REPHRAIN

Protecting citizens online

Recurring Contingent Service Payment

Aydin Abadi, University College London

Steven J. Murdoch, University College London

Thomas Zacharias, University of Edinburgh

October 2022

UK Research and Innovation · University of Bristol · UCL · The University of Edinburgh · King's College London · University of Bath

Recurring Contingent Service Payment†

Aydin Abadi‡[1] Steven J. Murdoch§[1] Thomas Zacharias¶[2]

[1] University College London
[2] University of Edinburgh

Abstract. Fair exchange protocols let two mutually distrustful parties exchange digital data in a way that neither party can cheat. They have various applications such as the exchange of digital items, or the exchange of digital coins and digital services between a buyer and seller. At CCS 2017, two blockchain-based protocols were proposed to support the fair exchange of digital coins and a certain service; namely, "proofs of retrievability" (PoR). In this work, we identify two notable issues of these protocols, (1) *waste of the seller's resources*, and (2) *real-time information leakage*. To rectify these issues, we formally define and propose a blockchain-based *generic* construction called "*recurring contingent service payment*" (RC-S-P). RC-S-P lets a fair exchange of digital coins and verifiable service occur periodically while ensuring that the buyer cannot waste the seller's resources, and the parties' privacy is preserved. It supports arbitrary verifiable services, such as PoR, or verifiable computation and imposes low on-chain overheads. Also, we present a concrete efficient instantiation of RC-S-P when the verifiable service is PoR. The instantiation is called "*recurring contingent PoR payment*" (RC-PoR-P). We have implemented RC-PoR-P and analysed its cost. When it deals with a 4-GB outsourced file, a verifier can check a proof in 90 milliseconds, and a dispute between prover and verifier is resolved in 0.1 milliseconds.

1 Introduction

Fair exchange is an interesting problem in which two mutually distrustful parties want to swap digital items such that neither party can cheat the other, in the sense that either each party gets the other's item, or neither party does. The problem has been drawing considerable attention, as the Internet's use for conducting business is rapidly growing. It captures various real-world scenarios; for instance, when two parties want to exchange digital items or when a seller wants to sell a digital verifiable service in exchange for digital coins. Solutions to the problem are usually certain cryptographic schemes, called fair exchange protocols, and have been studied for decades. It has been shown that fairness is unachievable without the aid of a trusted third party [19].

With the advent of decentralised cryptocurrencies and blockchain, it seemed fair exchange protocols can be designed without having to rely on a single trusted third party, in the sense that the third party's role can be turned into a computer program, i.e., smart contract, which is maintained and executed by the decentralised blockchain. This ultimately results in a stronger security guarantee, as there would be no need to trust a single entity, anymore. Ever since, various fair exchange protocols that rely on blockchain have been proposed, see Appendix A for a survey. The two (publicly and privately verifiable) schemes of Campanelli *et al.* at CCS 2017 [17] stand out from the rest, as they are the only ones designed so far to facilitate the fair exchange of digital coins and verifiable service (VS), on a blockchain.

Nevertheless, as we will show in this work, the schemes in [17] suffer from two serious issues: (1) *waste of seller's resources* and (2) *real-time information leakage*. In these schemes, a cheating buyer can waste the seller's resources and in certain cases can even get a free ride from the seller, without having to face any repercussions of its misbehaviour; moreover, the schemes leak in real-time non-trivial fresh information

† This work is the extension of our previous work in [5] with two important differences; namely, it offers new (1) generic definitions and (2) generic constructions.

‡ aydin.abadi@ucl.ac.uk
§ s.murdoch@ucl.ac.uk
¶ thomas.zacharias@ed.ac.uk

about the seller and buyer to the public, e.g., deposit's actual amount, proofs' status, buyer's file size, or even the file's location in some situations. The schemes' flaws leading to these issues are generic. If they are not dealt with appropriately, then future (blockchain-based) fair exchange protocols would inherit them.

