimum of $\$r/k$ to each $\mathcal{W}_i$ for correct and timely computation of $F(x)$
(4) As a condition, the workers who wish to compute $F(x)$ must pay a deposit of $\$d$ before τ_i. Let $\mathcal{WS} \subseteq \mathcal{W}$ be all the workers who paid deposits before τ_i. Any deposits made after τ_i are not considered. Let $\$d'$ be the sum of all the deposits made by workers and let $s = |\mathcal{WS}|$. If $|\mathcal{WS}| \le 2$ after τ_i then the contract is terminated and the reward is refunded to $\mathcal{D}$ and any deposit made by the workers is also refunded.
(5) Until $|\mathcal{WS}| < k$ do
  (a) Generate a random subset of workers $\Omega_r \subset \mathcal{WS}$ and notify them to compute $F(x)$.
  (b) Every $\mathcal{W}_i \in \Omega_r$ computes $F(x)$ and delivers its output before τ_c. If any $\mathcal{W}_i \in \Omega_r$ fails to send result by τ_c, mark it as cheated and set its output as *NULL*.
  (c) $\mathcal{SC}$ compares all the outputs obtained in a r^{th} round for equality, if all the outputs are equal then mark every $\mathcal{W}_i \in \Omega_r$ as honest and also mark all the workers in previous rounds who sent the same output as honest and go to step (7)
  (d) Otherwise mark all the workers in r^{th} round as cheated and set $r = r + 1,\ \tau_c = \tau_c + p,\ \mathcal{WS} = \mathcal{WS} - \Omega_r$.
(6) If $|\mathcal{WS}| < k$, then invoke the MULIOCV contract, similar to TWOIOCV contract and compare the output of the MULIOCV contract with all the hired workers output. Mark all the workers who sent the output same as MULIOCV as honest and mark all the workers who sent a different output as cheated.
(7) Send $\frac{\$d'}{s}\left(\frac{r*k}{r*k-|m|}\right) + \frac{\$r}{r*k-|m|}$ to each honest worker, where $|m|$ is the size of malicious workers who are marked as cheated and lost their deposit for behaving maliciously. Send $\$d'/s$ to all the workers who have not hired in any round and terminate the contract.

4.2.1. Analysis of MULIOC

As the protocol proceeds in the rounds, if there is no consensus on the output in the last round, then the timing parameter τ_c is updated in such a way that the contract can hire a fresh set of workers chosen from the intent list to compute and deliver the output. If all the workers hired in a round are honest, they all will get a reward of $\$r/k$; if any lazy worker is also hired in a round, who returns incorrect output, then the contract re-outsources the job. The lazy worker's deposit is shared among all the honest workers. If $\$d \ge \$r/k$, then delegator need not bother about paying the reward for the workers hired in $r > 1$ round. The sequence of interactions between delegator, workers, and smart contract are shown in Fig. 6. The ideal functionality for multiple-workers case is shown in Fig. 7 and the real world contract functionality is shown in Fig. 8. The protocols to interact with MULIOC contract are given in Fig. 9. Similar to TWOIOC contract, in MULIOC also we used time-based commitments to eliminate lazy workers listening to honest workers interactions with contract and submitting commitments without computing the $F(x)$. The MULIOCV contract is similar to TWOIOCV contract.

4.2.2. Excepted number of rounds

The contract outsources the job until all the workers hired in a round returns the same output. Let E be the probability of getting different outputs in a single round then the expected number of rounds the task is outsourced is $\frac{1}{1-E}$. The contract outsources the task at least once. If the outputs returned in the first round are not equal, then the contract outsources the task again, this will happen with a probability of E. Again if the outputs are not equal, the contract outsources with a probability of E^2 and so on till all the outputs in a round are equal. The expected number of rounds the task is outsourced is

$1 + E + E^2 + \cdots = \frac{1}{1-E}$

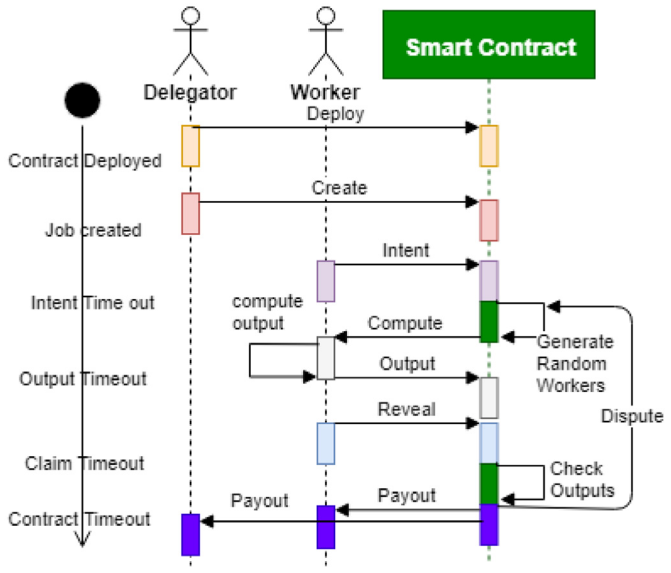

Fig. 6. MULIOC Sequence diagram.

