Verifiable Computation using Smart Contracts

Sepideh Avizheh, Mahmudun Nabi, Reihaneh Safavi-Naini, Muni Venkateswarlu K.*
University of Calgary, Canada

ABSTRACT

Outsourcing computation has been widely used to allow weak clients to access computational resources of a cloud. A natural security requirement for the client is to be able to efficiently verify the received computation result. An attractive approach to verifying a general computation is to send the computation to multiple clouds, and use carefully designed protocols to compare the results and achieve verifiability. This however requires a Trusted Third Party (TTP) to manage the interactions of the client and the clouds. Our goal is to employ a smart contract to act as the TTP. This also relieves the client from directly interacting with the clouds, and engaging in possibly a complex stateful protocol. We focus on a verifiable computation protocol of Canetti, Riva and Rothbulm (CRR) with provable security against a malicious cloud, and show that direct employment of the protocol with a smart contract will result in an attack that will undermine the security of the system. We describe and analyze the attack, and extend CRR protocol to protect against this attack, resulting in a secure verifiable computation system using smart contracts. We also give the pseudocode of a smart contract and the required functions that can be used to implement the protocol, written in the Solidity language, and explain its working.

CCS CONCEPTS

• **Security and privacy** → *Distributed systems security.*

KEYWORDS

Outsourcing computation; Verifiable computing; Blockchain; Smart contract

ACM Reference Format:
Sepideh Avizheh, Mahmudun Nabi, Reihaneh Safavi-Naini, Muni Venkateswarlu K.. 2019. Verifiable Computation using Smart Contracts. In *2019 Cloud Computing Security Workshop (CCSW'19), November 11, 2019, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338466.3358925

1 INTRODUCTION

Cloud computing is rapidly overtaking traditional local computation for application and service delivery in digital world. Companies and organizations move to cloud computing to lower infrastructure cost, speed up software deployment, and increase operational flexibility. For (computationally) weak clients cloud computing has opened a vast array of new possibilities by allowing a client to *outsource* its computation to a cloud, and benefit from a plethora of computational resources that are available to the cloud.

A basic security requirement (for a client) in outsourcing computation is the ability to efficiently verify the computation results that are provided by the cloud. A computation result may be incorrect for a variety of reasons that could be grouped into *accidental* and *intentional*. The focus of this paper is on intentional manipulation of the results by the cloud. To allow the client to verify the computation results, cloud must provide a proof that can be efficiently verifiable by the client. Efficiency is important as otherwise, the client could do the computation by itself. There has been a large body of work that consider the problem of verifiability of computation for a general computation[1] in different settings. In [4, 10, 13], a client outsources a computation that is represented by a Boolean circuit to a single cloud, and uses fully homomorphic encryption to allow the cloud to prove the correctness of the computation. These protocols however require initial processing and many computations on many inputs, so that the required initial offline computation cost can be amortized. There is also a large body of theoretical work on probabilistic proof systems, including the works on interactive proofs [5, 14] and probabilistically checkable proofs (PCP) [17, 18], that although are relevant [2], require many rounds of interactions, and so are of limited use in practice. A third intuitively appealing approach is to outsource the computation to a number of clouds, and design a protocol to verify the correctness of the result using multiple independent computations that are received from the clouds. This approach is attractive because of its generality (applicable to a general computation), and the relatively small work that is required from the client. Repeating the computation on multiple clouds is the price of achieving reliability and confidence in the result. A simple verifiable computation system using this approach is to select a solution that is generated by the majority of the clouds as the correct solution. This protocol will give correct result if there are at least three clouds, and one can assume that the majority of clouds are honest.

Canetti, Riva and Rothblum [9] considered verifiable outsourcing with only two clouds, and proposed a protocol, that we refer to as CRR, that builds on the *refereed game* framework of Feige and Kilian [12], and provides provable correctness guarantee *for any efficiently computable function.* This is the minimum number of clouds required if verifiability is based on multiple independent executions. CRR protocol guarantees correct output if *at least one of the two clouds is honest,* and in the case of two differing results,

*The authors' names are given in alphabetical order.

[1] There is also a large body of work that consider specific computations, for example computing inner product of two vectors.

[2] The cloud can apply the function f on the input x to generate the result y, and then prove that $f(x) = y$.

CCSW'19, November 11, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6826-1/19/11…$15.00
https://doi.org/10.1145/3338466.3358925

uses a *refereed game* between the two clouds that identifies the malicious cloud. This ensures that the honest cloud will be rewarded, and so correct behaviour is incentivized. They also provided an implementation of the system where the computation is expressed as a C program, and compiled and packaged into X86 executables (for Windows environment) that can be sent to the cloud.

CRR protocol. At a high level CRR protocol works as follows: the client sends the computation (the program and the input) to two clouds. Once the client receives results from the clouds, it proceeds as follows: (i) if the two results are the same, it accepts the results as correct and terminates (after paying the clouds); (ii) if the results do not match, the client runs a multi-round *refereed game* with the two clouds, where in each round the client asks a question from the two clouds, and based on the received results, asks a follow-up question. The sequence of questions ends with identifying the malicious cloud. We refer to this protocol as MCId, *Malicious Cloud Identification*, protocol. MCId always correctly (and efficiently) identifies the malicious cloud, assuming at least one of the clouds is honest. Thus, once the malicious cloud is found, the other one must be honest and their solution will be correct.

CRR protocol (also [11, 15]) assumes the client is trusted and correctly performs the protocol. In practice however, clouds also need to trust the client in honouring its promised payments for the computation. To remove these assumptions, one may use a Trusted Third Party (TTP) who will *manage* the required interactions and payments.

Smart contracts as a TTP for outsourcing. In this paper our goal is to use a smart contract for a verifiable computation system based on CRR. A blockchain-based smart contract [8] is a public piece of code that resides on a blockchain, specifies a protocol that is run on the underlying *consensus computer* (see Section 2.3) which guarantees trusted execution (assuming majority of consensus nodes are honest), and handles the required transfer of funds. Smart contracts provide an attractive digital alternative to employing a TTP in protocols, and are widely believed to revolutionize many of today's computing services. They provide correctness, transparency and immutability that are expected from a TTP by running the smart contract code on the consensus computer. Using the smart contract to verify computation according to CRR is particularly attractive because of provable guarantees of CRR (verifiability only doubles the computation cost) and its usage for a general computation without requiring excessive pre-processing, interaction, or cryptographic operations.

Using CRR protocol with smart contract. The smart contract will remove the need for trust assumption on the client in CRR, ensure the protocol will be correctly followed, and clouds will be rewarded as specified in the contract. Additionally, it will minimize the client's computation, communication and interaction, which could become unacceptable for computationally weak clients.

We will show however that this attractive proposal will fail completely because of the combination of a number of factors. We show an attack, that we call *copy attack,* that completely breaks the security guarantee of CRR protocol when used with a smart contract. We describe and analyze this attack, propose an extension of CRR protocol that can effectively protect against the attack, and give the pseudocode of our solution as a smart contract on Ethereum blockchain.

1.1 Our work

We consider verifiable outsourcing by using a smart contract over a cryptocurrency blockchain, and use the CRR protocol for verifiable computation using two clouds. The smart contract will act as a TTP to manage the interactions between parties, and the required transfer of funds between them. For concreteness, we will use Ethereum blockchain throughout, but our work is general and applicable to other commonly used decentralized, and distributed public ledger-based cryptocurrencies that support smart contracts such as Bitcoin (RSK [2]). Ethereum consists of a network of loosely synchronized computing nodes that collectively work to enable fund transfer among account holders, as well as simulating a *consensus computer*, where computational steps are sanctioned by a consensus algorithm (that uses proof-of-work to reach agreement) among active Ethereum nodes (miners). An overview of Ethereum, its blockchain and computation model is given in Section 2.3. Ethereum nodes store smart contracts, each specifying a set of functions, that can be called by users or other contracts. Each node processes transactions that include calls to the functions in the contract, as well as the cryptocurrency transactions that are posted to the network.

