

# Recurring Contingent Payment for Proofs of Retrievability

Aydin Abadi<sup>\*1</sup> Steven J. Murdoch<sup>\*\*1</sup> Thomas Zacharias<sup>\*\*\*2</sup>

<sup>1</sup> University College London

<sup>2</sup> University of Edinburgh

**Abstract.** Fair exchange protocols let two mutually distrusted parties exchange digital data in a way that neither can cheat. At CCS 2017, Campanelli *et al.* proposed two blockchain-based protocols for the fair exchange of digital coins and a certain service, i.e., “proofs of retrievability” (PoR), that take place between a buyer and seller. In this work, we identify two serious issues of these schemes; namely, (1) a malicious client can *waste the seller’s resources*, and (2) *real-time leakage* of information to non-participants in the exchange. To rectify the issues, we propose “*recurring contingent PoR payment*” (RC-PoR-P). It lets the fair exchange *reoccur* while ensuring that the seller’s resources are not wasted, and the parties’ privacy is preserved. We implemented the RC-PoR-P. Our cost analysis indicates that the RC-PoR-P is efficient. The RC-PoR-P is the first of its kind that offers all the above features.

## 1 Introduction

Fair exchange protocols let two mutually distrusted parties swap digital items such that either each party gets the other’s item, or neither party does. It captures various scenarios; for instance, when two parties want to exchange digital files or to exchange digital items and coins. Fair exchange protocols have been extensively studied for decades. It was shown that fairness is unachievable without a trusted third party’s aid [17]. With the advent of blockchain technology, it seemed fair exchange protocols can be designed without having to rely on a single trusted third party. Hence, various blockchain-based fair exchange protocols have been proposed. At CCS 2017, Campanelli *et al.* [15] proposed two schemes to facilitate the fair exchange of digital coins and a digital *service* (between a buyer and seller), on the blockchain. These two schemes stand out from the rest, as they are the only ones designed so far to support buying a service, through exchanging coins for proof that a service has been provided, rather than just buying a static item, such as a file.

Nevertheless, as we will show in this work, these two schemes in [15] suffer from two serious issues: (1) a malicious client can *waste the seller’s resources*

---

\* aydin.abadi@ucl.ac.uk

\*\* s.murdoch@ucl.ac.uk

\*\*\* thomas.zacharias@ed.ac.uk

and (2) *real-time leakage* of information to non-participants of the exchange (e.g., proofs’ status, deposit’s actual amount, buyer’s file size, or even the file’s location in some situations). When combined, these vulnerabilities allow a malicious client to discover, with high probability, whether the service has been provided, without making a payment. We identify two flaws in these schemes that led to Issue 1; namely, (a) incomplete fairness’ definition and (b) mismatch of security assumption/requirement between primitives. Also, we identify the improper use of public blockchain as the primary factor that led to Issue 2. The schemes’ flaws that lead to these issues are generic. If they are not dealt with appropriately, then future (blockchain-based) fair exchange protocols may inherit them.

**Our Contributions.** In this work, we:

1. identify two serious issues that the protocols of Campanelli *et al.* [15] suffer from, (1) *waste of the seller’s resources*, and (2) *real-time leakage*. We identify the protocols’ flaws causing these issues. Section 4 explains the issues.
2. formally define and propose a construction called “*recurring contingent PoR payment*” (RC-PoR-P) that addresses the above issues. The RC-PoR-P relies mainly on symmetric-key primitives and smart contracts. It is the first fair exchange scheme that offers the above features simultaneously. Section 6 presents the RC-PoR-P.
3. implement the RC-PoR-P and make its source code public. Our cost analysis illustrates that the RC-PoR-P is efficient. When it deals with a 4 GB file, a verifier can check a proof in 90 milliseconds, and a dispute between the prover and verifier can be resolved in 0.1 milliseconds. The contracts’ computation cost is also low, i.e.,  $O(1)$ . Section 7 evaluates the RC-PoR-P.

## 2 Related Work

In this section, we summarise related work (see Appendix A for a more detailed survey). Maxwell [38] proposes a fair exchange scheme, called “zero-knowledge contingent payment” that supports the fair exchange of digital goods and coins. It is based on Bitcoin’s smart contracts, a hash function, and zero-knowledge (zk) proofs. After the advancement of the “succinct non-interactive argument of knowledge” (zk-SNARK) [27] that yields more efficient zk proofs, the scheme was modified to use zk-SNARKs. Later, Campanelli *et al.* [15] identified an issue in the above scheme. The issue lets a malicious buyer receive the item without paying. To address it, the authors propose the “zero-knowledge Contingent Service Payments” (zkCSP) scheme that also supports contingent payment for digital services. It is based on Bitcoin smart contracts, hash functions, and witness indistinguishable proof of knowledge (WIPoK). To improve the efficiency, they use zk-SNARKs where the buyer generates a public parameter, i.e., CRS, and the seller performs minimal checks on the CRS. The authors, as the zkCSP’s concrete instantiations, propose public and private verifiable schemes where the service is “proofs of retrievability” (PoR) [47]. To date, they are the only ones designed for the fair exchange of digital coins and a digital service. Shortly, we will explain their shortcomings undetected in the literature.

Fuchsbauer [23] identifies a flaw in the zkCSP and shows that the seller’s minimal check in the zkCSP does not prevent the buyer from successfully cheating. Later, Nguyen *et al.* [44] show that by relying on a stronger notion of WI, the zkCSP remains secure. Tramer *et al.* [48] propose a fair exchange scheme that uses trusted hardware and Ethereum smart contracts. Dziembowski *et al.* [21] propose FairSwap, a fair exchange scheme using the Ethereum smart contracts and the notion of proof of misbehaviour [16]. Later, Eckey *et al.* [22] propose OPTISWAP that improves FairSwap’s performance. Similar to FairSwap, OPTISWAP uses a smart contract and proof of misbehaviour, but it relies on an *interactive* dispute resolution protocol. Recently, outsourced fair PoRs letting a client delegate the verifications to a smart contract were proposed in [3,20]. The scheme in [3] uses message authentication codes (MACs) and time-lock puzzles. The one in [20] uses polynomial commitment and involves a high number of exponentiations. As a result, it imposes higher costs, of proving and verifying, than the former scheme. The schemes in [3,20] rely on a stronger security assumption (i.e., the client is fully honest) than the rest of the above work.

### 3 Preliminaries

#### 3.1 Notation and Assumption

We use  $\lambda$  and  $\perp$  to denote security parameter and null value respectively. We use  $\mathcal{C}$ ,  $\mathcal{S}$ , and  $\mathcal{A}$  to denote the client, server, and arbiter. We let  $pl$  be  $\mathcal{S}$ ’s public price list,  $o$  be the amount paid to  $\mathcal{S}$  for each valid proof, and  $l$  be the amount (misbehaving)  $\mathcal{C}$  or  $\mathcal{S}$  pays to  $\mathcal{A}$  for resolving a dispute for each verification,  $o_{max}$  be the maximum amount paid to  $\mathcal{S}$  for a valid proof,  $l_{max}$  be the maximum amount to resolve a potential dispute, and  $z$  be the total number of verifications and  $(o, l, o_{max}, l_{max}) \in pl$ . We provide a notation table in Appendix B. Similar to the *optimistic* fair schemes that aim at efficiency, e.g., in [7,8,10,18], we assume a trusted third-party arbiter’s existence (e.g., smart contract or secure hardware) which mostly remains offline, and is invoked only to resolve disputes.

#### 3.2 Pseudorandom Function

Informally, a pseudorandom function (PRF) is a deterministic function that takes as input a key and some argument and outputs a value indistinguishable from that of a truly random function with the same domain and range. Pseudorandom functions have many applications in cryptography as they provide an efficient and deterministic way to turn input into a value that looks random. Below, we restate the formal definition of PRF, taken from [35].

**Definition 1.** *Let  $W : \{0, 1\}^\psi \times \{0, 1\}^n \rightarrow \{0, 1\}^t$  be an efficient keyed function. It is said  $W$  is a pseudorandom function if for all probabilistic polynomial-time distinguishers  $B$ , there is a negligible function,  $\mu(\cdot)$ , such that:*

$$\left| \Pr[B^{W_{\tilde{k}}(\cdot)}(1^\psi) = 1] - \Pr[B^{\omega(\cdot)}(1^\psi) = 1] \right| \leq \mu(\psi)$$

where the key,  $\hat{k} \xleftarrow{\$} \{0,1\}^\psi$ , is chosen uniformly at random and  $\omega$  is chosen uniformly at random from the set of functions mapping  $\eta$ -bit strings to  $\iota$ -bit strings.

### 3.3 Smart Contract

Beyond offering a decentralised currency, cryptocurrencies (e.g., Bitcoin [43] and Ethereum [49]) support computations on transactions. In this setting, often a computation logic is encoded in a computer program, called a “smart contract”. To date, Ethereum is the most predominant cryptocurrency framework that enables users to define arbitrary smart contracts. In this framework, contract code is stored on the blockchain and run by all parties maintaining the cryptocurrency. The program execution’s correctness is guaranteed by the security of the underlying blockchain components. To prevent a denial of service attack, the framework requires a transaction creator to pay a fee, called “gas”.

### 3.4 Commitment Scheme

A commitment scheme involves two parties, *sender* and *receiver*, and includes two phases: *commit* and *open*. In the commit phase, the sender commits to a message:  $x$  as  $\text{Com}(x, r) = \text{Com}_x$ , that involves a secret value:  $r \xleftarrow{\$} \{0,1\}^\lambda$ . In the end of the commit phase, the commitment  $\text{Com}_x$  is sent to the receiver. In the open phase, the sender sends the opening  $\tilde{x} := (x, r)$  to the receiver who verifies its correctness:  $\text{Ver}(\text{Com}_x, \tilde{x}) \stackrel{?}{=} 1$  and accepts if the output is 1. A commitment scheme must satisfy two properties: (a) *hiding*: it is infeasible for an adversary (i.e., the receiver) to learn any information about the committed message  $x$ , until the commitment  $\text{Com}_x$  is opened, and (b) *binding*: it is infeasible for an adversary (i.e., the sender) to open a commitment  $\text{Com}_x$  to different values  $\tilde{x}' := (x', r')$  than that was used in the commit phase, i.e., infeasible to find  $\tilde{x}'$ , *s.t.*  $\text{Ver}(\text{Com}_x, \tilde{x}) = \text{Ver}(\text{Com}_x, \tilde{x}') = 1$ , where  $\tilde{x} \neq \tilde{x}'$ . There exist efficient non-interactive commitment schemes both in (a) the standard model, e.g., Pedersen scheme [45], and (b) the random oracle model using the well-known hash-based scheme such that committing is :  $\text{H}(x||r) = \text{Com}_x$  and  $\text{Ver}(\text{Com}_x, \tilde{x})$  requires checking:  $\text{H}(x||r) \stackrel{?}{=} \text{Com}_x$ , where  $\text{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$  is a collision resistant hash function; i.e., the probability to find  $x$  and  $x'$  such that  $\text{H}(x) = \text{H}(x')$  is negligible in the security parameter  $\lambda$ .

### 3.5 Merkle Tree

A Merkle tree scheme introduced by Merkle [39,40], is a data structure often used for efficiently checking the integrity of an outsourced file. In this setting, there are two roles involved, a prover  $\mathcal{P}$  and a verifier  $\mathcal{V}$ . The Merkle tree scheme includes three algorithms (`MT.genTree`, `MT.prove`, `MT.verify`). Briefly, the first algorithm constructs a Merkle tree on file blocks, the second algorithm generates a proof of a block’s (or set of blocks’) membership, and the third one verifies the proof.

The security of the Merkle tree scheme requires that a computationally bounded malicious  $\mathcal{P}$  cannot convince  $\mathcal{V}$  into accepting an incorrect proof, e.g., a proof for non-member block. We refer readers to Appendix C for more detail.

### 3.6 Proofs of Retrievability (PoR)

A PoR scheme considers the case where an honest client wants to outsource the storage of its file to a potentially malicious server, i.e., an active adversary. It is a challenge-response interactive protocol, where the server proves to the client that its file is intact and retrievable. Below, we restate PoR’s formal definition initially proposed in [33,47]. A PoR scheme comprises five algorithms:

- **PoR.keyGen**( $1^\lambda$ )  $\rightarrow k := (sk, pk)$ . A probabilistic algorithm, run by a client,  $\mathcal{C}$ . It takes as input the security parameter  $1^\lambda$ . It outputs private-public verification key,  $k := (sk, pk)$ .
- **PoR.setup**( $1^\lambda, u, k$ )  $\rightarrow (u^*, \sigma, pp)$ . A probabilistic algorithm, run by  $\mathcal{C}$ . It takes as input  $1^\lambda$ , a file  $u$ , and key  $k$ . It encodes  $u$  yielding  $u^*$  and generates metadata,  $\sigma$ . It outputs  $u^*$ ,  $\sigma$ , and public parameters  $pp$ .
- **PoR.genQuery**( $1^\lambda, k, pp$ )  $\rightarrow \mathbf{q}$ . A probabilistic algorithm, run by  $\mathcal{C}$ . It takes as input  $1^\lambda$ , key  $k$ , and public parameters  $pp$ . It outputs a query vector  $\mathbf{q}$ , possibly picked uniformly at random.
- **PoR.prove**( $u^*, \sigma, \mathbf{q}, pk, pp$ )  $\rightarrow \pi$ . It is run by the server,  $\mathcal{S}$ . It takes as input the encoded file  $u^*$ , metadata  $\sigma$ , query  $\mathbf{q}$ , public key  $pk$ , and public parameters  $pp$ . It outputs a proof,  $\pi$ .
- **PoR.verify**( $\pi, \mathbf{q}, k, pp$ )  $\rightarrow d \in \{0, 1\}$ . It is run by  $\mathcal{C}$ . It takes as input  $\pi$ ,  $\mathbf{q}$ ,  $k$ , and  $pp$ . It outputs 0 if it rejects the proof, or 1 if it accepts the proof.

Informally, a PoR’s soundness requires that if a prover convinces the verifier, then the file is stored by the prover. Note, the above definition is generic, so certain realisations of this definition may not use all its algorithms and inputs/outputs. Appendix D presents the PoR’s formal definition.

## 4 Previous Work’s Limitations

In this section, we explain the previous schemes’ shortcomings and their flaws causing such shortcomings. Our focus is on the zkCSP schemes of Campanelli *et al.* [15], as they were specifically designed for the exchange of services and coins.

### 4.1 Issue 1: Waste of Server’s Resources

A malicious client, in each zkCSP scheme, can misbehave to benefit itself and waste the servers’ resources. Its misbehaviours include: (i) not participating in the payment although it has been using the service, or (ii) participating in the payment but making the server fail to pass the verification; for a concrete example, we refer readers to Appendix E. As we will show shortly, in the recurring payment, the client can collect convincing background information about an

honest server. A combination of this information and the above issues lets the client conclude that it has been served honestly, without paying the server.

The sources of Issue 1 are *incomplete fairness definition* and *mismatch of security assumptions*. In particular, the misbehaviour (i) is allowed because the zkCSP’s definition (Section 4.1 in [15]) is incomplete (or too weak). It only captures the moment when the client and server trade the service’s proof for coins, but it does not capture a crucial property, “resource fairness”. It is vital that this property is taken into consideration, because in the real world the server invests resources to serve the client before participating in the fair payment. Hence, in the exchange of services and coins, it is not fair if the client does not participate in the payment. The misbehaviour (ii) is allowed because each zkCSP scheme assumes *either party* is corrupt, but it uses a service scheme (i.e., PoR) that is secure against *only* a malicious server and assumes the client is honest. The incomplete definition and lack of rigorous security proof, of the zkCSP’s concrete instantiation, also played vital roles in misbehaviour (ii) remaining undetected.

## 4.2 Issue 2: Real-time Leakage

The zkCSP protocols leak in real-time, fresh information to the public. The leakage includes: (a) *proofs’ status* and (b) *deposit amount*. First, we focus on the proofs’ status leakage. In the zkCSP schemes (where a blockchain plays a role in the verification and payment) it becomes visible in *real-time* to *everyone* whether the service proofs were accepted. This issue remains even if the proofs are not stored in plaintext in the blockchain, as *coins transfer* itself reveals the proofs’ status. Thus, the blockchain provides to the public fresh information about the server’s status that could have not been easily attained otherwise. This leakage can have *serious immediate* consequences for both the server and clients, e.g., stock value drop [12,30], or opening doors for attackers to exploit such incidents. For example, this leakage lets a malicious client construct background knowledge of the server’s current behaviour (towards other clients). Such information can assist it to wisely exploit Issue 1, in the sense that when it observes that recently the server has been acting honestly, it refuses to participate in the payment. In this case, it still has high confidence that the server delivered the service.

As another example, a malicious observer can simply find out that the service is suffering from failure and exploit such vulnerability to harm the parties, e.g., mount social engineering attacks.<sup>3</sup> Also, in the zkCSP schemes, the deposit amount in the contract swiftly leaks information about the client to the public, as the server’s price list is public. For instance, when the service is the PoR, the public can learn the approximate file’s size, service type, or sometimes the region of the outsourced file. The main source of Issue 2 is an *improper use of public blockchain*. Briefly, the leakages occur because of the use of public blockchain for transferring deposits without having any privacy-preserving mechanism in place. We refer readers to Appendix F for further discussion of Issue 2.

---

<sup>3</sup> A survey conducted by Kaspersky lab in 2018 suggests that 33% of attacks that affect business clients, of cloud computing servers, are of type social engineering [37].

## 5 Our Solution’s Overview

In this section, we outline how we address the two issues. In Appendix G, we explain why naive approaches are not suitable to address these issues.

### 5.1 Addressing Issue 1

To address Issue 1, first we upgrade PoR to a “PoR with identifiable abort”<sup>4</sup> (PoRID) to (a) ensure security against a malicious client and (b) let an arbiter identify the misbehaving party and resolve disputes. PoRID requires the parties to post (some of) their messages to a smart contract, to avoid any repudiation issue. Also, the client is required to provide correct inputs, e.g., tags, or challenges; otherwise, it is identified as a misbehaving party. Then, we upgrade PoRID to a “recurring contingent PoR payment” (RC-PoR-P) scheme to (a) ensure fair payment and (b) prevent real-time leakage. To address Issue 1, RC-PoR-P requires (i) the client to deposit its coins before uploading its file to the server, and (ii) the server is paid by a misbehaving client. Also, RC-PoR-P ensures that the misbehaving party pays the arbiter.

### 5.2 Addressing Issue 2

To address Issue 2, first we let the client and server take control of the time of the information release and agree on the period in which the information remains hidden, a “*private time bubble*”. In this period, all messages sent to the smart contract are encrypted. They raise disputes after this period ends (or the bubble bursts) and when the data loses its sensitivity.<sup>5</sup> But, the client can still locally check a proof’s validity once it is provided. Second, to hide the actual deposit amount, we let each party mask its coins, by increasing the actual coins amount to the maximum amount of coin in the server’s price list. However, this raises another challenge: *how can the mutually distrusted parties claim back their masking coins after the bubble bursts, while hiding the actual coins amount from the public in the private bubble?* To deal with it, we let the client and server initially agree on a *private* statement specifying the deposit details. Later, when they want to reclaim their coins, they provide the statement to the smart contract which checks the statement’s validity and if the check passes, it distributes coins according to the statement’s specification. We will show how they can efficiently agree on such a statement, by using a “statement agreement protocol” (SAP).

## 6 Recurring Contingent PoR Payment (RC-PoR-P)

In this section, we present RC-PoR-P. It offers two features (in addition to addressing the two issues), it (a) does not use zk proofs although either client

---

<sup>4</sup> It was inspired by the notion of secure multi-party computation with identifiable abort [32].

<sup>5</sup> The concept of delayed information release has already been used by researchers, e.g., in smart metering in [31] or in the real world through the declassification approach.

or server can be malicious, and (b) has a low arbiter-side cost. As we will show, the latter also enables a smart contract to efficiently play the arbiter’s role.

*Avoiding the use of zero-knowledge proofs:* PoRs often use tags (e.g., MACs) or a Merkle tree’s root, as metadata, for the verification. Most PoRs assume that the client is honest. When it is also considered corrupt, if tags are used, using zk proofs is an obvious approach for the client to prove that it correctly generated the tags (e.g., in [6]). But, this imposes significant overheads. We observed that using a Merkle tree would benefit our protocol from a couple of perspectives; it (i) removes the need for zk proofs and (ii) supports “proof of misbehaviour”. If a Merkle tree is used, then the server can efficiently check the metadata by simply rebuilding this tree on the file with low costs and without using zk proofs.

*Low arbiter-side cost:* In a Merkle tree-based PoR, the number of proofs (or paths) are linear with the number of blocks that are probed, say  $\phi$ . In this scheme, the verifier checks all proofs and rejects them if one of them is invalid. We observed that if it is used in the RC-PoR-P, then once the client rejects a proof, it can send only that single invalid proof as a *proof of misbehaviour* to the arbiter.<sup>6</sup> This technique significantly reduces the arbiter’s computation cost, i.e., from  $\phi \log_2(n)$  to  $\log_2(n)$ , where  $n$  is the number of file blocks.

In the rest of this section, we first present a modified Merkle tree-based PoR. Then, we upgrade it to the PoRID and after that build the RC-PoR-P upon it.

## 6.1 Modified Merkle tree-based PoR

In this section, we first present a modified version of the standard Merkle tree-based PoR, and then explain the applied modifications. At a high level, the client encodes its file using an error-correcting code, splits the result into blocks, and builds a Merkle tree on the blocks. It locally stores the tree’s root. It sends the blocks to the server which rebuilds the tree. At the verification time, the client sends a PRF’s key to the server which derives a number of blocks’ indices showing which blocks are probed. The server for each probed block generates a proof. It sends all proofs to the client which checks them. If it accepts all proofs, it concludes that its file is retrievable. But, if it rejects some proofs, it stores only one index of the blocks whose proofs were rejected. Below, we present the PoR.

1. **Client-side Setup.**  $\text{PoR.setup}(1^\lambda, u)$ 
  - (a) Uses an error-correcting code, to encode the file:  $u$ . Let  $u'$  be the encoded file. It splits  $u'$  into blocks as follows,  $u^* = u'_0 || 0, \dots, u'_m || m$ .
  - (b) Constructs a Merkle tree on  $u^*$ ’s blocks, i.e.,  $\text{MT.genTree}(u^*)$ . Let  $\sigma$  be the root of the tree, and  $\phi$  be the number of blocks that will be probed. It sets public parameters as  $pp := (\sigma, \phi, m, \zeta)$ , where  $\zeta := (\psi, \eta, \iota)$  is a PRF’s description, as it was defined in Section 3. It sends  $pp$  and  $u^*$  to  $\mathcal{S}$ .
2. **Client-side Query Generation.**  $\text{PoR.genQuery}(1^\lambda, pp)$ .
  - Picks a key  $\hat{k}$  for PRF. It sends  $\hat{k}$  to  $\mathcal{S}$ .
3. **Server-side Proof Generation.**  $\text{PoR.prove}(u^*, \hat{k}, pp)$

<sup>6</sup> This idea is akin to the proof of misbehaviour proposed in [16].