4.2.3. MULIOC contract states

- When the delegator successfully calls the create functionality, the MULIOC contract moves from Init state to Created state.
- From Created state, the contract moves into one of the two states: Compute or Aborted. The contract moves to Compute state if $\tau > \tau_i$ and at least k number of workers has called the intent functionality. The contract moves to Aborted state if $\tau > \tau_{end}$. This happens when the number of workers shown intent are less than k.
- From Compute state, the contract moves into one of the two states: Agreed or Discord. The contract moves into the Agreed state if all the outputs sent by the workers are equal, else the contract moves into Discord state. Before moving into Discord state the parameters τ_c, τ_r and r are updated.
- From Discord state, the contract moves into one of the two states: Compute or Dispute. The contract moves into compute state indicating the start of a new round. The contract moves into Dispute state if $\tau > \tau_c$ and the number of workers to be hired for the new round are less than k.
- From Dispute state, the contract moves into one of the two states: Agreed or Terminated. The contract moves into the Agreed state if MULIOCV contract returns the result before τ_{end}. Before moving to Agreed state a list of honest and cheated workers is prepared based on the output returned by MULIOCV. The contract moves into Terminated state if $\tau > \tau_{end}$ and the MULIOCV contract fails to return the output before τ_{end}. Before moving to the Terminated state, the workers are paid with reward and their deposit. This reward is smaller than what the delegator initially agreed to pay each worker. However, this scenario does not happen if the MULIOCV contract returns before $\tau > \tau_{end}$.
- From Agreed state, the contract moves into Terminated state if $\tau > \tau_{end}$. Before moving to Terminated state every honest worker is paid with reward, and their deposit is refunded. The unhired workers' deposit is also refunded.

```
                              Ideal-MULIOC
Init:     Set $state := Init$, $r := 1$, $output := \{\}$, $\$reward := 0$, $\$deposit := 0$, $n := 0$, $H := \{\}$, $M := \{\}$ and
          $\mathcal{W} := \{\}$
Create:   Upon receiving ("create", $F$, $x$, $\tau_i$, $\tau_c$, $\tau_r$, $\tau_{end}$, $\$r$) from $\mathcal{D}$
          assert $state = Init$ and $ledger[\mathcal{D}] \geq \$r$.
          assert $\tau < \tau_i < \tau_c < \tau_r < \tau_{end}$
          set $ledger[\mathcal{D}] := ledger[\mathcal{D}]$ - $\$r$ and $\$reward := \$r$.
          set $state := Created$
          Notify $\mathcal{S}$ of ("create", $F$, $x$, $\tau_i$, $\tau_c$, $\tau_r$, $\tau_{end}$, $\$r$,$\mathcal{D}$)
Intent:   Upon receiving ("intent",$\$d$) from worker $\mathcal{W}_i$
          assert $state := Created.$
          assert $\tau \leq \tau_i$ and $\mathcal{W}_i \notin \mathcal{W}$
          assert $ledger[\mathcal{W}_i] \geq \$d$ and $\$d \geq \$reward$
          set $ledger[\mathcal{W}_i] := ledger[\mathcal{W}_i]$ - $\$d$.
          set $\$deposit := \$deposit + \$d$.
          set $\mathcal{W} := \mathcal{W} \cup \mathcal{W}_i$.
          set $n := n + 1$
          Notify $\mathcal{S}$ of ("intent",$\$d$,$\mathcal{W}_i$)
Output:   Upon receiving ("output",$y_i$, $ish_i$) from $\mathcal{W}_i$
          assert $\mathcal{W}_i \in \Omega_r$ and $\tau < \tau_c$
          assert $state = Compute$
          assert $\mathcal{W}_i$ had sent an "intent" message earlier
          set $output := output \cup (y_i, ish_i, \mathcal{W}_i)$
          Notify $\mathcal{S}$ of ("output",$\mathcal{W}_i$)
Claim:    Upon receiving ("claim") from $\mathcal{W}_i$
          assert $\tau_c < \tau < \tau_r$
          assert $\mathcal{W}_i$ had sent an "output" message earlier
          Notify $\mathcal{S}$ of ("claim",$\mathcal{W}_i$)
Dispute:  Same as in Contract-MULIOC(Figure 8)
Timer:    Same as in Contract-MULIOC(Figure 8)
```

Fig. 7. Ideal-MULIOC.

Next, we prove our protocols given in Fig. 9 securely emulates the ideal functionality given in 7 and no polynomial time environment can distinguish whether it is in ideal-world or real-world.

Theorem 1. *Assuming the commitment scheme comm is adaptively secure, then the protocol described in Fig. 9 securely emulates* $\mathcal{F}(Ideal - MULIOC)$

4.3. Proofs for MULIOC contract

We now prove Theorem 1. For any real-world adversary $\mathcal{A}$ we construct an ideal-world simulator $\mathcal{S}$, such that no polynomial-time environment $\mathcal{E}$ can distinguish whether it is in real-world or ideal-world. We first show the construction of the simulator $\mathcal{S}$ and then prove the indistinguishability of ideal and real worlds.

4.3.1. Ideal world simulator

We construct a simulator $\mathcal{S}$ for dummy adversary which will pass messages to and from the environment $\mathcal{E}$. $\mathcal{S}$ can also interact with $\mathcal{F}(ideal - MULIOC)$. The ideal adversary $\mathcal{S}$ can be obtained by applying the simulator wrapper $\mathcal{S}(simP)$ (Juels et al., 2016). *Init* The simP generates the public parameters (G, P, Q) such that G is an order-q elliptic curve group over $\mathbb{F}_p$, P and Q are random generators of G. *Simulation for honest parties* When the environment sends messages to the honest parties, the simulator $\mathcal{S}$ needs to simulates messages for the corrupted parties, from honest parties or from functionalities in the real world. The honest parties will be simulated as below.

- *Create:* Environment $\mathcal{E}$ sends input ("create", F, x, τ_i, τ_c, τ_r, τ_{end}, $\$r$, $\mathcal{D}$) to a honest delegator $\mathcal{D}$: simP receives ("create",F, x, τ_i, T_c, T_{end}, $\$r$, $\mathcal{D}$) from ideal functionality $\mathcal{F}(Ideal - MULIOC)$. simP forwards the messages to the internally simulated $\mathcal{G}(Contract - IOC)$ contract functionality, as well to the environment $\mathcal{E}$.
- *Intent*: Environment $\mathcal{E}$ sends ("intent",$\$d$) to the honest worker $\mathcal{W}_i$: simP receives ("intent",$\$d, \mathcal{W}_i$) from the ideal functionality $\mathcal{F}(Ideal - MULIOC)$. Now simP sends ("intent",$\$d, \mathcal{W}_i$) to the internally simulated $\mathcal{G}(Contract - MULIOC)$ contract functionality.
- *Output*: Environment $\mathcal{E}$ sends ("output",y_i, ish_i) to the an honest worker: Simulator simP receives notification from the ideal $\mathcal{F}(Ideal - MULIOC)$ without seeing (y_i, ish_i). simP now computes y_i and ish_i to be as commitments of two 0 vectors. simP sends ("output",cm_{y_i}, cm_{ish_i}) to the simulated $\mathcal{G}(Contract - MULIOC)$ functionality and simulates the contract in the obvious manner.
- *Claim*: Environment $\mathcal{E}$ sends ("claim") the worker $\mathcal{W}_i$
  - Case 1: if the worker is honest. simP sends ("claim",y_i, ish_i, s_1, s_2) to $\mathcal{G}(contract - MULIOC)$ where y_i and ish_i are previously simulated values and s_1 and s_2 are random values used in computation of commitment of y and ish respectively.
  - Case 2: if worker is corrupted. simP receives (y_i, ish_i) from $\mathcal{F}(Ideal - MULIOC)$ and computes $(y_i^{'}, ish_i^{'})$ using the honest algorithm. Suppose the corresponding randomness are $s_1^{'}$ and $s_2^{'}$ simP now sends ("reveal",$y_i^{'}$, $ish_i^{'}$, $s_1^{'}$, $s_2^{'}$) to $\mathcal{G}(contract - MULIOC)$ and simulates the contract functionality in oblivious manner.