Our Contributions. In this work, we:

1. identify two issues that the protocols of Campanelli *et al.* [17] suffer from; namely, (1) *waste of seller's resources*, and (2) *real-time information leakage*. We identify two drawbacks in these protocols that led to Issue 1; namely, (a) incomplete fairness definition and (b) mismatch of security assumptions/requirements between primitives. We identify the improper use of public blockchain as the primary factor that led to Issue 2.
2. define and propose the first generic construction, called "*recurring contingent service payment*" (RC-S-P), that addresses the above issues simultaneously. RC-S-P makes black-box use of any scheme that offers a verifiable service.
3. propose the first recurring contingent PoR payment (RC-PoR-P) which is a concrete instantiation of the RC-S-P scheme. By avoiding generic cryptographic tools with substantial overhead and utilising symmetric-key primitives and smart contracts, RC-PoR-P achieves efficiency while preserving all desired RC-S-P properties.
4. implement RC-PoR-P and analyse its cost. Our cost analysis illustrates RC-PoR-P is highly efficient. When it deals with a 4-GB outsourced file, in each verification, a verifier can check a proof in only 90 milliseconds, and a dispute between prover and verifier can be resolved in 0.1 milliseconds. Also, the contracts' computation is constant in file size.

Defining and designing generic RC-S-P is particularly challenging, for three reasons: (i) there exists no generic definition for VS schemes in the literature, (ii) most of the application-specific VS schemes (e.g., proofs of retrievability [58], or verifiable computation [30], verifiable searchable encryption [51]) assume the client is fully trusted, while in a fair exchange setting either party can be an active adversary, and (iii) the majority of VS schemes do not (need to) take the privacy of exchanged messages into account, as they are in the traditional setting where the client and server directly interact with each other, without the involvement of a public blockchain; hence, their messages' privacy can be protected from the public by using standard tools, e.g., secure channels. Our protocols can be used to prevent a variant of "Authorised Push Payment" (APP) fraud, called Purchase fraud, where the sender of an item may wish to receive a certain amount of coin without sending the item[3].

2 Related Work

In this section, we summarise related work. In Appendix A, we present a detailed survey. Maxwell [47] 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) [31] that yields more efficient zk proofs, the scheme was modified to use zk-SNARKs. Later, Campanelli *et al.* [17] 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. To improve 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) [58]. 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 [28] 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.* [54] show that by relying on a

[3] We refer readers to [4,61] for further discussion about APP fraud.

stronger assumption, the zkCSP remains secure. Tramer *et al.* [60] propose a fair exchange scheme that uses trusted hardware and Ethereum smart contracts. Dziembowski *et al.* [26] propose FairSwap, a fair exchange scheme using the Ethereum smart contracts and the notion of proof of misbehaviour [18]. Later, Eckey *et al.* [27] 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 [3,25]. The scheme in [3] uses message authentication codes (MACs) and time-lock puzzles. The one in [25] 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 [3,25] 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

We write $x \xleftarrow{\$} X$ to denote that x is chosen uniformly at random from set X. We write $\mathsf{negl}(\lambda)$ to denote that a function is negligible in λ, i.e., asymptotically smaller than the inverse of any polynomial. In the formal definitions in this paper, we use the notation $\Pr\left[\frac{\mathsf{Exp}}{\mathsf{Cond}}\right]$, where Exp is an experiment that involves an adversary $\mathcal{A}$, and Cond is the set of the corresponding winning conditions for $\mathcal{A}$. We summarise our notations in Table 1. Similar to the *optimistic* fair cryptographic protocols that aim efficiency, e.g., in [9,10,12,23], we assume the existence of a trusted third party arbiter (e.g., secure hardware) which remains offline most of the time but can be invoked to resolve any dispute.

Table 1: Notation Table.

Setting	Symbol	Description
Generic	z	Number of verifications
	λ	Security parameter
	PRF	Pseudorandom function
	ζ	PRF's description
	Pr	Probability
	Com	Commit algorithm in commitment
	Ver	Verify algorithm in commitment
	μ	Negligible function
	H	Hash function
	MT	Merkle tree
	sk, pk	Secret and public keys
	PoR	Proof of retrievability
	u	Service input, e.g., file
	u^*	Encoded input
	σ	Metadata
	ω_σ	Proof for metadata's correctness
	e	$e := (\sigma, \omega_\sigma)$
	pp	Public parameter
	$q, \boldsymbol{q}$	Query and query vector
	$\boldsymbol{\omega}_q$	Proof for $\boldsymbol{q}$'s correctness
	c	$c := (\boldsymbol{q}, \boldsymbol{\omega}_q)$
	$\pi, \boldsymbol{\pi}$	Service proof and proof vector
	VS	Verifiable service
	VSID	Verifiable service with identifiable abort
	RCSP	Recurring contingent service payment
	SAP	Statement agreement protocol
	$\mathcal{C}$	Client
	$\mathcal{S}$	Server
	$\mathcal{A}r$	Arbiter
	SC	Smart contract