- (a) Derives  $\phi$  pseudorandom indices from  $\hat{k}$  as follows.  
 $\forall i, 1 \leq i \leq \phi : q_i = \text{PRF}(\hat{k}, i) \bmod m + 1$ . Let  $\mathbf{q} = [q_1, \dots, q_\phi]$ .
  - (b) For each random index  $q_i$ , generates a proof,  $\pi_{q_i} = \text{MT.prove}(u^*, q_i)$ . The final result is  $\boldsymbol{\pi} = [(u_{q_i}^*, \pi_{q_i})]_{q_i \in \mathbf{q}}$ , where  $i$ -th element in  $\boldsymbol{\pi}$  corresponds to  $q_i$ , and the probed block is  $u_{q_i}^*$ . It sends  $\boldsymbol{\pi}$  to  $\mathcal{C}$ .
- 4. Client-side Proof Verification.**  $\text{PoR.verify}(\boldsymbol{\pi}, \mathbf{q}, pp)$
- (a) If  $|\boldsymbol{\pi}| = |\mathbf{q}| = 1$ , then set  $\phi = 1$ . This step is only for the case where single proof and query is provided (e.g., in the proof of misbehaviour).
  - (b) Checks if the server sent all required proofs, by parsing each element of  $\boldsymbol{\pi}$  as:  $\text{parse}(u_{q_i}^*) = u'_{q_i} || q_i$ , and checking if its index  $q_i$  equals to  $\mathbf{q}$ 's  $i$ -th element. If all checks pass, it takes the next step. Otherwise, it outputs  $\mathbf{d} = [0, i]$ , where  $i$  is the index of  $\boldsymbol{\pi}$ 's element that did not pass the check.
  - (c) Checks if every path in  $\boldsymbol{\pi}$  is valid, by calling  $\text{MT.verify}(u_{q_i}^*, \pi_{q_i}, \sigma)$ . If all checks pass, it outputs  $\mathbf{d} = [1, \perp]$ ; otherwise, it outputs  $\mathbf{d} : [0, i]$ , where  $i$  refers to the index of the element in  $\boldsymbol{\pi}$  that does not pass the check.

**Theorem 1 (informal).** *The above PoR scheme is sound if Merkle tree and pseudorandom function PRF are secure.*

We refer readers to Appendix J for the theorem's formal statement and proof. This PoR differs from the standard Merkle tree-based PoR from two perspectives; First, in step 4, the client also outputs one of the rejected proofs' indices. This will let a third party, given that index (and vectors of proofs and challenges), *efficiently* verify that the server did not pass the verification. Second, in step 2, instead of sending  $\phi$  challenges, we let the client send only a key of the PRF to the server which can derive a set of challenges from it. This will lead to a decrease in (the client's communication and a smart contract's storage) costs.

## 6.2 PoRID Scheme

Most PoRs are secure against a malicious server and assume the client is honest. Although this assumption may suffice in theory, it may not hold in the real world, when there are monetary incentives (e.g., service payment) tempting a client to misbehave. In this section, we present PoRID that addresses this limitation.

**PoRID Model's Overview.** The PoRID model is built upon the PoR (presented in Section 3.6). However, the PoRID offers two additional properties; namely, (1) *inputs well-formedness*: a malicious client cannot persuade a server to serve it on ill-structured inputs, and (2) *detectable abort*: a corrupt party is identified by a trusted third-party arbiter. In the PoRID, four parties are involved: a client, server, arbiter, and standard smart contract. In Appendix H, we present our PoRID's formal definition.

**PoRID Protocol.** In the PoRID, at the setup  $\mathcal{C}$  encodes its file  $u$  and generates public parameters and metadata. It sends these parameters and metadata to the smart contract. It sends the encoded file to  $\mathcal{S}$  which *efficiently* checks the correctness of the parameters and metadata. It agrees to serve if the checks pass. At the verification time,  $\mathcal{C}$  sends a query to the smart contract.  $\mathcal{S}$  checks the query and generates proofs, if the check passes. The prove and verify algorithms are similar to those in the PoR with a difference that  $\mathcal{S}$  sends the proofs to the contract. In case of a dispute,  $\mathcal{C}$  or  $\mathcal{S}$  invokes the arbiter that checks the query and proofs to identify a misbehaving party. Below, we present the PoRID.

1. **Client-side Setup.**  $\text{PoRID.setup}(1^\lambda, u)$ 
  - (a) Calls  $\text{PoR.setup}(1^\lambda, u) \rightarrow (u^*, pp)$ . Recall,  $pp := (\sigma, \phi, m, \zeta)$ .
  - (b) Sends the public parameter  $pp$  to the smart contract and sends  $u^*$  to  $\mathcal{S}$ .
2. **Server-side Setup.**  $\text{PoRID.serve}(u^*, pp)$ 
  - (a) rebuilds the Merkle tree on  $u^*$  and checks if the result root equals  $\sigma \in pp$ .
  - (b) checks  $|u^*| = m$  and  $\phi \leq m$ .

If the checks pass, it outputs  $a = 1$ . Otherwise, it outputs  $a = 0$  and halts.
3. **Client-side Query Generation.**  $\text{PoRID.genQuery}(1^\lambda, pp)$ 
  - (a) Calls  $\text{PoR.genQuery}(1^\lambda, pp) \rightarrow \hat{k}$ .
  - (b) Sends  $\hat{k}$  to the smart contract.
4. **Server-side Query Verification.**  $\text{PoRID.checkQuery}(\hat{k}, pp)$ 
  - (a) Checks if  $\hat{k}$  is not empty, i.e.,  $\hat{k} \neq \perp$ , and is in the key's universe.
  - (b) If the checks pass, it outputs  $b = 1$ ; otherwise, it outputs  $b = 0$  and halts.
5. **Server-side Service Proof Generation.**  $\text{PoRID.prove}(u^*, \hat{k}, pp)$ 
  - (a) Calls  $\text{PoR.prove}(u^*, \hat{k}, pp) \rightarrow \pi$ , to generate a proof vector,  $\pi$ .
  - (b) Sends  $\pi$  to the smart contract.
6. **Client-side Proof Verification.**  $\text{PoRID.verify}(\pi, \hat{k}, pp)$ 
  - Calls  $\text{PoR.verify}(\pi, \hat{k}, pp) \rightarrow d$ , to verify the proof. If  $d[0] = 1$ , it returns  $d = 1$ ; otherwise, it returns  $d = 0$ .
7. **Arbiter-side Identification.**  $\text{PoRID.identify}(\pi, q, pp)$ 

It is invoked by  $\mathcal{C}$  or  $\mathcal{S}$ , in the case of a dispute. Let  $q := (\hat{k}, i)$ . If  $\mathcal{C}$  invokes it, then  $i$  is an invalid proof's index. If  $\mathcal{S}$  invokes it, then  $i = \perp$ .

  - (a) Checks if  $\hat{k}$  is well-structured by calling  $\text{PoRID.checkQuery}(\hat{k}, pp) \rightarrow b$ .
    - if  $b = 0$ , it outputs  $I = \mathcal{C}$  and halts.
    - if  $b = 1$  and  $i = \perp$ , it halts.
    - if  $b = 1$  and  $i \neq \perp$ , it takes the next step.
  - (b) Generates the probed block's index, i.e.,  $q_i = \text{PRF}(\hat{k}, i) \bmod m + 1$ .
  - (c) Verifies  $i$ -th proof, by setting  $\hat{\pi} = \pi[i]$ ,  $\hat{q} = q_i$  and calling  $\text{PoR.verify}(\hat{\pi}, \hat{q}, pp) \rightarrow d'$ . If  $d'[0] = 0$ , it outputs  $I = \mathcal{S}$ . Otherwise, it outputs  $I = \perp$ .

**Theorem 2 (informal).** *The PoRID satisfies the soundness, detectable abort, and inputs well-formedness, if the PoR is sound and the blockchain is secure.*

We provide the above theorem's formal statement and proof in Appendix K.

### 6.3 Recurring Contingent PoR Payment (RC-PoR-P) Protocol

As stated in Section 5, the RC-PoR-P relies on the SAP that lets the server and client provably agree on private statements efficiently. In the RC-PoR-P, SAP will let a party (i) reclaim its masking coins or (ii) prove it has an agreement with its counter-party on secret parameters, after the private bubble bursts. In this section, we first present the SAP and then the RC-PoR-P.

**Statement Agreement Protocol (SAP).** Informally, a SAP is secure if it meets four properties: (1) neither party can persuade a third-party verifier that it has agreed with its counter-party on a *different statement* than the one both parties initially agreed on, (2) after they agree on a statement, an honest party can (almost) *always* prove to have the agreement to the verifier, (3) the *privacy* of the statement is preserved (from the public before the proving phase), and (4) after both parties reach an agreement, neither can later *deny* it. The SAP uses a smart contract and commitment scheme. To agree on a private statement  $x$  with  $\mathcal{S}$ ,  $\mathcal{C}$  picks a random value  $r$  and uses it to commit to  $x$ . It sends the commitment to the smart contract and the commitment's opening (i.e.,  $x$  and  $r$ ) to  $\mathcal{S}$  that checks if the opening matches the commitment and if so, it commits to the statement using the same random value. It sends the result to the contract. Later, for a party to prove to the contract (i.e., the verifier) that it has agreed on  $x$  with the other party, it sends the commitment's opening to the verifier which checks if the opening matches *both* commitments and accepts if they do. Below, we present the SAP. It assumes that each party  $\mathcal{R} \in \{\mathcal{C}, \mathcal{S}\}$  already has a blockchain public address,  $adr_{\mathcal{R}}$ .

1. **Initiate.**  $\text{SAP.init}(1^\lambda, adr_{\mathcal{C}}, adr_{\mathcal{S}}, x)$   
The following steps are taken by  $\mathcal{C}$ .
  - (a) Deploys a smart contract that explicitly states both parties' addresses,  $adr_{\mathcal{C}}$  and  $adr_{\mathcal{S}}$ . Let  $adr_{\text{SAP}}$  be the deployed contract's address.
  - (b) Picks a random value  $r$ , and commits to the statement,  $\text{Com}(x, r) = g_{\mathcal{C}}$ .
  - (c) Sends  $adr_{\text{SAP}}$  and  $\ddot{x} := (x, r)$  to  $\mathcal{S}$ , and  $g_{\mathcal{C}}$  to the contract.
2. **Agreement.**  $\text{SAP.agree}(x, r, g_{\mathcal{C}}, adr_{\mathcal{C}}, adr_{\text{SAP}})$   
The following steps are taken by  $\mathcal{S}$ .
  - (a) Checks if  $g_{\mathcal{C}}$  was sent from  $adr_{\mathcal{C}}$ , and checks  $\text{Ver}(g_{\mathcal{C}}, \ddot{x}) = 1$ .
  - (b) If the checks pass, it sets  $b = 1$ , computes  $\text{Com}(x, r) = g_{\mathcal{S}}$ , and sends  $g_{\mathcal{S}}$  to the contract. Otherwise, it sets  $b = 0$  and  $g_{\mathcal{S}} = \perp$ .
3. **Prove.** For either  $\mathcal{C}$  or  $\mathcal{S}$  to prove, it sends  $\ddot{x} := (x, r)$  to the smart contract.
4. **Verify.**  $\text{SAP.verify}(\ddot{x}, g_{\mathcal{C}}, g_{\mathcal{S}}, adr_{\mathcal{C}}, adr_{\mathcal{S}})$   
The following steps are taken by the smart contract.
  - (a) Ensures  $g_{\mathcal{C}}$  and  $g_{\mathcal{S}}$  were sent from  $adr_{\mathcal{C}}$  and  $adr_{\mathcal{S}}$  respectively.
  - (b) Ensures  $\text{Ver}(g_{\mathcal{C}}, \ddot{x}) = \text{Ver}(g_{\mathcal{S}}, \ddot{x}) = 1$ .
  - (c) Outputs  $d = 1$ , if the checks, in steps 4a and 4b, pass. It outputs  $d = 0$ , otherwise.

In Appendix L, we discuss the SAP's security and explain why naive solutions are not suitable.

**RC-PoR-P Model’s Overview.** The RC-PoR-P model is built upon the PoRID, but it captures fair payment and privacy too. Informally, the RC-PoR-P is secure if it meets three properties: (1) *security against a malicious server*: for each verification,  $\mathcal{S}$  cannot make  $\mathcal{C}$  or the arbiter receive an incorrect amount regardless of whether it convinces  $\mathcal{C}$  that it stored the file, (2) *security against a malicious client*: for each verification, a malicious  $\mathcal{C}$  cannot provide: (2.a) valid metadata and query but makes  $\mathcal{S}$  or the arbiter receive an incorrect amount, or (2.b) invalid metadata or query but convinces  $\mathcal{S}$  to accept either of them, or (2.c) invalid query but persuades the arbiter to accept it, or makes them withdraw an incorrect amount, and (3) *privacy*: no information about the file’s content and the proof’s status is leaked, during the private time bubble. We present the RC-PoR-P’s formal model in Appendix I.

**RC-PoR-P Protocol.** Below we present the RC-PoR-P’s outline and then its detailed description. In this protocol, initially  $\mathcal{C}$  and  $\mathcal{S}$  use the SAP to agree on two private statements, one includes the payment’s detail, and another one specifies an encryption secret key and a pad’s length, used to encode private messages sent to a smart contract.  $\mathcal{C}$  deploys a smart contract SC specifying public parameters, e.g., the total amount of masked coin each party deposits, and the private time bubble’s length:  $z + J$ , where  $z$  is the total number of billing cycles, and  $J$  is a waiting time. Each party deposits its masked coin in SC.

$\mathcal{C}$  encodes the file and generates metadata. It sends the metadata’s encryption to SC and the encoded file to  $\mathcal{S}$  that decrypts the ciphertext, checks the result, and declares that it wants to serve, if the check passes. At the end of each billing cycle,  $\mathcal{C}$  sends an encrypted query to SC.  $\mathcal{S}$  decrypts it and checks the result’s correctness. If it rejects it, it locally stores that cycle’s index and generates dummy proofs. Otherwise, it generates actual proofs. In either case,  $\mathcal{S}$  encodes the proofs, and sends the result to SC.  $\mathcal{C}$  decodes and locally verifies them. If it accepts them,  $\mathcal{C}$  concludes that the file is retrievable. Otherwise,  $\mathcal{C}$  locally stores that cycle’s index and detail of *one of the invalid proofs* (in the same cycle).

During the dispute resolution time,  $\mathcal{S}$  or  $\mathcal{C}$  sends to the arbiter the detail of invalid values in SC and the statement, containing the decoding parameters, and its proof. The arbiter checks the statement’s validity. If the check passes, it decodes the specified values. The arbiter checks all claims. Next, it informs SC about how many times each party misbehaved (and unnecessarily invoked it). In the next phase, to distribute the coins, either  $\mathcal{S}$  or  $\mathcal{C}$  sends to SC: (a) “pay” message, (b) the statement specifying the payment’s detail, and (c) the statement’s proof. The contract verifies the statement and if approves, distributes the coins according to the statement’s detail, and the number of times each party misbehaved. Below, we present the RC-PoR-P.

1. **Key Generation.**  $\text{RCPoRP.keyGen}(1^\lambda)$

- (a)  $\mathcal{C}$  picks a fresh key  $\bar{k}$ . It sets parameter  $pad_\pi$ : the number of dummy values used to pad encrypted proofs. Let  $sk' := (pad_\pi, \bar{k})$  and  $k := (sk', pk')$ , where  $pk' := (adr_{\mathcal{C}}, adr_{\mathcal{S}})$ .

2. **Client-side Initiation.**  $\text{RCPoRP.cInit}(1^\lambda, u, k, z, pl)$ 
  - (a) Calls  $\text{PoRID.setup}(1^\lambda, u) \rightarrow (u^*, pp)$  to encode  $u$ . It sets  $qp := (sk', pp)$ .
  - (b) Sets  $cp := (o, o_{max}, l, l_{max}, z)$ ,  $coin_c^* = z \cdot (o_{max} + l_{max})$  and  $p_s = z \cdot l_{max}$ , given the price list  $pl$ , where  $coin_c^*$  and  $p_s$  are the total number of masked coins  $\mathcal{C}$  and  $\mathcal{S}$  should deposit. Section 3.1 defines the parameters.
  - (c) Calls  $\text{SAP.init}(1^\lambda, adr_c, adr_s, qp) \rightarrow (r_{qp}, g_{qp}, adr_{\text{SAP}_1})$  and  $\text{SAP.init}(1^\lambda, adr_c, adr_s, cp) \rightarrow (r_{cp}, g_{cp}, adr_{\text{SAP}_2})$  to initiate agreements on  $qp$  and  $cp$  with  $\mathcal{S}$ . Let  $T_{qp} := (\dot{x}_{qp}, g_{qp})$  and  $T_{cp} := (\dot{x}_{cp}, g_{cp})$ , s.t.  $\dot{x}_{qp} := (qp, r_{qp})$  and  $\dot{x}_{cp} := (cp, r_{cp})$  are the openings of  $g_{qp}$  and  $g_{cp}$ . Let  $T := \{T_{qp}, T_{cp}\}$ .
  - (d) Sets a smart contract, SC, that explicitly specifies parameters  $z, coin_c^*, p_s, adr_{\text{SAP}_1}, adr_{\text{SAP}_2}, pk', \text{Time} : \{T_0, \dots, T_2, G_{1,1}, \dots, G_{z,2}, J, K_1, \dots, K_6, L\}$ , and  $[y_c, y'_c, y_s, y'_s]$ . The vector's elements value is 0. It deploys SC. Let  $adr_{\text{SC}}$  be the address of the deployed SC and  $\mathbf{y} : [y_c, y'_c, y_s, y'_s, adr_{\text{SC}}]$ .
  - (e) Deposits  $coin_c^*$  coins in the contract. It sends  $u^*, z, \dot{x}_{qp}$ , and  $\dot{x}_{cp}$  (along with  $adr_{\text{SC}}$ ) to  $\mathcal{S}$ . Let  $T_0$  be the time that the above process finishes.
3. **Server-side Initiation.**  $\text{RCPoRP.sInit}(u^*, z, T, p_s, \mathbf{y})$ 
  - (a) Checks the parameters in  $T$  (e.g.,  $qp$  and  $cp$ ) and in SC (e.g.,  $p_s, \mathbf{y}$ ) and ensures sufficient amount has been deposited by  $\mathcal{C}$ .
  - (b) Calls  $\text{SAP.agree}(qp, r_{qp}, g_{qp}, adr_c, adr_{\text{SAP}_1}) \rightarrow (g'_{qp}, b_1)$  and  $\text{SAP.agree}(cp, r_{cp}, g_{cp}, adr_c, adr_{\text{SAP}_2}) \rightarrow (g'_{cp}, b_2)$ , to check and agree on  $qp$  and  $cp$ .
  - (c) If  $b_1 = 0$  or  $b_2 = 0$ , sets  $a = 0$ . Otherwise, it calls  $\text{PoRID.serve}(u^*, pp) \rightarrow a$ . It sends  $a$  and  $coin_s^* = p_s$  coins to SC at time  $T_1$  (if  $a = 0$ ,  $coin_s^* = \perp$ ).  
 $\mathcal{S}$  and  $\mathcal{C}$  can withdraw their coins at time  $T_2$ , if  $\mathcal{S}$  sends  $a = 0$  or fewer coins than  $p_s$  to SC. To withdraw,  $\mathcal{S}$  or  $\mathcal{C}$  sends “pay” to  $\text{RCPoRP.pay}(\cdot)$  at  $T_2$ .  
**Billing-cycles Onset.**  $\mathcal{C}$  and  $\mathcal{S}$  engage in phases 4-6, at the end of every  $j$ -th billing cycle, where  $1 \leq j \leq z$ . Each  $j$ -th cycle includes two time points,  $G_{j,1}$  and  $G_{j,2}$ , where  $G_{j,2} > G_{j,1}$ , and  $G_{1,1} > T_2$ .
4. **Client-side Query Generation.**  $\text{RCPoRP.genQuery}(1^\lambda, T_{qp})$ 
  - (a) Calls  $\text{PoRID.genQuery}(1^\lambda, pp) \rightarrow \hat{k}_j$  to generate a query, where  $pp \in T_{qp}$ .
  - (b) Sends encrypted query  $\hat{k}_j^* = \text{Enc}(\bar{k}, \hat{k}_j)$  to SC at  $G_{j,1}$ .
5. **Server-side Proof Generation.**  $\text{RCPoRP.prove}(u^*, \hat{k}_j^*, T_{qp})$ 
  - (a) Constructs an empty vector,  $\mathbf{m}_s = \perp$ , if  $j = 1$ .
  - (b) Decrypts the query,  $\hat{k}_j = \text{Dec}(\bar{k}, \hat{k}_j^*)$ , where  $\bar{k} \in T_{qp}$ .
  - (c) Calls  $\text{PoRID.checkQuery}(\hat{k}_j, pp) \rightarrow b_j$  to check the query's correctness.
    - if the check passes, it calls  $\text{PoRID.prove}(u^*, \hat{k}_j, pp) \rightarrow \pi_j$ , to generate proofs. It encrypts them,  $\forall i, 1 \leq i \leq |\pi_j| : \text{Enc}(\bar{k}, \pi_j[i]) = \pi'_j[i]$ . Let  $\pi'_j$  contain the encrypted proofs. It pads every encrypted proof in  $\pi'_j$  with  $pad_\pi \in T_{qp}$  random values, picked from the encryption's output range  $U$ . Let  $\pi_j^*$  be the result. It sends  $\pi_j^*$  to SC at  $G_{j,2}$ .
    - otherwise, it appends  $j$  to  $\mathbf{m}_s$ , generates dummy proofs  $\pi'_j$ , whose elements are randomly picked from  $U$ , pads the result as above, and sends the final result,  $\pi_j^*$ , to SC at  $G_{j,2}$ .
6. **Client-side Proof Verification.**  $\text{RCPoRP.verify}(\pi_j^*, \hat{k}_j^*, T_{qp})$ 
  - (a) Constructs an empty vector,  $\mathbf{m}_c = \perp$ , if  $j = 1$ .