Copy attack: CRR protocol is a *full information* (public coin) protocol that uses public communication (messages from the servers to the client are digitally signed but not encrypted). Message transmission in Ethereum network is not instantaneous, and messages that are generated by different nodes (e.g. clouds) take different amounts of time to propagate through the network. This provides an opportunity for a malicious cloud to see the result of the computation that is generated by the other (honest) cloud, and *copy* the result into its own (signed) message (transaction) that is sent to the contract. That is, without doing the actual computation, a malicious cloud is able to produce the correct result (assuming the published result is from an honest cloud). This is a *rushing adversary* in multi-party computation where in each round of computation, chooses its message after seeing the messages of honest parties. The copy attack results in the smart contract to receive two identical solutions, that it will accept as correct, and reward both clouds. Note that this does not break security of CRR because of the requirement that at least one cloud is honest, and will ensure that the copied solution is the correct solution. The effect of this copy attack is that the malicious cloud will receive the reward without doing any computation. In other words the malicious cloud can always be a *free-rider,* and benefit from the computation of the honest cloud.

A more serious negative effect of the copy attack is that CRR argued that the MCId protocol will incentivize honest behaviour of clouds, which are assumed to be *rational*: "we view the servers as rational self-interested parties (say cloud computing service providers)." Knowing the availability of copying the result and the possibility of being rewarded without doing the computation, will be a strong disincentive for honest behavior. A cloud can always wait for the other cloud to submit a solution and copy it. If both clouds use this waiting strategy, there will be a deadlock and no solution will be provided. A rational cloud however can avoid the deadlock by randomizing the time of submitting its solution. Knowing that its solution would be copied by the other cloud (as both are rational), the cloud may find it sufficient to generate a random solution and submit it at a (random) time that takes into account the cloud's estimate of the network delays, and factors such as smart contract time-out (specified deadline to submit a solution). If the solution is seen by the second cloud before publishing its own solution, the copy attack will succeed with a very high probability. Note that the smart contract time-out will be significantly larger than the travel time of a message in the network, and so it will be very likely that copy attack succeeds. There is also a chance that two clouds generate two individual solutions (a random, or maliciously constructed result) that will result in a mismatch. The original CRR does not have any security guarantee for this case.

In summary, copy attack dis-incentivizes honest behavior, and there will be a real possibility that the protocol accepts a random value as the correct result.

In Section 3.1, we show possible action sequences of clouds using a game like action tree, and describe the outcomes in each case. The analysis clearly shows the case (*Case 5*) that a random, or even maliciously constructed solution can be accepted, and there is no detection/prevention mechanism for this case in the system. We note that the copy attack will be attractive to a rational cloud whose utility is defined by the reward, and this will provide an opportunity for a malicious cloud to post and propagate its maliciously constructed solution in the network, in the hope of being copied and accepted.

Protection mechanism: The above discussion shows that the correctness of the result in the above setting cannot be based on having two identical *independent* solutions, and copying effectively removes independence assumption even if the two clouds are not cooperating.

We propose modifications to CRR protocol to maintain its security guarantee when it is used by a smart contract. We modify the format of a submitted solution to include the length of the execution tape,[3] and a Merkle proof (See Section 2.1) for the execution trace, and introduce a new *Result Confirmation (RC)* protocol in CRR that will be executed when the two results do match. RC protocol can efficiently detect if a solution provider has actually performed the computation using the submitted proof of execution. RC protocol consists of sending an individual randomly generated query to each cloud, that enables the smart contract to efficiently identify a cloud that has not performed the computation. The combination of the RC and MCId protocols ensures that rational clouds will (i) be incentivized to correctly perform the computation, instead of being lazy and relying on the work of the other cloud, or simply producing a random result with the hope of being copied, and (ii) be confident that their work will be rewarded (same as the original CRR). That is, RC incentivizes an honest cloud to avoid copy attack and correctly perform the computation, and MCId will ensure the original incentive of CRR will be maintained.

Generating random queries. A query is a random integer in a range that is determined by the execution claim of the cloud. Since a smart contract cannot hold a secret, one needs to use other sources of randomness that are available to the smart contract. We use an approach similar to [7] to generate a random query by applying a collision-resistant hash function to a sequence of past blocks. In Section 3.3.2, we will describe our random query generation algorithm, and in Section 3.3.1, prove that assuming the queries are random, the smart contract CRR (referred to as scCRR) (modifying messages and including the RC protocol) will restore the original correctness guarantee of CRR.

Implementation. In Section 4, we give the pseudocode of a smart contract for verifiable computation using scCRR. The scCRR contract consists of a set of functions that are shown in Figure 3, and can be called using the smart contract (through a public address). Client publishes the contract on the Ethereum blockchain and calls the *Initialize()* function to initialize the contract. Other functions and the flow of control in the contract is described in Section 4.1.

As noted in [9], adaptation of CRR for real-world execution is very delicate, and comparing two executions need determinism in execution (not affected by OS), and abilities such as executing the program from any intermediate state given the set of state variables. Their implementation uses C program and their executable code will maintain the state of computation.

We use EVM for the implementation of the smart contract. The computation of the smart contract (on the consensus computer) consists of (i) comparing the results of the two clouds, (ii) generating queries that are sent to the clouds, and verifying the response against the submitted proof (when there is matching result), and (iii) executing MCId (when results do not match). The computation of each cloud can be prepared using the same approach of [9]. However, this will require the intermediate computation results that are used by RC and MCId protocols to be "understandable" by the Ethereum Virtual Machine (EVM). For simplicity, we assume the computation package of each cloud will be in EVM bytecode and run on an EVM in the cloud. We note that this computation on EVM will not be sent to the Ethereum network (will not executed by the consensus computer). The required computation (the function and input description together with a link to the EVM bytecode and input data, and the fees and rewards) will be posted on the web, and advertised through relevant websites and social platforms. Successful registration of clouds will result in a state change of the contract that will be recorded on the blockchain. Using smart contract as a TTP will introduce delay in the interactions (clouds interactions with the smart contract). We will quantify this delay by the number of Ethereum transactions that are required for execution of CRR.

1.2 Related work

Verifying general computation has been studied in theoretical and application oriented research [4, 5, 10, 13, 14, 17, 18]. Motivation of these works are outsourcing the computation to cloud, in particular when the client is a resource bounded device. These results despite their generality and importance have limited practical application. The work of [9] is inspired by the elegant work of Feige and Kilian[12] on *refereed game* where a computationally bounded honest referee must decide on the honesty of one of the two players. Feige et al.'s computational assumptions nicely match the smart contract setting that is considered in our work: clouds have limitless

[3]CRR assumes a Turing machine to describe the computation. We use the same model to describe our theoretical results.

computation and the smart contract is bounded in its computation (each instruction of a smart contract will run over consensus computer and must be paid for through cryptocurrency). Canetti et al. follow this model and introduce *Refereed Delegation of Computation (RDoC)* and propose a refereed game (interactive protocol, CRR) that we use in our verifiable computation system using smart contracts.

Using multiple solutions by independent computational agents to provide verifiability, and assuming many rational agents who participate in the computation and are rewarded for their work, was formally studied in [6]. Authors considered a setting where a central authority, called *boss*, outsources computation to *contractors* who will be rewarded if they correctly perform the computation, and will be fined if found to have generated incorrect result. Authors assume the boss is honest and computational agents are rational, and the correctness guarantee of the system is over many rounds of the protocol executions, and contractor participations. The guarantee in this system is against rational nodes [6]. This model has been used in [16, 20] with goals such as optimizing the rewards and fines, and auditing rate that minimizes the cost of the outsourcer.