Setting	Symbol	Description
Generic	F	Function run on u^* by $\mathcal{S}$
	M	Metadata generator function
	Q	Query generator function
	aux	Auxiliary information
	m	Number of a file blocks, $m = \lvert u^* \rvert$
	$\lvert\lvert u^* \rvert\rvert$	Bit size of u^*
	δ	Proof of F's evaluation correctness
	j	Verification index, $1 \leq j \leq z$
	adr	Address
	ϕ	Number of challenged blocks
RC-PoR-P or RC-S-P	r_{qp}, r_{cp}	Random values
	$\ddot{x}_{qp}, \ddot{x}_{cp}$	$\ddot{x}_{qp} := (qp, r_{qp}), \ddot{x}_{cp} := (cp, r_{cp})$
	$coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}}$	Encoded coins deposited by $\mathcal{C}$ and $\mathcal{S}$
	enc	Encoding/decoding functions $enc := (E, D)$
	$m_{\mathcal{C}}, m_{\mathcal{S}}$	Complaints of $\mathcal{C}$ and $\mathcal{S}$
	pad_π, pad_q	Number of elements used to pad π and q
	$y_{\mathcal{C}}, y_{\mathcal{S}}$	Number of times $\mathcal{C}$ and $\mathcal{S}$ misbehave towards each other
	$y'_{\mathcal{C}}, y'_{\mathcal{S}}$	Number of times $\mathcal{C}$ and $\mathcal{S}$ unnecessarily invoke $\mathcal{A}r$
	cp	Coin secret parameters
	T_{cp}	Coin encoding token
	qp	Query/proof secret parameters
	T_{qp}	Query/proof encoding token
	T	$T := (T_{cp}, T_{qp})$
	$g_{\mathcal{C}}, g_{\mathcal{S}}$	Commitments computed by $\mathcal{C}$ and $\mathcal{S}$
	pl	Public price list: $\{(o, l), ..., (o'', l'')\}$
	o	Coins $\mathcal{S}$ must get for a valid proof, where $o \in pl$
	l	Coins $\mathcal{A}r$ must get for resolving a dispute, where $l \in pl$
	l_{max}	$Max(l, ..., l'')$
	o_{max}	$Max(o, ..., o'')$
	$p_{\mathcal{S}}$	Total coins $\mathcal{S}$ should deposit

3.2 Smart Contract

Cryptocurrencies, such as Bitcoin [53] and Ethereum [63], beyond offering a decentralised currency, support computations on transactions. In this setting, often a certain 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 executed by all parties maintaining the cryptocurrency, when the program inputs are provided by transactions. 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"*.

3.3 Building Blocks

We outline the main cryptographic primitives that we utilize in our protocols. For completeness, we provide a detailed description of the said primitives in Appendix B.

- *Pseudorandom Fuction* (PRF): we apply a pseudorandom function $\texttt{PRF} : \{0,1\}^{\psi} \times \{0,1\}^{\eta} \rightarrow \{0,1\}^{\iota}$ that takes as input a random ψ-bit key and η-bit message and outputs a ι-bit pseudorandom value (cf. Appendix B.1).
- *Commitment Scheme*: we deploy a binding and hiding commitment scheme that comprises the *commit* and *open* phases. In the commit phase, the sender commits to a message x as $\texttt{Com}(x, r) = \texttt{Com}_x$, that involves a secret value, r. In the open phase, the sender sends the opening $\ddot{x} := (x, r)$ to the receiver which verifies its correctness: $\texttt{Ver}(\texttt{Com}_x, \ddot{x}) \stackrel{?}{=} 1$ and accepts if the output is 1 (cf. Appendix B.2).
- *Publicly Verifiable Non-interactive Zero-knowledge Proof* (NIZK): is a non-interactive proof where a prover $\mathcal{P}$, given a witness w for some statement x in an NP language L, wants to convince in zero-knowledge a verifier $\mathcal{V}$ of the validity of $x \in L$. A NIZK is publicly verifiable when any party can verify the validity of $x \in L$ by obtaining the proof (cf. Appendix B.3).
- *Symmetric-key Encryption Scheme*: it consists of a key generation algorithm `SKE.keyGen`, an encryption algorithm `Enc`, and a decryption algorithm `Dec`. We require that the scheme satisfies IND-CPA security (cf. Appendix B.4).
- *Digital Signature Scheme*: it consists of a key generation algorithm `Sig.keyGen`, a signing algorithm `Sig.sign`, and a verification algorithm `Sig.ver`. We require that the digital signature scheme satisfies EUF-CMA security (cf. Appendix B.5).
- *Merkle Tree*: A Merkle tree scheme [48,49] is a data structure often used for efficiently checking the integrity of an outsourced file. The Merkle tree scheme includes three algorithms; namely, `MT.genTree`, `MT.prove` and `MT.verify`. Briefly, the first algorithm constructs a Merkle tree on file blocks, the second generates a proof of a block's (or set of blocks') membership, and the third verifies the proof (cf. Appendix B.6).