- (b) Removes the pads from  $\pi_j^*$  yielding  $\pi'_j$ . It decrypts  $\pi_j^*$  as:  $\text{Dec}(\bar{k}, \pi'_j) = \pi_j$  and runs  $\text{PoRID.verify}(\pi_j, \hat{k}_j, pp) \rightarrow \mathbf{d}_j$ , where  $\hat{k}_j = \text{Dec}(\bar{k}, \hat{k}_j^*)$ .
- if  $\pi_j$  passes the verification, i.e.,  $\mathbf{d}_j[0] = 1$ , then  $\mathcal{C}$  concludes that the service for this verification has been delivered successfully.
  - otherwise, it sets  $i = \mathbf{d}_j[1]$  and appends vector  $[j, i]$  to  $\mathbf{m}_c$ . Recall,  $\mathbf{d}_j[1]$  refers to a rejected proof's index in proof vector  $\pi_j$ .
7. **Dispute Resolution.**  $\text{RCPoRP.resolve}(\mathbf{m}_c, \mathbf{m}_s, z, \pi^*, \mathbf{q}^*, T_{qp})$
- (a) The arbiter sets  $y_c, y'_c, y_s$  and  $y'_s$  initially to 0, before time  $K_1 > G_{z,2} + J$ .
- (b)  $\mathcal{S}$  sends  $\mathbf{m}_s$  and  $\hat{x}_{qp}$  to the arbiter, at  $K_1$ .
- (c) The arbiter after receiving  $\mathbf{m}_s$ , takes the following steps, at time  $K_2$ .
- i. checks  $\hat{x}_{qp}$ 's validity, by calling the SAP's verification which returns  $d$ . If  $d = 0$ , it discards  $\mathbf{m}_s$  and does not take steps 7(c)ii and 7(c)iii. Otherwise, it proceeds to the next step.
  - ii. removes from  $\mathbf{m}_s$  any element that is duplicated or is not in the range  $[1, z]$ . It constructs an empty vector,  $\mathbf{v}$ .
  - iii. for every element  $j \in \mathbf{m}_s$ : decrypts the related query  $\hat{k}_j^* \in \mathbf{q}^*$  as  $\hat{k}_j = \text{Dec}(\bar{k}, \hat{k}_j^*)$ ; and checks the query, by calling  $\text{PoRID.checkQuery}(\hat{k}_j, pp) \rightarrow b_j$ . If  $b_j = 0$ , it increments  $y_c$  by 1 and appends  $j$  to  $\mathbf{v}$ . Otherwise, it increments  $y'_s$  by 1. Let  $K_3$  be the time the checks are complete.
- (d)  $\mathcal{C}$  sends  $\mathbf{m}_c$  and  $\hat{x}_{qp}$  to the arbiter, at time  $K_4$ .
- (e) The arbiter after receiving  $\mathbf{m}_c$ , takes the following steps, at time  $K_5$ .
- i. checks  $\hat{x}_{qp}$ 's validity, by calling the SAP's verification which returns  $d'$ . If  $d' = 0$ , it discards  $\mathbf{m}_c$ , and does not take steps 7(e)ii-7(e)iii. Otherwise, it proceeds to the next step.
  - ii. ensures each vector  $\mathbf{m} \in \mathbf{m}_c$  is well-formed. Specifically, it checks there exist no  $\mathbf{m}, \mathbf{m}' \in \mathbf{m}_c$  such that  $\mathbf{m}[0] = \mathbf{m}'[0]$ . If such vectors exist, it deletes the redundant ones from  $\mathbf{m}_c$ . It also removes any vector  $\mathbf{m}$  from  $\mathbf{m}_c$  if  $\mathbf{m}[0]$  is not in the range  $[1, z]$ , or if  $\mathbf{m}[0] \in \mathbf{v}$ .
  - iii. for every vector  $\mathbf{m} \in \mathbf{m}_c$ :
    - retrieves a rejected proof's detail, by setting  $j = \mathbf{m}[0]$  and  $i = \mathbf{m}[1]$ . Recall,  $i$  is a rejected proof's index in the proofs,  $\pi_j$ .
    - decrypts the related query  $\hat{k}_j^* \in \mathbf{q}^*$  as  $\hat{k}_j = \text{Dec}(\bar{k}, \hat{k}_j^*)$ .
    - removes the pads from  $i$ -th encoded proof. Let  $\pi'_j[i]$  be the result. It decrypts the encrypted proof,  $\text{Dec}(\bar{k}, \pi'_j[i]) = \pi_j[i]$ .
    - generates a fresh vector:  $\pi''_j$ , such that its  $i$ -th element equals  $\pi_j[i]$  (i.e.,  $\pi''_j[i] = \pi_j[i]$  and  $|\pi''_j| = |\pi_j|$ ) and the rest of its elements are dummy values.
    - sets  $q := (\hat{k}_j, i)$  and calls  $\text{PoRID.identify}(\pi''_j, q, pp) \rightarrow I_j$ . If  $I_j = \mathcal{S}$ , it increments  $y_s$  by 1. If  $I_j = \perp$ , it increments  $y'_c$  by 1.
- (f) The arbiter at time  $K_6$  sends  $[y_c, y_s, y'_c, y'_s]$  to SC, that adds them to  $\mathbf{y}$ .
8. **Coin Transfer.**  $\text{RCPoRP.pay}(\mathbf{y}, T_{cp}, a, p_s, \text{coin}_c^*, \text{coin}_s^*)$
- (a) If SC receives "pay" at time  $T_2$ , where  $a = 0$  or  $\text{coins}_s^* < p_s$ , it sends  $\text{coin}_c^*$  coins to  $\mathcal{C}$  and  $\text{coin}_s^*$  coins to  $\mathcal{S}$ .

- (b) If SC receives “pay” and  $\ddot{x}_{cp} \in T_{cp}$  at time  $L > K_6$ , it checks  $\ddot{x}_{cp}$ ’s validity by calling the SAP’s verification which returns  $d''$ . SC proceeds to the next step if  $d'' = 1$ .
- (c) SC distributes the coins to the parties as follows:
- $coin_{\mathcal{C}} = coin_{\mathcal{C}}^* - o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_{\mathcal{C}} + y'_{\mathcal{C}})$  coins to  $\mathcal{C}$ .
  - $coin_{\mathcal{S}} = coin_{\mathcal{S}}^* + o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_{\mathcal{S}} + y'_{\mathcal{S}})$  coins to  $\mathcal{S}$ .
  - $coin_{Ar} = l \cdot (y_{\mathcal{S}} + y_{\mathcal{C}} + y'_{\mathcal{S}} + y'_{\mathcal{C}})$  coins to the arbiter.

**Theorem 3 (informal).** *The RC-PoR-P protocol is secure if the PoRID, SAP, blockchain, and encryption scheme are secure.*

We present the above theorem’s formal statement and proof in Appendix M. Moreover, in Appendix O we provide several extensions and remarks on the RC-PoR-P, such as (i) how  $\mathcal{S}$  and  $\mathcal{C}$  even during the private time bubble can promise to a third party the amount they own in the contract, or (ii) how they can send proofs *directly* to each other, without running into any deniability issue.

**Delegating the Arbiter’s Role to a Smart Contract.** In the RC-PoR-P, due to the efficiency of the arbiter-side algorithm, we can delegate the arbiter’s role to the smart contract, SC. To this end, the RC-PoR-P requires slight amendments, e.g., to the amount of deposit. In Appendix N, we explain how we can construct the RC-PoR-P’s new variant.

## 7 Evaluation

In this section, we provide the RC-PoR-P’s brief analysis and compare it with the (i) zero-knowledge contingent (publicly verifiable) PoR payment in [15], and (ii) fair PoR payment scheme in [3]. Tables 1 and 2 summarises the RC-PoR-P’s *asymptotic* and *concrete* costs, while Table 3 compares the three schemes. To conduct the concrete cost study, we have implemented the RC-PoR-P. The protocol’s off-chain and on-chain source code is in [1] and [2] respectively. We used random files whose size is in the range [64 MB, 4 GB] and set a file block to 128 bits. This results in the number of file blocks in the range  $[2^{22}, 2^{28}]$ .

Table 1: The RC-PoR-P’s complexity (in big O), for  $z$  verifications.  $\phi$ : the number of probed blocks,  $z'$ : the max number of complaints,  $m$ : the number of file blocks,  $\|u^*\|$ : the file bit-size, and  $\|\pi^*\|$ : the number of elements in the encoded proof.

Phase	Party	Computation Cost	Communication Cost
Client and Server Init. (i.e., outsourcing: 2 and 3)	Client	$m$	$\ u^*\ $
	Server	$m$	1
The rest of phases (i.e., 4- 8)	Client	$z\phi \log_2(m)$	$z \log_2(\ u^*\ )$
	Server	$z\phi \log_2(m)$	$z\ \pi^*\ $
	Arbiter	$z' \log_2(m)$	1
	Smart Contract	1	-

Table 2: The RC-PoR-P off-chain run-time (in seconds) and on-chain cost, for  $z$  verifications.  $z'$ : the max number of complaints, and  $m$ : the number of a file blocks.

Phase	Off-chain cost			On-chain cost	
	$m : 2^{22}$	$m : 2^{25}$	$m : 2^{28}$	Ether	US Dollar
Client-side Init.	23.1	185.8	1596.6	$123 \cdot 10^{-5}$	3.42
Server-side Init.	20.9	144.6	1548.8	$9 \cdot 10^{-5}$	0.22
Client-side Query Gen.	-	-	-	$6 \cdot 10^{-5} \cdot z$	$0.17 \cdot z$
Server-side Proof Gen.	$18.4 \cdot z$	$106.8 \cdot z$	$1320.7 \cdot z$	-	-
Client-side Proof Ver.	$0.09 \cdot z$	$0.16 \cdot z$	$0.24 \cdot z$	-	-
Arbiter-side Dispute Res.	$2 \cdot 10^{-5} \cdot z'$	$8 \cdot 10^{-5} \cdot z'$	$10^{-4} \cdot z'$	$10^{-4}$	0.27
Coin Transfer	-	-	-	$6 \cdot 10^{-5}$	0.17

Table 3: Contingent PoRs' complexity (in big O) and property comparison.  $m$ : the number of the file blocks,  $T$ : a time parameter, and  $\phi$ : the number of probed blocks.

Protocols	Operation	Computation Complexity				Proof Size	Malicious		Offers Privacy
		Initiate	Solve Puzzle	Prove	Verify		Client	Server	
[3]	Exp.	$z$	$Tz$	-	-	1	×	✓	×
	Add./Mul.	$m + z\phi$	$z$	$z\phi$	$z\phi$				
[15]	Exp.	$m$	-	$z\phi$	$z\phi$	1	×	✓	×
	Add./Mul.	-	-	$z\phi$	$z\phi$				
	Hash	$m$	-	1	1				
	ZK proof	-	-	$z\phi$	$z\phi$				
RC-PoR-P	Hash	$m$	-	$z\phi \log_2(m)$	$z\phi \log_2(m)$	$\phi \log_2(m)$	✓	✓	✓
	Sym. enc.	-	-	$z\phi \log_2(m)$	$z\phi \log_2(m)$				

Briefly, the initiation phase in the RC-PoR-P is more efficient than the one in [3,15]. Specifically, in the RC-PoR-P, this phase requires  $O(m)$  hash function invocations taking at most 1596.6 seconds for a 4-GB file (or  $2^{28}$  blocks). But, this phase requires  $O(m)$  exponentiations in [15], and requires  $O(m + z\phi)$  addition/multiplication and  $O(z)$  exponentiations in [3]. The prove and verify in the RC-PoR-P is faster than the scheme in [15]. Specifically, in the RC-PoR-P, these phases require  $\mathcal{S}$ ,  $\mathcal{C}$ , and the arbiter to invoke at most  $O(z\phi \log_2(m))$  symmetric key encryption and hash function. The prove, verify, and resolve take  $\mathcal{S}$ ,  $\mathcal{C}$ , and the arbiter  $1320.7 \cdot z$ ,  $0.24 \cdot z$ , and  $10^{-4} \cdot z'$  seconds respectively. Nevertheless, in [15], the prove and verify require  $\mathcal{S}$  and  $\mathcal{C}$  to do  $O(z\phi)$  exponentiations. In [3], these two phases have a better complexity than those in the RC-PoR-P. But, in [3],  $\mathcal{S}$  continuously performs exponentiations for solving puzzles until all  $z$  verifications are complete. Such very costly operations are not needed in the RC-PoR-P. The schemes in [3,15] have a better proof size complexity than the RC-PoR-P has. Furthermore, the RC-PoR-P offers (i) security against malicious  $\mathcal{S}$  and  $\mathcal{C}$ , and (ii) privacy, unlike the schemes in [3,15] that cannot offer them. Thus, the RC-PoR-P is computationally more efficient and offers stronger security than those in [3,15]. In Appendix P, we provide a full analysis.

## 8 Conclusion

Fair exchange protocols are interesting solutions to various real-world problems. At CCS 2017, Campanelli *et al.* proposed two schemes for the fair exchange of digital coins and a certain service, i.e., proofs of retrievability. To date, these schemes are the only ones designed to support a digital service. In this work, we identified two serious issues of these schemes; namely, (1) waste of the seller’s resources, and (2) real-time leakage. We also identified flaws in these protocols’ design and definition. To rectify these issues, we proposed “recurring contingent PoR payment” (RC-PoR-P) that remains secure even if the fair exchange reoccurs. We implemented the RC-PoR-P. Our analysis indicates that the RC-PoR-P is more efficient than the state of the art, while offering stronger security. Future research can investigate the design of protocols for the fair exchange of coins and other vital (verifiable) services, e.g., verifiable computation [26], verifiable searchable encryption [41], or verifiable private information retrieval [50].

## Acknowledgments

Aydin Abadi is supported by REPHRAIN: The National Research Centre on Privacy, Harm Reduction and Adversarial Influence Online, under UKRI grant EP/V011189/1. Steven J. Murdoch is supported by The Royal Society under grant UF160505.

## References

1. Abadi, A.: Off-chain source code of “recurring contingent proofs of retrievability payment” (RC-PoR-P) (2021), <https://github.com/AydinAbadi/RC-S-P/blob/main/RC-PoR-P-Source-cod/RC-PoR-P.cpp>
2. Abadi, A.: On-chain source code of “recurring contingent proofs of retrievability payment” (RC-PoR-P) (2021), <https://github.com/AydinAbadi/RC-S-P/blob/main/RC-PoR-P-Source-cod/RC-PoR-P-Smart-Contract.sol>
3. Abadi, A., Kiayias, A.: Multi-instance publicly verifiable time-lock puzzle and its applications. In: , FC (2021)
4. Amazon: Amazon s3 pricing (2021), <https://aws.amazon.com/s3/pricing/>
5. Androulaki, E., Karame, G., Roeschlin, M., Scherer, T., Capkun, S.: Evaluating user privacy in bitcoin. In: FC (2013)
6. Armknecht, F., Bohli, J.M., Karame, G.O., Liu, Z., Reuter, C.A.: Outsourced proofs of retrievability. In: CCS (2010)
7. Asokan, N., Schunter, M., Waidner, M.: Optimistic protocols for fair exchange. In: CCS (1997)
8. Asokan, N., Shoup, V., Waidner, M.: Optimistic fair exchange of digital signatures (extended abstract). In: EUROCRYPT (1998)
9. Ateniese, G., Burns, R.C., Curtmola, R., Herring, J., Kissner, L., Peterson, Z.N.J., Song, D.X.: Provable data possession at untrusted stores. In: CCS (2007)
10. Bao, F., Deng, R.H., Mao, W.: Efficient and practical fair exchange protocols with off-line TTP. In: S&P (1998)

11. Barber, S., Boyen, X., Shi, E., Uzun, E.: Bitter to better - how to make bitcoin a better currency. In: FC (2012)
12. Bary, E.: Zoom stock falls after service outage (2020), <https://www.marketwatch.com/story/zoom-stock-falls-amid-service-outage-2020-08-24>
13. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: S&P (2014)
14. Boneh, D., Naor, M.: Timed commitments. In: CRYPTO (2000)
15. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS (2017)
16. Canetti, R., Riva, B., Rothblum, G.N.: Practical delegation of computation using multiple servers. In: CCS (2011)
17. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: STOC (1986)
18. Dong, C., Chen, L., Camenisch, J., Russello, G.: Fair private set intersection with a semi-trusted arbiter. In: DBSec (2013)
19. Dropbox: Choose the right dropbox for you (2021), <https://www.dropbox.com/plans?tab=personal>
20. Du, Y., Duan, H., Zhou, A., Wang, C., Au, M.H., Wang, Q.: Enabling secure and efficient decentralized storage auditing with blockchain. TDSC (2021)
21. Dziembowski, S., Eckey, L., Faust, S.: Fairswap: How to fairly exchange digital goods. In: CCS (2018)
22. Eckey, L., Faust, S., Schlosser, B.: Optiswap: Fast optimistic fair exchange. In: ASIA CCS (2020)
23. Fuchsbauer, G.: WI is not enough: Zero-knowledge contingent (service) payments revisited. In: CCS (2019)
24. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: FC (2002)
25. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: EUROCRYPT (2015)
26. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: CRYPTO (2010)
27. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcs. In: EUROCRYPT (2013)
28. GoogleOne: Upgrade to a plan that works for you (2021), [https://one.google.com/about/plans?hl=en\\_GB](https://one.google.com/about/plans?hl=en_GB)
29. Halevi, S., Harnik, D., Pinkas, B., Shulman-Peleg, A.: Proofs of ownership in remote storage systems. In: CCS (2011)
30. Haselton, T.: Slack service goes down for more than three hours (2021), <https://www.cnn.com/2021/01/04/slack-outage-on-first-monday-of-2021.html>
31. Hu, C., Cheng, X., Tian, Z., Yu, J., Lv, W.: Achieving privacy preservation and billing via delayed information release. IEEE/ACM Transactions on Networking (2021)
32. Ishai, Y., Ostrovsky, R., Zikas, V.: Secure multi-party computation with identifiable abort. In: CRYPTO (2014)
33. Juels, A., Jr., B.S.K.: Pors: Proofs of retrievability for large files. In: CCS (2007)
34. Kalodner, H.A., Goldfeder, S., Chen, X., Weinberg, S.M., Felten, E.W.: Arbitrum: Scalable, private smart contracts. In: USENIX Security (2018)
35. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)
36. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: S&P (2016)

37. Lab, K.: Understanding security of the cloud: from adoption benefits to threats and concerns. Kaspersky daily (2018), <https://www.kaspersky.com/blog/understanding-security-of-the-cloud>
38. Maxwell, G.: Zero knowledge contingent payment (2011)
39. Merkle, R.C.: Protocols for public key cryptosystems. In: S&P (1980)
40. Merkle, R.C.: A certified digital signature. In: CRYPTO (1989)
41. Miao, Y., Tong, Q., Deng, R., Choo, K.R., Liu, X., Li, H.: Verifiable searchable encryption framework against insider keyword-guessing attack in cloud storage. IEEE Transactions on Cloud Computing (2020)
42. Miller, A., Juels, A., Shi, E., Parno, B., Katz, J.: Permacoin: Repurposing bitcoin work for data preservation. In: S&P'14
43. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2019)
44. Nguyen, K., Ambrona, M., Abe, M.: WI is almost enough: Contingent payment all over again. In: CCS (2020)
45. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO (1991)
46. Reid, F., Harrigan, M.: An analysis of anonymity in the bitcoin system. In: PASSAT (2011)
47. Shacham, H., Waters, B.: Compact proofs of retrievability. In: ASIACRYPT (2008)
48. Tramèr, F., Zhang, F., Lin, H., Hubaux, J., Juels, A., Shi, E.: Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In: EuroS&P (2017)
49. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2014)
50. Zhang, L.F., Safavi-Naini, R.: Verifiable multi-server private information retrieval. In: ACNS (2014)

## A Survey of Related Work

As stated in the introduction, blockchain technology and in particular smart contracts have the potentials to replace the third party in fair exchange protocols. Ethereum is the most predominant generic smart contract platform. Although the third-party’s role can be directly encoded/programmed in an Ethereum smart contract, it would not be efficient. Moreover, Bitcoin, as the most popular cryptocurrency, supports smart contracts with very limited functionalities. Therefore, the third party’s full role cannot be directly encoded in a contract on the Bitcoin blockchain.

### A.1 Zero-knowledge Contingent Payment

For the first time in [38] it was shown how to construct a fair exchange protocol, called “zero-knowledge contingent payment”, that utilises Bitcoin’s smart contract capabilities. The protocol allows a fair exchange of digital goods and payments over Bitcoin’s network. Its main security requirement is that a seller is paid if and only if a buyer learns a correct secret. The protocol uses a feature of Bitcoin’s scripting language, called “hash-lock transaction”. This type of transaction lets one create a payment transaction that specifies a hash value  $y$  and allows anyone that can provide its preimage  $k$ , i.e  $H(k) = y$ , to claim the amount of coin specified in the transaction. The contingent payment protocol in [38] works as follows. The seller first picks a secret key,  $k$ , of a symmetric-key encryption and uses it to encrypt the secret information,  $s$ . This yields a ciphertext,  $c$ . It also computes the key’s hash,  $y = H(k)$ . The seller sends  $c, y$ , and a (zero-knowledge) proof to the buyer, where the proof asserts that  $c$  is the encryption of  $s$  under key  $k$  and  $H(k) = y$ .

After the buyer verifies and accepts the proof, it sends a transaction to the blockchain that pays the seller a fixed amount of coin if the seller provides, to the blockchain, a value  $k$  such that  $H(k) = y$ . Next, the seller sends  $k$  to the blockchain and receives the coins. Now, the buyer can read the blockchain and learn  $k$  which allows it to decrypt  $c$ , and extract the secret,  $s$ . Later, after the advancement of the “succinct non-interactive argument of knowledge” (zk-SNARK) [27], that results in a more efficient implementation of zero-knowledge proofs, the contingent payment protocol was modified to use zk-SNARK. However, all zk-SNARKs require a trusted third party for a trusted setup, i.e., to generate a “common reference string” (CRS), which means there would be a need for the involvement of an additional third party in those protocols that use them, including the contingent payment protocol. As such involvement is undesirable, the contingent payment protocol, that uses zk-SNARK, lets the buyer play the role of the third party and generate the parameter.

### A.2 Zero-knowledge Contingent Service Payment

Later, Campanelli *et al.* [15] identify a serious security issue of the above contingent payment (that uses zk-SNARK and lets a buyer pick a CRS). In particular,

the authors show that a malicious buyer (which generates the CRS) can construct the CRS in a way that lets it learn the secret from the seller’s proof without paying the seller. Campanelli *et al.* propose a set of fixes; namely, (a) jointly computing the CRS using a secure two-party computation, (b) allowing the seller to check the well-formedness of the buyer’s CRS, or (c) using a new scheme called “zero-knowledge Contingent Service Payments” (zkCSP). The latter solution is a more efficient approach than the other two and offers an additional interesting feature; namely, supporting contingent payment for *digital (verifiable) services*. In short, zkCSP works as follows. Let  $v(\cdot)$  be the verification algorithm for a certain service and  $s$  be the service’s proof, where if the proof is valid it holds that  $v(s) = 1$ . The parties agree on two claw-free hash functions, e.g.,  $H_1(\cdot)$  and  $H_2(\cdot)$ . The seller picks a random value,  $r$ . Then, it computes either  $y = H_1(r)$  if it knows  $s$  such that  $v(s) = 1$ , or  $y = H_2(r)$  otherwise. The seller also generates a witness indistinguishable proof of knowledge (WIPoK),  $\pi$ , using a compound sigma protocol to prove that it knows either the preimage of  $y = H_1(r)$  if it knows a valid  $s$ , i.e.,  $v(s) = 1$ , or the preimage of  $y = H_2(r)$ . Note, due to the witness indistinguishability of  $\pi$  and the flaw-freeness of the hash functions, the verifier cannot tell which statement the prover is proving.

The seller sends the proof along with  $y$  to the buyer which first ensures  $\pi$  is valid. Then, if the check passes, the buyer sends to the blockchain a hash-lock transaction that would send  $n$  coins to the party that can provide  $r$  to the blockchain such that  $y = H_1(r)$ . After a seller provides a valid  $r$  to the blockchain it gets paid, accordingly the buyer concludes that it has been served honestly by the seller, as the seller demonstrated the knowledge of the service proof,  $s$ . Otherwise (if the seller does not provide a valid  $r$ ) it would not get paid and the buyer learns nothing about  $s$ . To improve the efficiency of the above zkCSP and to make it practical, the authors suggest using SNARKs in the setting that the buyer generates the CRS but the seller initially performs minimal efficient checks. Also, as concrete instantiations of the zkCSP, the authors propose two schemes in which the service is “proof of retrievability” (PoR) [47]. One of the schemes relies on a publicly verifiable PoR and the other one relies on a privately verifiable one. In these schemes, the buyer uploads its data to a server and pays if and only if the server provides valid proof that asserts the buyer’s data is retrievable.