Simulation for corrupted parties

- *Create:* simulator simP receives a message ("create", F, x, τ_i, τ_c, τ_r, τ_{end}, $\$r$, $\mathcal{D}$). do nothing.
- *Intent:* simP receives message ("intent",$\$d, \mathcal{W}_i$). Forwards it to the internally simulated $\mathcal{G}(contract - MULIOC)$.
- *Output:* simP receives a output message ("output",cm_{y_i}, cm_{ish_i}): simP runs a commitment extraction algorithm and recovers y_i and ish_i. simP now computes cm_{y_i} and cm_{ish_i} with different s_1 and s_2 and sends ("output", cm_{y_i}, cm_{ish_i}) to $\mathcal{G}(contract - MULIOC)$.
- *Claim:* simP receives ("claim",y_i, ish_i, s_1, s_2). Forwards it to the internally simulated $\mathcal{G}(contract - MULIOC)$.

Indistinguishability of real and ideal worlds A sequence of hybrid games are required to prove the indistinguishability of real and ideal worlds from the perspective of environment $\mathcal{E}$.

```
Contract-MULIOC
Init:     Set $state := Init$, $r = 1$, $output = \{\}$, $\$reward = 0$, $\$deposit = 0$, $n = 0$, $commitments = \{\}$, $H = \{\}$,
          $M = \{\}$ and $\mathcal{W} := \{\}$
Create:   Upon receiving ("create", $F$, $x$, $\tau_i$, $\tau_c$, $\tau_r$, $\tau_{end}$, $\$r$) from $\mathcal{D}$
          assert $state = Init$ and $ledger[\mathcal{D}] \geq \$r$.
          assert $\tau < \tau_i < \tau_c < \tau_r < \tau_{end}$
          set $ledger[\mathcal{D}] := ledger[\mathcal{D}]$ - $\$r$ and $\$reward := \$r$.
          set $state := Created$
Intent:   Upon receiving ("intent",$\$d$) from worker $\mathcal{W}_i$
          assert $state := Created$ and $\tau \leq \tau_i$.
          assert $\mathcal{W}_i \notin \mathcal{W}$
          assert $ledger[\mathcal{W}_i] \geq \$d$ and $\$d \geq \$reward$
          set $ledger[\mathcal{W}_i] := ledger[\mathcal{W}_i]$ - $\$d$.
          set $\$deposit := \$deposit + \$d$.
          set $\mathcal{W} := \mathcal{W} \cup \mathcal{W}_i$.
          set $n := n + 1$
Output:   Upon receiving ("commit",$cm_{y_i}$, $cm_{ish_i}$) from $\mathcal{W}_i$
          assert $\mathcal{W}_i \in \Omega_r$
          assert $\tau \leq \tau_c$ and $state = Compute$
          assert $(\mathcal{W}_i, *, *) \notin commitments$
          $commitments = commitments \cup (\mathcal{W}_i, cm_{y_i}, cm_{ish_i})$
Claim:    Upon receiving ("claim",$y_i, ish_i, s_1, s_2$) from $\mathcal{W}_i$
          assert $\tau_c < \tau < \tau_r$
          assert $\mathcal{W}_i \in \Omega_r$ and $(\mathcal{W}_i, *, *) \notin output$
          assert $cm_{y_i} = y_i P + s_1 Q$ and $cm_{ish_i} = ish_i P + s_2 Q$
          set $output = output \cup (\mathcal{W}_i, y_i, ish_i)$
Dispute:  Upon receiving ("resolve",$y_t$,$ish_t$)
          assert $\tau_c < \tau < \tau_{end}$ and $state = Dispute$
          while $r > 0$
             $(H, M) \leftarrow getHonest(r, \Omega_r, output, y_t, ish_t)$
             $r := r - 1$
          set $state := Agreed$
Timer:    If $\tau > \tau_i$ and $state = Created||Discord$
            assert $|\mathcal{W}| \geq k$
            select a random subset $\Omega_r \subset \mathcal{W}$ of size $k$
            set $\mathcal{W} := \mathcal{W} - \Omega_r$.
            set $state := Compute$.
            send ("compute",$\Omega_r$) to all parties.
          if $\tau > \tau_r$ and $state = Compute$
            $(status, y, ish) \leftarrow compare(\Omega_r, output)$
            if $status == agreed$
               while $r > 0$
                  $(H, M) \leftarrow getHonest(r, \Omega_r, output, y, ish)$
                  $r := r - 1$
               set $state := Agreed$
            else set $\tau_c = \tau_c + p, \tau_r = \tau_r + p, r = r + 1, state := Discord$
          if $\tau > \tau_c$ and $state = Discord$
            send ("dispute") to MULIOCV and set $state := Dispute$
          if $\tau > \tau_{end}$
            if $state = Agreed$
               set $\$mdeposit = \$deposit/(n - |M| - |\mathcal{W}|)$ and $\$deposit = \$deposit - \$mdeposit$
               for every $\mathcal{W}_i \in H$ set
                  $ledger[\mathcal{W}_i] := ledger[\mathcal{W}_i]$ + $(\$deposit/n)$ + $(\$reward/(k * (r) - |M|))$ + $(\$mdeposit/|H|)$.
               for every $\mathcal{W}_i \in \mathcal{W}$ set
                  $ledger[\mathcal{W}_i] := ledger[\mathcal{W}_i]$ + $(\$deposit/n)$.
               set $state := Terminated$
            if $state = Created$
               set $ledger[\mathcal{D}] = ledger[\mathcal{D}] + \$reward$
               for every $\mathcal{W}_i \in \mathcal{W}$ set $ledger[\mathcal{W}_i] = ledger[\mathcal{W}_i] + \$deposit/n$
               set $state = Aborted$.
            if $state = Dispute$
               for every $\mathcal{W}_i \in \mathcal{W}$ set $ledger[\mathcal{W}_i] = ledger[\mathcal{W}_i] + \$deposit/n$
               while $r > 0$
                  for every $\mathcal{W}_i \in \Omega_r$ set $ledger[\mathcal{W}_i] = ledger[\mathcal{W}_i] + (\$deposit/n)$+$(\$reward/(r * k))$
                  set $r = r - 1$
               set $state = Terminated$.
```