Our work follows the model of Canetti et al. and its security does not assume well-defined utility functions. Security guarantee of our protocol is against an arbitrary malicious behaviour of the cloud, and holds for *any* malicious strategy used by the corrupted cloud. The correctness guarantees in our case holds for any run of the protocol assuming one of the clouds is honest. The notion of rationality in Canetti et al. is only to say that, the guarantee that an honest cloud will be rewarded for their work and malicious clouds will be caught[4], will motivate honest behaviour.

An important assumption that is used in [6, 9, 16, 20] is that the nodes do not cooperate (collude). This is well justified when the computational nodes are cloud. This assumption however is not used in [15, 21] where computational nodes may collude, resulting in a much more complex security analysis. Distribution of a large computation to volunteer computing nodes goes back to the projects such as BOINC [3] that provide a platform for outsourcing computation to volunteers.

Paper organization. Section 2 is preliminaries. Section 3 shows how CRR can be implemented using a smart contract. In Section 3.1, we discuss the copy attack and our proposed protection mechanism. Section 4 gives pseudocode of our proposed system and estimation of the delay that is introduced by using smart contracts. Section 5 is the concluding remarks.

2 PRELIMINARIES

We first give a brief overview of the Merkle hash tree and CRR protocol [9], and then introduce the basics of the Ethereum blockchain and smart contracts over Ethereum.

2.1 Merkle hash tree (MHT)

A Merkle hash tree is a binary tree that is constructed using a *collision-resistant* hash function, on leaves that are the hashes of individual data blocks and each parent node is the hash value of the concatenation of its two children. The hash tree construction starts from the leaves and moves to the root.

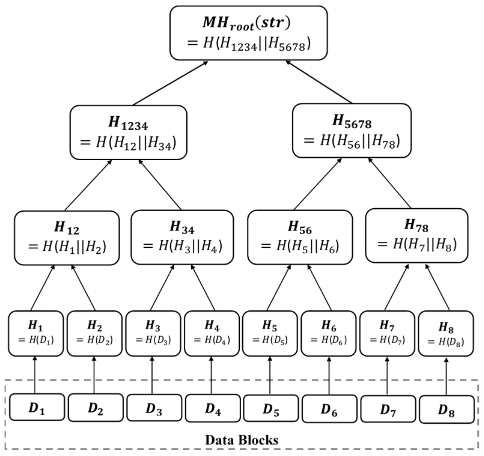


Figure 1: Merkle tree of a byte array $str[]$.

Figure 1 is an example of MHT on 8 data blocks. Data blocks $D_i = str[i]$ $(i = 1, \cdots, 8)$ are hashed to $H_i = H(D_i), i \in \{1, \cdots, 8\}$, respectively. For the second layer, the node $H_{12} = H(H_1||H_2)$ and H_{34}, H_{56} and H_{78} are defined similarly. In the third layer, $H_{1234} = H(H_{12}||H_{34})$ and $H_{5678} = H(H_{56}||H_{78})$. Finally, the root of the tree is $MH_{root}(str) = H(H_{1234}||H_{5678})$.

Given the root $MH_{root}(str)$, the Merkle proof $p_i = MH_{proof}(str, i)$ that a string is in the i-th position of str consists of the hash values of all the siblings on a path from the i-th leaf value (i.e., H_i) to the root. For example, the Merkle proof for the leaf node H_3 at index $i = 3$ is $p_3 = MH_{proof}(str, 3) = (H_3, H_4, H_{12}, H_{5678})$. In CRR [9] the proof is called *proof of consistency*.

Given a Merkle root, $root = MH_{root}(str)$, for a string str, a proof p_i, is verified using $VerifyMHProof(root, i, str, p_i)$. The function will return $True$ if the sequence of hash values in p_i, starting from the i-th element of str, will match the $root$, and $False$ otherwise.

2.2 CRR protocol

The model of [9], called Refereed Delegation of Computation (RDoC) by authors, includes a client who has an input x, and two clouds that are hired to compute $y = f(x)$. The computation is represented by a Turing Machine (TM), whose state is shown by a *reduced Turing Machine (TM) configuration*, which for an index j is given by $rc_j = (s_j, h_j, v_j, r_j)$, where s_j and h_j represent the TM's current *state* and the *head* position, respectively, v_j represents the value on the tape at the TM's head position, *tape[head]*, and $r_j = MH_{root}(t_j)$ is the *root* of the Merkle tree on the tape t_j. The reduced configuration rc_j captures the computation state s_j.
The CRR protocol works as follows:
i) The client requests the two clouds to execute the function f on the input x. Cloud $i, i \in \{1, 2\}$, returns its result $y_i = f(x)$ to the client.
ii) If the two results match the result is *accepted.*
iii) Otherwise, the client initiates the *Malicious Cloud Identification* (MCId) protocol to detect the malicious cloud. The pseudocode of

[4]They do not analyze the case that both clouds are malicious. The likelihood of a matching result in this situation is negligible, and their protocol can identify both clouds as malicious.

MCId protocol is given in Algorithm 3 in Appendix A. The protocol has two functions: (i) *binary-search* and (ii) *verify-reduced-step.*

- The *binary-search* function takes n_b, the minimum number of steps that is taken by one of the clouds to compute $f(x)$, as input, and returns the index n_g of the last reduced configuration matched for the two clouds; that is, $s^{(1)}_{n_g} = s^{(2)}_{n_g}$, $h^{(1)}_{n_g} = h^{(2)}_{n_g}$, $v^{(1)}_{n_g} = v^{(2)}_{n_g}$ and $r^{(1)}_{n_g} = r^{(2)}_{n_g}$.
- The *verify-reduced-step* function takes two consecutive reduced configurations, $rc_{n_g} = (s_{n_g}, h_{n_g}, v_{n_g}, r_{n_g})$ and $rc_{n_b} = (s_{n_b}, h_{n_b}, v_{n_b}, r_{n_b})$ where $n_b = n_g + 1$, and the proof of consistency $p_{n_g} = MH_{proof}(t_{n_g}, h_{n_g})$ for the tree index h_{n_g} from one of the clouds (say $C1$), where t_{n_g} is the tape at computation step n_g.

Then it calls $VerifyMHProof(r_{n_g}, h_{n_g}, v_{n_g}, p_{n_g})$ to recompute the Merkle root for the configuration rc_{n_g}, and performs the computation from rc_{n_g} to rc_{n_b}, and verifies the Merkle root for rc_{n_b} (invoking $VerifyMHProof(r_{n_b}, h_{n_b}, v_{n_b}, p_{n_b})$).
If the two outcomes (of $VerifyMHProof()$) match, the cloud ($C1$) is considered honest and its result is accepted. Otherwise, the other cloud ($C2$) is honest and its result is accepted.

2.3 Ethereum and Smart contracts

Ethereum is an open source, decentralized and distributed computing platform that enables users to develop smart contracts and decentralized applications, known as DApps. A smart contract is a computer protocol that facilitates, verifies, and enforces an agreement between parties without the need for a TTP. The main building blocks of Ethereum smart contract platform are, (i) peer-to-peer networking where users' computers form a network that is used for data exchange without a central server; (ii) a consensus algorithm, that is used by the blockchain users to reach a consensus about the current state of the blockchain. Ethereum uses a consensus algorithm that uses proof-of-work. (iii) Ethereum Virtual Machine (EVM), a Turing complete virtual machine that is a software that can be run at a layer of abstraction above an underlying hardware, and (iv) cryptographic tokens and addresses that enable secure transfer of assets on the blockchain. The cryptocurrency that is used for exchanges in Ethereum is called *Ether.*

There is a large number of Ethereum nodes that verify and execute computational tasks and reach agreement through the consensus protocol. In [19], this framework of scriptable cryptocurrencies is called *consensus computer* emphasizing the agreement of all the nodes in the network on the executed instructions. The consensus process introduces a delay from issuing a transaction, to its appearing on the chain. Executing programs on this consensus computer bear costs. The cost of running operations on Ethereum is measured in *gas* units, and the total gas associated with a contract can be converted to ETH. Smart contracts come with a user interface that is usually implemented as a web page, a mobile application or an application.