### A.3 Known Zero-knowledge Contingent (Service) Payment’s Flaw in the Literature

Fuchsbauer [23] identifies a flaw in the above zkCSP. The author shows that the minimal efficient check that the seller performs in the zkCSP is not sufficient, because it does not prevent the buyer from cheating and learning the secret. He highlights that the use of computationally expensive verification on the CRS is inevitable to address the issue. Very recently, Nguyen *et al.* [44] show that by relying on a slightly stronger notion of WI (i.e., trapdoor subversion witness indistinguishability), the zkCSP can remain secure and would not be susceptible to the issues Fuchsbauer pointed out. Moreover, they propose an efficient scheme

that relies on an *interactive* ZK proof system which is based on garbled circuits and oblivious transfer. However, the above two issues, we highlighted in Section 4, are not identified and addressed in [23,44].

#### A.4 Using Ethereum Smart Contracts in Contingent Payment

Tramer *et al.* [48] propose a fair exchange scheme that uses a combination of trusted hardware, i.e., Intel SGX, and Ethereum smart contracts. Interestingly, unlike the common assumption that secure hardware maintains private states, this scheme relies on weaker security assumptions, i.e., it only relies on the integrity of SGX's computation and the authenticity of a message it sends. At a high level, in this scheme, the buyer and seller agree on a smart contract and then the buyer deposits a fixed amount of coin in the smart contract. Then, the seller sends its messages (that contains proofs) to SGX which verifies the messages' correctness and then sends its verdict to the smart contract. Next, the contract distributes the deposit according to the SGX's verdicts. The scheme in addition to achieving fair exchange wants to ensure that after the parties' initial interaction and after the seller makes an offer, the buyer cannot abort without paying the seller. To this end, in the scheme, the contract needs to validate SGX's signature (or in general attestation). However, as the authors state, in practice the signature scheme used in SGX (i.e., EPID signature) is not supported by standard Ethereum contracts. Therefore, the suggested technique, to ensure the buyer cannot abort, remains only of theoretical interest. Also, in the protocol SGX is always involved, regardless of the parties' behaviour.

Later, Dziembowski *et al.* [21] propose FairSwap, an efficient protocol for a fair exchange of digital goods (i.e., files) and coins. It is mainly based on the Ethereum smart contracts and the notion of proof of misbehaviour [16]. Briefly, a proof of misbehaviour scheme is usually based on a Merkle tree; in this scheme, proving that a party has misbehaved is much cheaper than proving it has behaved honestly. FairSwap offers two main features: (a) imposes a low computation cost to a smart contract, and (b) avoids using zero-knowledge proofs. At a high level, FairSwap works as follows. First, the seller and buyer agree on a smart contract. Then, the seller picks a key  $k$  (for symmetric-key encryption), encrypts the secret (i.e., file) under  $k$ , and sends the ciphertext to the buyer. The seller also commits to  $k$  and sends the commitment to the smart contract. Next, the buyer verifies the correctness of the buyer's messages and if approved, it sends a fixed amount of coin to the smart contract. After that, the seller reveals the opening of the commitment, that contains  $k$ , to the smart contract. This allows the buyer to read from the contract and learn  $k$  with which it can decrypt the ciphertext, extract the secret, and then verify the secret's correctness. In the case where the buyer rejects the secret, it can send a short proof (of misbehaviour) to the contract which performs an efficient verification and distributes the deposit according to the verification's result.

Very recently, Eckey *et al.* [22] propose OPTISWAP that improves FairSwap's performance. It also ensures a malicious seller cannot force the buyer to submit a large transaction to the blockchain, which ultimately imposes transaction

costs to the buyer, i.e., the grieving attack. Similar to FairSwap, OPTISWAP uses a smart contract and proof of misbehaviour. Nevertheless, to achieve a better efficiency (than FairSwap), OPTISWAP uses an *interactive* dispute resolution protocol, previously proposed and used in [34]. The interactive phase is a challenge-response procedure between the two parties and lets an honest buyer efficiently generate proof of misbehaviour. After computing the proof, the buyer sends it to the contract which verifies the proof and distributes the deposit according to the verification result. To prevent the grieving attack, the protocol requires the seller to deposit coins to the contract as well, which allows the contract to compensate an honest buyer which reports the seller's misbehaviour.

We highlight that the protocols in [21,22,48] have been designed and are suitable for a fair exchange of digital items, e.g., file, and digital coins. Nevertheless, they are not suitable for verifiable services, e.g., PoR. If they are *directly* used for verifiable services, then they would suffer from the two issues we stated in Section 4 (i.e., a malicious client can waste an honest server's resources and lack of privacy). For instance, if they are naively used for PoR, then a malicious client (as a buyer) can simply avoid engaging in the payment protocol with the server (as a seller), even though the server has honestly maintained the buyer's data. This means the client can waste the server's resources. This issue would not be fully addressed by simply forcing the client to deposit coins at the point where it outsources its data. Because, the client can encode its data in a way that makes the server compute an invalid proof, that ultimately allows the client to withdraw its deposit and avoid paying the server. Moreover, the amount of deposit leaks non-trivial information about the secret (or the file in the PoR context) in real-time to the public.

Very recently, outsourced (fair) PoR schemes that allow a client to delegate the verifications to a smart contract have been proposed in [3,20]. The scheme in [3] uses message authentication code (MAC) and time-lock puzzle that results in low cost in the proof generation and verification phases while the one in [20] is based on polynomial commitment and involves a high number of modular exponentiations that lead to higher proof generation and verification cost than the former scheme. The schemes in [3,20] do not address the above privacy issue either and rely on a stronger security assumption than the rest of the work studied in this section, as these two protocols assume the client is fully honest while the rest assume either party can be corrupt.

## B Notations

We summarise our notations in Table 4.

Table 4: Notation Table.

Setting	Symbol	Description	Setting	Symbol	Description
Generic	$z$	Number of verifications	Generic	$\ u^*\ $	Bit size of $u^*$
	$\lambda$	Security parameter		$j$	Verification index, $1 \leq j \leq z$
	PRF	Pseudorandom function		$adr$	Address
	$\zeta$	PRF's description		$\phi$	Number of challenged blocks
	$Pr$	Probability	RC-PoR-P	$r_{qp}, r_{cp}$	Random values
	Com	Commitment's commit		$\tilde{x}_{cp}$	$\tilde{x}_{cp} := (cp, r_{cp})$
	Ver	Commitment's verify		$\tilde{x}_{qp}$	$\tilde{x}_{qp} := (qp, r_{qp})$
	$\mu$	Negligible function		$coin_C^*, coin_S^*$	Encoded coins deposited by $C$ and $S$
	H	Hash function		$enc$	Encode/decode functions $enc := (E, D)$
	MT	Merkle tree		$m_C, m_S$	Complaints of $C$ and $S$
	$sk, pk$	Secret and public keys		$pad_\pi, pad_q$	Number of elements used to pad $\pi$ and $q$
	$u$	File		$y_C, y_S$	Number of times $C$ and $S$ misbehave
	$u^*$	Encoded file		$y'_C, y'_S$	Number of times $\mathcal{A}$ is unnecessarily invoked
	$\sigma$	Metadata		$cp$	Coin secret parameters
	$e$	$e := (\sigma, \omega_\sigma)$		$T_{cp}$	Coin encoding token
	$pp$	Public parameter		$qp$	Query/proof secret parameters
	$q, \hat{k}$	Query		$T_{qp}$	Query/proof encoding token
	$\pi$	Proof vector		$T$	$T := (T_{cp}, T_{qp})$
	$C$	Client		$g_C, g_S$	Commitments computed by $C$ and $S$
	$S$	Server		$pl$	Price list: $\{(o, l), \dots, (o', l')\}$
$\mathcal{A}$	Arbiter	$o$	Coins $S$ receives for a valid proof		
SC	Smart contract	$l$	Coins $\mathcal{A}$ must get for resolving a dispute		
$M$	Metadata gen. function	$l_{max}$	$Max(l, \dots, l')$		
$Q$	Query ver. function	$o_{max}$	$Max(o, \dots, o')$		
$m$	Number of blocks, $m =  u^* $	$ps$	Total coins $S$ should deposit		

## C Merkle Tree

In the setting where a Merkle tree is used to remotely check a file, the file is split into blocks and the tree is built on top of the file blocks. Usually, for the sake of simplicity, it is assumed the number of blocks,  $m$ , is a power of 2. The height of the tree, constructed on  $m$  blocks, is  $\log_2(m)$ . The Merkle tree scheme includes three algorithms (`MT.genTree`, `MT.prove`, `MT.verify`) as follows:

- The algorithm that constructs a Merkle tree, `MT.genTree`, is run by  $\mathcal{V}$ . It takes file blocks,  $u := u_1, \dots, u_m$ , as input. Then, it groups the blocks in pairs. Next, a collision-resistant hash function,  $H(\cdot)$ , is used to hash each pair. After that, the hash values are grouped in pairs and each pair is further hashed, and this process is repeated until only a single hash value, called “root”, remains. This yields a tree with the leaves corresponding to the blocks of the input file and the root corresponding to the last remaining hash value.  $\mathcal{V}$  locally stores the root, and sends the file and tree to  $\mathcal{P}$ .

- The proving algorithm, `MT.prove`, is run by  $\mathcal{P}$ . It takes a block index,  $i$ , and a tree as inputs. It outputs a vector proof, of  $\log_2(m)$  elements. The proof asserts the membership of  $i$ -th block in the tree, and consists of all the sibling nodes on a path from the  $i$ -th block to the root of the Merkle tree (including  $i$ -th block). The proof is given to  $\mathcal{V}$ .
- The verification algorithm, `MT.verify`, is run by  $\mathcal{V}$ . It takes as input  $i$ -th block, a proof and tree's root. It checks if the  $i$ -th block corresponds to the root. If the verification passes, it outputs 1; otherwise, it outputs 0.

The Merkle tree-based scheme has two properties: *correctness* and *security*. Informally, the correctness requires that if both parties run the algorithms correctly, then a proof is always accepted by  $\mathcal{V}$ . The security requires that a computationally bounded malicious  $\mathcal{P}$  cannot convince  $\mathcal{V}$  into accepting an incorrect proof, e.g., proof for non-member block. The security relies on the assumption that it is computationally infeasible to find the hash function's collision.

## D PoR's Definition

A PoR scheme has two properties: *correctness* and *soundness*. Correctness requires that the verification algorithm accepts proofs generated by an honest verifier; formally, PoR requires that for any key  $k$ , any file  $u \in \{0, 1\}^*$ , and any pair  $(u^*, \sigma)$  output by `PoR.setup` $(1^\lambda, u, k)$ , and any query  $\mathbf{q}$ , the verifier accepts when it interacts with an honest prover. Soundness requires that if a prover convinces the verifier (with high probability) then the file is stored by the prover. This is formalized via the notion of an extractor algorithm, that is able to extract the file in interaction with the adversary using a polynomial number of rounds. Before we define soundness, we restate the experiment, defined in [47], that takes place between an environment  $\mathcal{E}$  and adversary  $\mathcal{A}$ . In this experiment,  $\mathcal{A}$  plays the role of a corrupt party and  $\mathcal{E}$  simulates an honest party's role.

1.  $\mathcal{E}$  executes `PoR.keyGen` $(1^\lambda)$  algorithm and provides public key,  $pk$ , to  $\mathcal{A}$ .
2.  $\mathcal{A}$  can pick arbitrary file  $u$ , and uses it to make queries to  $\mathcal{E}$  who runs `PoR.setup` $(1^\lambda, u, k) \rightarrow (u^*, \sigma, pp)$  and returns the output to  $\mathcal{A}$ . Also, upon receiving the output of `PoR.setup` $(1^\lambda, u, k)$ ,  $\mathcal{A}$  can ask  $\mathcal{E}$  to run `PoR.genQuery` $(1^\lambda, k, pp) \rightarrow \mathbf{q}$  and give the output to it.  $\mathcal{A}$  can locally run `PoR.prove` $(u^*, \sigma, \mathbf{q}, pk, pp) \rightarrow \pi$  to get its outputs as well.
3.  $\mathcal{A}$  can request from  $\mathcal{E}$  the execution of `PoR.verify` $(\pi, \mathbf{q}, k, pp)$  for any  $u$  used to query `PoR.setup` $(\cdot)$ . Accordingly,  $\mathcal{E}$  informs  $\mathcal{A}$  about the verification output. The adversary can send to  $\mathcal{E}$  a polynomial number of queries. Finally,  $\mathcal{A}$  outputs metadata  $\sigma$  returned from a setup query and the description of a prover,  $\hat{\mathcal{A}}$ , for any file it has already chosen above.

It is said that a cheating prover,  $\hat{\mathcal{A}}_\epsilon$ , is  $\epsilon$ -admissible if it convincingly answers  $\epsilon$  fraction of verification challenges (for a certain file). Informally, a PoR scheme supports extractability, if there is an extractor algorithm `Ext` $(k, \sigma, \hat{\mathcal{A}}_\epsilon)$ , that takes as input the key  $k$ , metadata  $\sigma$ , and the description of the machine implementing

the prover’s role  $\hat{\mathcal{A}}_\epsilon$  and outputs the file,  $u$ . The extractor has the ability to reset the adversary to the beginning of the challenge phase and repeat this step polynomially many times for the purpose of extraction, i.e., the extractor can rewind  $\hat{\mathcal{A}}_\epsilon$ . Therefore, the above experiment, with regards to an  $\epsilon$ -admissible adversary, can be written as  $\text{Exp}_{\text{PoR}}^{\mathcal{A}}$  presented below:

```

ExpPoRA(1λ) :

PoR.keyGen(1λ) → k := (sk, pk)
A(1λ, pk) → u
PoR.setup(1λ, u, k) → (u*, σ, pp)
A(u*, σ, pp) → state
PoR.genQuery(1λ, k, pp) → q
((A(q, state) → π) = (PoR.verify(π, q, k, pp))) → Aε

```

**Definition 2** ( $\epsilon$ -soundness). *A PoR scheme is  $\epsilon$ -sound if there exists an extraction algorithm  $\text{Ext}(\cdot)$  such that, for every adversary  $\mathcal{A}$  who plays experiment  $\text{Exp}_{\text{PoR}}^{\mathcal{A}}$  and outputs an  $\epsilon$ -admissible cheating prover  $\hat{\mathcal{A}}_\epsilon$  for a file  $u$ , the extraction algorithm recovers  $u$  from  $\hat{\mathcal{A}}_\epsilon$ , given honest party’s private key, public parameters, metadata and the description of  $\hat{\mathcal{A}}_\epsilon$ , except with a negligible probability in the security parameter  $\lambda$ :*

$$\Pr [\text{Ext}(k, pp, \sigma, \hat{\mathcal{A}}_\epsilon) \neq u : \text{Exp}_{\text{PoR}}^{\mathcal{A}}] \leq \mu(\lambda).$$

In contrast to the PoR definition in [33,47] where  $\text{PoR.genQuery}(\cdot)$  is implicit, in the above definition we have explicitly defined  $\text{PoR.genQuery}(\cdot)$ , as it plays an important role in this paper. Also, there are PoR protocols, e.g., in [42], that do not involve  $\text{PoR.keyGen}(\cdot)$ . Instead, a set of public parameters/keys (e.g., file size or a root of Merkle tree) are output by  $\text{PoR.setup}(\cdot)$ . To make the PoR definition generic to capture both cases, we have explicitly included the public parameters  $pp$  in the algorithms’ definitions too.

## E Further Discussion of Issue 1

In this section, we provide a concrete example of how a malicious client can make the server provide an invalid proof in the zkCSP in [15]. Recall, the zkCSP schemes offer privately and publicly verifiable PoR [47] as digital (verifiable) services. Without loss of generality, we will focus on the privately verifiable one. The idea is that the client can generate a tag of the file block in a way that the server cannot generate a valid proof when that block is probed (in the verification phase). In this PoR scheme, at the setup, the malicious client instead of honestly generating a tag  $\sigma_i$  on a file block  $m_i$  as  $\sigma_i = r_i + \alpha \cdot m_i$ , it generates an invalid tag as:  $\sigma'_i = r_i + \alpha \cdot m'_i$ , where  $r_i$  and  $\alpha$  are two random values and  $m_i \neq m'_i$ . In this case, given the file blocks  $\{m_1, \dots, m_i, \dots, m_n\}$  and a set of maliciously generated tag that includes  $\sigma'_i$ , the honest server cannot pass the verification when block  $m_i$  is probed. Hence, in the zkCSP, it would not get paid. In this case, the client will not be detected, as neither the zkCSP nor the PoR offers any mechanism to let the server check the correctness of each tag.

## F Further Discussion of Issue 2

In this section, we elaborate on the two types of information leakages from which the schemes in [15] suffer.

### F.1 Proofs' Status Leakage

In the traditional setting, the client and server directly interact with each other to verify and prove the integrity of agreed-upon services, e.g., PoR. In this case, the verification's result is only apparent to the two parties. Nevertheless, in the blockchain era, where a blockchain plays a role in the verification and payment phases, e.g., in the zkCSP schemes, it becomes visible in *real-time* to *everyone* whether the verification (proof) has been accepted, that reflects if the server has successfully delivered the agreed-upon service or failed to do so, i.e., violation of "service level agreement" (SLA). This issue remains even if the service proofs are not stored in plaintext in the blockchain, as *coins transfer* itself reveals the proofs' status. Therefore, in this setting, the blockchain provides to the public first-hand up-to-date information about the servers' status that could have not been easily attained otherwise.

This leakage can have *serious immediate* consequences for both the server and (business) clients, e.g., stock value drop [12,30], or opening doors for attackers to exploit such incidents. As an example, observing the proof's verification outputs (when a server deals with multiple clients) allows a malicious client to immediately construct comprehensive background knowledge of the server's current behaviour and status, e.g., the server has been acting honestly. Such auxiliary information can assist the malicious client to more wisely exploit the above deposit issue (that can avoid sending the deposit). For instance, when the server always acts honestly towards its clients, the malicious client refuses to send the deposit and still has high confidence that the server delivered the service. As another example, in the case of PoR, a malicious observer can simply find out that the service is suffering from (hardware or software) failure and exploit such vulnerability to harm the parties, e.g., mount social engineering attacks on clients or penetrate to the server. A survey conducted by Kaspersky lab (in 2018) suggests that 33% of attacks that affect business clients of cloud computing servers are of type social engineering [37] which is a high rate. Given real-time evidence of servers' failure, attackers as a part of social engineering can provide more convincing evidence to their victims. This ultimately increases the attackers' chance of success.<sup>7</sup>

### F.2 Deposit's Amount Leakage

The amount of deposit placed in the contract, swiftly leaks non-trivial information about the client to the public. In the case of PoR, an observer can learn the approximate size of outsourced file, service type, or in certain cases even the region of clients' outsourced file, by comparing the amount of deposit with the service provider's price list, which is usually publicly available, e.g., in [4,19,28]. For instance, at the time of writing this paper, the "Amazon S3 One Zone - Infrequent Access" monthly price is \$0.0208 per GB if the data is stored in "South America (Sao Paulo)" [4]. Interestingly,

---

<sup>7</sup> An adversary can target a large set of people a subset of which is likely to be the cloud's clients (akin to the phishing attack), or it can target specific cloud's clients by using the techniques used to link the blockchain's addresses to certain parties [5,11,46].

that region has a unique price. Hence, if the client deposits about \$208 in cryptocurrency in the contract, then the public knows that the client has outsourced about 10000 GB data, using Amazon S3 One Zone and its data location is Sao Paulo.

## G Strawman Solutions for the Two Issues

In this section, we discuss why naive solutions are not suitable to address the two issues that we highlighted in Section 4, i.e., (1) a malicious client can waste the seller’s resources, and (2) real-time information leakage.

To address Issue 1, one may slightly adjust each zkCSP protocol such that it would require the client to deposit coins long before the server provides the (ZK) proof to it, with the hope that the client cannot avoid depositing after the server provides proofs. Nevertheless, this would not work, as the client after accepting the proofs, needs to send a confirmation message/transaction to the contract. In this setting, a malicious client can avoid doing so or make the server compute invalid (PoR) proofs, that ultimately allows the client to get its deposit back. Alternatively, one may let a smart contract perform the verification on the client’s behalf, such that the client deposits its coins in the contract when it starts using the service. Then, the server sends its proof to the contract which performs the verification and pays the server if the proof is accepted. Even though this approach would address (only) Issue 1, it imposes a high cost and defeats the purpose of the zkCSP’s design. Because the contract has to *always* be involved to run the verification algorithm that has to be a publicly verifiable one, which usually imposes a high (computation or communication) cost. To address Issue 2, one may use privacy-preserving cryptocurrency frameworks, e.g., Zerocash [13] or Hawk [36]. Although such frameworks partially address this issue (i.e., they can hide deposit amount but not proofs’ status), they impose an additional high cost to their users, as each transaction involves a generic (zk) proofs system that are computationally expensive. Moreover, one might want to let the server pick a fresh address for each verifier/verification, to preserve its pseudonymity with the hope that an observer cannot link clients to a server (so issues 1 and 2 can be addressed). However, for this to work, we have to assume that multiple service providers use the same protocol on the blockchain and all of them are pseudonymous. This is a strong assumption and may not be always feasible.

## H Verifiable PoR with Identifiable Abort (PoRID) Definition

A protocol that realises only the PoR’s definition, would be merely secure against a malicious server and assumes the client is honest. Although this assumption would suffice in certain settings and has been used before (e.g., in [3,33,47]), it is rather strong and not suitable for real-world settings, especially when there are monetary incentives (e.g., service payment) that encourage a client to misbehave. Therefore, we enhance the PoR’s definition to allow (a) either party to be malicious and (b) a trusted third party, *arbiter*, to identify a corrupt party. We call an upgraded verifiable service scheme with these features “verifiable service with identifiable abort” (PoRID), inspired by the notion of secure multi-party computation with identifiable abort [32]. Below, we present a formal generic definition of the PoRID which is generic and suitable for any PoR scheme.

**Definition 3 (PoRID Scheme).** A PoR with identifiable abort  $\text{PoRID} := (\text{PoRID.keyGen}, \text{PoRID.setup}, \text{PoRID.serve}, \text{PoRID.genQuery}, \text{PoRID.checkQuery}, \text{PoRID.prove}, \text{PoRID.verify}, \text{PoRID.identify})$  involves four entities; namely, client, server, arbiter, and bulletin board (e.g., smart contract). It consists of eight algorithms defined below.