Fig. 8. Contract-MULIOC.

```
Proto-MULIOC
Init:              Let $(G, P, Q)$ be the public parameters generated through a
                   trusted setup such that $G$ is an order-$q$ elliptic curve group
                   over $\mathbb{F}_p$, $P$ and $Q$ are random generators of $G$
As Delegator $D$:
  Create:          Upon receiving ("create", $F, x, \tau_i, \tau_c, \tau_r, \tau_{end}, \mathcal{D}$) from environment $\mathcal{E}$,
                   send ("create", $F, x, \tau_i, \tau_c, \tau_r, \tau_{end}, \$r$) to $\mathcal{G}$(Contract-MULIOC)
As Worker $\mathcal{W}_i$:
  Intent:          Upon receiving ("intent", $\mathcal{W}_i$) from environment $\mathcal{E}$
                   Send ("intent", $\$d$) to $\mathcal{G}$(Contract-MULIOC)
  Compute:         Upon receiving ("compute", $\Omega_r$) from $\mathcal{E}$
                   if $\mathcal{W}_i \in \Omega_r$ then
                       compute $(ish_i, y_i) \leftarrow F(x)$.
                       generate two random numbers $s_1 \in_R Z_q$ and $s_2 \in_R Z_q$
                       compute $cm_{y_i} = y_i P + s_1 Q$ and $cm_{ish_i} = ish_i P + s_2 Q$
                       send ("output", $cm_{y_i}, cm_{ish_i}$) to $\mathcal{G}$(Contract-MULIOC).
  Claim:           Upon receiving ("claim") from $\mathcal{E}$
                   send ("claim", $y_i, ish_i, s_1, s_2$) to $\mathcal{G}$(Contract-MULIOC)
```

Fig. 9. User-side programs for incentivized outsourced computation.

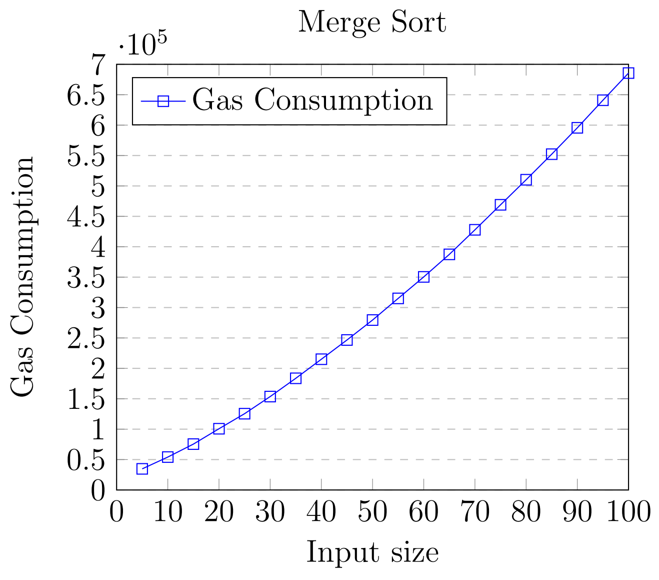

Fig. 10. Gas consumption of merge sort.

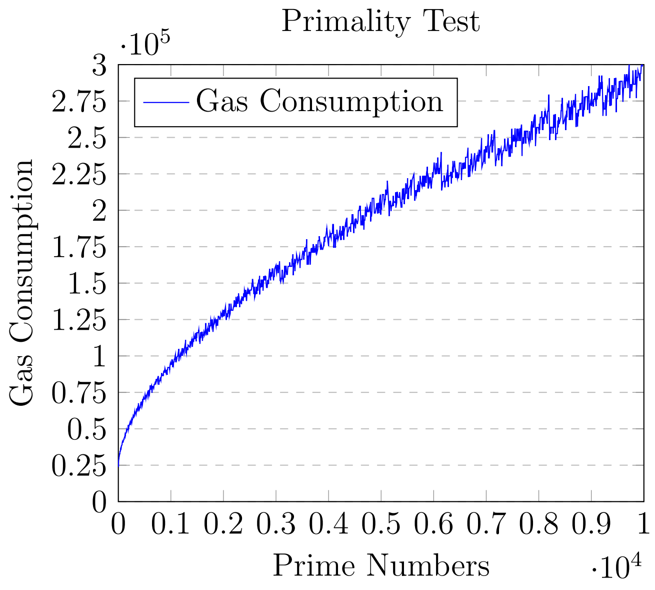

Fig. 11. Gas consumption of prime numbers.

- **Real World**. We start with the real world with a dummy adversary that simply passes messages to and fro from the environment $\mathcal{E}$.
- **Hybrid 1**. Hybrid 1 is same as Real world except now the adversary/simulator will generate (G, P, Q) to perform simulated setup for the commitment scheme. The simulator will pass the simulated (G, P, Q) to the environment $\mathcal{E}$.
  Fact 1. If the commitment scheme is adaptively secure then no polynomial-time environment $\mathcal{E}$ can distinguish Hybrid 1 from the real world except with negligible probability.
- **Hybrid 2**. The simulator simulates the $\mathcal{G}(Contract - MULIOC)$ functionality. Since all the messages to the $\mathcal{G}(Contract - MULIOC)$ are public, simulating the contract functionality is trivial. Therefore, Hybrid 2 is identically distributed as Hybrid 1 from the environment $\mathcal{E}$'s view.
- **Hybrid 3**. Hybrid 3 is same as Hybrid 2 except for the following changes. When an honest party sends a message to the contract(now simulated by the simulator $\mathcal{S}$), it will sign the message with a signature verifiable under an honestly generated nym. In Hybrid 3, the simulator will replace all honest parties' nyms and generate these nyms itself. In this way, the simulator will simulate honest parties' signatures by signing them itself. Hybrid 3 is identically distributed as Hybrid 2 from the environments $\mathcal{E}$'s view.

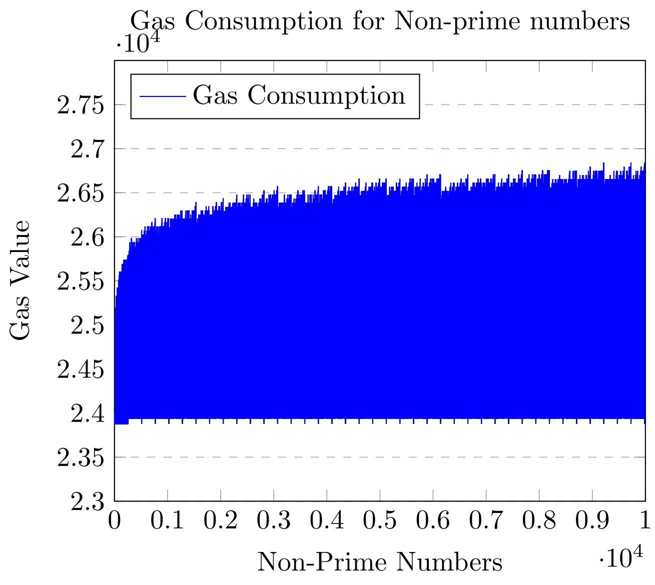

Fig. 12. Gas consumption of non-prime numbers.