To deploy a smart contract, the program code is compiled into EVM bytecode and is sent within a transaction to the Ethereum network. When the transaction appears on the blockchain, a distinct address is generated by the Ethereum computing nodes which is then used to interact with the smart contract. Interaction with a contract is by calling the functions in the contract via transactions. A posted transaction is propagated in the Ethereum network until it is added to a block. The *network latency* is the elapsed time for a transaction to appear on the blockchain after it has been proposed.

3 DEPLOYING CRR

Implementing verifiable computation using CRR protocol that is managed by a smart contract can take different forms. For example, it can be implemented as a service that takes a tuple $[(x, f()), \alpha]$ from a problem giver that specifies the required computation $(x, f())$ and the reward α, manages the whole process, and returns the result to the problem giver[5]. A smart contract code is public and so the required computation $(x, f())$, can be posted on a public dashboard or web page. To focus on the new attack, we consider a basic setting where a client (problem giver) launches a smart contract that contains a *url* to a reputable web hosting site, that includes the computing task $f()$ and the input x, the CRR algorithm and the EVM bytecode, and the processes that describe the flow of cryptocurrency. The smart contract execution (run over the consensus computer) (i) recruits two clouds to perform the computation, (ii) executes CRR, and (iii) makes the required payments. The entities in the system are (i) the problem giver (PG) who writes and launches the contract on the Ethereum blockchain, (ii) two clouds, $C1$ and $C2$, who are assumed to be independent and not cooperating, and (iii) a "trusted computer" that is implemented by the Ethereum consensus computer. We assume that the probability of generating the correct result without performing the computation is negligible. To attract volunteer clouds for the computation the fees and rewards must be appropriately chosen. CRR protocol ensures that honest cloud will always be rewarded for its work. Each cloud pays a deposit δ to participate in the computation. To demotivate malicious behaviour, the deposit of an identified malicious cloud will be used towards the execution of MCId protocol. Determining the fees and rewards can take many factors. A typical case would be: PG estimates the reward α' that it should offer for the computation to a cloud by estimating the cost of computation α'_c (i.e. the required resources for performing the computation), and include an additional incentive, α'_r, to encourage the cloud's participation. PG deposits $\alpha = 2\alpha'$ in the smart contract as a fee that it will pay to the clouds for their computation (i.e., to receive correct result). Additionally, PG will pay e_{pg} as the contract deployment transaction fee. Thus, each cloud pays δ, receives α' if they correctly perform the computation, and pays for the execution of the MCId protocol, if found malicious. A malicious cloud does not have a well-defined utility and may plan its strategy irrespective of the reward.

3.1 Copy attack and protection

In the following, we use a smart contract verifiable computation system that uses CRR, and show that direct employment of CRR by the smart contract will make the system vulnerable to an attack that we refer to as *copy attack.*

Model. Our computation model consists of the following entities: (i) PG, (ii) smart contract, and (iii) two clouds, all communicating over the Ethereum network which is modeled as a synchronous network

[5]For example the contract be written to repeat the computation if a computation round fails.

with bounded delay. The two clouds act independently. We consider a rushing adversary that can corrupt one of the clouds, in which case the cloud will act maliciously. A malicious cloud has copying as one of its strategies, but may use other strategies to construct its solution. A non-corrupted cloud is rational with its utility defined by the computation reward. A rational cloud will use available strategies that ensure it receives the reward while minimizing its resource cost to perform the computation. It will avoid all strategies that will deny the reward, or even worse penalize it.

Copy attack. A smart contract cannot hold any secret and so using a smart contract as a TTP implies that the communication channels to the TTP will be public and authenticated. Network latency in Ethereum network provides a window of time that could be used by the *rushing adversary* to perform *Copy attack*. A rushing adversary will delay the message of the corrupted cloud until after it receives the message of the other cloud in the current round; this means that the round-$i + 1$ message of the corrupted cloud can depend on the round-$i + 1$ message of the other cloud. If copy attack is used by a cloud, whether malicious or rational, it will result in a match, irrespective of the correctness of the solution of the first cloud, and undermines the basis of security of the CRR which is the existence of two independent solutions. That is, although the two clouds are not cooperating, the solutions will become dependent. The attack motivates an honest (and rational) cloud to divert from honest behavior of correctly computing the function, and attempt to receive the reward without performing the computation.

3.2 Analysis

Inspired by action trees in game theoretic analysis, we show the effect of copy attack on the CRR system using a tree representation of possible scenarios. The tree shows possible sequence of events for the cloud that sends its computation result first, and the cloud that sends its result later. We consider copy attack as an attractive strategy for rational (uncorrupted) cloud. We do not consider the very risky strategy where the rational cloud randomly generates a solution and *hopes* that it will be copied by the other cloud that could be malicious.

The first cloud, say C1, (1) may be honest, compute the function and send the correct result. This is an honest cloud who has not seen the solution of the other cloud. (2) may be malicious; it generates an incorrect solution and sends it to the smart contract. The second cloud, say C2, if honest, in addition to computing the result, (3) can copy the solution of the first cloud and send it to the smart contract. Figure 2 shows the sequences of events, taking into account that if C1 is corrupted, then C2 will be honest, and its two available strategies are computing or copying.

The sequence of events and the corresponding outcomes are below.

Case 1: (*Both clouds are honest and correctly compute the function.*) The results match. The PG will receive the correct result. Each cloud receives the reward α', and its deposit δ. Each cloud has spent α'_c for its computation.

Case 2: (*$C1$ is honest and $C2$ is malicious.*) The results do not match. Smart contract will initiate the MCId protocol of [9], and will identify the malicious cloud. The result of $C1$ will be considered as the correct result and will be delivered to the PG. $C1$ will receive reward α', spends α'_c on the computation cost, and gets back its deposit δ. $C2$ will be charged ξ for the cost of executing MCId protocol and gets back its deposit (i.e., $\delta - \xi$).

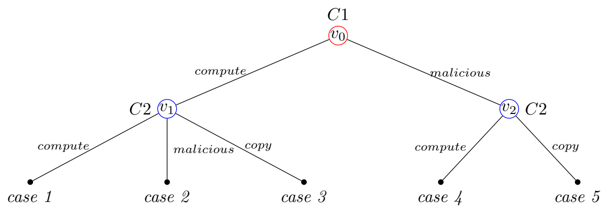


Figure 2: Action tree for two clouds.

Case 3: (*$C1$ is honest and $C2$ uses copy strategy.*) The results match. PG will receive the correct result. Each cloud receives the reward α' and its deposit δ. However, $C1$ will bear the computation cost and $C2$ will be a free-rider. Note that *copying* is a strategy that is available to both honest (rational), and malicious cloud.

Case 4: (*$C1$ is malicious and $C2$ is honest and computes the function.*) Same outcome as case 2, with the role of clouds and the fund distribution among them, reversed.

Case 5: (*$C1$ is malicious and $C2$ is honest (and rational), uses copy strategy.*) The results match. The ***PG will receive an incorrect result***. Each cloud receives reward α' and gets back the deposit δ. However, $C1$ will have computation cost ($< \alpha'_c$) whereas $C2$ is a free-rider.

The above cases show that direct implementation of CRR using smart contract fails to prevent free-riding of the second cloud, and in *Case 5* leads to an incorrect result to be accepted.