3.4 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 [39,58]. A PoR scheme comprises of five algorithms:

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

- $\texttt{PoR.verify}(\pi, \boldsymbol{q}, k, pp) \rightarrow d \in \{0,1\}$. It is run by $\mathcal{C}$. It takes as input π, $\boldsymbol{q}$, k, and pp. It outputs 0 if it rejects the proof, or 1 if it accepts the proof.

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^*, σ) output by $\texttt{PoR.setup}(1^\lambda, u, k)$, and any query $\boldsymbol{q}$, 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 [58], 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 $\texttt{PoR.keyGen}(1^\lambda)$ 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 $\texttt{PoR.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma, pp)$ and returns the output to $\mathcal{A}$. Also, upon receiving the output of $\texttt{PoR.setup}(1^\lambda, u, k)$, $\mathcal{A}$ can ask $\mathcal{E}$ to run $\texttt{PoR.genQuery}(1^\lambda, k, pp) \rightarrow \boldsymbol{q}$ and give the output to it. $\mathcal{A}$ can locally run $\texttt{PoR.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \pi$ to get its outputs as well.
3. $\mathcal{A}$ can request from $\mathcal{E}$ the execution of $\texttt{PoR.verify}(\pi, \boldsymbol{q}, k, pp)$ for any u used to query $\texttt{PoR.setup}(\cdot)$. 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 σ 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 ϵ fraction of verification challenges (for a certain file). Informally, a PoR scheme supports extractability, if there is an extractor algorithm $\texttt{Ext}(k, \sigma, \hat{\mathcal{A}}_\epsilon)$, that takes as input the key k, metadata σ, 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 $\hat{\mathcal{A}}_\epsilon$.

Definition 1 (ϵ-soundness). *A PoR scheme is ϵ-sound if there exists an extraction algorithm $\texttt{Ext}(\cdot)$ such that, for every adversary $\mathcal{A}$ who plays experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{PoR}}$ and outputs an ϵ-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 $\mathsf{negl}(\lambda)$ probability. Formally:*

$$\Pr\left[\begin{array}{l}\texttt{PoR.keyGen}(1^\lambda) \rightarrow k := (sk, pk)\\ \mathcal{A}(1^\lambda, pk) \rightarrow u\\ \texttt{PoR.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma, pp)\\ \mathcal{A}(u^*, \sigma, pp) \rightarrow state\\ \texttt{PoR.genQuery}(1^\lambda, k, pp) \rightarrow \boldsymbol{q}\\ \Big(\big(\mathcal{A}(\boldsymbol{q}, state) \rightarrow \pi\big) \rightleftharpoons \big(\texttt{PoR.verify}(\pi, \boldsymbol{q}, k, pp)\big)\Big) \rightarrow \hat{\mathcal{A}}_\epsilon\\ \hline \texttt{Ext}(k, pp, \sigma, \hat{\mathcal{A}}_\epsilon) \neq u\end{array}\right] = \mathsf{negl}(\lambda).$$

In contrast to the PoR definition in [39,58] where $\texttt{PoR.genQuery}(\cdot)$ is implicit, in the above definition we have explicitly defined $\texttt{PoR.genQuery}(\cdot)$, as it plays an important role in this paper. Also, there are PoR protocols, e.g., in [52], that do not involve $\texttt{PoR.keyGen}(\cdot)$. Instead, a set of public parameters/keys (e.g., file size or a root of Merkle tree) are output by $\texttt{PoR.setup}(\cdot)$. To make the PoR definition generic to capture both cases, we have explicitly included the public parameters pp in the algorithms' definitions too.