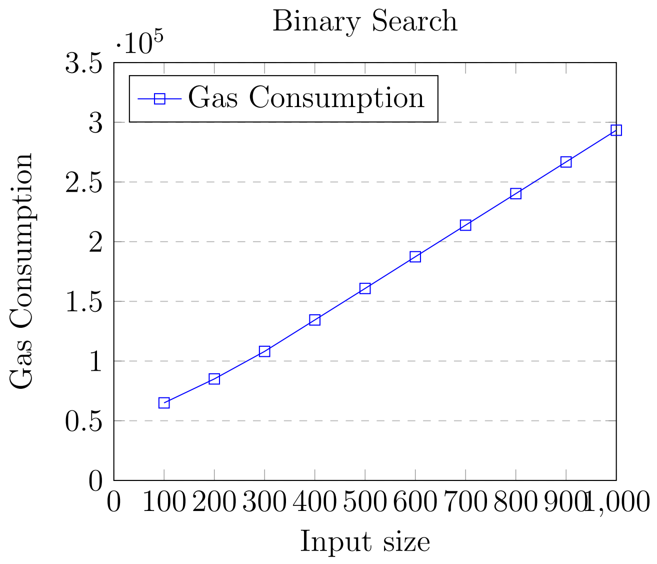

Fig. 13. Gas consumption of Binary Search.

- **Hybrid 4**. Hybrid 4 is same as Hybrid 3 except for the following changes. When an honest worker computes the commitments cm_{y_i} and cm_{ish_i}, then the simulator will replace these commitments with the commitments of two 0 vectors before passing it to the environment $\mathcal{E}$.
  Fact 2. If the commitment scheme is adaptively secure then no polynomial-time environment $\mathcal{E}$ can distinguish Hybrid 4 from Hybrid 3 except with a negligible probability.
- **Hybrid 5**. Hybrid 5 is same as Hybrid 4 except the following changes. Whenever the environment $\mathcal{E}$ passes to the simulator $\mathcal{S}$ a message signed on behalf of an honest party's nym, if the message and signature pair was not among the ones previously passed to the environment $\mathcal{E}$, then the simulator $\mathcal{S}$ aborts.
  Fact 3. Assume that the signature scheme used is secure, then the probability of aborting in Hybrid 5 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 5 would otherwise be identically distributed as Hybrid 4 modulo aborting.
- **Hybrid 6**. Hybrid 6 is same as Hybrid 5 except for the following changes. Whenever the environment passes ("claim",y_i, ish_i, s_1, s_2) to the simulator, if $cm_{y_i} \neq y_iP + s_1Q$ and $cm_{ish_i} \neq ish_iP + s_2Q$, then abort the simulation.
  Fact 4. Assume that the commitment scheme is adaptively secure then the probability of aborting Hybrid 6 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 5 would otherwise be identically distributed as Hybrid 4 modulo aborting.
  Finally, observe that Hybrid 6 is computationally indistinguishable from the ideal simulation $\mathcal{S}$ unless the following bad events occur
  The honest commitment scheme results in collisions. Obviously this will happen with negligible probability if the commitment is scheme is secure and collision-resistant.
  Fact 5. Given that the commitment scheme is adaptively secure, then Hybrid 6 is computationally indistinguishable from the ideal simulation to any polynomial time environment $\mathcal{E}$.

5. Implementation

We have implemented the contracts in solidity 0.5.0 (https://solidity.readthedocs.io/en/v0.5.0/) using truffle framework (https://github.com/trufflesuite). We ran the experiments with a 2.50 GHz Intel Core i5 CPU and a 16GB RAM. The actual tasks are treated as black boxes, and the contracts do not need to know their internal states. The contracts are called before, during or after the execution of the tasks. We have implemented all the contracts in our own private Ethereum network which mimics the Ethereum production network. However, our goal of this implementation is to deploy it in public Blockchain networks in real scenarios.

5.1. Implementation of PBIOC

We ran our experiments multiple times, and each transaction's computational and financial cost is listed in Table 4. Observe that the Init function(Contract deployment) is consuming a large amount of gas, but this will be amortized over multiple agreements between delegator and worker. We designed our contract to be used for outsourcing multiple tasks so that the delegator and worker can run multiple agreements on a single deployment. For implementation feasibility, we modified our PBIOC contract in Fig. 1, because the current blockchain networks support calling of the function in a contract only through an Ethereum account or from a function in the same contract or another contract, i.e., the scheduled function calls which are executed when the timer expires are not possible. The Return and Timer functionalities in Fig. 1 are implemented as Verify and Payout functionalities. Observe that the cost of running the PBIOC contract is minimal. The lack of possibility of running heavy cryptography operations on Ethereum blockchain limits the implementation of the PBIOCV contract. Recently, a new library ZoKrates (Eberhardt and Tai, 2018) using zkSNARKs (Ben-Sasson et al., 2014; Parno et al., 2013) is introduced to perform heavy computations off-chain and the proof of off-chain computations is verified on the Blockchain network. The delegator and worker runs a off-chain setup phase to

Table 4
Costs of interacting with PBIOC Contract. We approximated the gas price as 1 Gwei and 1 ETH = $267.76 which are the real world costs in June 2019. We have rounded off the cost in $ value upto three decimals.

Function	Caller	Cost in Gas	Cost in $
Init	Delegator	21,17,086	0.565
Create	Delegator	302,889	0.080
Intent	Worker	92,255	0.024
Commit	Worker	85,709	0.022
Agree	Delegator	41,905	0.011
Verify	Worker	30,275 +cost(PBIOCV)	0.008+cost(PBIOCV)
Payout	Anyone	37,742	0.010