- $\text{PoRID.keyGen}(1^\lambda) \rightarrow k := (sk, pk)$ . A probabilistic algorithm run by the client,  $C$ . It takes as input the security parameter  $1^\lambda$ . It outputs a secret/public verification key pair  $k$ . It sends  $pk$  to the bulletin board.
- $\text{PoRID.setup}(1^\lambda, u, k, M) \rightarrow (u^*, e, pp)$ . It is run by  $C$ . It takes as input the security parameter  $1^\lambda$ , file  $u$ , key pair  $k$ , and metadata generator deterministic function  $M$ . It encodes  $u$ , that results  $u^*$ . It outputs  $u^*$ , public parameters  $pp$ , and  $e := (\sigma, w_\sigma)$ , where  $\sigma = M(u^*, k, pp)$  is metadata and  $w_\sigma$  is a proof asserting the metadata is well-structured. It sends the output (i.e.,  $u^*, e, pp$ ) to the bulletin board.
- $\text{PoRID.serve}(u^*, e, pk, pp) \rightarrow a \in \{0, 1\}$ . It is run by the server,  $S$ . It takes as input the encoded file  $u^*$ , the pair  $e := (\sigma, w_\sigma)$ , public key  $pk$ , and public parameters  $pp$ . If the metadata is well-formed (i.e., the proof  $w_\sigma$  is accepted), it outputs  $a = 1$ . Otherwise, it outputs  $a = 0$ . The output is sent to the board.
- $\text{PoRID.genQuery}(1^\lambda, Q, pp) \rightarrow \mathbf{q}$ . A probabilistic algorithm run by  $C$ . It takes as input the security parameter  $1^\lambda$ , a deterministic function  $Q$  that ensures queries are well-structured, and public parameters  $pp$ . It generates a query vector  $\mathbf{q}$ , and ensures it is well-structured, i.e.,  $Q(\mathbf{q}, pp) = 1$ . It sends  $\mathbf{q}$  to the board.
- $\text{PoRID.checkQuery}(\mathbf{q}, pk, pp) \rightarrow b \in \{0, 1\}$ . It is run by  $S$ . It takes as input the query  $\mathbf{q}$ , public key  $pk$ , and public parameters  $pp$ . It checks if the query is well structured. If the check is passed, it outputs  $b = 1$ . Otherwise, it outputs  $b = 0$ .
- $\text{PoRID.prove}(u^*, \sigma, \mathbf{q}, pk, pp) \rightarrow \pi$ . It is run by  $S$ . It takes as input the encoded file  $u^*$ , metadata  $\sigma$ , query  $\mathbf{q}$ , public key  $pk$ , and public parameters  $pp$ . It outputs a proof  $\pi$ . It sends  $\pi$  to the board.
- $\text{PoRID.verify}(\pi, \mathbf{q}, k, pp) \rightarrow d \in \{0, 1\}$ . It is run by  $C$ . It takes as input the proof  $\pi$ , queries  $\mathbf{q}$ , key pair  $k$ , and public parameters  $pp$ . If the proof is accepted, it outputs  $d = 1$ ; otherwise, it outputs  $d = 0$ .
- $\text{PoRID.identify}(\pi, \mathbf{q}, k, pp) \rightarrow I \in \{C, S, \perp\}$ . It is run by a third party arbiter. It takes as input the proof  $\pi$ , query  $\mathbf{q}$ , key pair  $k$ , and public parameters  $pp$ . It checks the query  $\mathbf{q}$ . If it is not well-structured, it outputs  $I = C$ ; otherwise, if proof  $\pi$  is rejected, it outputs  $I = S$ . Otherwise, if  $\mathbf{q}$  and  $\pi$  are accepted, it outputs  $I = \perp$ .

The definition of PoRID (unlike PoR) includes two additional algorithms: (i) algorithm  $M(\cdot)$  that generates metadata (or tags), and (ii) algorithm  $Q(\cdot)$  that checks whether queries (or challenges) are well-structured.<sup>8</sup> A PoRID scheme has four main properties; namely, it is (a) correct, (b) sound, (c) inputs of clients are well-formed, and (d) a corrupt party can be identified by an arbiter, i.e., detectable abort. Below, we formally define each of them. Correctness requires that the verification algorithm always accepts a proof generated by an honest prover and both parties are identified as honest. We formally state it below.

**Definition 4 (PoRID Correctness).** A PoR with identifiable abort scheme is correct for functions  $M$  and  $Q$ , if the key generation algorithm produces keys  $\text{PoRID.keyGen}(1^\lambda) \rightarrow k := (sk, pk)$  such that for any file  $u$ , if  $\text{PoRID.setup}(1^\lambda, u, k, M)$

<sup>8</sup> In the original definition of the PoR scheme,  $M(\cdot)$  is implicit; however,  $Q(\cdot)$  is not needed because the scheme assumes the client is fully honest, which is not the case in the PoRID.

$\rightarrow (u^*, e, pp)$ ,  $\text{PoRID.serve}(u^*, e, pk, pp) \rightarrow a$ ,  $\text{PoRID.genQuery}(1^\lambda, Q, pp) \rightarrow \mathbf{q}$ ,  
 $\text{PoRID.checkQuery}(\mathbf{q}, pk, pp) \rightarrow b$ ,  $\text{PoRID.prove}(u^*, \sigma, \mathbf{q}, pk, pp) \rightarrow \pi$ , and  
 $\text{PoRID.verify}(\pi, \mathbf{q}, k, pp) \rightarrow d$ , then  $\text{PoRID.identify}(\pi, \mathbf{q}, k, pp) \rightarrow I = \perp \wedge a = 1 \wedge b = 1 \wedge d = 1$ .

Informally, the PoRID's soundness (similar to PoR) requires that if a prover convinces the verifier, then the file is stored by the prover (with high probability). We formally state it below.

**Definition 5 (PoRID Soundness).** *A PoRID is sound for functions  $M$  and  $Q$ , if for any probabilistic polynomial time adversary  $\mathcal{A}$ , there exists a negligible function  $\mu(\cdot)$  and an extraction algorithm  $\text{Ext}(\cdot)$ , such that for any security parameter  $\lambda$ :*

$$\Pr \left[ \text{Ext}(k, pp, e, \hat{\mathcal{A}}_\epsilon) \neq u \mid \begin{array}{l} \text{PoRID.keyGen}(1^\lambda) \rightarrow k := (sk, pk) \\ \mathcal{A}(1^\lambda, pk) \rightarrow u \\ \text{PoRID.setup}(1^\lambda, u, k, M) \rightarrow (u^*, e, pp) \\ \mathcal{A}(u^*, e, pp) \rightarrow \text{state} \\ \text{PoRID.genQuery}(1^\lambda, Q, pp) \rightarrow \mathbf{q} \\ \left( \mathcal{A}(\mathbf{q}, \text{state}) \rightarrow \pi \right) \equiv \\ \left( \text{PoRID.verify}(\pi, \mathbf{q}, k, pp) \right) \end{array} \right] \leq \mu(\lambda).$$

A PoRID has well-formed inputs, if a malicious client cannot persuade a server to serve it on ill-structured inputs (i.e., to accept incorrect metadata or query). Below, we state the property formally.

**Definition 6 (PoRID Inputs Well-formedness).** *A PoRID has well-formed inputs for functions  $M$  and  $Q$ , if for any probabilistic polynomial time adversary  $\mathcal{A}$ , there exists a negligible function  $\mu(\cdot)$ , such that for any security parameter  $\lambda$ :*

$$\Pr \left[ \begin{array}{l} (M(u^*, k, pp) \neq \sigma \\ \wedge a = 1) \vee \\ (Q(\mathbf{q}, pp) \neq 1 \wedge \\ b = 1) \end{array} \mid \begin{array}{l} \mathcal{A}(1^\lambda, M, Q) \rightarrow (u^*, k := (sk, pk), e := (\sigma, w_\sigma), pp) \\ \text{PoRID.serve}(u^*, e, pk, pp) \rightarrow a \\ \mathcal{A}(1^\lambda, Q, pp) \rightarrow \mathbf{q} \\ \text{PoRID.checkQuery}(\mathbf{q}, pk, pp) \rightarrow b \end{array} \right] \leq \mu(\lambda).$$

It is further required that a malicious party be identified by an honest third party, arbiter. This ensures that in the case of dispute (or false accusation) a malicious party can be pinpointed. A PoRID supports detectable abort if a corrupt party can escape from being identified, by the arbiter, with only negligible probability. Formally:

**Definition 7 (PoRID Detectable Abort).** *A PoRID supports detectable abort for functions  $M$  and  $Q$ , if the following hold:*

1. For any PPT adversary  $\mathcal{A}_1$  there exist a negligible function  $\mu_1(\cdot)$  and an extraction algorithm  $\text{Ext}(\cdot)$ , such that for any security parameter  $\lambda$ :

$$\Pr \left[ \begin{array}{l} I \neq \mathcal{S} \wedge \\ \text{Ext}(k, pp, e, \hat{\mathcal{A}}_\epsilon) \neq u \end{array} \left| \begin{array}{l} \text{PoRID.keyGen}(1^\lambda) \rightarrow k := (sk, pk) \\ \mathcal{A}_1(1^\lambda, pk) \rightarrow u \\ \text{PoRID.setup}(1^\lambda, u, k, M) \rightarrow (u^*, e, pp) \\ \mathcal{A}_1(u^*, e, pp) \rightarrow \text{state} \\ \text{PoRID.genQuery}(1^\lambda, Q, pp) \rightarrow \mathbf{q} \\ \left( (\mathcal{A}_1(\mathbf{q}, \text{state}) \rightarrow \pi) \right) \right. \\ \left. (\text{PoRID.verify}(\pi, \mathbf{q}, k, pp)) \right) \rightarrow \hat{\mathcal{A}}_\epsilon \\ \text{PoRID.identify}(\pi, \mathbf{q}, k, pp) \rightarrow I \end{array} \right] \leq \mu_1(\lambda).$$

2. For any PPT adversary  $\mathcal{A}_2$  there exists a negligible function  $\mu_2(\cdot)$  such that for any security parameter  $\lambda$ :

$$\Pr \left[ \begin{array}{l} b = 0 \wedge I \neq \mathcal{C} \end{array} \left| \begin{array}{l} \mathcal{A}_2(1^\lambda, M, Q) \rightarrow (u^*, k := (sk, pk), e := (\sigma, w_\sigma), pp) \\ \text{PoRID.serve}(u^*, e, pk, pp) \rightarrow a \\ \mathcal{A}_2(1^\lambda, Q, pp) \rightarrow \mathbf{q} \\ \text{PoRID.checkQuery}(\mathbf{q}, pk, pp) \rightarrow b \\ \text{PoRID.prove}(u^*, \sigma, \mathbf{q}, pk, pp) \rightarrow \pi \\ \text{PoRID.identify}(\pi, \mathbf{q}, k, pp) \rightarrow I \end{array} \right] \leq \mu_2(\lambda).$$

## I Recurring Contingent PoR Payment (RC-PoR-P) Definition

Although the PoRID scheme offers two appealing features, it is not sufficient to address all the issues we identified in Section 4. In particular, it does not take the privacy of service input and proofs' status into consideration and does not take into account a secure and fair payment (so it cannot deal with the issue related to wasting the server's resources). Thus, we present an upgraded PoRID's definition that takes the above points into account. We call the enhanced PoRID, “*recurring contingent PoR payment*” (RC-PoR-P). Below, we present the RC-PoR-P's formal definition.

**Definition 8 (RC-PoR-P Scheme).** *A recurring contingent PoR payment scheme RC-PoR-P = (RCSP.keyGen, RCSP.cInit, RCSP.sInit, RCSP.genQuery, RCSP.prove, RCSP.verify, RCSP.resolve, RCSP.pay) involves four parties; namely, client, server, arbiter and smart contract (which represents a bulletin board), and consists of eight algorithms defined as follows.*

- $\text{RCPoRP.keyGen}(1^\lambda) \rightarrow \mathbf{k}$ . A probabilistic algorithm run by the client,  $\mathcal{C}$ . It takes as input security parameter  $1^\lambda$ . It outputs  $\mathbf{k}$  that contains a secret and public verification key pair  $k := (sk, pk)$  and a set of secret and public parameters,  $k' := (sk', pk')$ . It sends  $pk$  and  $pk'$  to the smart contract.

- **RCPoRP.cInit** $(1^\lambda, u, \mathbf{k}, M, z, pl, enc) \rightarrow (u^*, e, T, p_S, \mathbf{y}, coin_C^*)$ . It is run by  $\mathcal{C}$ . It takes as input  $1^\lambda$ , a file  $u$ , key pair  $\mathbf{k} := (k, k')$ , metadata generator function  $M$ , the total number of verifications  $z$ , and price list  $pl$  containing pairs of actual amount of coin for each valid service proof and the amount for covering each potential dispute resolution's cost. It also takes as input encoding/decoding functions  $enc := (E, D)$  used to encode/decode the queries/proofs. It encodes  $u$ , that yields  $u^*$ . It sets  $pp$  as (possibly) input dependent parameters, e.g., file size. It computes metadata  $\sigma = M(u^*, k, pp)$  and a proof  $w_\sigma$  asserting the metadata is well-structured. It sets the value of  $p_S$  to the total coins the server should deposit. It picks a private price pair  $(o, l) \in pl$ . It sets coin secret parameters  $cp$  that include  $(o, l)$  and parameters of  $pl$ , e.g., its maximum values. It constructs coin encoding token  $T_{cp}$  containing  $cp$  and  $cp$ 's witness,  $g_{cp}$ . It constructs encoding token  $T_{qp}$  that contains secret parameters  $qp$  including  $pp$ , (a representation of  $\sigma$ ) and parameters (in  $sk'$ ) that will be used to encode the queries/proofs.  $T_{qp}$  contains  $qp$ 's witness,  $g_{qp}$ . Given a valid value and its witness anyone can check if they match. It sets a vector of parameters  $\mathbf{y}$  that includes binary vectors  $[\mathbf{y}_C, \mathbf{y}_S, \mathbf{y}'_C, \mathbf{y}'_S]$  each of which is set to 0 and its length is  $z$ . Note  $\mathbf{y}$  may contain other public parameters (e.g., the contract's address). It outputs  $u^*$ ,  $e := (\sigma, w_\sigma)$ ,  $T := (T_{cp}, T_{qp})$ ,  $p_S$ ,  $\mathbf{y}$ , and the encoded coins amount  $coin_C^*$  (that contains  $o$  and  $l$  coins in an encoded form). The client sends  $u^*$ ,  $z$ ,  $e$ ,  $T_{cp} \setminus \{g_{cp}\}$  and  $T_{qp} \setminus \{g_{qp}\}$  to the server and sends  $g_{cp}, g_{qp}, p_S$ , and  $\mathbf{y}$ , and  $coin_C^*$  coins to the contract.
- **RCPoRP.sInit** $(u^*, e, pk, z, T, p_S, \mathbf{y}, enc) \rightarrow (coin_S^*, a)$ . It is run by the server,  $\mathcal{S}$ . It takes as input the encoded file  $u^*$ , metadata-proof pair  $e := (\sigma, w_\sigma)$ , public key  $pk$  (read from the contract), the total number of verifications  $z$ , and  $T := (T_{cp}, T_{qp})$  (where  $\{g_{cp}, g_{qp}\}$  are read from the smart contract). Also, it reads  $p_S$ , and  $\mathbf{y}$  from the smart contract and takes as input the encoding/decoding functions  $enc := (E, D)$ . It verifies the validity of  $e$  and  $T$  elements. Also, it checks elements of  $\mathbf{y}$  and ensures each element of  $\mathbf{y}_C, \mathbf{y}_S, \mathbf{y}'_C, \mathbf{y}'_S \in \mathbf{y}$  have been set to 0. If all checks are successful, then it encodes the amount of its coins that yields  $coin_S^*$ ; and it sets  $a = 1$ . Otherwise, it sets  $coin_S^* = \perp$  and  $a = 0$ . It outputs  $coin_S^*$  and  $a$ . The smart contract is given  $coin_S^*$  coins and  $a$ .
- **RCPoRP.genQuery** $(1^\lambda, Q, T_{qp}, enc) \rightarrow c_j^*$ . A probabilistic algorithm run by  $\mathcal{C}$ . It takes as input  $1^\lambda$ , deterministic function  $Q$  that ensures queries are well-structured, encoding token  $T_{qp}$  and  $enc := (E, D)$ . It computes a query vector  $\mathbf{q}_j$  and ensures it is well-structured, i.e.,  $Q(\mathbf{q}_j, pp) = 1$ . It outputs the encoding of the query,  $c_j^* = E(\mathbf{q}_j, T_{qp})$ , and sends the output to the contract.
- **RCPoRP.prove** $(u^*, \sigma, c_j^*, pk, T_{qp}, enc) \rightarrow (b_j, m_{S,j}, \pi_j^*)$ . It is run by  $\mathcal{S}$ . It takes as input the encoded file  $u^*$ , metadata  $\sigma$ , encoded query pair  $c_j^*$ , public key  $pk$ , the encoding token  $T_{qp}$ , and  $enc := (E, D)$ . It checks the validity of decoded query pair  $\mathbf{q}_j = D(c_j^*, T_{qp})$ . If it is rejected, then it sets  $b_j = 0$  and constructs a complaint  $m_{S,j}$ . Otherwise, it sets  $b_j = 1$  and  $m_{S,j} = \perp$ . It outputs  $b_j, m_{S,j}$ , and encoded proof  $\pi_j^* = E(\pi_j, T_{qp})$ . Note,  $\pi_j$  may contain dummy values if  $b_j = 0$ . The smart contract is given  $\pi_j^*$ .
- **RCPoRP.verify** $(\pi_j^*, c_j^*, k, T_{qp}, enc) \rightarrow (d_j, m_{C,j})$ . A deterministic algorithm run by  $\mathcal{C}$ . It takes as input the encoded proof  $\pi_j^*$ , query vector  $\mathbf{q}_j \in c_j^*$ , key pair  $k$ , the encoding token  $T_{qp}$  and  $enc := (E, D)$ . If the decoded proof  $\pi_j = D(\pi_j^*, T_{qp})$  is rejected, it outputs  $d_j = 0$  and a complaint  $m_{C,j}$ . Otherwise, it outputs  $d_j = 1$  and  $m_{C,j} = \perp$ .
- **RCPoRP.resolve** $(\mathbf{m}_C, \mathbf{m}_S, z, \boldsymbol{\pi}^*, \mathbf{c}^*, pk, T_{qp}, enc) \rightarrow \mathbf{y}$ . It is run by the arbiter,  $\mathcal{Ar}$ . It takes as input the client's complaints  $\mathbf{m}_C$ , the server's complaints  $\mathbf{m}_S$ , the total

number of verifications  $z$ , all encoded proofs  $\pi^*$ , all encoded query pairs  $c^*$ , public key  $pk$ , encoding token  $T_{qp}$ , and  $enc := (E, D)$ . It verifies the token, decoded queries, and proofs. It reads the binary vectors  $[\mathbf{y}_C, \mathbf{y}_S, \mathbf{y}'_C, \mathbf{y}'_S]$  from the smart contract. It updates  $\mathbf{y}_E$  by setting an element of it to one, i.e.,  $y_{E,j} = 1$ , if party  $E \in \{C, S\}$  has misbehaved in the  $j$ -th verification (i.e., provided invalid query or service proof). It also updates  $\mathbf{y}'_E$  (by setting an element of it to one) if party  $E$  has provided a complain that does not allow it to identify a misbehaved party, in the  $j$ -th verification, i.e., when the arbiter is unnecessarily invoked.

- **RCPoRP.pay** $(\mathbf{y}, T_{cp}, a, p_S, coin_C^*, coin_S^*) \rightarrow (coin_C, coin_S, coin_{Ar})$ . It is run by the smart contract. It takes as input the binary vectors  $[\mathbf{y}_C, \mathbf{y}_S, \mathbf{y}'_C, \mathbf{y}'_S] \in \mathbf{y}$  that indicate which party misbehaved, or sent an invalid complaint in each verification, coins' token  $T_{cp} := \{cp, g_{cp}\}$ , the output of the checks that server-side initiation algorithm performed  $a$ , the total coins the server should deposit  $p_S$ , and the total coins amount the client and server deposited, i.e.,  $coin_C^*$  and  $coin_S^*$  respectively. If  $a = 1$  and  $coin_S^* = p_S$ , then it verifies the validity of  $T_{cp}$ . If  $T_{cp}$  is rejected, then it aborts. If it is accepted, then it constructs vector  $coin_{\mathcal{I}}$ , where  $\mathcal{I} \in \{C, S, Ar\}$ ; It sends  $coin_{\mathcal{I},j} \in coin_{\mathcal{I}}$  coins to party  $\mathcal{I}$  for each  $j$ -th verification. Otherwise,  $a = 0$  or  $coin_S^* \neq p_S$  it sends  $coin_C^*$  and  $coin_S^*$  coins to  $C$  and  $S$  respectively.

In the definition, algorithms **RCPoRP.genQuery** $(\cdot)$ , **RCPoRP.prove** $(\cdot)$ , **RCPoRP.verify** $(\cdot)$  and **RCPoRP.resolve** $(\cdot)$  implicitly take  $a, coin_S^*, p_S$  as other inputs and execute only if  $a = 1$  and  $coin_S^* = p_S$ ; however, for the sake of simplicity we avoided explicitly stating it in the definition.

A recurring contingent PoR payment (RC-PoR-P) scheme satisfies correctness and security. At a high level, correctness requires that by the end of the protocol's execution (that involves honest client and server) the client receives all  $z$  valid proofs (of retrievability) while the server gets paid for the proofs, without the involvement of the arbiter. More specifically, it requires that the server accepts an honest client's encoded data and query while the honest client accepts the server's valid proofs (and no one is identified as misbehaving party). Also, the honest client gets back all its deposited coins minus the service payment, the honest server gets back all its deposited coins plus the service payment and the arbiter receives nothing. Below, we formally define the correctness.

**Definition 9 (RC-PoR-P Correctness).** A recurring contingent PoR payment scheme is correct for functions  $M, Q, E$ , and  $D$ , if for any price list  $pl$ , the key generation algorithm produces keys  $\text{RCPoRP.keyGen}(1^\lambda) \rightarrow \mathbf{k}$ , such that for any file  $u$ , if  $\text{RCPoRP.cInit}(1^\lambda, u, \mathbf{k}, M, z, pl, enc) \rightarrow (u^*, e, T, p_S, \mathbf{y}, coin_C^*)$ ,  $\text{RCPoRP.sInit}(u^*, e, pk, z, T, p_S, \mathbf{y}, enc) \rightarrow (coin_S^*, a), \forall j \in [z] : \left( \text{RCPoRP.genQuery}(1^\lambda, Q, T_{qp}, enc) \rightarrow c_j^*, \text{RCPoRP.prove}(u^*, \sigma, c_j^*, pk, T_{qp}, enc) \rightarrow (b_j, m_{S,j}, \pi_j^*), \text{RCPoRP.verify}(\pi_j^*, c_j^*, k, T_{qp}, enc) \rightarrow (d_j, m_{C,j}) \right)$ ,  $\text{RCPoRP.resolve}(m_C, m_S, z, \pi^*, c^*, pk, T_{qp}, enc) \rightarrow \mathbf{y}$ ,  $\text{RCPoRP.pay}(\mathbf{y}, T_{cp}, a, p_S, coin_C^*, coin_S^*) \rightarrow (coin_C, coin_S, coin_{Ar})$ , then  $(a = 1) \wedge (\bigwedge_{j=1}^z b_j = \bigwedge_{j=1}^z d_j = 1) \wedge (\mathbf{y}_C = \mathbf{y}_S = \mathbf{y}'_C = \mathbf{y}'_S = 0) \wedge (\sum_{j=1}^z coin_{C,j} = coin_C^* - o \cdot z) \wedge (\sum_{j=1}^z coin_{S,j} = coin_S^* + o \cdot z) \wedge (\sum_{j=1}^z coin_{Ar,j} = 0)$ , where  $\mathbf{y}_C, \mathbf{y}_S, \mathbf{y}'_C, \mathbf{y}'_S \in \mathbf{y}$ .