4 Previous Work's Limitations and Our Solution's Overview

In this section, we first elaborate on the limitations of previous work, and then outline how we address them. We focus only on the zero-knowledge contingent service payment (zkCSP) protocols in [17], as they

have been specifically designed for a fair exchange of verifiable services and digital coins, whereas the other protocols studied in Section 2 were designed for a fair exchange of digital items, e.g., file and coins. If they are used for verifiable services, then they would suffer from the same issues as the ones in [17] do.

4.1 Limitations of zkCSP

We first elaborate on the shortcomings of the schemes in [17] and explain the schemes' flaws that caused such shortcomings. Then, we outline why trivial solutions would not work.

Issue 1: Waste of Server's Resources. A malicious client, in each zkCSP scheme, can deviate from the protocol to benefit itself and waste the servers' resources (depending on the service type it may include computation or storage) without having to face any consequence of its misbehaviour. Its misbehaviours include:

(i) not participating in the payment phase despite it has been using the server's service, e.g., the server's storage in PoR.
(ii) participating in the payment phase but making the server generate invalid service proofs. For instance, in PoR, at the setup the client may generate ill-formed tags that prevent the honest server from passing the verification. To give a concrete example, in the privately verifiable PoR [58], at the setup, the malicious client instead of honestly generating a tag σ_i on a file block m_i as $\sigma_i = r_i + \alpha \cdot m_i$, it generates a tag as: $\sigma'_i = r_i + \alpha \cdot m'_i$, where r_i and α are two random values and $m_i \neq m'_i$. In this case, given the file and its maliciously generated tag, the server cannot pass the verification when block m_i is challenged. Hence, it would not get paid.

At first glance, it seems the malicious client can only waste the server's resources without gaining anything. But, as we will show shortly, in the recurring payment (i.e., when the server interacts with a client multiple times and/or the server interacts with multiple clients) the client can collect convincing background information about an honest server. This lets the client conclude that it has been served honestly, although it does not pay the server and does not check the proof. Thus, it can get a free ride from the server.

Sources of Issue 1: Incomplete Fairness Definition and Mismatch of Security Assumptions between Primitives. First, we focus on misbehaviour (i). Briefly, the reason this misbehaviour is allowed in the zkCSP scheme is that the scheme's fairness definition is incomplete. Specifically, the fairness definition (presented in Section 4.1) in [17] only captures the moment when the client and server want to trade proof of service for coins (i.e., fair payment phase). Nevertheless, this definition does not take into account the "resource fairness", i.e., the server is paid for the resources it allocated. The server needs to invest resources (in PoR for a long time) to serve the client before it participates in the fair payment phase. Therefore, in the fair exchange of services and coins, it would not be fair if the client does not participate in the payment phase. Now we move on the misbehaviour (ii). The main reason that is allowed is that the zkCSP scheme uses a subprotocol that offers a weaker security guarantee than required. Specifically, the zkCSP scheme assumes *either party* can be potentially corrupted by an active adversary, but it uses a certain verifiable service protocol that is secure against *only* a malicious server and assumes the client is fully honest. This mismatch of security assumption/requirement lets a malicious client misbehave. The incomplete definition and the lack of rigorous security proof for the concrete instantiation of the zkCSP scheme also played vital roles in misbehaviour (ii) remaining undetected.

Issue 2: Real-time Leakage. The zkCSP protocols leak in real-time non-trivial fresh information, about the server and clients to the public. The leakage includes:

- *Deposit amount.* The amount of deposit placed in the smart contract, swiftly leaks non-trivial information about the client to the public. In the case of PoR, an observer learns the approximate size of outsourced data, service type, or in certain cases even the region of clients' outsourced data, by comparing the amount

of deposit with the service provider's price list which is usually publicly available, e.g., in [6,24,34]. 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)" Interestingly, 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.

- *Proofs' status.* In the traditional setting, the client and server directly interact with each other to verify and prove the integrity of agreed-upon services. In this case, the verification's result is only apparent to them. 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 whether the server has successfully delivered the service. This issue remains even if the service proofs are not stored in plaintext in the blockchain, as *coins transfer* itself reveals the proofs' status. In certain settings, this leakage might be undesirable and could have *immediate* consequences for both the server and (business) clients, e.g., stock value drop [14,36], 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) lets a malicious client 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. 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. For more discussion on proofs' status issue we refer readers to Appendix C.

Source of Issue 2: Improper Use of Public Blockchain. This issue occurred due to the use of a public blockchain for holding and transferring deposits without using any privacy-preserving mechanism to preserve the confidentiality of the deposits' amount, in the real-time from the public view.