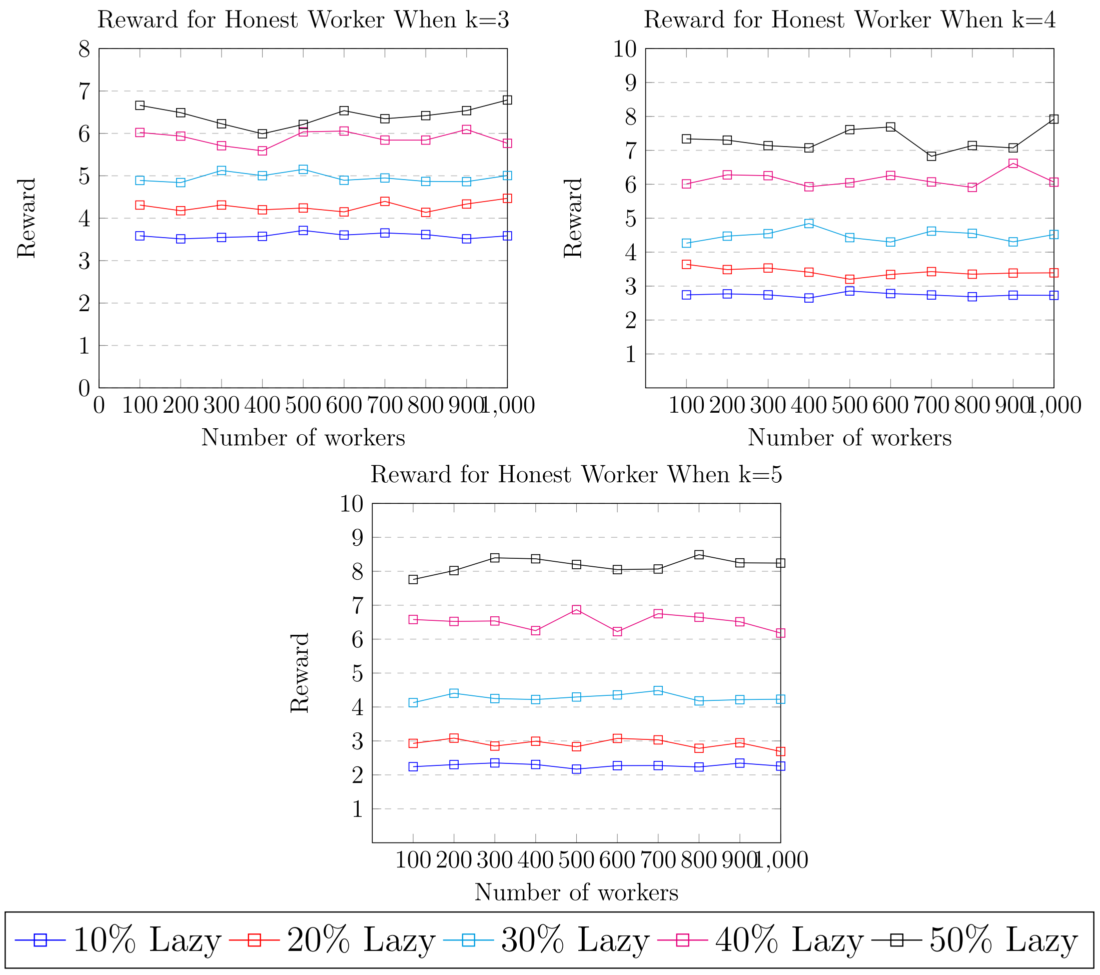

Fig. 14. Reward for honest worker in different scenarios. The reward for the computation is set as 10 Ethers, which is shared by all the honest workers.

Table 5
Deployment and Execution costs of PBIOCV contracts. * The factors multiplication test is conducted only for non-prime number. The primality test for prime numbers is not possible with current Zokrates implementation as modulo operations are expensive on prime fields.

PBIOCV contract	Deployment cost in gas	Execution cost in gas
Sort(input size 10)	1,281,467	736,975
Primality Test*	1,005,959	563,902
Search(input size 10)	1,311,503	999,455

establish a common reference string(CRS) which is used to derive a proving key and verification key. The PBIOCV contract is generated by the delegator using verification key and the delegator deploys the PBIOCV contract and sends its address to the worker. The worker computes the witness and generates a proof. The worker then sends this proof to verify functionality in Fig. 1. We ran the experiments for three problems (1) Sorting integers (2) Primality test on non-prime numbers and 3) Searching in a array of integers. The costs of deploying and running the PBIOCV contracts is shown in Table 5.

5.2. Implementation of TWOIOC

5.2.1. Inner state hash computation

The worker has to send the inner state hash along with the output of the algorithm to prove that he executed the prescribed algorithm given by the delegator. We created an AspectJ aspect which hashes all the inputs and outputs of a method without modifying the actual java code. The delegator outsources the java code, which is compiled along with AspectJ aspect which, when executed, returns the output of the algorithm along with the hash of the inner state. We computed the inner state hash using a Merkle tree (Merkle, 1987). The input string and output string of all the methods in the algorithm will become leaf nodes in the Merkle tree (i.e., two nodes for one method call). Once the execution is completed, a Merkle tree is constructed with all the leaf nodes. The root of the Merkle tree is returned as the inner state hash of an algorithm.

The workers will compute the commitments using Pedersen (1991) commitments based on the public parameters given in McCorry et al. (2017). We also use the Cryptocon contract for verifying the commitments submitted by the workers. We invoked several instances of the TWOIOC contract, and each transaction's computational and financial cost is listed in Table 6. As we discussed earlier, our TWOIOC contract is simple and invokes TWOIOCV contract only in case disputes. We modeled (1) Merge sort (2) Primality test and (3) Binary search as TWOIOCV contracts and deployed them on Ganache Blockchain. Merge sort contract takes an array of integers as an input and outputs the sorted array of numbers. We used a recursive merge sort implementation and hence we could not test it for large input sizes due to the limit on the number of call stacks of a smart contract in Ethereum Blockchain. The number of call stacks limit is hardcoded as 1024 stack frames which limit the number of function calls allowed. We also could not test for sorting fractional numbers as Ethereum does not support operations on floating-point numbers.

Table 6
Costs of interacting with TWOIOC Contract. We approximated the gas price as 1 Gwei and 1 ETH = \$267.76 which are the real world costs in June 2019.

Functions	Caller	Cost in Gas	Cost in \$
Init	Delegator	2,526,981	0.67
Create	Delegator	64,424	0.01
Intent	Worker	64,320	0.01
Commit	Worker	118,142	0.031
Reveal	Worker	386,452	0.10
Check	Anyone	40,364	0.01
Payout	Anyone	39,640	0.01

The Merge sort contract is tested for input sizes from 5 to 100, where all the numbers in a single test are generated randomly, and the transaction costs are listed in Fig. 10. The second TWOIOCV contract we deployed will check whether the given number is prime or not. We tested the primality contract for integers up to 10000, and the transaction cost of prime numbers is listed in Fig. 11 and transactional cost of non-prime numbers is given in Fig. 12. The gas cost to test non-prime number is very less when compared to the gas cost for testing prime numbers. Also, the gas cost to test prime numbers increases as we move to higher numbers. Observe that the gas consumption of non-prime number varies dramatically. This is because we are using Fermat's little theorem for primality test. To reduce the gas cost, we included a condition "if $num\%2 == 0 \mid\mid num\%3 == 0$ then return *false*" where *num* is the input to primality test. This statement consumes an almost constant amount of gas for all the non-prime numbers which are having 2 or 3 as one of the factors. For all the remaining non-prime numbers we check the condition "$a^{num-1} \equiv 1 \bmod (num)$" for some random $1 < a < num - 1$. This condition is checked minimum once and a maximum of $\sqrt{num}$ times. As a third TWOIOCV contract, we implemented and tested the Binary Search contract. The binary search contract is tested for input sizes from 100 to 1000, and the transactional costs for searching an element which doesn't exist in the input are shown in Fig. 13.