A RC-PoR-P scheme is said to be secure if it satisfies three main properties: (1) security against a malicious server, (2) security against a malicious client, and (3) privacy. Now, we formally define each of them. Intuitively, Property 1 states that at

the end of the protocol's execution the server which (1.a) convinced the client that it has stored the file, cannot make the client to receive an incorrect amount of coin (i.e., its deposit minus the service payment), or make the arbiter receive an incorrect amount of coin, if it unnecessarily invokes the arbiter, or (1.b) did not convince the client, cannot make the client to receive an incorrect amount of coin (i.e., full deposit for that verification) or makes the arbiter receive an incorrect amount of coin (i.e., anything other than  $l$ ). Below, we formalize this intuition.

**Definition 10 (Security Against Malicious Server).** *A RC-S-P is secure against a malicious server for functions  $M, Q, E$ , and  $D$ , if for any price list  $pl$ , every  $j$  (where  $1 \leq j \leq z$ ), and any PPT adversary  $\mathcal{A}$ , there exist a negligible function  $\mu(\cdot)$ , and extraction algorithm  $\text{Ext}(\cdot)$ , such that for any security parameter  $\lambda$  and experiment  $\text{Exp}_1^{\mathcal{A},j,pl}$ :*

$$\begin{array}{l}
\text{Exp}_1^{\mathcal{A},j,pl}(\text{In} := (1^\lambda, M, Q, E, D, z)) : \\
\text{RCPoRP.keyGen}(1^\lambda) \rightarrow \mathbf{k} \\
\mathcal{A}(1^\lambda, pk) \rightarrow u \\
\text{RCPoRP.cInit}(1^\lambda, u, \mathbf{k}, M, z, pl, enc) \rightarrow (u^*, e, T, p_S, \mathbf{y}, \text{coin}_C^*) \\
\mathcal{A}(u^*, e, pk, z, T, p_S, \mathbf{y}, enc) \rightarrow (\text{coin}_S^*, a, \text{state}) \\
\text{RCPoRP.genQuery}(1^\lambda, Q, T_{qp}, enc) \rightarrow c_j^* \\
\left( \mathcal{A}(c_j^*, \text{state}, a) \rightarrow (b_j, m_{S,j}, \pi_j^*) \right) \Leftrightarrow \\
\text{RCPoRP.verify}(\pi_j^*, c_j^*, k, T_{qp}, enc) \rightarrow (d_j, m_{C,j}) \Big) \rightarrow \mathcal{A}_\epsilon \\
\text{RCPoRP.resolve}(\mathbf{m}_C, \mathbf{m}_S, z, \pi^*, \mathbf{c}^*, pk, T_{qp}, enc) \rightarrow \mathbf{y} \\
\text{RCPoRP.pay}(\mathbf{y}, T_{cp}, a, p_S, \text{coin}_C^*, \text{coin}_S^*) \rightarrow (\text{coin}_C, \text{coin}_S, \text{coin}_{Ar})
\end{array}$$

it holds:

$$\Pr \left[ \left( \begin{array}{l} \text{Ext}(\mathbf{k}, pp, e, \hat{\mathcal{A}}_\epsilon) = u \wedge \\ (\text{coin}_{C,j} \neq \frac{\text{coin}_C^*}{z} - o) \vee \\ (\text{coin}_{Ar,j} \neq l \wedge y'_{S,j} = 1) \end{array} \right) \vee \left( \begin{array}{l} \text{Ext}(\mathbf{k}, pp, e, \hat{\mathcal{A}}_\epsilon) \neq u \wedge \\ (y_{S,j} = 0 \vee \text{coin}_{C,j} \neq \\ \frac{\text{coin}_C^*}{z} \vee \text{coin}_{Ar,j} \neq l) \end{array} \right) \mid \text{Exp}_1^{\mathcal{A},j,pl}(\text{In}) \right] \leq \mu(\lambda)$$

where  $m_{C,j} \in \mathbf{m}_C, m_{S,j} \in \mathbf{m}_S, y'_{S,j} \in \mathbf{y}'_S \in \mathbf{y}, y_{S,j} \in \mathbf{y}_S \in \mathbf{y}$ , and  $pp \in T_{qp}$ .

Informally, Property 2 (i.e., security against a malicious client) requires that for each  $j$ -th verification, a malicious client with a negligible probability wins if it provides either (2.a) valid metadata and query but either makes the server receive an incorrect amount of coin (something other than its deposit plus the service payment), or makes the arbiter withdraw an incorrect amount of coin if it unnecessarily invokes the arbiter or (2.b) invalid metadata or query but convinces the server to accept either of them (i.e., the invalid metadata or query), or (2.c) invalid query but persuades the arbiter to accept it, or makes them withdraw an incorrect amount of coin (i.e.,  $\text{coin}_{S,j} \neq \frac{\text{coin}_S^*}{z} + o$  or  $\text{coin}_{Ar,j} \neq l$  coins). Below, we formally state this property.

**Definition 11 (Security Against Malicious Client).** A RC-PoR-P is secure against a malicious client for functions  $M, Q, E$ , and  $D$ , if for every  $j$  (where  $1 \leq j \leq z$ ), and any probabilistic polynomial time adversary  $\mathcal{A}$ , there exists a negligible function  $\mu(\cdot)$ , such that for any security parameter  $\lambda$  and experiment  $\text{Exp}_2^{A,j}$ :

$$\begin{array}{l} \text{Exp}_2^{A,j}(\text{In} := (1^\lambda, M, Q, E, D, z)) : \\ \mathcal{A}(1^\lambda) \rightarrow (u^*, z, \mathbf{k}, e, T, pl, p_S, \text{coin}_c^*, \text{enc}, \mathbf{y}, \text{enc}, pk) \\ \text{RCPoRP.sInit}(u^*, e, pk, z, T, p_S, \mathbf{y}, \text{enc}) \rightarrow (\text{coin}_S^*, a) \\ \mathcal{A}(\text{coin}_S^*, a, 1^\lambda, k, Q, T_{qp}, \text{enc}) \rightarrow c_j^* \\ \text{RCPoRP.prove}(u^*, \sigma, c_j^*, pk, T_{qp}, \text{enc}) \rightarrow (b_j, m_{S,j}, \pi_j^*) \\ \mathcal{A}(\pi_j^*, c_j^*, k, T_{qp}, \text{enc}) \rightarrow (d_j, m_{S,j}) \\ \text{RCPoRP.resolve}(\mathbf{m}_c, \mathbf{m}_S, z, \pi^*, c^*, pk, T_{qp}, \text{enc}) \rightarrow \mathbf{y} \\ \text{RCPoRP.pay}(\mathbf{y}, T_{cp}, a, p_S, \text{coin}_c^*, \text{coin}_S^*) \rightarrow (\text{coin}_c, \text{coin}_S, \text{coin}_{Ar}) \end{array}$$

it holds:

$$\Pr \left[ \left( \begin{array}{l} (M(u^*, k, pp) = \sigma \wedge \\ Q(\mathbf{q}_j, pp) = 1) \wedge \\ (\text{coin}_{S,j} \neq \frac{\text{coin}_S^*}{z} + o \vee \\ \text{coin}_{Ar,j} \neq l \wedge y'_{c,j} = 1) \end{array} \right) \vee \left( \begin{array}{l} (M(u^*, k, pp) \neq \sigma \wedge a = 1) \vee \\ (Q(\mathbf{q}_j, pp) \neq 1 \wedge (b_j = 1 \vee \\ y_{c,j} = 0 \vee \text{coin}_{S,j} \neq \frac{\text{coin}_S^*}{z} + o \\ \vee \text{coin}_{Ar,j} \neq l)) \end{array} \right) \mid \text{Exp}_2^{A,j}(\text{In}) \right] \leq \mu(\lambda)$$

where  $\mathbf{q}_j \in D(c_j^*, t_{qp})$ ,  $D \in \text{enc}$ ,  $\sigma \in e$ ,  $y'_{c,j} \in \mathbf{y}'_c \in \mathbf{y}$ ,  $y_{c,j} \in \mathbf{y}_c \in \mathbf{y}$ , and  $pp \in T_{qp}$ .

In the above definition, an honest server either does not deposit (e.g., when  $a = 0$ ) or if it deposits (i.e., agrees to serve) ultimately receives its deposit *plus the service payment* (with a high probability). Informally, Property 3 (i.e., privacy) requires that the privacy of (3.a) the service input, i.e., outsourced file, and (3. b) the proof's status during the private time bubble are preserved. Below, we formally define this property.

**Definition 12 (Privacy).** A RC-PoR-P preserves privacy for functions  $M, Q, E$ , and  $D$ , if for any price list  $pl$ , the following hold:

1. For any PPT adversary  $\mathcal{A}_1$  there exists a negligible function  $\mu(\cdot)$ , such that for any security parameter  $\lambda$  and experiment  $\text{Exp}_3^{A_1, pl}$ :

$$\begin{aligned}
& \text{Exp}_3^{A_1, pl}(\text{In} := (1^\lambda, M, Q, E, D)) : \\
& \text{RCPoRP.keyGen}(1^\lambda) \rightarrow \mathbf{k} \\
& \mathcal{A}_1(1^\lambda, pk) \rightarrow (u_0, u_1) \\
& \beta \xleftarrow{\$} \{0, 1\} \\
& \text{RCPoRP.cInit}(1^\lambda, u_\beta, \mathbf{k}, M, z, pl, enc) \rightarrow (u_\beta^*, e, T, p_S, \mathbf{y}, \text{coin}_C^*) \\
& \text{RCPoRP.sInit}(u_\beta^*, e, pk, z, T, p_S, \mathbf{y}, enc) \rightarrow (\text{coin}_S^*, a) \\
& \forall j \in [z] : \\
& \quad \left( \text{RCPoRP.genQuery}(1^\lambda, Q, T_{qp}, enc) \rightarrow c_j^* \right. \\
& \quad \left. \text{RCPoRP.prove}(u_\beta^*, \sigma, c_j^*, pk, T_{qp}, enc) \rightarrow (b_j, m_{S,j}, \pi_j^*) \right. \\
& \quad \left. \text{RCPoRP.verify}(\pi_j^*, c_j^*, k, T_{qp}, enc) \rightarrow (d_j, m_{C,j}) \right)
\end{aligned}$$

it holds:

$$\Pr \left[ \mathcal{A}_1(\mathbf{c}^*, \text{coin}_S^*, \text{coin}_C^*, g_{cp}, \left| \text{Exp}_3^{A_1, pl}(\text{In}) \right. \right] \leq \frac{1}{2} + \mu(\lambda).$$

2. For any PPT adversaries  $\mathcal{A}_2$  and  $\mathcal{A}_3$  there exists a negligible function  $\mu(\cdot)$  such that for any security parameter  $\lambda$  and experiment  $\text{Exp}_4^{A_2, pl}$ :

$$\begin{aligned}
& \text{Exp}_4^{A_2, pl}(\text{In} := (1^\lambda, M, Q, E, D)) : \\
& \text{RCPoRP.keyGen}(1^\lambda) \rightarrow \mathbf{k} \\
& \mathcal{A}_2(1^\lambda, pk) \rightarrow u \\
& \text{RCPoRP.cInit}(1^\lambda, u, \mathbf{k}, M, z, pl, enc) \rightarrow (u^*, e, T, p_S, \mathbf{y}, \text{coin}_C^*) \\
& \text{RCPoRP.sInit}(u^*, e, pk, z, T, p_S, \mathbf{y}, enc) \rightarrow (\text{coin}_S^*, a) \\
& \forall j \in [z] : \\
& \quad \left( \mathcal{A}_2(1^\lambda, k, Q, T_{qp}, enc) \rightarrow c_j^* \right. \\
& \quad \left. \text{RCPoRP.prove}(u^*, \sigma, c_j^*, pk, T_{qp}, enc) \rightarrow (b_j, m_{S,j}, \pi_j^*) \right. \\
& \quad \left. \text{RCPoRP.verify}(\pi_j^*, c_j^*, k, T_{qp}, enc) \rightarrow (d_j, m_{C,j}) \right)
\end{aligned}$$

it holds:

$$\Pr \left[ \mathcal{A}_3(\mathbf{c}^*, \text{coin}_S^*, \text{coin}_C^*, g_{cp}, \left| \text{Exp}_4^{A_2, pl}(\text{In}) \right. \right] \leq Pr' + \mu(\lambda)$$

where  $\mathbf{c}^* = [c_1^*, \dots, c_z^*]$ ,  $\boldsymbol{\pi}^* = [\pi_1^*, \dots, \pi_z^*]$ , and  $Pr'$  is defined as follows. Let  $\mathbf{q}_j \in D(c_j^*, T_{qp})$  and  $pp \in T_{qp}$ . We define the events  $Con_{0,j}^{(1)} : Q(\mathbf{q}_j, pp) \neq 1$ ,  $Con_{0,j}^{(2)} : b_j = 0$ ,  $Con_{1,j}^{(1)} : Q(\mathbf{q}_j, pp) = 1$ , and  $Con_{1,j}^{(2)} : b_j = 1$ . For  $i \in \{0, 1\}$  and  $j \in [z]$ , we define

$$Pr_{i,j} := \Pr \left[ \left( Con_{i,j}^{(1)} \wedge Con_{i,j}^{(2)} \right) \left| \text{Exp}_4^{A_2, pl}(\text{In}) \right. \right].$$

Then, we have  $Pr' := \text{Max}\{Pr_{0,1}, Pr_{1,1}, \dots, Pr_{0,z}, Pr_{1,z}\}$ .

Note, in the above definition, for each  $j$ -th verification, adversary  $\mathcal{A}_2$  with probability  $Pr_{0,j}$  produces an invalid query and with probability  $Pr_{1,j}$  produces a valid query. It is required that the privacy holds regardless of the queries and proofs status, i.e., whether they are valid or invalid, as long as they are correctly encoded and provided. In the above definitions, the private time bubble is a time period from the point when

`RCPoRP.keyGen(.)` is executed up to the time when `RCPoRP.resolve(.)` is run. In other words, the privacy holds up to the point where `RCPoRP.resolve(.)` is run. That is why the latter algorithm is excluded from the experiments in Definition 12. Now, we state the RC-PoR-P’s main security theorem.

**Definition 13 (RC-PoR-P Security).** *A RC-PoR-P scheme is secure if it satisfies security against a malicious server, security against a malicious client, and preserves privacy, w.r.t. Definitions 10-12.*

## J Proof of Modified Merkle tree-based PoR

Below, we prove Theorem 1, i.e., the security of the PoR in section 6.1.

**Theorem 1.** *The PoR scheme, presented in Section 6.1, is  $\epsilon$ -sound, w.r.t. Definitions 2, if Merkle tree and pseudorandom function PRF are secure.*

*Proof.* As stated above, the proposed PoR differs from the standard Merkle tree-based PoR from a couple of perspectives. However, the changes do not affect the security and soundness of the proposed PoR. Its security proof is similar to the existing Merkle tree-based PoR schemes, e.g., [29,33,42]. Alternatively, our protocol can be proven based on the security analysis of the PoR schemes that use MACs or BLS signatures, e.g., [47]. In this case, the extractor design (in the Merkle tree-based PoR) would be simpler because it does not need to extract blocks from a linear combination of MACs or signatures, as the blocks are included in proofs, i.e., they are part of the Merkle tree proofs. Intuitively, in either case, the extractor interacts with any adversarial prover that passes a non-negligible  $\epsilon$  fraction of audits. It initialises an empty array. Then it challenges a subset of file blocks and asks the prover to generate a proof. If the received proof passes the verification, then it adds the related block (in the proof) to the array. It then rewinds the prover and challenges a fresh set of blocks, and repeats the process many times. Since the prover has a good chance of passing the audit, it is easy to show that the extractor can eventually extract a large fraction of the entire file, as it is shown in [47]. Due to the security of the Merkle tree, the retrieved values are the valid and correct file blocks and due to the security of the pseudorandom function, the challenges (or the function’s outputs) are not predictable. After collecting a sufficient number of blocks, the extractor can use the error-correcting code to decode and recover the entire file blocks, given the retrieved ones.  $\square$

## K Proof of PoRID

In this section, we prove the security of the PoRID.

**Theorem 2.** *The PoRID protocol satisfies the  $\epsilon$ -soundness, inputs well-formedness, and detectable abort properties, w.r.t. Definitions 5, 6, and 7, if the PoR is  $\epsilon$ -sound and the blockchain is secure.*

*Proof.* The  $\epsilon$ -soundness of the PoRID stems from the security of the PoR scheme, i.e.,  $\epsilon$ -soundness. Specifically, in the PoRID the (honest) client makes black-box calls to the algorithms of the PoR, to ensure the soundness. In the PoRID no additional message is provided to the adversary/server. It receives the same messages as it receives in the

PoR. The latter scheme’s soundness ensures that an extractor can recover the entire file interacting with a corrupt server which passes  $\epsilon$  fraction of challenges. On the other hand, the inputs well-formedness holds for the following reasons. The metadata generation algorithm, i.e., the Merkle tree algorithm that builds a tree and computes a root, is deterministic and involves only public parameters. Thus, given the tree’s leaves (i.e., file blocks), its parameters, and the root, anyone can reconstruct it, check if it yields the same root, and verify the tree’s parameters. Also, a query contains a single random key,  $\hat{k}$ , whose correctness can be checked deterministically, i.e., by checking  $\hat{k} \neq \perp$  and  $\hat{k} \in \{0, 1\}^\psi$ . The detectable abort property holds as long as the soundness and inputs well-formedness hold and the blockchain is secure. The reason is that algorithm `PoRID.identify(.)`, which ensures detectable abort, is a wrapper function that makes black-box calls to algorithms `PoRID.checkQuery(.)` and `PoR.verify(.)`, where the former ensures input, i.e., query, well-formedness, and the latter ensures soundness.

The (blockchain’s) signature security ensures if a proof (or a transaction in general) is signed correctly, then it cannot be repudiated by the signer later, this guarantees the signer is held accountable for a rejected proof it provided. Moreover, a malicious party cannot frame an honest party for proving an ill-formed or no proof. In particular, to do that, it has to either (a) forge the honest party’s signature, so it can send an ill-formed proof on its behalf, or (b) fork the blockchain so the chain comprising the accepting proof is discarded. In the former case, the adversary’s probability of success is negligible as long as the signature is secure. The adversary has the same success probability in the latter case; because it has to generate a long enough chain that excludes the accepting proof which has a negligible success probability, under the assumption that the hash power of the adversary is lower than those of honest miners and due to the blockchain’s *liveness* property an honestly generated transaction will eventually appear on an honest miner’s chain [25].  $\square$

## L Further Discussion of the SAP

In this section, first we outline why the SAP satisfies all four security properties set out in Section 6.3 and then discuss why naive solutions are not suitable replacements of the SAP. After that, we highlight that the SAP’s verification phase can be locally performed with low costs.

### L.1 SAP’s Security Analysis

Intuitively, the SAP meets Property 1 due to the binding property of the commitment scheme. Property 2 is satisfied due to the security of the blockchain and smart contract; namely, due to blockchain’s liveness property an honestly generated transaction, containing the opening, eventually gets into chains of honest miners, and due to the security and correctness of smart contracts a valid opening is always accepted by the contract. Property 3 is met due to the hiding property of the commitment, while Property 4 is satisfied due to the signature scheme’s security.

### L.2 Unsuitability of Naive Solutions

As a replacement of the SAP, one may let each party sign the statement and send it to the other party, so later each party can send both signatures to the contract

which verifies them. However, this would not work, as the party who first receives the other party’s signature may refuse to send its own signature, that prevents the other party from proving that it has agreed on the statement with its counter-party, i.e., cannot satisfy Property 2. Alternatively, one may want to use a protocol for a fair exchange of digital signature (or fair contract signing) such as those in [14,24]. In this case, after both parties have the other party’s signature, they can sign the statement themselves and send the two signatures to the contract which first checks the validity of both signatures. Although this satisfies the four security requirements, it yields two main *efficiency* and *practical* issues; namely, it (a) imposes very high computation costs, as protocols for a fair exchange of signatures involve generic zero-knowledge proofs and require a high number of modular exponentiations, and (b) is impractical because protocols for the fair exchange of signatures support only certain signature schemes (e.g., RSA, Rabin, or Schnorr) that are not directly supported by the most predominant smart contract framework, Ethereum, that only supports Elliptic Curve Digital Signature Algorithm (EDCSA).

### L.3 Off-chain Verification in the SAP

The SAP’s verification algorithm can be executed *off-chain*. In particular, given statement  $\tilde{x}$ , anyone can read  $(g_c, g_s, adr_c, adr_s)$  from the SAP smart contract and locally run  $\text{SAP.verify}(\tilde{x}, g_c, g_s, adr_c, adr_s)$  to check the statement’s correctness. This relieves the verifier from the transaction and smart contract’s execution costs.

## M Security Analysis of RC-PoR-P

In this section, we analyse the security of the RC-PoR-P. We start by presenting the protocol’s primary security theorem.

**Theorem 3.** *The RC-PoR-P protocol is secure, w.r.t. Definition 13, if the PoRID, SAP, and blockchain are secure and the encryption scheme is semantically secure.*

To prove the above theorem, we show that the RC-PoR-P satisfies all security properties defined in Section I. We first prove that the RC-PoR-P meets the security against a malicious server.

**Lemma 1.** *If the SAP and blockchain are secure and the PoRID scheme supports correctness, soundness, and detectable abort, then the RC-PoR-P is secure against a malicious server, w.r.t. Definition 10.*

*Proof.* First, we focus on event  $\left(\text{Ext}(\mathbf{k}, pp, e, \hat{A}_e) = u \wedge \left( (coin_{c,j} \neq \frac{coin_c^*}{z} - o) \vee (coin_{a,r,j} \neq l \wedge y'_{s,j} = 1) \right)\right)$  that captures the case where the server provides an accepting proof (and as a result the file is extractable), but makes an honest client withdraw an incorrect amount of coin, i.e.,  $coin_{c,j} \neq \frac{coin_c^*}{z} - o$ , or it makes the arbiter withdraw an incorrect amount of coin, i.e.,  $coin_{a,r,j} \neq l$ , if it unnecessarily invokes the arbiter, i.e.,  $y'_{s,j} = 1$ . Since the server’s proof is valid, an honest client accepts it and does not raise a dispute. But, the server could make the client withdraw an incorrect amount, if it manages to either convince the arbiter that the client has misbehaved, by making the arbiter output  $y_{c,j} = 1$  through the dispute resolution phase, or submit

to the contract (in the coin transfer phase) an accepting statement  $\tilde{x}'_{cp}$  other than what was agreed in the initiation phase, i.e.,  $\tilde{x}'_{cp} \neq \tilde{x}_{cp}$ , so it can change the payment's parameters, or send a message on the client's behalf to unnecessarily invoke the arbiter.