Strawman Solutions for the Two Issues To address Issue 1, one may adjust each zkCSP protocol such that it would require the client to deposit coins before the server provides the ZK to it, with the hope that the client cannot avoid depositing after the server provides ZK proofs. Nevertheless, this would not work, as the client after accepting the ZK proof, needs to send a confirmation message to the contract. A malicious client can avoid doing so or make the server compute invalid (PoR) proofs, that ultimately lets the client 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 solve (only) Issue 1, it imposes a high cost and defeats the purpose of zkCSP design. Because the contract has to *always* be involved to run the verification algorithm that has to be a publicly verifiable one, which often imposes high costs. To address Issue 2, one may use privacy-preserving cryptocurrency frameworks, e.g., Zerocash [15] or Hawk [43]. Although such frameworks partially solve this problem (i.e., they can hide deposit amount but not proofs' status), they impose additional high cost to their users, as each transaction involves a generic proofs system that are computationally expensive. Also, 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.

4.2 Overview of Our Solution

Addressing Issue 1. To prevent a malicious client from wasting server's resources, we use a combination of the following techniques. First, we upgrade a verifiable service scheme to a "verifiable service with identifiable abort" (VSID). This guarantees that not only the service takes into consideration that the client can be

malicious too, but also the public or an arbiter can identify the misbehaving party and resolve any potential disputes between the two. Second, we require a client to deposit its coins to the contract right before it starts using the service (e.g., in the case of PoR before it uploaded its data to the server) and it is forced to provide correct inputs via NIZK; otherwise, its deposit is sent to the server. Third, we require parties to post (some of) their messages to the contract, to avoid any potential repudiation issue. Forth, we allow the party which resolves disputes to get paid by a corrupt party. Now we explain how the solution works. The client before using the service, deposits a fixed amount of coins in a smart contract, where the deposit amount covers the service payment: o coins, and dispute resolutions' cost: l coins. Also, the server deposits l coins. Then, the client and server engage in the VSID protocol such that (the encryption of) messages exchanged between the parties are put in the contract. The parties perform the verifications locally, off-chain. In the case where a party detects misbehavior, it has a chance to raise a dispute that invokes the arbiter which checks the party's claim, off-chain. The arbiter sends the output of the verification to the contract. If the party's claim is valid, then it can withdraw its coins and the arbiter is paid by the misbehaving party, i.e., l coins from the misbehaving party's deposit are transferred to the arbiter. If the party's claim is invalid, then that party has to pay the arbiter and the other party can withdraw its deposit. In the case where both the client and server behave honestly, then the arbiter is never invoked; in this case, the server (after a fixed time) gets its deposit back and is paid for the service, while the client gets l coins back. Later, we will show in a certain case, i.e., PoR, the arbiter's role can be efficiently played by a smart contract, so its involvement is not needed in that case. Specifically, In a concrete instantiation of the generic solution in which the VS is PoR, we will use a Merkle tree and proof of misbehaviour letting us avoid using NIZK and reduce arbiter-side computation.

Addressing Issue 2. To prevent real-time information leakage, we use the following ideas in our protocols. First, we let the client and server take control of the time of the information release. This enables them to keep the information confidential from the public within an agreed-upon period, and release it when it becomes stale and loses its sensitivity.[4] In particular, the client and server agree on the period in which the information should remain hidden, "private time bubble". During this period, all messages sent to the contract are encrypted and the parties do not raise any dispute. They raise disputes only after the private time bubble ends (or bubble bursts). Nevertheless, the client/server can still find out whether a proof is valid as soon as it is provided by its counter-party, because it can locally verify the proof. Note, due to the above solutions to Issue 1, a malicious client that has seen all stale information on the blockchain, cannot waste the server's resources. Second, to further hide the amount of deposit, we let each party mask its coins, by increasing the actual coins amount to the maximum amount of coins in the server's price list. So, the masked coins hide the actual coins amount from the public. However, this raises another challenge: *how can the mutually untrustful parties claim back their masking coins (i.e., the difference between the maximum and actual coins amount) after the bubble bursts, while hiding the actual coins amount from the public in the private time bubble?* One may want to explicitly state in the contract the amount of masking coins, but this would not suffice, as it would reveal the masking coins' amount to the public at the beginning of the protocol. Our third idea, which addresses this challenge, is to let the client and server, at the beginning of the protocol, agree on a private statement specifying the deposit details, e.g., parties' actual coins amount for the service, dispute resolution, or masking. Later, when they want to claim their coins, they also provide the statement to the contract which checks the statement validity and if it is accepted, it distributes coins according to the statement (and the contract status). We will show how they can efficiently agree on such a statement, by using a statement agreement protocol (SAP). In Section 7.3, we also show how they can promise their locked share of coins to a third party.