5.3. Implementation of MULIOC

We deployed our MULIOC contract in the Ganache Blockchain and tested it under different parameters. The first parameter we varied is the number of workers. we varied the number of workers from 10 to 90. The second parameter we varied is the number of lazy workers who show their intent to compute the task. The percentage of the lazy workers varies from 10% to 50%. The third parameter we varied is the number of workers selected per round. We tested our contract for the number of workers selected per round from 3 to 5. In Table 7 we show cost of transactions sent to the MULIOC contract. As currently Ganache doesn't support for large number of accounts we simulated the contract in Java for n=100 to 1000 and we show the pay received by the honest workers in all the above scenarios in Fig. 14.

6. Conclusion and future work

Blockchains and smart contracts came up with new solutions to the problems which are thought hard to solve. One such problem is achieving fair payments for verifiable computations without a trusted intermediary. We have designed fair protocols using smart contracts for both types of verifiable computations: (1) Proof-based verifiable computation and (2) Replication-based verifiable computation. Our experiments show that in fair proof-based verifi-

Table 7
Costs of running MULIOC contract.

Function			Init	Create	Intent	Commit			Reveal		
Cost in Gas			2,908,677	126,179	150,966	66576			33,838		
Cost in \$			0.77662	0.0337	0.04032	0.01778			0.00902		
Function			Compute	Check	Payout				Compute	Check	payout
$n = 10$	$k = 3$	gas	219,779	165,631	131,866	$n = 60$	$k = 3$	gas	424,140	2,483,879	660,366
		\$	0.05869	0.04422	0.03522			\$	0.11323	0.6632	0.17633
	$k = 4$	gas	265,589	190,749	132,541		$k = 4$	gas	471,637	2,522,335	661,041
		\$	0.07092	0.05092	0.03538			\$	0.12592	0.67345	0.17649
	$k = 5$	gas	314,773	220,080	133,216		$k = 5$	gas	519,135	2,562,952	661,716
		\$	0.08405	0.05877	0.03556			\$	0.1386	0.68432	0.17667
$n = 20$	$k = 3$	gas	259,300	373,656	237,566	$n = 70$	$k = 3$	gas	465,350	3,353,794	766,066
		\$	0.06923	0.09978	0.06344			\$	0.12426	0.89546	0.20455
	$k = 4$	gas	306,797	403,904	238,241		$k = 4$	gas	512,847	3,378,912	766,741
		\$	0.08192	0.10784	0.0636			\$	0.13692	0.90217	0.20471
	$k = 5$	gas	355,153	421,436	238,916		$k = 5$	gas	560,345	3,425,172	767,416
		\$	0.09484	0.11251	0.06379			\$	0.1496	0.91453	0.2049
$n = 30$	$k = 3$	gas	300,510	700,721	343,266	$n = 80$	$k = 3$	gas	506,560	4,331,463	871,766
		\$	0.08023	0.18709	0.09166			\$	0.13526	1.15651	0.23277
	$k = 4$	gas	348,007	733,021	343,941		$k = 4$	gas	554,057	4,381,718	872,441
		\$	0.09292	0.19571	0.09182			\$	0.14794	1.16991	0.23293
	$k = 5$	gas	395,505	755,170	344,616		$k = 5$	gas	601,555	4,356,671	873,116
		\$	0.1056	0.20164	0.09201			\$	0.16063	1.16324	0.23312
$n = 40$	$k = 3$	gas	341,720	1,177,093	448,966	$n = 90$	$k = 3$	gas	547,770	5,409,704	977,466
		\$	0.09123	0.31429	0.11988			\$	0.14626	1.44439	0.26099
	$k = 4$	gas	389,217	1,203,750	449,641		$k = 4$	gas	595,267	5,440,978	978,141
		\$	0.10392	0.32141	0.12004			\$	0.15895	1.45275	0.26115
	$k = 5$	gas	436,715	1,238,211	450,316		$k = 5$	gas	643,609	5,497,498	978,816
		\$	0.1166	0.3306	0.12023			\$	0.17184	1.46783	0.26134
$n = 50$	$k = 3$	gas	382,930	1,770,453	554,666	$n = 100$	$k = 3$	gas	584,859	6,537,458	1,072,596
		\$	0.10223	0.47272	0.1481			\$	0.15617	1.74551	0.28638
	$k = 4$	gas	430,427	1,828,916	555,341		$k = 4$	gas	632,356	6,613,876	1,073,271
		\$	0.11492	0.48832	0.14827			\$	0.16885	1.76591	0.28657
	$k = 5$	gas	478,769	1,801,817	556,016		$k = 5$	gas	679,854	6,628,843	1,073,946
		\$	0.12784	0.48108	0.14845			\$	0.18153	1.76989	0.28673

able computation, the overhead of using smart contract is minimal when both delegator and worker are honest. We achieved fairness in replication-based verifiable computation by imposing fines on cheating workers and offering bounties to honest workers. We have shown that monetized penalties are an efficient way to deter cheating workers. We ran experiments for both types of verifiable computations and presented the transactional and financial costs of interacting with smart contracts. Our work could serve as a founding stone to future works which can design more robust, secure smart contracts for fair verifiable computations with less number of rounds of interactions. However, our protocols do not provide privacy to the output. One area of future work will mainly focus on the development of fair protocols for verifiable computations using smart contracts which will also provide privacy of the inputs and outputs of an outsourced problem.

Declaration of Competing Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

We confirm that the manuscript has been read and approved by all named authors and that are no other persons who satisfied the criteria for authorship but are not listed. We further confirm that all of us have approved the order of authors listed in the manuscript.

We confirm that we have given due consideration to the protection of intellectual property associated with this work and that there are no impediments to publication, including the timing of publication, with respect to intellectual property. In so doing we confirm that we have followed the regulations of our institutions concerning intellectual property.

We understand that the Corresponding Author is the sole contact for the Editorial process (including Editorial Manager and direct communications with the office). He/she is responsible for communicating with the other authors about progress, submissions of revisions and final approval of proofs. We confirm that we have provided a current, correct email address, which is accessible by the Corresponding Author arjun753016@gmail.com.

References

Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D., 2002. Seti@ home: an experiment in public-resource computing. Commun. ACM 45 (11), 56–61.

Belenkiy, M., Chase, M., Erway, C.C., Jannotti, J., Küpçü, A., Lysyanskaya, A., 2008. Incentivizing outsourced computation. In: Proceedings of the 3rd International Workshop on Economics of Networked Systems. ACM, pp. 85–90.

Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M., 2014. Succinct non-interactive zero knowledge for a Von Neumann architecture.. In: USENIX Security Symposium, pp. 781–796.