Nevertheless, it cannot falsely accuse the client of misbehaviour. Because, due to the security of the SAP, it cannot convince the arbiter to accept a different decryption key (that will be used to decrypt queries) other than what was agreed with the client in the initiation phase. In particular, it cannot persuade the arbiter to accept  $\tilde{x}'_{qp}$ , where  $\tilde{x}'_{qp} \neq \tilde{x}_{qp}$ , except with a negligible probability,  $\mu(\lambda)$ . This ensures that the honest client's queries are accessed by the arbiter with a high probability. Furthermore, if the adversary provides a valid statement, i.e.,  $\tilde{x}_{qp}$ , then due to the PoRID's correctness, algorithm `PoRID.identify(.)` outputs  $I_j = \perp$ . Therefore, due to the security of the SAP and correctness of the PoRID, the following holds  $y_{c,j} = y_{s,j} = 0$ . Furthermore, because of the SAP's security, the server cannot change the payment's parameters by convincing the contract to accept any statement  $\tilde{x}'_{cp}$  other than what was agreed initially between the client and server, except with a negligible probability,  $\mu(\lambda)$ . Also, due to the (blockchain) signature's security, the adversary cannot send a message on behalf of the client, to unnecessarily invoke the arbiter and make it output  $y'_{c,j} = 1$ , except with a negligible probability  $\mu(\lambda)$ ; so with a high probability  $y'_{c,j} = 0$ . Recall, in the protocol, the total amount the client should receive after  $z$  verifications is  $coin_{\mathcal{C}}^* - o \cdot (z - y_s) - l \cdot (y_c + y'_c)$ . Since we focus on the  $j$ -th verification, the amount that should be credited to the client for that verification is:

$$coin_{c,j} = \frac{coin_{\mathcal{C}}^*}{z} - o \cdot (1 - y_{s,j}) - l \cdot (y_{c,j} + y'_{c,j}) \quad (1)$$

Since it holds that  $y_{c,j} = y_{s,j} = y'_{c,j} = 0$ , the client is credited  $\frac{coin_{\mathcal{C}}^*}{z} - o$  coins for the  $j$ -th verification, with a high probability. Furthermore, as stated above, if the adversary invokes the arbiter, the arbiter with a high probability outputs  $I_j = \perp$  that yields  $y'_{s,j} = 1$ . Recall, in the RC-PoR-P, the total amount the arbiter should receive for  $z$  verifications is  $l \cdot (y_s + y_c + y'_s + y'_c)$ , so for the  $j$ -th the credited amount is:

$$coin_{Ar,j} = l \cdot (y_{s,j} + y_{c,j} + y'_{s,j} + y'_{c,j}) \quad (2)$$

As shown above  $y_{c,j} = y_{s,j} = y'_{c,j} = 0$  and  $y'_{s,j} = 1$ , which means  $l$  coins is credited to the arbiter for the  $j$ -th verification if it is unnecessarily invoked by the adversary. In this case, for the server to make the arbiter withdraw other than that amount, it has to send to the smart contract (in the coin transfer phase) an accepting statement  $\tilde{x}'_{cp}$  other than what was agreed in the initiation phase, i.e.,  $\tilde{x}'_{cp} \neq \tilde{x}_{cp}$ , so it can change the payment's parameters. Nevertheless, as stated above, it cannot succeed with a probability significantly greater than  $\mu(\lambda)$ .

We now move on to event  $\left( (\mathbf{Ext}(\mathbf{k}, pp, e, \hat{A}_\epsilon) \neq u) \wedge (y_{s,j} = 0 \vee coin_{c,j} \neq \frac{coin_{\mathcal{C}}^*}{z} \vee coin_{Ar,j} \neq l) \right)$  which captures the case where the server provides an invalid proof (and the file cannot be extracted) but it either convinces the client to accept the proof, or persuades the arbiter to accept it or makes the client or arbiter withdraw incorrect amount of coin, i.e.,  $coin_{c,j} \neq \frac{coin_{\mathcal{C}}^*}{z}$  or  $coin_{Ar,j} \neq l$  respectively. Due to the soundness of the PoRID, the probability that the adversary can convince an honest client to accept invalid proof is negligible,  $\mu(\lambda)$ . Therefore, the client outputs  $d_j = 0$  with a high probability and raises a dispute. Furthermore, the server may try to make the arbiter keep  $y_{s,j} = 0$ . For the adversary to succeed, it has to make the arbiter identify the client as the misbehaving party, and output  $y_{c,j} = 1$ . In this case, according

to the RC-PoR-P protocol, the client's complaint (for the  $j$ -th verification) would not be processed by the arbiter. This allows  $y_{S,j}$  to remain 0. But, as we argued above, the probability that the adversary makes the arbiter recognise the client as misbehaving is at most  $\mu(\lambda)$ . So, with a high probability  $y_{S,j} = 1$  and  $y_{C,j} = 0$ , after the arbiter is invoked by the client or server. It also holds that  $y'_{C,j} = y'_{S,j} = 0$ , because the arbiter has already identified a misbehaving party. Moreover, due to the SAP's security, the probability that the adversary succeeds in changing the payment's parameters to make the client or arbiter withdraw an incorrect amount of coin is negligible too. Therefore, according to Equations 1 and 2, the client and arbiter are credited  $\frac{coin_C^*}{z}$  and  $l$  coins for the  $j$ -th verification respectively.  $\square$

Next, we provide a lemma which formally states that the RC-PoR-P is secure against a malicious client. Then, we prove this lemma.

**Lemma 2.** *If the SAP and blockchain are secure and the PoRID supports correctness, inputs well-formedness, and detectable abort, then the RC-PoR-P is secure against a malicious client, w.r.t. Definition 11.*

*Proof.* We first consider event  $\left( (M(u^*, k, pp) = \sigma \wedge Q(\mathbf{q}_j, pp) = 1) \wedge \left( (coin_{S,j} \neq \frac{coin_S^*}{z} + o) \vee (coin_{Ar,j} \neq l \wedge y'_{C,j} = 1) \right) \right)$ . It captures the case where the client provides accepting metadata (i.e., a correct Merkle tree's root) and query but makes the server withdraw incorrect amount of coin, i.e.,  $coin_{S,j} \neq \frac{coin_S^*}{z} + o$ , or makes the arbiter withdraw an incorrect amount of coin, i.e.  $coin_{Ar,j} \neq l$ , if it unnecessarily invokes the arbiter, i.e.,  $y'_{C,j} = 1$ . Since the metadata and queries are valid and correctly structured, an honest server accepts them and does not raise a dispute, so  $y_{C,j} = 0$ . However, the client could make the server withdraw an incorrect amount of coin if it manages to either persuade the arbiter to recognise the server as the misbehaving party, i.e., makes the arbiter output  $y_{S,j} = 1$ , or submit to the contract an accepting statement  $\tilde{x}'_{cp}$  other than what was agreed at the initiation phase, i.e.,  $\tilde{x}_{cp}$ , or send a message on the server's behalf to unnecessarily invoke the arbiter. Nevertheless, it cannot falsely accuse the server of misbehaviour. Because, due to the SAP's security, it cannot convince the arbiter to accept different decryption key and pads' detail, by providing a different accepting statement  $\tilde{x}'_{qp}$  than what was initially agreed with the server (i.e.,  $\tilde{x}'_{qp} \neq \tilde{x}_{qp}$ ), except with a negligible probability,  $\mu(\lambda)$ . This ensures the arbiter is given the honest server's messages, with a high probability. Therefore, with a high probability  $y_{S,j} = 0$ . Also, if the adversary provides a valid statement, i.e.,  $\tilde{x}_{qp}$ , then due to the correctness of the PoRID, algorithm `PoRID.identify(.)` outputs  $I_j = \perp$ . So, due to the SAP's security and PoRID's correctness, it holds  $y_{C,j} = y_{S,j} = 0$ , with a high probability. Moreover, it holds that  $y'_S = 0$  because the honest server never invokes the arbiter when the client's queries are well-structured and due to the (blockchain) signature's security, the client cannot send a message on the server's behalf to unnecessarily invoke the arbiter. Note, due to the SAP's security, the client cannot change the payment's parameters by convincing the contract to accept any statement  $\tilde{x}'_{cp}$  other than what was initially agreed between the client and server (i.e.,  $\tilde{x}'_{cp} \neq \tilde{x}_{cp}$ ), except with a negligible probability,  $\mu(\lambda)$ . According to the RC-PoR-P protocol, the total coins the server should receive after  $z$  verifications is  $coin_S^* + o \cdot (z - y_S) - l \cdot (y_S + y'_S)$ . Since we focus on the  $j$ -th verification, the amount of coins that should be credited to the server for the  $j$ -th verification is:

$$coin_{S,j} = \frac{coin_S^*}{z} + o \cdot (1 - y_{S,j}) - l \cdot (y_{S,j} + y'_{S,j}) \quad (3)$$

Thus, given  $y_{S,j} = y'_{S,j} = 0$ , the server is credited  $\frac{coin_S^*}{z} + o$  coins for the  $j$ -th verification, with a high probability. Furthermore, as stated above, if the adversary invokes the arbiter, the arbiter with a high probability outputs  $I_j = \perp$  which yields  $y'_{C,j} = 1$ . Hence, according to Equation 2, the arbiter for the  $j$ -th verification is credited  $l$  coins, with a high probability. As previously stated, due to the SAP's security, the client cannot make the arbiter withdraw an incorrect amount by changing the payment's parameters and persuading the contract to accept any statement  $\hat{x}'_{cp}$  other than what was agreed between the client and server, except with a negligible probability  $\mu(\lambda)$ .

Now we turn our attention to  $\left(M(u^*, k, pp) \neq \sigma \wedge a = 1\right)$  which captures the case where the server accepts ill-formed metadata. However, due to the PoRID's inputs well-formedness, the probability that the event happens is negligible,  $\mu(\lambda)$ ; therefore, with a high probability  $a = 0$ . In this case, the server does not raise any dispute, instead it avoids serving the client. Now, we move on to  $\left((Q(\mathbf{q}_j, pp) \neq 1) \wedge (b_j = 1 \vee y_{C,j} = 0 \vee coin_{S,j} \neq \frac{coin_S^*}{z} + o \vee coin_{Ar,j} \neq l)\right)$  which considers the case where the client provides an invalid query, but either convinces the server or arbiter to accept it, or makes the server or arbiter withdraw an incorrect amount of coin, i.e.,  $coin_{S,j} \neq \frac{coin_S^*}{z} + o$  or  $coin_{Ar,j} \neq l$  respectively. However, due to the PoRID's inputs well-formedness, the probability that the server outputs  $b_j = 1$  is negligible,  $\mu(\lambda)$ . Note, when the honest server rejects the query and raises a dispute, the arbiter checks the query and sets  $y_{C,j} = 1$ . After that, due to the RC-PoR-P design, the client cannot make the arbiter set  $y_{C,j} = 0$  (unless it manages to modify the blockchain's content later, but its probability of success is negligible due to the security of blockchain). As already stated, the client cannot make the arbiter recognise the honest server as a misbehaving party with a probability significantly greater than  $\mu(\lambda)$ . That means, with a high probability  $y_{S,j} = 0$ . Furthermore, since the arbiter has identified a misbehaving party, it holds  $y'_{C,j} = y'_{S,j} = 0$ . The adversary may still try to make them withdraw an incorrect amount. To this end, in the coin transfer phase, it has to send a different accepting statement than what was initially agreed with the server. But, due to the SAP's security, its success probability is  $\mu(\lambda)$ . Hence, according to Equations 3 and 2 the server and arbiter are credited  $\frac{coin_S^*}{z} + o$  and  $l$  coins respectively for the  $j$ -th verification, with a high probability.  $\square$

Prior to proving the RC-PoR-P's privacy, we provide a lemma that will be used in the privacy's proof. Informally, the lemma states that encoded coins leak no information about the actual amount of coins  $(o, l)$ , agreed between the client and server.

**Lemma 3.** *Let  $\beta \xleftarrow{\$} \{0, 1\}$ , the price list be  $pl : \{(o_0, l_0), (o_1, l_1)\}$ , and the encoded amount of coin be  $coin_C^* = z \cdot (\text{Max}(o_\beta, o_{1-\beta}) + \text{Max}(l_\beta, l_{1-\beta}))$  and  $coin_S^* = z \cdot (\text{Max}(l_\beta, l_{1-\beta}))$ . Then, given  $pl$ ,  $z$ ,  $coin_C^*$ , and  $coin_S^*$ , an adversary  $\mathcal{A}$  cannot tell the value of  $\beta$  with a probability significantly greater than  $\frac{1}{2}$  (where the probability is taken over the choice of  $\beta$  and the randomness of  $\mathcal{A}$ ).*

*Proof.* As it is evident, the list and  $z$  contain no information about  $\beta$ . Also, since  $z$  is a public value, it holds that  $coin_C^* = \frac{coin_C}{z} = \text{Max}(o_\beta, o_{1-\beta}) + \text{Max}(l_\beta, l_{1-\beta})$ . It is not hard to see  $coin_C^*$  is a function of the maximum value of  $(o_0, o_1)$ , and the maximum value of  $(l_0, l_1)$ . It is also independent of  $\beta$ . Therefore (given  $pl$ ,  $z$ , and  $coin_C^*$ ) the adversary learns nothing about  $\beta$ , unless it guesses the value, with success probability  $\frac{1}{2}$ . The same also holds for  $coin_S^*$ .  $\square$

**Lemma 4.** *If the SAP is secure and the encryption scheme is semantically secure, then the RC-PoR-P is privacy-preserving, w.r.t. Definition 12.*

*Proof.* We start with case 1, i.e., the privacy of service input (or the file). Due to the SAP’s privacy, that stems from the hiding property of the commitment scheme, given the commitments  $g_{qp}$  and  $g_{cp}$ , (stored in the blockchain as a result of running the SAP) the adversary learns no information about the committed values (e.g.,  $o, l, pad_\pi, pad_q$ , and  $\bar{k}$ ), except with a negligible probability,  $\mu(\lambda)$ . Moreover, given price list  $pl$ , encoded amount  $coin_c^* = z \cdot (o_{max} + l_{max})$  and  $coin_s^* = z \cdot l_{max}$ , the adversary learns nothing about the actual price that was agreed between the server and client, i.e.,  $(o, l)$ , for each verification, due to Lemma 3.

Next, we analyse the privacy of padded encrypted proof vector  $\pi^*$ . We focus on the vector  $\pi_j^*$  (where  $\pi_j^* \in \pi^*$ ), that is a padded encrypted proof for  $j$ -th verification. Let  $\pi_{j,0}$  and  $\pi_{j,1}$  be plaintext proofs, for  $j$ -th verification, related to files  $u_0$  and  $u_1$ , where the files are picked by the adversary, w.r.t. Definition 12 in which the environment picks  $\beta \xleftarrow{\$} \{0, 1\}$ . Let us assume all proofs have the same size (which will be relaxed shortly). If we assume  $\pi_{j,\beta}$  is only encrypted (but not padded), then given the ciphertext, due to the semantical security of the encryption, an adversary cannot tell if the ciphertext corresponds to  $\pi_{j,0}$  or  $\pi_{j,1}$  (accordingly to  $u_0$  or  $u_1$ ) with a probability significantly greater than  $\frac{1}{2} + \mu(\lambda)$ . However, the above assumption, that all proofs have the same, can be relaxed with the use of a pad. In particular, since each encrypted proof is padded to the proofs’ maximum size and the pad’s elements are picked randomly from the encryption’s ciphertext range, the adversary cannot tell with a probability greater than  $\frac{1}{2} + \mu(\lambda)$  if the padded encrypted proof corresponds to  $\pi_{j,0}$  or  $\pi_{j,1}$ . Furthermore, because each query  $\hat{k}_j^*$  is an output of semantically secure symmetric-key encryption and has a fixed size, it leaks nothing to the adversary. Also, the value of  $a$  is independent of  $u_0$  or  $u_1$ , and only depends on whether the metadata is well-formed, so it leaks nothing about the file  $u_\beta$ ,  $\beta$ , the encrypted query, and encoded proofs. Hence, the adversary cannot tell with a probability significantly greater than  $\frac{1}{2} + \mu(\lambda)$  which file of its choice has been used as the server input.

Now we move on to case 2, i.e., the privacy of proof’s status. We know that each encoded query-proof pair has a fixed size and contains random elements of  $U$ ; therefore, they leak no information except with a negligible probability,  $\mu(\lambda)$ . Recall, in the experiment for each  $j$ -th verification, an invalid query-proof pair is computed with probability  $Pr_{0,j}$  and a valid query-proof pair is generated with probability  $Pr_{1,j}$ . So, an adversary can correctly guess a query’s status with a probability at most  $Pr' + \mu(\lambda)$ , where  $Pr' := \text{Max}\{Pr_{0,1}, Pr_{1,1}, \dots, Pr_{0,z}, Pr_{1,z}\}$ . Also, the encoded amount and commitments (i.e.,  $coin_s^*$ ,  $coin_c^*$ ,  $g_{cp}$ , and  $g_{qp}$ ) are independent of the status of the queries or proofs. Hence, the adversary cannot tell a proof’s status with a probability significantly greater than  $Pr' + \mu(\lambda)$ .  $\square$

## N Protocol for RC-PoR-P without Arbiter’s Involvement

As we highlighted in Section 6.3, in the RC-PoR-P, due to the efficiency of the arbiter-side algorithm, i.e., `RCSPoR.resolve(.)`, we can delegate the arbiter’s role to the smart contract, SC. In this case, the involvement of the single third party arbiter is not needed anymore. To this end, we need to adjust the protocol, primarily from two perspectives. First, the way a party pays to resolve a dispute would change, that ultimately affects the amounts of coin each party receives in the coin distribution phase. Recall, in the

protocol, the party which raises a dispute does not pay the arbiter when it sends to it the dispute query. Instead, loosely speaking, the arbiter is paid later, in the coin distribution phase, by a misbehaving party.

In contrast, when the arbiter's role is played by a smart contract, the party which raises a dispute and sends the dispute query to the contract (due to the nature of the smart contracts' platform) has to pay the contract before the contract processes its query. This means an honest party which sends a complaint to the contract needs to be compensated (by the corrupt party) for the amount of coin it sent to the contract to resolve the dispute. Therefore, the amount of coin each party receives in the coin distribution phase would change, compare to the original RC-PoR-P. Second, there would be no need to keep track of the number of times a party unnecessarily raises a dispute, as it pays the contract when it sends a query before the contract processes its claim.

Now, we elaborate on how the original RC-PoR-P is adjusted. Briefly, Phases 1-6 remain unchanged, with an exception. Namely, in step 2d, only two counters  $y_C$  and  $y_S$  are created, instead of four counters; accordingly, in the same step, vector  $\mathbf{y}$  is now  $\mathbf{y} : [y_C, y_S, adr_{SC}]$ , so counters  $y'_C$  and  $y'_S$  are excluded from this vector. At a high level, the changes applied to Phase 7 are as follows: the parties send their complaints to SC now, SC does not maintain  $y'_C$  and  $y'_S$  anymore, SC takes the related steps (on the arbiter's behalf), and it reads its internal state (instead of receiving it from the arbiter). Moreover, the main adjustment to Phase 8 is that the amount of coin each party receives changes. For the sake of clarity, we present the modified version of phases 7 and 8, below.

#### 7. Dispute Resolution. $RCPoRP.resolve(\mathbf{m}_C, \mathbf{m}_S, z, \pi^*, \mathbf{q}^*, T_{qp})$

- (a)  $\mathcal{S}$  sends  $\mathbf{m}_S$  and  $\ddot{x}_{qp}$  to SC, at time  $K_1$ , where  $K_1 > G_{z,2} + J$
- (b) SC upon receiving  $\mathbf{m}_S$  takes the below steps, at time  $K_2$ .
  - i. checks  $\ddot{x}_{qp}$ 's validity, by calling the SAP's verification which returns  $d$ . If  $d = 0$ , SC discards  $\mathcal{S}$ 's complaint,  $\mathbf{m}_S$ , and does not take steps 7(c)ii and 7(c)iii. Otherwise, it proceeds to the next step.
  - ii. Removes from  $\mathbf{m}_S$  any element that is duplicated or is not in the range  $[1, z]$ . It also constructs an empty vector  $\mathbf{v}$ .
  - iii. For every element  $j \in \mathbf{m}_S$ :
    - decrypts the related query  $\hat{k}_j^* \in \mathbf{q}^*$  as  $\hat{k}_j = \text{Dec}(\bar{k}, \hat{k}_j^*)$ .
    - checks the query, by calling  $\text{PoRID.checkQuery}(\hat{k}_j, pp) \rightarrow b_j$ . If  $b_j = 0$ , it increments  $y_C$  by 1 and appends  $j$  to  $\mathbf{v}$ .

Let  $K_3$  be the time SC finishes the above checks.
- (c)  $\mathcal{C}$  sends  $\mathbf{m}_C$  and  $\ddot{x}_{qp}$  to SC, at time  $K_4$ .
- (d) SC upon receiving  $\mathbf{m}_C$ , takes the following steps, at time  $K_5$ .
  - i. checks the validity of statement  $\ddot{x}_{qp}$ , by calling the SAP's verification which returns  $d'$ . If  $d' = 0$ , SC discards the  $\mathcal{C}$ 's complaint,  $\mathbf{m}_C$ , and does not take steps 7(e)ii-7(e)iii. Otherwise, it proceeds to the next step.
  - ii. ensures each vector  $\mathbf{m} \in \mathbf{m}_C$  is well-formed. In particular, it verifies there exist no two vectors:  $\mathbf{m}, \mathbf{m}' \in \mathbf{m}_C$  such that  $\mathbf{m}[0] = \mathbf{m}'[0]$ . If such vectors exist, it deletes the redundant ones from  $\mathbf{m}_C$ . Also, it removes any vector  $\mathbf{m}$  from  $\mathbf{m}_C$  if  $\mathbf{m}[0]$  is not in the range  $[1, z]$  or if  $\mathbf{m}[0] \in \mathbf{v}$ .
  - iii. For every vector  $\mathbf{m} \in \mathbf{m}_C$ :
    - retrieves a rejected proof's detail, by setting  $j = \mathbf{m}[0]$  and  $i = \mathbf{m}[1]$ .
    - decrypts the related query  $\hat{k}_j^* \in \mathbf{q}^*$  as  $\hat{k}_j = \text{Dec}(\bar{k}, \hat{k}_j^*)$ .

- removes the pads only from  $i$ -th padded encrypted proof. Let  $\pi'_j[i]$  be the result. Next, it decrypts the encrypted proof,  $\text{Dec}(\bar{k}, \pi'_j[i]) = \pi_j[i]$
- generates a fresh vector:  $\pi''_j$ , such that its  $i$ -th element equals  $\pi_j[i]$  (i.e.,  $\pi''_j[i] = \pi_j[i]$  and  $|\pi''_j| = |\pi_j|$ ) and the rest of its elements are dummy.
- sets  $q := (\hat{k}_j, i)$  and calls  $\text{PoRID.identify}(\pi''_j, q, pp) \rightarrow I_j$ . If  $I_j = \mathcal{S}$ , then it increments  $y_{\mathcal{S}}$  by 1. Otherwise, it does nothing.

Let  $K_6$  be the time that SC finishes all the above checks.

8. **Coin Transfer.**  $\text{RCPoRP.pay}(\mathbf{y}, T_{cp}, a, p_{\mathcal{S}}, \text{coin}_{\mathcal{C}}^*, \text{coin}_{\mathcal{S}}^*)$

- If SC receives “pay” message at time  $T_2$ , where  $a = 0$  or  $\text{coins}_{\mathcal{S}}^* < p_{\mathcal{S}}$ , then it sends  $\text{coin}_{\mathcal{C}}^*$  coins to  $\mathcal{C}$  and  $\text{coin}_{\mathcal{S}}^*$  coins to  $\mathcal{S}$ . Otherwise (i.e., they reach an agreement), they take the following step.
- If SC receives “pay” message and statement  $\tilde{x}_{cp} \in T_{cp}$  to SC at time  $L > K_6$ , it checks the validity of the statement by calling the SAP’s verification which returns  $d''$ . SC only proceeds to the next step if  $d'' = 1$ .
- SC distributes the coins to the parties as follows:
  - $\text{coin}_{\mathcal{C}}^* - o \cdot (z - y_{\mathcal{S}}) + l \cdot (y_{\mathcal{S}} - y_{\mathcal{C}})$  coins to  $\mathcal{C}$ .
  - $\text{coin}_{\mathcal{S}}^* + o \cdot (z - y_{\mathcal{S}}) + l \cdot (y_{\mathcal{C}} - y_{\mathcal{S}})$  coins to  $\mathcal{S}$ .