3.3 Protecting against copy attack

We modify CRR protocol to protect against copy attack when run in the smart contract managed verifiable computation. We call the modified protocol *scCRR* (smart contract using CRR).

scCRR includes an additional checking protocol, *Result Confirmation (RC)*, that is used when the results match. RC consists of a single round of interaction between the smart contract and each cloud. In the following we give the details of the protocol and prove its security guarantee.

3.3.1 Result confirmation (RC). In scCRR a cloud sends its result which is of the form, $(Res, MH_{root}(C), N)$ where, Res is the computation result, $MH_{root}(C)$ is the root of the MHT that is constructed on an array of reduced configurations C, and N is the length of the array C. Each leaf node of the MHT stores the value $H(rc_j), j \in \{1, \cdots, N\}$, where H is a collision-resistant hash function. When the computation results of the two clouds match, RC protocol is executed as follows:

(1) Smart contract generates a pair of distinct random queries $q_i, i = 1, 2$, one for each cloud, and sends it to them.
A query is of the form (i, x_i) where i is the cloud label, and x_i's are random integers in the range $[1, N - 1]$ such that

$|x_2 - x_1| > 1$ (i.e., x_2 and x_1 are non-consecutive) (see Section 3.3.2 for details on random query generation).

(2) Each cloud $i \in \{1, 2\}$, responds with a proof of consistency of the x_i-th configuration rc_{x_i} with respect to the root $MH_{root}(C)$.

(3) *Once the results are received,* smart contract does the following for each cloud i.
   (a) It verifies the proof p_{x_i} against the root $MH_{root}(C)$. If the verification succeeds, the proof is considered as valid.
   (b) If the above check fails, the computation of the cloud is rejected.

Step 3a, will detect free-riding with overwhelming probability.

Discussion. *RC* protocol can efficiently identify a cloud that uses the *copy* strategy and cannot provide the required response to the query about its execution trace. In the following, we evaluate the effect of using RC protocol in the five cases that were considered in Section 3.2.

Copy attack happens in two cases, *Case 3* and *Case 5*. *RC* protocol can always identify a copying cloud (because of the failure of a copier to provide correct response to the query by the smart contract, see Theorem 3.1). The effect of this for an honest (rational) cloud is that it will remove copying from its available strategies (because it knows that it will be caught), and so will always compute correctly. A corrupted cloud however may still consider copying as a valid strategy. This case again will be detected by RC. This means that RC effectively removes *Cases 3* and *5*. The remaining cases are *Case 1*, where a correct result will be generated, and *Cases 2* and *4*, where there are two non-matching results, and MCId will be used to identify the honest cloud. Theorem 3.1 gives security of the RC protocol.

Algorithm 1 Result confirmation.

```
1:  function resultConfirmation
2:      $(q_1, q_2)$=$QueryGen(\kappa, d, N, idx)$;      ▷ generates two random queries $q_i = (i, x_i)$
3:      Send $x_i$ to cloud $i$;      ▷ Cloud $i$ responds by sending the proof $p_{x_i}$
4:      bool $st_i$, $i = \{1, 2\}$
5:      for each cloud $i$ do
6:          if (VerifyMHProof ($MH_{root}(RC)$, $x_i$, $rc_{x_i}$, $p_{x_i}$)==$True$) then
7:              $st_i$ = True;      ▷ Cloud $i$ is honest
8:          else
9:              $st_i$ = False      ▷ Cloud $i$ is a free-rider
10:         end if
11:     end for
12:     return ($st_i$, $i = \{1, 2\}$)
13: end function
```

Theorem 3.1. *Consider the copy attack in Section 3 and RC protocol in Section 3.3.1, and let H be a collision resistant hash function that is used to construct the Merkle hash tree on the array of reduced configurations, C. Then RC protocol provide protection against copy attack of honest cloud.*

Proof. (Sketch) Consider a copy attack where cloud C1 has produced and published the result (Res, MH_{root}, N), and cloud C2 has copied the result. C1 will receive query x_1 and C2 will receive query x_2, where $|x_2 - x_1| > 1$. Each cloud must provide the proof of consistency for the corresponding query rc_{x_1} and rc_{x_2}, with respect to the root MH_{root}. Let C1 generate a proof $\pi_1 = (p_1^{x_1}, p_2^{x_1}, ..., p_{log(N)}^{x_1})$.

First consider the case that C1 is the honest cloud, then $VerifyMHProof(MH_{root}, x_1, C, \pi_1) = True$. C2 can copy any part of this proof. However it needs to generate a proof, $\pi_2 = (p_1^{x_2}, p_2^{x_2}, \cdots, p_{log(N)}^{x_2})$. Each leaf of the MHT tree will have a path of length $\log N$ to the root, and because the two leaves are not consecutive, they will not have a common parent, and so their path to the root will include at least one hash value that cannot be calculated from published path. For example, in Figure 1, the query $x_1 = 1$ corresponds to the leaf that is hashed to H_1 and has the path (H_2, H_{34}, H_{5678}). On the other hand the path for the leaf $x_2 = 3$ is (H_4, H_{12}, H_{5678}). Thus, there is at least one j for which $p_j^{i_1} \neq p_j^{i_2}$. To succeed, C2 must find $p_j^{i_2}$ that satisfies $VerifyMHProof(MH_{root}, x_2, C, \pi_2) = True$, and so its success probability is bounded by the collision probability of the hash family.

Next consider the case that C1 is malicious and so $VerifyMHProof(MH_{root}, x_1, C, \pi_1) = False$. The success probability of the copying cloud C2, will be even lower as the published path of C1 does not provide useful information. □

In [9], a *Refereed Delegation of Computation (RDoC)* protocol is defined as follows.

Definition 3.2. $(P_1, P_2 \cdots P_N, R)$ is an ϵ-RDoC with N servers for a function f if the following holds:
—For any input x and for any $i \in [1, N]$, if P_i is honest, then for any $(P_1^*, P_2^* \cdots P_{i-1}^*, P_{i+1}^*, \cdots P_N^*)$ the output of R is $f(x)$ with probability at least $1 - \epsilon$.
—The complexity of the client is at most quasi-linear in $|x|$ and the complexity of the (honest) servers is polynomial in the complexity of evaluating $f()$.

Using Theorem 3.1 and Theorem 1 in [9], we have the following.

Corollary 3.3. *Assuming the hash function is collision resistant, then scCRR protocol is a computationally sound, full information RDoC with two servers and with negligible soundness ϵ for any function computable in polynomial time.*

3.3.2 Random query generation.

Let each block in the blockchain have an index that shows its position in the chain; for example the index of the first block (genesis block) is 0, the next block after that is 1 and so on. The smart contract must generate two query indexes x_1 and x_2 that are unpredictable (except with negligible chance of random guess) and satisfy $|x_2 - x_1| > 1$. We use an approach in [7] that uses a subset of blocks on the blockchain to extract a random string. We generate an index x_1 to be random, and then choose x_2 to be a shift of x_1. For the query to C1, we use a collision-resistant hash function on the set of consecutive published blocks on the blockchain in the range $[idx - 2\kappa, idx - \kappa]$, where idx is the block index where the results of clouds are published (if there are two such blocks we consider the smaller index). The query index x_1 is calculated as $H(H(B^{idx-2\kappa}), .., H(B^{idx-\kappa})) \pmod{N-1}$. The second query is simply $x_2 = x_1 + d \pmod{N-1}$ where d is a random value chosen by PG in the range $[2, N - 3]$. Here κ is the security parameter of the blockchain (see below). The two queries will be (i) distinct and because of the choice of d will not be consecutive,

(ii) each will be individually random and unpredictable by the corresponding cloud, and (iii) by using sufficiently large κ, they will not be biased by the malicious parties. Algorithm 2 shows the steps for generating the random queries. The value of κ, among other things, will be related to the underlying consensus algorithm, and will be chosen such that (i) in every κ consecutive blocks there is at least one block that is guaranteed to be honestly generated, and (ii) by removing the last κ blocks, the blockchain will be consistent among the majority of the computing nodes. Condition (i) ensures that at least one of the hashes $\{H(B^{idx-2\kappa}), .., H(B^{idx-\kappa})\}$ is unbiased by the malicious cloud. Condition (ii) guarantees that the randomness is extracted from the stable part of the blockchain and so the majority of the nodes will see the blocks that are used for randomness generation, and can generate and verify the queries (for details see Section 1.1 in [7]).