Brassard, G., Chaum, D., Crépeau, C., 1988. Minimum disclosure proofs of knowledge. J. Comput. Syst. Sci. 37 (2), 156–189.

Canetti, R., 2001. Universally composable security: a new paradigm for cryptographic protocols. In: Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on. IEEE, pp. 136–145.

Canetti, R., Riva, B., Rothblum, G.N., 2011. Practical delegation of computation using multiple servers. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 445–454. doi:10.1145/2046707.2046759.

Carbunar, B., Tripunitara, M., 2010. Fair payments for outsourced computations. In: Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference. IEEE, pp. 1–9.

Carbunar, B., Tripunitara, M.V., 2012. Payments for outsourced computations. IEEE Trans. Parallel and Distrib.Syst. 23 (2), 313–320.

Chen, X., Li, J., Susilo, W., 2012. Efficient fair conditional payments for outsourcing computations. IEEE Trans. Inf. Forensics Secur. 7 (6), 1687–1694.

Chung, K.-M., Kalai, Y., Vadhan, S., 2010. Improved delegation of computation using fully homomorphic encryption. In: Advances in Cryptology – CRYPTO 2010, pp. 483–501.

Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., Zahur, S., 2015. Geppetto: versatile verifiable computation. In: 2015 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 253–270.

Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A., 2017. Betrayal, distrust, and rationality: smart counter-collusion contracts for verifiable cloud computing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 211–227.

Dorsala, M.R., Sastry, V.N., chapram, S., 2018. Fair protocols for verifiable computations using bitcoin and ethereum. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, USA, pp. 786–793. doi:10.1109/CLOUD.2018.00107.

Eberhardt, J., Tai, S., 2018. Zokrates - scalable privacy-preserving off-chain computations. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 1084–1091.

Gennaro, R., Gentry, C., Parno, B., 2010. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In: Advances in Cryptology – CRYPTO 2010. Springer Berlin Heidelberg, pp. 465–482.

Goldreich, O., 1999. Probabilistic proof systems. In: Modern Cryptography, Probabilistic Proofs and Pseudorandomness, pp. 39–72.

Goldwasser, S., Micali, S., Rackoff, C., 1989. The knowledge complexity of interactive proof systems. SIAM J. Comput. 18 (1), 186–208.

Golle, P., Mironov, I., 2001. Uncheatable computations. In: Cryptographers' Track at the RSA Conference. Springer, pp. 425–440.

Juels, A., Kosba, A., Shi, E., 2016. The ring of gyges: investigating the future of criminal smart contracts. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 283–295.

Kilian, J., 1992. 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.

Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C., 2016. Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 839–858.

Kosba, A., Papamanthou, C., Shi, E., 2018. xjsnark: a framework for efficient verifiable computation. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 944–961.

Kumaresan, R., Bentov, I., 2014. How to use bitcoin to incentivize correct computations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 30–41.

Küpçü, A., 2017. Incentivized outsourced computation resistant to malicious contractors. IEEE Trans. Depend. Secure Comput. 14 (6), 633–649.

Li, H.C., Clement, A., Wong, E.L., Napper, J., Roy, I., Alvisi, L., Dahlin, M., 2006. Bar gossip. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. USENIX Association, pp. 191–204.

Lund, C., Fortnow, L., Karloff, H., Nisan, N., 1990. Algebraic methods for interactive proof systems. In: Proceedings of the 31st Annual Symposium on Foundations of Computer Science, pp. 2–10 vol.1.

Luu, L., Teutsch, J., Kulkarni, R., Saxena, P., 2015. Demystifying incentives in the consensus computer. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 706–719.

McCorry, P., Shahandashti, S.F., Hao, F., 2017. A smart contract for boardroom voting with maximum voter privacy. In: Financial Cryptography and Data Security, pp. 357–375.

Merkle, R.C., 1987. A digital signature based on a conventional encryption function. In: Conference on the Theory and Application of Cryptographic Techniques. Springer, pp. 369–378.

Molnar, D., 2000. The seti@ Home Problem. ACM Crossroads, Sep 2000. https://www.acm.org/crossroads/columns/onpatrol/september2000.html

Nakamoto, S., 2008. Bitcoin: A Peer-to-Peer Electronic Cash System.

Parno, B., Howell, J., Gentry, C., Raykova, M., 2013. Pinocchio: Nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy. IEEE, pp. 238–252.

Pedersen, T.P., 1991. Non-interactive and information-theoretic secure verifiable secret sharing. In: Annual International Cryptology Conference. Springer, pp. 129–140.

Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M., 2013. Resolving the conflict between generality and plausibility in verified computation. In: Proceedings of the 8th ACM European Conference on Computer Systems. ACM, pp. 71–84.

Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M., 2012. Taking proof-based verified computation a few steps closer to practicality. In: 21st USENIX Security Symposium (USENIX Security 12), pp. 253–268.

Setty, S.T., McPherson, R., Blumberg, A.J., Walfish, M., 2012. Making argument systems for outsourced computation practical (sometimes). NDSS.

Sudan, M., 2009. Probabilistically checkable proofs. Commun. ACM 52 (3), 76–84.

Wood, G., 2014. Ethereum: a secure decentralised generalised transaction ledger. In: Ethereum Project Yellow Paper, 151, pp. 1–32.

Zhang, Y., Deng, R., Liu, X., Zheng, D., 2018. Outsourcing service fair payment based on blockchain and its applications in cloud computing. IEEE Trans. Serv. Comput. doi:10.1109/TSC.2018.2864191.

Zhang, Y., Deng, R.H., Liu, X., Zheng, D., 2018. Blockchain based efficient and robust fair payment for outsourcing services in cloud computing. Inf. Sci. 462, 262–277. doi:10.1016/j.ins.2018.06.018.

Dorsala Mallikarjun Reddy received the MTech degree in computer science and systems engineering from Andhra University. He is currently doing PhD at National Institute of Technology, Warangal. His research interest includes Blockchain networks, Distributed Systems, and Cryptography.

V. N. Sastry is currently working as professor in the Institute for Development and Research in Banking Technology (IDRBT), Hyderabad, India. He received his Ph.D. and M.Sc from Indian Institute of Technology, Kharagpur, India. His research and teaching interests include Operations Research, Multiple Objective Network Optimization, Security Models, Mobile Computing, Mobile Payments, Mobile Governance, Fuzzy Optimization and Fuzzy Control, Financial Engineering, Asset Liability Management, Portfolio Optimization.

Chapram Sudhakar is currently working as an associate professor in National Institute of Technology, Warangal. He received the MTech degree in computer science from Indian Institute of Technology, Kanpur and the PhD degree in computer science from National Institute of Technology, Warangal. His research interest includes Cloud Computing, Distributed Systems, High-Performance Computing, and Operating Systems.