Our generic framework that offers the above features is called "recurring contingent service payment" (RC-S-P).

[4] The concept of delayed information release has already been used by researchers, e.g., in smart metering in [37], and in the real world through the declassification approach taken by most democratic countries which declassify sensitive information after the information loses its sensitivity.

5 Verifiable Service (VS) Definition

At a high level, a verifiable service scheme is a two-party protocol in which a client chooses a function, F, and provides (an encoding of) F, its input u, and a query $\boldsymbol{q}$ to a server. The server is expected to evaluate F on u and $\boldsymbol{q}$ (and some public parameters) and respond with the output. Then, the client verifies that the output is indeed the output of the function computed on the provided input. In verifiable services, either the computation (on the input) or both the computation and storage of the input are delegated to the server. A verifiable service is defined as follows.

Definition 2 (VS Scheme). *A verifiable service scheme* $VS := (\texttt{VS.keyGen}, \texttt{VS.setup}, \texttt{VS.genQuery}, \texttt{VS.prove}, \texttt{VS.verify})$ *with function* F*, metadata generator function* M*, and query generator function* Q *consists of five algorithms defined as follows.*

- $\texttt{VS.keyGen}(1^\lambda) \rightarrow k := (sk, pk)$*. A probabilistic algorithm run by the client. It takes as input the security parameter* 1^λ *and outputs a secret/public verification key pair* k*. The server is given* pk*.*
- $\texttt{VS.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma, pp)$*. It is run by the client. It takes as input the security parameter* 1^λ*, the service input* u*, and key pair* k*. If an encoding is needed, then it encodes* u*, that results in* u^**; otherwise,* $u^* = u$*. It outputs encoded input* u^**, metadata* $\sigma = M(u^*, k, pp)$*, and (possibly input dependent) public parameters* pp*. Right after that, the server is given* u^**,* σ*, and* pp*.*
- $\texttt{VS.genQuery}(1^\lambda, aux, k, pp) \rightarrow \boldsymbol{q}$*. A probabilistic algorithm run by the client. It takes as input the security parameter* 1^λ*, auxiliary information* aux*, the key pair* k*, and public parameters* pp*. It outputs a query vector* $\boldsymbol{q} = Q(aux, k, pp)$*. Depending on service types,* $\boldsymbol{q}$ *may be empty or contain only random strings. The output is given to the server.*
- $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \boldsymbol{\pi}$*. It is run by the server. It takes as input the service encoded input* u^**, metadata* σ*, queries* $\boldsymbol{q}$*, public key* pk*, and public parameters* pp*. It outputs a proof,* $\boldsymbol{\pi} = [F(u^*, \boldsymbol{q}, pp), \delta]$*, containing the function evaluation for service input* u*, public parameters* pp*, and query* $\boldsymbol{q}$*, i.e.,* $h = F(u^*, \boldsymbol{q}, pp)$*, and a proof* δ *asserting the evaluation is performed correctly, where generating* δ *may involve* σ*. The output is given to the client.*
- $\texttt{VS.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \rightarrow d \in \{0, 1\}$*. It is run by the client. It takes as input the proof* $\boldsymbol{\pi}$*, query vector* $\boldsymbol{q}$*, key* k*, and public parameters* pp*. In the case where* $\texttt{VS.verify}(\cdot)$ *is publicly verifiable then* $k := (\perp, pk)$*, and when it is privately verifiable* $k := (sk, pk)$*. The algorithm outputs* $d = 1$*, if the proof is accepted; otherwise, it outputs* $d = 0$*.*

A verifiable service scheme has two main properties, *correctness* and *soundness*. Correctness requires that the verification algorithm always accepts a proof generated by an honest prover. It is formally stated below.

Definition 3 (VS Correctness). *A verifiable service scheme VS with functions* F, M, Q *is correct for an auxiliary information* aux*, if for any service input* u *it holds that:*

$$\Pr\left[\begin{array}{l}\texttt{VS.keyGen}(1^\lambda) \rightarrow k := (sk, pk) \\ \texttt{VS.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma, pp) \\ \texttt{VS.genQuery}(1^\lambda, aux, k, pp) \rightarrow \boldsymbol{q} \\ \texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \boldsymbol{\pi} \\ \hline \texttt{VS.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \rightarrow 1.\end{array}\right] = 1$$

Intuitively, a verifiable service is sound if a malicious server cannot convince the verification algorithm to accept an incorrect output of F except with negligible probability. Soundness is formally stated as follows.