Algorithm 2 Random query generation

```
1: function QueryGen($\kappa$, $d$, $N$, $idx$)
2:     Compute $x_1 = H(H(B^{idx-2\kappa}), .., H(B^{idx-\kappa})) \pmod{N-1}$;
3:     Compute $x_2 = x_1 + d \pmod{N-1}, 2 \le d \le N-3$;
4:     Return $((i, x_i), i = \{1, 2\})$.
5: end function
```

3.4 Modifying CRR to be used in a smart contract

Using CRR within a smart contract needs the following modifications. We refer to the resulting protocol as *scCRR*.

scCRR Protocol:

(i) The *first message* in scCRR is $(Res = f(x), MH_{root}(C), N)$ as described in Section 3.3.1. (The corresponding message in CRR is $Res = f(x)$.)

(ii) If the results match, RC protocol is used. The protocol selects two distinct random queries, sends one to each cloud, and receives the response. The response is accepted if it is consistent with $MH_{root}(C)$, and rejected otherwise. The algorithm for generating random queries is given in Section 3.3.2.

(ii) If the results do not match, the MCId protocol in CRR and its associated decision will be used.

4 IMPLEMENTATION

In the following section we describe (i) our design of the smart contract, and (ii) its deployment and execution.

4.1 Smart contract construction and interactions

The smart contract in Figure 3 is written by the PG, and consists of a set of functions that are available for interaction with the contract, and their corresponding inputs and outputs. PG deploys and initializes the smart contract but cannot update or interact with it until receives the final result. The contract will be killed at that stage, and re-deployed if needed. Restricting PG's interaction with the smart contract after its initialization, ensures that PG cannot subvert the contract.

```
pragma solidity >=0.4.0 <0.6.0;
contract scCRR {
constructor () public;
function Initialize (uint256 _task_url, uint256 _web_hash, uint256 _comp_hash, uint _reward, uint min_deposit) public onlyOwner;
function Register (address _sender, uint _amount) public payable;
function receiveResults (uint256 _result, uint256 _root, uint tape_length) public;
function Compare (uint256 _result1, uint256 root1, uint256 _result2, uint256 root2) internal;
function resultConfirmation () internal returns (bool, bool);
function QueryGen (uint256 _k, uint256 d, uint256 N, uint256 idx) internal returns (uint, uint);
function binary-search (uint min_steps) internal returns (uint);
function verify-reduced-step (uint256 rc_ng, uint256 rc_nb, uint256 p_ng) internal returns (bool);
function Pay (uint _case) internal;
function shutDown() internal;
}
```

Figure 3: Abstract *scCRR* smart contract.

The pseudocodes of the functions are given in Figures 4 to 8. In the following we give a brief description of each function.

(1) *Constructor().* A smart contract code is deployed on the Ethereum blockchain once its constructor function (declared with the keyword *constructor*) is executed. The function assigns the smart contract's variables, if any, and sets the problem-giver as the creator of the smart contract ($PG = msg.sender$). Here, the $msg.sender$ refers to the address of the PG.

(2) *Initialize().* PG first calls the *initialize* function to initiate the smart contract execution. For this, PG sends a transaction to the smart contract address specifying the following.
(i) A *url* that provides information about the computation, and a link to the computation EVM bytecode f_{byte} and the inputs x;
(ii) The hash values $h_c = H(f_{byte}, x)$, and h_w of the webpage (pointed by the *url*). h_w and h_c, are used by the clouds to check the integrity of the web page and the computation, respectively.
(iii) The values that are required for the execution of the contract including reward α and minimum deposit δ for registration, will be stored in the contract storage. *Initialize()* is annotated with the keyword *onlyOwner* to ensure that only PG can call the function.

After successful initialization, an event *taskInitialized* is emitted to the blockchain. This event will allow the clouds to observe the required computation, and be able to express their interest by registration.

(3) *Register().* A cloud who is interested in participating in the computation sends a transaction to the smart contract address calling *Register ()*, and provides its address and the minimum deposit amount. This function is annotated with the keyword *payable* to be able to receive deposit from the clouds. The contract adds the deposits to its balance and saves the clouds addresses in the array *Cloud*[]. When two clouds are registered, the *state* variable is set

```
function INITIALIZE( task,$h_w$, $h_c$,$\alpha$,$\delta$) public onlyOwner
    assert ($msg.sender == PG$);
    //initialize contract parameters
    set $index$, $count$, $address(this).balance \leftarrow 0$;
    set $task\_url$ = $task$;
    set $web\_hash$ = $h_w$;
    set $comp\_hash$ = $h_c$;
    set address(this).balance = address(this).balance + $\alpha$;
    set $depositRequire = \delta$; $reward = \alpha/2$;
    emit $taskInitialized$(task_url, web_hash,comp_hash, $reward$, $depositRequire$);
end function
```

Figure 4: Pseudocode of *Initialize* function.

to *Compute*, and the event *registerComplete* is issued to reflect the state and indicate that the clouds have been registered successfully. At this stage, each cloud will use the *task_url* to download the EVM bytecode of the computation and the *input* data, verify their integrity by comparing the corresponding hash value with *comp_hash*, and initiates the execution.

```
function REGISTER(sender_address, amount) public payable
    if $count < 2$ then
        set $deposit[index]$ = amount;
        assert ($deposit == depositRequire$);
        set address(this).balance = address(this).balance + $deposit[index]$;
        set Cloud[index] = msg.sender;
        set $index+ = 1$;
    end if
    update $count+ = 1$;
    assert(count < 2); ▷ Checking whether two clouds registered or not
    set state = Compute;
    emit registerComplete($state$);
end function
```

Figure 5: Cloud Registration.

(4) *receiveResults().* Each cloud performs the computation in its local machine, and delivers the result (by sending a transaction) using the *receiveResults().* The transaction includes ($Res = f(x), MH_{root}(C), N$) parameters as specified in Section 3.3.1. The contract saves the results and the root values in the arrays *Result*[] and *MRoot*[], respectively. After receiving the results, the smart contract invokes the internal function *Compare()* to check the results.

(5) *Compare().* The compare function initiates *RC protocol* (Algorithm 1) if both the clouds' results match the root. Otherwise, it initiates *MCId protocol* (Algorithm 3).

When both results match, *Compare()* invokes *resultConfirmation()* that returns a pair of Boolean values ($true, true$), ($true, false$), and ($false, false$) representing the *Cases* 1, 3, and 5 in Section 3.2, respectively, and assigns it to the variable *case*.

```
function RECEIVERESULTS($r$, $h_{root}$, $N$) public
    assert ($msg.sender == Cloud[0]$ || $msg.sender == Cloud[1]$);
    if idx==0 then
        set idx = block.number; ▷ idx: stores block number of the first received result
    end if
    if $msg.sender == Cloud[0]$ then
        set Result[0] = $r$; MRoot[0] = $h_{root}$; Tape_length[0] = $N$;
        ▷ (Result[], MRoot[], Tape_length[]: array to store result, root, tape length respectively)
    else
        set Result[1] = $r$; MRoot[1] = $h_{root}$; Tape_length[1] = $N$;
    end if
end function
```

Figure 6: Receive results from clouds.