## O Remarks on the RC-PoR-P

*Remark 1.* The client or server even during the private time bubble can promise to a third party the amount of coin it owns in the smart contract, SC. In this section, we briefly explain how the RC-PoR-P can support it if the protocol is slightly adjusted. For the sake of simplicity, we assume  $\mathcal{S}$  will receive  $\text{coin}_{\mathcal{S}}$  coins after the bubble bursts and wants to promise  $\hat{\text{coin}}_{\mathcal{S}}$  coins (where  $\hat{\text{coin}}_{\mathcal{S}} \leq \text{coin}_{\mathcal{S}}$ ) to the third party  $\mathcal{D}$  within this bubble. First,  $\mathcal{S}$  proves to  $\mathcal{D}$  that it will receive  $\text{coin}_{\mathcal{S}}$  coins after the bubble bursts. To do that, it sends the RC-PoR-P’s transcripts (that includes the existing proofs and queries) to  $\mathcal{D}$ , which can verify  $\mathcal{S}$ ’s claim, as they are publicly verifiable.

If  $\mathcal{D}$  is convinced, then  $\mathcal{S}$  and  $\mathcal{D}$  invoke a new instance of the SAP and insert the value  $\hat{\text{coin}}_{\mathcal{S}}$  into the SAP’s private statement. Invoking the SAP results in a new smart contract, e.g.,  $\text{SC}_{\text{SAP}_3}$ . Next, if both parties agree on the parameters of the  $\text{SC}_{\text{SAP}_3}$ , then  $\mathcal{S}$  sends the address of the  $\text{SC}_{\text{SAP}_3}$  to the RC-PoR-P’s contract, i.e., SC. At this point,  $\mathcal{D}$  can check whether  $\mathcal{S}$  has inserted a correct address into SC. If the check passes, it can conclude that it will receive the agreed amount. When the bubble bursts, SC transfers the  $\mathcal{C}$ ’s share to  $\mathcal{C}$  as before. But, SC distributes the  $\mathcal{S}$ ’s coins if  $\mathcal{S}$  or  $\mathcal{D}$  sends to it a valid proof for the above (new) private statement (in addition to the proofs required in Phase 8 of the original RC-PoR-P). Upon receiving that proof, SC invokes  $\text{SC}_{\text{SAP}_3}$  to check the validity of the proof. If the check passes, then SC sends  $\hat{\text{coin}}_{\mathcal{S}}$  to  $\mathcal{D}$  and  $\text{coin}_{\mathcal{S}} - \hat{\text{coin}}_{\mathcal{S}}$  to  $\mathcal{S}$ . As it is evident, this approach leaks no information about the amount (including  $\hat{\text{coin}}_{\mathcal{S}}$ ) during the bubble to the public, due to the SAP’s security. The above idea can be further extended to support multiple third parties.

*Remark 2.* In the protocol, the pads are added *after* the actual values are encrypted. This is done to save computation cost. Otherwise (if the pads are added prior to the encryption), then the pads would have to be encrypted too, which imposes additional computation cost.

*Remark 3.* The reason in step 7(e)iii  $\pi_i''$  is constructed is to let SC make *black-box* use of `PoRID.identify(.)`. Alternatively, SC could decrypt all proofs in  $\text{Enc}(\bar{k}, \pi_i)$  and pass them to `PoRID.identify(.)`. However, this approach would impose a high cost, as all proofs have to be decrypted.

*Remark 4.* In the protocol, for the sake of simplicity, it is assumed that the cost imposed by a malicious  $\mathcal{C}$  to the arbiter (to resolve a dispute) is the same as the cost imposed by a malicious  $\mathcal{S}$ . To relax the assumption, we can simply introduce another parameter  $l'$ . We let  $l$  and  $l'$  be the amount of coin malicious  $\mathcal{C}$  and  $\mathcal{S}$  must pay to the arbiter respectively. In this case, (a) in step 2b,  $\mathcal{C}$  appends  $l'$  to  $cp$  and (b) in the coin transfer phase, the amounts of coin each party receives would be as follow:  $\text{coin}_{\mathcal{C}}^* - o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_{\mathcal{C}} + y'_{\mathcal{C}})$  coins to  $\mathcal{C}$ ,  $\text{coin}_{\mathcal{S}}^* + o \cdot (z - y_{\mathcal{S}}) - l' \cdot (y_{\mathcal{S}} + y'_{\mathcal{S}})$  coins to  $\mathcal{S}$ , and  $l \cdot (y_{\mathcal{C}} + y'_{\mathcal{C}}) + l' \cdot (y_{\mathcal{S}} + y'_{\mathcal{S}})$  coins to the arbiter.

*Remark 5.* As stated previously, the proofs are sent to the contract to avoid running into the deniability issue, i.e., a malicious  $\mathcal{C}$  wrongly claims that  $\mathcal{S}$  never sent a proof or a malicious  $\mathcal{S}$  claims that it sent its proof to  $\mathcal{C}$ . But, in the case where the proof size is large and posting it to the smart contract imposes a high cost, the parties can use the following technique to directly communicate with each other to send and receive the proof.  $\mathcal{S}$  sends a signed proof directly to  $\mathcal{C}$  which needs to send back to  $\mathcal{S}$  a signed acknowledgment stating that it received the proof, within a fixed period. If  $\mathcal{S}$  does not receive a valid acknowledgment on time, it sends the signed proof to the arbiter. Also, if  $\mathcal{C}$  does not receive the proof on time, it needs to let the arbiter know about it. In this case, if the arbiter has already received the proof, it sends the proof to  $\mathcal{C}$  which allows  $\mathcal{C}$  to perform the rest of the computation. On the other hand, if the arbiter does not have the proof, it asks  $\mathcal{S}$  to send to it  $\mathcal{C}$ 's acknowledgment. If  $\mathcal{S}$  provides a valid acknowledgment, then the arbiter considers  $\mathcal{C}$  as a misbehaving party; otherwise (if  $\mathcal{S}$  could not provide the acknowledgment), it considers  $\mathcal{S}$  as a misbehaving one. But, if both  $\mathcal{S}$  and  $\mathcal{C}$  behave honestly in sending and receiving the proof, then they do not need to invoke the arbiter for this matter and the proof is never stored on the blockchain.

## P Full Evaluation

In this section, we provide a full analysis of the RC-PoR-P. Tables 5 depicts the protocol's runtime. Moreover, we compare the RC-PoR-P with the (i) zero-knowledge contingent (publicly verifiable) PoR payment in [15] and (ii) fair PoR payment scheme in [3]. To conduct a concrete cost study, we have implemented the RC-PoR-P. The protocol's off-chain and on-chain parts have been implemented in C++ and Solidity respectively. To carry out the off-chain experiment, we used a server with dual Intel Xeon Gold 5118, 2.30 GHz CPU and 256 GB RAM. To conduct the on-chain experiment, we used a MacBook Pro laptop with quad-core Intel core i5, 2 GHz CPU and 16 GB RAM that interacts with the Ethereum private blockchain. We did not take advantage of parallelisation, although our protocol is highly parallelisable. We ran the experiment on average 10 times. In the experiment, we used the SHA-2 hash function and set its output's length and the security parameter to 128 bits. We used random files whose size are in the range [64 MB, 4 GB]. We set the size of every block to 128 bits, similar to the scheme in [47]. This results in the number of file blocks in the range  $[2^{22}, 2^{28}]$ . Since in the experiment we used relatively large file sizes, to lower on-chain

transaction costs, we let  $\mathcal{S}$  send the proofs directly to  $\mathcal{C}$ , by using the technique explained in Appendix O. The prototype implementation utilises the Cryptopp<sup>9</sup> library for cryptographic primitives and the GMP<sup>10</sup> library for arbitrary precision arithmetics. The protocol’s off-chain and on-chain source code is publicly available in [1] and [2] respectively.

Table 5: RC-PoR-P off-chain run-time (in seconds), for  $z$  verifications.  $z'$  : the max number of complaints, and  $m$ : the number of blocks.

Phase	Off-chain cost						
	$m : 2^{22}$	$m : 2^{23}$	$m : 2^{24}$	$m : 2^{25}$	$m : 2^{26}$	$m : 2^{27}$	$m : 2^{28}$
Client-side Init.	23.1	45.5	89.7	185.8	413	732.1	1596.6
Server-side Init.	20.9	36.5	73.2	144.6	395.4	728.8	1548.8
Client-side Query Gen.	-	-	-	-	-	-	-
Server-side Proof Gen.	18.4z	30.4z	57.4z	106.8z	376.1z	703.1z	1320.7z
Client-side Proof Ver.	0.09z	0.11z	0.12 · z	0.16z	0.18z	0.21z	0.24z
Arbiter-side Dispute Res.	$2 \cdot 10^{-5} z'$	$4 \cdot 10^{-5} z'$	$8 \cdot 10^{-5} z'$	$8 \cdot 10^{-5} z'$	$9 \cdot 10^{-5} \cdot z'$	$9 \cdot 10^{-5} z'$	$10^{-4} z'$

**Computation Cost.** In our analysis, the cost of erasure-coding a file is not taken into consideration, as it is identical in all PoR schemes. We first analyse the computation cost of the RC-PoR-P.  $\mathcal{C}$ ’s cost is as follows. In the client-side initiation phase (i.e., Phase 2),  $\mathcal{C}$ ’s cost in step 2a involves  $m \cdot \sum_{i=1}^{\log_2(m)} \frac{1}{2^i}$  invocations of a hash function to construct a Merkle tree on the encoded file,  $u^*$ . So its complexity in this step is  $O(m)$ . Also, its total cost in steps 2c includes two invocations of the hash function when it calls `SAP.init(.)` twice, one for  $qp$  and the other for  $cp$ . Therefore,  $\mathcal{C}$ ’s total complexity in this phase is  $O(m)$ , which is a one-off cost. In this phase,  $\mathcal{C}$ ’s off-chain *run-time* increases gradually (i.e., 23.1, 45.5, ..., 1596.6 seconds) when the number of file blocks increases (i.e.,  $2^{22}, 2^{23}, \dots, 2^{28}$  blocks). This phase also costs it  $123 \cdot 10^{-5}$  ether, which stems from the SC and SAP contracts’ deployment and initiation. In the client-side query generation phase (i.e., Phase 4), in step 4a,  $\mathcal{C}$  calls `PoRID.genQuery(.)` that involves picking a random key for the PRF. In step 4b,  $\mathcal{C}$  uses the symmetric-key encryption to encrypt that single key. Note,  $\mathcal{C}$ ’s off-chain run-time in this phase is negligibly small. This phase also costs it  $6 \cdot 10^{-5} \cdot z$  ether for sending a transaction to SC. In the client-side proof verification phase (i.e., Phase 6), in step 6b,  $\mathcal{C}$  for each verification decrypts and verifies the proofs (i.e., the Merkle tree paths) which mainly involves  $\phi \cdot (\log_2(m) + 1)$  invocations of the symmetric key encryption and  $\phi \cdot \log_2(m)$  invocations of the hash function. So,  $\mathcal{C}$ ’s total complexity in Phase 6 is  $O(z \cdot \phi \cdot \log_2(m))$ . Also,  $\mathcal{C}$ ’s off-chain run-time in this phase is very low and gradually grows (i.e.,  $0.09 \cdot z, 0.11 \cdot z, \dots, 0.24 \cdot z$  seconds) when the number of file blocks increases.

Now, we analyse  $\mathcal{S}$ ’s computation cost. In the server-side initiation phase (i.e., Phase 3), in step 3b,  $\mathcal{S}$  calls `SAP.agree(.)` twice, to check and agree on parameters of  $cp$  and  $qp$ , that results in four invocations of the hash function. Also, in step 3c,

<sup>9</sup> <https://cryptopp.com>

<sup>10</sup> <https://gmplib.org>

$\mathcal{S}$  calls `PoRID.serve(.)` that requires it to construct a Merkle tree on the file. This results in  $O(m)$  invocations of the hash function. So,  $\mathcal{S}$ 's total complexity to check and agree on the protocol's parameters is  $O(m)$ .  $\mathcal{S}$ 's off-chain run-time in this phase increases gently (i.e., 8.9, 16.5, ..., 548.8 seconds) when the number of blocks grows. This phase also costs it  $9 \cdot 10^{-5}$  ether for sending transactions to the smart contracts. In the server-side proof generation phase (i.e., Phase 5), in step 5b,  $\mathcal{S}$  decrypts a single value for each verification. Also, in step 5c, checks a query's correctness that imposes a negligible computation cost. In the same step, it generates and encrypts proofs that require  $\phi \cdot \log_2(m)$  invocations of the hash function (to generate Merkle tree's paths) and  $\phi \cdot (\log_2(m) + 1)$  invocations of symmetric key encryption, for each verification. Therefore,  $\mathcal{S}$ 's total complexity in Phase 5 is  $O(z \cdot \phi \cdot \log_2(m))$ . In this phase,  $\mathcal{S}$ 's off-chain run-time slowly grows (i.e.,  $20.4 \cdot z$ ,  $36.5 \cdot z$ , ...,  $1596.6 \cdot z$  seconds) when the number of the file blocks increases.

Next, we analyse an arbiter's cost in the RC-PoR-P in the dispute resolution phase (i.e., Phase 7). Note, if both  $\mathcal{C}$  and  $\mathcal{S}$  behave honestly, then the arbiter is not invoked, accordingly it performs no computation. Therefore, we consider the case where one of the parties complains about its counter-party's behaviour. First, we evaluate the arbiter's cost when it is invoked by an honest  $\mathcal{S}$ . In step 7(c)i, the arbiter invokes the hash function twice to check the correctness of the statement,  $\ddot{x}_{qp}$ . In step 7(c)iii, it decrypts  $|\mathbf{v}_S|$  queries, where  $|\mathbf{v}_S|$  is the total number of verifications that  $\mathcal{S}$  has complained about and  $|\mathbf{v}_S| \leq z$ . Note, in the same step the arbiter calls `PoRID.checkQuery(.)` to check the queries; however, its cost is negligibly small. Now, we evaluate the arbiter's cost when it is invoked by an honest  $\mathcal{C}$ . In step 7(e)i, the arbiter invokes the hash function twice to check the correctness of the statement,  $\ddot{x}_{qp}$ , sent by  $\mathcal{C}$ . Also, in step 7(e)iii, it decrypts  $|\mathbf{v}_C|$  queries, where  $|\mathbf{v}_C|$  is the total number of verifications that  $\mathcal{C}$  complained about. In the same step, the arbiter also invokes the hash function  $|\mathbf{v}_C| \cdot \log_2(m)$  times and the symmetric key encryption  $|\mathbf{v}_C| \cdot (\log_2(m) + 1)$  times in total, to process  $\mathcal{C}$ 's complaints. Thus, the arbiter's cost, in Phase 7 is at most  $O(z' \cdot \log_2(m))$ , where  $z' = \text{Max}(|\mathbf{v}_C|, |\mathbf{v}_S|)$  and  $z' \leq z$ . Note that due to the use of the proof of misbehaviour in the protocol, the arbiter's computation cost is about  $\frac{1}{\phi} = \frac{1}{460}$  of its computation cost in the absence of such technique where it has to check all  $\phi$  proofs for each verification. The arbiter's off-chain run-time is very low and increases gently (i.e.,  $2 \cdot 10^{-5} \cdot z'$ ,  $4 \cdot 10^{-5} \cdot z'$ , ...,  $10^{-4} \cdot z'$  seconds) when the number of the file blocks increases. This phase also imposes  $10^{-4}$  ether to the arbiter as a result of sending its inputs to SC. The smart contract performs computations only in the coin transfer phase (i.e., Phase 8) that involves two invocations of the hash function, in step 8b, to check the correctness of the statement,  $\ddot{x}_{cp}$ . So its computation complexity is constant,  $O(1)$ . This phase imposes  $6 \cdot 10^{-5}$  ether to the party that calls `RCPoRP.pay(.)`.

**Communication Cost.** Since in the above analysis we have taken into account the gas cost that also covers the cost of transacting with the contracts, below we focus on the off-chain communication costs. We first analyse the  $\mathcal{C}$ 's communication cost. In the client-side initiation phase (i.e., Phase 2), in step 2e, it sends  $u^*$ ,  $z$ ,  $\ddot{x}_{qp}$ , and  $\ddot{x}_{cp}$  to  $\mathcal{S}$ , where (a)  $\ddot{x}_{qp}$  contains padding information whose size is negligible, the symmetric-key encryption's key of size 128-bit and a random value of size 128-bit, and (b)  $\ddot{x}_{cp}$  contains five small-sized values, and a random value of size 128-bit. Therefore,  $\mathcal{C}$ 's communication cost in the initiation phase is  $\|u^*\| + 384$  bits or  $O(\|u^*\|)$ . In the dispute resolution phase (i.e., Phase 7),  $\mathcal{C}$ 's communication cost in step 7d is low; because, in this step, it sends statement  $\ddot{x}_{qp}$  and its complaint  $\mathbf{m}_c$  to the arbiter, such

that  $\ddot{x}_{qp}$  contains (a) padding information whose size is at most a few bits and (b) the symmetric-key encryption’s key whose size is 128 bits. Moreover,  $\mathbf{m}_c$  contains at most  $z$  invalid paths of the Merkle tree. Thus,  $\mathcal{C}$ ’s total communication cost (excluding the initiation phase) is  $z \cdot \log_2(\|u^*\|) + 128$  bits or  $O(z \cdot \log_2(\|u^*\|))$ .

Now, we analyse  $\mathcal{S}$ ’s communication cost. In the server-side proof generation phase (i.e., Phase 5),  $\mathcal{S}$  in step 5c, for each verification sends out a proof vector  $\boldsymbol{\pi}_j^*$ . Therefore, its complexity for  $z$  verifications is  $O(z \cdot \|\boldsymbol{\pi}_j^*\|)$ . Next, we provide a concrete communication cost of  $\mathcal{S}$ , for each verification, in Phase 5, where the size of encoded file  $u^*$  is 4 GB (or  $2^{28}$  blocks). In general, a proof (or path) in a Merkle tree, has  $\log_2(m) + 1$  elements, where  $m$  is the number of leaf nodes. Therefore, for the above  $u^*$ , a proof would contain  $\log_2(2^{28}) + 1 = 29$  elements. If we set the hash function output’s length to 128 bits, then a proof’s total bit-size would be 3712. Also, if we set the number of challenged blocks to 460, then the total size of proofs (related to the challenged blocks) would be about  $214 \times 10^3$  bytes<sup>11</sup>. So, in Phase 5, for  $z$  verifications its cost is  $214 \times 10^3 \cdot z$  bytes.  $\mathcal{S}$ ’s communication cost in the dispute resolution phase (i.e., Phase 7) is low. Because it sends  $\ddot{x}_{qp}$  and its complaint  $\mathbf{m}_s$  to the arbiter, where  $\ddot{x}_{qp}$  contains padding information whose size is negligible and the symmetric-key encryption’s key, whose size is at most 128 bits. Also,  $\mathbf{m}_s$  contains at most  $z$  negligible size indices, where each index is at most a few bits. Thus,  $\mathcal{S}$ ’s total communication cost is  $O(z \cdot \|\boldsymbol{\pi}_j^*\|)$ . The arbiter’s communication cost is constant, as it only sends a transaction containing four values to SC in step 7f, in the dispute resolution phase (i.e., Phase 7).

**Comparison.** The fair PoR scheme in [3] assumes that the client is fully trusted. This protocol mainly uses a time-lock puzzle scheme, efficient MAC-based PoR, and a smart contract. In the initiation phase, the client generates  $z$  puzzles that involve  $z$  modular exponentiations. In the same phase, the client generates  $m$  permanent MACs and  $z\phi$  disposable MACs. In this scheme, generating each MAC involves a couple of modular multiplication and additions. Therefore, the initiation phase involves  $O(m)$  modular exponentiations and  $O(m + z\phi)$  modular additions and multiplications. Once the puzzles are given to  $\mathcal{S}$ , it needs to *continuously* solve different puzzles until all  $z$  verifications end. This means  $\mathcal{S}$  performs the exponentiations even between two consecutive verifications where no challenges have been sent to it. This requires  $\mathcal{S}$  to perform  $O(Tz)$  exponentiations and  $z$  modular additions and multiplications, where  $T$  is a time parameter in this scheme. Also, for  $z$  verification,  $\mathcal{S}$  needs to perform  $O(z\phi)$  multiplications and additions to generate  $z$  proofs. Likewise, a verifier (i.e., a smart contract) performs  $O(z\phi)$  multiplications and additions to verify all proofs.

Now we turn our attention to the scheme in [15]. As we showed in Section 4, this scheme is not secure against a malicious client. It is based on BLS signatures, (compound) zero-knowledge proofs, and hash functions. In the initiation phase,  $\mathcal{C}$  needs to generate a signature for each file block which involves  $O(m)$  exponentiations and  $O(m)$  hash function invocations. For  $\mathcal{S}$  to generate  $z$  proofs, it requires (i)  $O(z\phi)$  exponentiations to combine the signatures, (ii)  $O(1)$  invocations of the hash function, and (iii)  $O(z\phi)$  invocations of the zero-knowledge proof. The scheme imposes the same computation complexity to the verifier as it does to the prover. Since both schemes in [3] and [15] use homomorphic tags, the proofs for each verification can be combined which results in constant proof size, i.e.,  $O(1)$ . Moreover, the above two schemes do

<sup>11</sup> As shown in [9], to ensure 99% of file blocks is retrievable, it would be sufficient to set the number of challenged blocks to 460.

not address the privacy issue we highlighted in Section 4. On the other hand, the RC-PoR-P is secure against a malicious client (as well as a malicious server) and addresses the privacy issue. Similar to the other two schemes, its initiation complexity is  $O(m)$ ; however, unlike the other two schemes, it does not require any modular exponentiations. Instead, it involves only invocations of hash function which imposes a much lower overhead. Moreover, unlike the other two schemes that have  $O(z\phi)$  complexity in the prove and verify phases, RC-PoR-P's complexity, in theory, is slightly higher, i.e., it is  $O(z\phi \log_2(m))$ . However, the extra factor:  $\log_2(m)$  is not very high in practice. For instance, for a 4-GB file (or  $2^{28}$  blocks),  $\log_2(m)$  is only 28. The RC-PoR-P's prove and verify algorithms, similar to the ones [3], involve only symmetric key operations; in contrast, the ones in [15] require asymmetric key operations. Furthermore, the proof size (for each verification) complexity is larger than the other two schemes; nevertheless, each message length in RC-PoR-P is much shorter than the one in [15], i.e., 128-bit vs 2048-bit. Thus, overall RC-PoR-P is computationally more efficient than [15,3] while offering stronger security guarantees.<sup>12</sup>

---

<sup>12</sup> Note, Campanelli *et al.* in [15] provide an implementation of zkCSP for publicly and privately verifiable PoRs. However, we have been informed by Campanelli that the total size of the outsourced file used in their experiment is at most 256 bits, which is small in the context of PoR where a large file is often outsourced to a remote server. Therefore, we could not compare their protocols' run-time with RC-PoR-P's run-time when a large file is outsourced.