If the results are not matching, *binary-search()* is invoked to find the last matched configuration index. Then, one of the two clouds is asked to supply two consecutive configurations, starting from the matched index, and a proof of consistency. This will be verified using the internal function *verify-reduced-step.* This verification step results in, ($true, false$) or ($false, true$), representing the *Cases* 2 and 4 as given in Section 3.2, respectively. This will be assigned to the variable *case*. After executing *RC* (or the *MCId*) protocol, the *Pay()* function is called internally for *case* value to correctly distribute and/or penalize the clouds.

(6) *QueryGen.* This function takes four integers, i.e. the security parameter κ, length of C denoted by N, a random value $d \in [2, N-3]$, and the index *idx* of the block that contains the first published result. The *idx* value is obtained by *block.number.* Finally, *QueryGen*() returns two random queries of the form $q_i = (i, x_i), i = 1, 2$. (See Section 3.3.2 for more details and Appendix B for Solidity code.)

(7) *Pay().* This function distributes the rewards to the clouds according to the cases described in Section 3.2, returns the (remaining) deposits, and sends the (correct) result to the PG.

(8) *shutDown().* At the end of the computation the *shutDown()* function is called. Once this function is called, the contract will no longer be executable. If the computation was unsuccessful, the PG may repeat the above process, re-deploying the smart contract and re-outsourcing the computation.

This is a basic contract and it can be extended to include time-out to specify the execution time.

4.2 Deployment and execution

Deployment. The smart contract is written in Solidity which is a high level language. The code will be compiled into low-level machine instructions, *op-codes*, which are used by the EVM. Op-codes are then encoded into bytecode, which will be deployed (by the PG) on the Ethereum network using a specific transaction called *contract creation transaction*, with the following structure:
Once the contract creation transaction is published on the blockchain, the *To* field in the transaction will be filled with the *contract address* created by the network. The *contract address* is deterministically

```
function Pay(case) internal
    if (case=1) then                                    ▷ Both honest
        send Result[0] to PG;
        pay $\alpha' + \delta$ to $C1$;
        pay $\alpha' + \delta$ to $C2$;
    end if
    if (case=2||case=3) then                ▷ $C1$ honest, $C2$ malicious
        send Result[0] to PG; return $\alpha'$ to PG;
        pay $\alpha' + \delta$ to C1;
        return $\delta - \xi$ to $C2$;
    end if
    if (case=5) then                                    ▷ Both malicious
        return $\alpha = 2\alpha'$ to PG;
        return $\delta - \frac{\xi}{2}$ to $C1$;
        return $\delta - \frac{\xi}{2}$ to $C2$;
    else                                    ▷ $C1$ malicious, $C2$ honest
        send Result[1] to PG; return $\alpha'$ to PG;
        return $(\delta - \xi)$ to $C1$;
        pay $\alpha' + \delta$ to $C2$;
    end if
end function
```

Figure 7: Payments to clouds.

```
function shutDown () internal
    call selfdestruct();
end function
```

Figure 8: Destroying the contract.

From	Address of PG (an EOA)
To	Empty
Value	Amount (in Ether)
Gas Limit	Amount of Gas
Gas Price	Amount (in Gwei)
Nonce	Number of transaction sent by PG
Data/Input	608060405260006008 · · · (contract creation bytecode)

Figure 9: Sample transaction to deploy the smart contract.

computed from the PG's address and the nonce value that are in the transaction. This completes the deployment of the smart contract.

Interaction with the deployed smart contract. In *scCRR*, functions with the keyword *public* can only be called by PG or the registered clouds, and functions with the keyword *internal* can be invoked by the contract only. To call a public function, the calling party sends a transaction to the contract address, specifying the *function id* and the input arguments in the data field. The first 4 bytes of the hash value of a function signature are considered as the *function id*. The structure of a transaction that calls a smart contract function is given in Figure 10.

Execution. Each participating cloud will install an Ethereum client (e.g., Geth) that connects to the Ethereum network. This client

From	Address of Cloud 1 (an EOA)
To	scCRR contract address
Value	Minimum_deposit_amount (in Ether)
Gas Limit	Amount of Gas
Gas Price	Amount (in Gwei)
Nonce	Number of transaction sent by Cloud 1
Data/Input	$3de3a6fac\cdots$ (Function signature: Register(address _sender, uint _amount))

Figure 10: Transaction to execute *Register* function.

allows the cloud to work as an Ethereum node having its own EVM, a stateful machine that runs assembly instructions (op-codes).

The contract code is stored in a virtual ROM of the EVM [22]. During execution, each EVM maintains (i) a *stack* to store the outputs of the intermittent execution of op-codes, (ii) a volatile *memory* to store temporary data such as function arguments, local variables and return values, and (iii) a *program counter* to point to the current instruction (i.e., op-code). At the start of execution, the stack and the memory are empty and the program counter is set to zero by the EVM. Each time an instruction is executed, the stack and the memory are updated accordingly, and the program counter is incremented.

State and Reduced Configuration. We set the execution state of a program in EVM to be the tuple (*stack contents*, *memory contents*, *program counter*), and assume that each cloud has a database to store all the intermediate execution states. The reduced configuration for a specific state is defined as the root hash of the Merkle tree that is built on top of that state. The reduced configurations are encoded as EVM bytecodes, and are sent for *binary-search* when the results mismatch.

After the binary search phase, one of the clouds sends two consecutive full states, that is, stack, memory contents, and program counter, to the Ethereum network for verification. Since the smart contract execution in Ethereum is fully deterministic, it produces an identical state transition for any conforming implementation beginning in an identical state [22]. In this way, Ethereum network can verify the correctness of the computation.

4.3 Delay analysis

Using a smart contract will introduce delay in the execution of scCRR because communications to and from the smart contract uses the Ethereum network. Each message is sent to the smart contract in the form of a transaction, which takes time to be included in a block; the block must be broadcasted to the network, mined and after reaching consensus, appended to the blockchain. The required time for these steps depend on the blockchain and can be estimated from published statistics of the network. In Ethereum, the delay between sending a transaction and its inclusion in a block depends on the gas price for that transaction [1] and can be determined by the PG when creating the smart contract.

In the following, we discuss the number of transactions that will be sent and received between the clouds and the smart contract for a given computation; note that we do not differentiate the transactions that smart contract sends from the cloud transactions. For implementation convenience, we assume that each cloud has

a smart contract deployed on the blockchain for sending and receiving messages. This is required, because in Ethereum a smart contract can send a message only to another smart contract.
–For registration, each cloud sends a transaction to pay the deposit. Smart contract issues an event, i.e. *registerComplete*, to acknowledge the completion of the registration. This requires two transactions in total.
– Once the task is executed, each cloud sends its results as a transaction to the smart contract (i.e., two transactions in total).
– If the results are same, the smart contract calls the result confirmation function which in turn issues two queries to the clouds. In response, the clouds provide proof of consistency with respect to $MRoot[i]$ ($i \in 1, 2$) as a transaction. Therefore, there are four transactions involved in this phase.
– If the results do not match, the clouds engage in the MCId protocol execution. First, they will run a *binary-search* which could take up to $2log(N)$ transactions. At each step, scCRR sends the index of the next step that the clouds should respond. This needs two transactions and so, in total, up to $4log(N)$ transactions will be needed. Second, they will engage in $verify\text{-}reduced\text{-}step$ function; one of the clouds provides two consecutive configurations and also the proof of consistency, requiring three transactions in total.

Table 1 shows the number of transactions required in different phases of the smart contract execution.

Table 1: Delay of the scCRR in terms of the number of transactions.

Phase	Register	receiveResult	RC	MCId
# **Transactions**	2	2	4	$4log(N) + 3$

5 CONCLUSIONS

Smart contracts could be used as a TTP to restructure many information communication systems that are in use today. Using a smart contract to construct a verifiable computation system based on CRR provides an attractive solution to the problem of verifiable outsourced (general) computation. Running CRR by a trusted third party will not affect the security of the CRR if the channel from cloud to the TTP can be made confidential, for example by using a public key encryption algorithm. Using a smart contract as a TTP however removes this possibility and opens the system to a new attack that we studied here.

Our work raises a number of interesting research questions, including using the smart contract as a TTP in other verifiable computation systems, and relaxing the requirement that the cloud must run the computation in an EVM. These are our directions for future work.

ACKNOWLEDGMENTS

This work is in part supported by the Industrial Research Chair Grant program of Natural Sciences and Engineering Research Council of Canada, and supported by Telus Communications.

REFERENCES

[1] 2019. ETH gas station. https://ethgasstation.info/ Accessed on August 19, 2019.
[2] 2019. RSK. https://www.rsk.co/solutions/ Accessed on August 19, 2019.
[3] David P Anderson. 2004. Boinc: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 4–10.
[4] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. 2010. From secrecy to soundness: Efficient verification via secure computation. In *International Colloquium on Automata, Languages, and Programming*. Springer, 152–163.
[5] László Babai. 1985. Trading group theory for randomness. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. ACM, 421–429.
[6] Mira Belenkiy, Melissa Chase, C Chris Erway, John Jannotti, Alptekin Küpçü, and Anna Lysyanskaya. 2008. Incentivizing outsourced computation. In *Proceedings of the 3rd international workshop on Economics of networked systems*. ACM, 85–90.
[7] Iddo Bentov, Rafael Pass, and Elaine Shi. 2016. Snow White: Provably Secure Proofs of Stake. *IACR Cryptology ePrint Archive* 2016 (2016), 919.
[8] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).
[9] Ran Canetti, Ben Riva, and Guy N Rothblum. 2011. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 445–454.
[10] Kai-Min Chung, Yael Kalai, and Salil Vadhan. 2010. Improved delegation of computation using fully homomorphic encryption. In *Annual Cryptology Conference*. Springer, 483–501.
[11] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. 2017. Betrayal, distrust, and rationality. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 211–227.
[12] Uriel Feige and Joe Kilian. 1997. Making games short. In *STOC*, Vol. 97. 506–516.
[13] Rosario Gennaro, Craig Gentry, and Bryan Parno. 2010. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Annual Cryptology Conference*. Springer, 465–482.
[14] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The knowledge complexity of interactive proof systems. *SIAM Journal on computing* 18, 1 (1989), 186–208.
[15] Dominik Harz and Magnus Boman. 2018. The Scalability of Trustless Trust. In *International Conference on Financial Cryptography and Data Security*. Springer, 279–293.
[16] MHR Khouzani, Viet Pham, and Carlos Cid. 2014. Incentive engineering for outsourced computation in the face of collusion. In *Proceedings of WEIS*.
[17] Joe Kilian. 1992. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. ACM, 723–732.
[18] Joe Kilian. 1995. Improved efficient arguments. In *Annual International Cryptology Conference*. Springer, 311–324.
[19] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. 2015. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 706–719.
[20] Viet Pham, MHR Khouzani, and Carlos Cid. 2014. Optimal contracts for outsourced computation. In *International Conference on Decision and Game Theory for Security*. Springer, 79–98.
[21] Jason Teutsch and Christian Reitwießner. 2017. A scalable verification solution for blockchains. *url: https://people. cs. uchicago. edu/teutsch/papers/truebit pdf* (2017).
[22] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

A MCID PROTOCOL

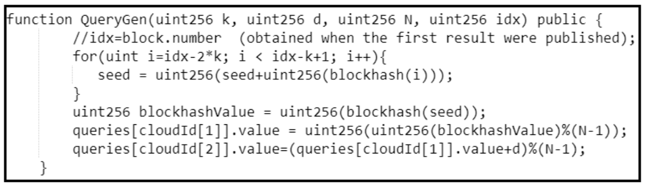
Pseudocode for malicious cloud identification (MCId) protocol of CRR [9] is given in Algorithm 3.

Algorithm 3 Malicious cloud identification protocol.

```
1: Set $n_b = min\ (numOfsteps\_of\_C1,\ numOfsteps\_of\_C2)$;
2: Set $n_g = binary - search(n_b)$;   ▷ index of last matched configuration
3: function BINARY-SEARCH($n_b$) returns ($int$)
4:     Set $idx_s = 0$, $idx_e = n_b$;   ▷ $idx_s$=starting index where clouds agree, $idx_e$=last index where clouds disagree
5:     Set $mid = \frac{idx_e - idx_s}{2} + idx_s$;
6:     while $idx_e > idx_s + 1$ do
7:         PG asks $C1$ to send its reduced-configurations $rc^{(1)}_{mid}$ at index $mid$.
8:         PG asks $C2$ to send its reduced-configurations $rc^{(2)}_{mid}$ at index $mid$.
9:         if ($rc^{(1)}_{mid}$ and $rc^{(2)}_{mid}$ are valid) then   ▷ having same tape $t_{mid}$
10:            if $rc^{(1)}_{mid} = rc^{(2)}_{mid}$ then
11:                ▷ mismatch is at the second half $(mid, idx_e]$
12:                set $idx_s = mid$;
13:                update $mid = \frac{idx_e - idx_s}{2} + idx_s$;
14:            else   ▷ mismatch is at the first half $[idx_s, mid]$.
15:                set $idx_e = mid$;
16:                update $mid = \frac{idx_e - idx_s}{2} + idx_s$;
17:            end if
18:        end if
19:    end while
20:    return $idx_s$;
21: end function
22: PG asks $C1$ for configurations $rc_{n_g}$ and $rc_{n_b}$, $n_b = n_g + 1$;
23: PG asks $C1$ for proof of consistency $p_{n_g} = MH_{proof}(t_{n_g}, h_{n_g})$;   ▷ $t_{n_g}$ is the tape of configuration $rc_{n_g}$
24: cloudHonest = verify-reduced-step($rc_{n_g}$, $rc_{n_b}$, $p_1$, $p_2$);
25: function VERIFY-REDUCED-STEP($rc_{n_g}$, $rc_{n_b}$, $p_{n_g}$) returns ($bool$)
26:     if (VerifyMHProof($r_{n_g}$, $h_{n_g}$, $v_{n_g}$, $p_{n_g}$) == True) then
27:         Simulate one step of TM on ($s_{n_g}$, $h_{n_g}$, $v_{n_g}$) and get new ($s'$, $h'$, $v'$);
28:         if ($s' = s_{n_b}$ && $h' = h_{n_b}$) then
29:             Compute $r' = MH_{root}(t_{[h_{n_g}]=v'})$;   ▷ using $p_{n_g}$ and $v'$
30:             if $r' = r_{n_b}$ then
31:                 return True;
32:             else
33:                 return False;
34:             end if
35:         end if
36:     end if
37: end function
```

B IMPLEMENTATION OF RANDOM QUERY GENERATION FUNCTION

Implementation of the random query generation function written in Solidity language is given in Figure 11.



```
function QueryGen(uint256 k, uint256 d, uint256 N, uint256 idx) public {
        //idx=block.number  (obtained when the first result were published);
        for(uint i=idx-2*k; i < idx-k+1; i++){
           seed = uint256(seed+uint256(blockhash(i)));
        }
        uint256 blockhashValue = uint256(blockhash(seed));
        queries[cloudId[1]].value = uint256(uint256(blockhashValue)%(N-1));
        queries[cloudId[2]].value=(queries[cloudId[1]].value+d)%(N-1);
    }
```

Figure 11: Random query generator.