Definition 4 (VS Soundness). *A verifiable service VS with functions* F, M, Q *is sound for an auxiliary information* aux*, if for any probabilistic polynomial time adversary* $\mathcal{A}$*, it holds that:*

$$\Pr\left[\begin{array}{l}\texttt{VS.keyGen}(1^\lambda) \rightarrow k := (sk, pk)\\ \mathcal{A}(1^\lambda, pk, F, M, Q) \rightarrow u\\ \texttt{VS.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma, pp)\\ \texttt{VS.genQuery}(1^\lambda, aux, k, pp) \rightarrow \boldsymbol{q}\\ \mathcal{A}(\boldsymbol{q}, u^*, \sigma, pp) \rightarrow \boldsymbol{\pi} = [h, \delta]\\ \texttt{VS.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \rightarrow d\\ \hline F(u^*, \boldsymbol{q}, pp) \neq h \wedge d = 1\end{array}\right] = \mathsf{negl}(\lambda).$$

The above generic definition captures the core requirements of a wide range of verifiable services such as verifiable outsourced storage, i.e., Proofs of Retrievability [39,58] or Provable Data Possession [11,59], verifiable computation [30,45], verifiable searchable encryption [51,46], and verifiable information retrieval [64,62], to name a few. Other additional security properties (e.g., privacy) mandated by certain services can be added to the above definition. Alternatively, the definition can be upgraded to capture the additional requirements. The verifiable service with identifiable abort (VSID) and recurring contingent service payment (RC-S-P) definitions presented in this paper are two examples.

Remark 1. It is not hard to see that the original PoR definition (presented in Section 3.4) is a VS's special case. In particular, PoR's ϵ-soundness captures VS's soundness; in ϵ-soundness, the extractor algorithm interacts (many times) with the cheating prover which must not be able to persuade the extractor to accept an invalid proof with a high probability and should provide accepting proofs for non-negligible ϵ fraction of verification challenges. The former property is exactly what VS soundness states.

6 Verifiable Service with Identifiable Abort (VSID)

A protocol that realises only VS's definition (cf. Appendix 5) 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 [59,51]), it is rather strong and not suitable in the real world, especially when there are monetary incentives (e.g., service payment) that encourage a client to misbehave. Therefore, in the following we enhance the VS notion 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* (VSID), inspired by the notion of secure multi-party computation with identifiable abort [38].

6.1 VSID Definition

This section presents the definition of a VSID scheme.

Definition 5 (VSID Scheme). *A verifiable service with identifiable abort VSID* := (`VSID.keyGen`, `VSID.setup`, `VSID.serve`, `VSID.genQuery`, `VSID.checkQuery`, `VSID.prove`, `VSID.verify`, `VSID.identify`) *with function F, metadata generator function M, and query generator function Q involves four entities; namely, client, server, arbiter, and bulletin board. It consists of eight algorithms defined below.*

- $\texttt{VSID.keyGen}(1^\lambda) \rightarrow k := (sk, pk)$. *A probabilistic algorithm run by the client $\mathcal{C}$. It takes as input the security parameter 1^λ and outputs a secret/public verification key pair k. It sends pk to the bulletin board.*
- $\texttt{VSID.setup}(1^\lambda, u, k) \rightarrow (u^*, pp, e)$. *It is run by the client. It takes as input the security parameter 1^λ, the service input u, and the key pair k. If an encoding is needed, then it encodes u, that results u^*; otherwise, $u^* = u$. It outputs u^*, (possibly file dependent) public parameters pp and $e := (\sigma, w_\sigma)$, where $\sigma = M(u^*, k, pp)$ is metadata and w_σ is a proof asserting the metadata is well-structured. It sends the output (i.e., u^*, pp, e) to the bulletin board.*
- $\texttt{VSID.serve}(u^*, e, pk, pp) \rightarrow a \in \{0, 1\}$. *It is run by the server $\mathcal{S}$. It takes as input the encoded service input u^*, the pair $e := (\sigma, w_\sigma)$, public key pk, and public parameters pp. It outputs $a = 1$, if the proof w_σ is accepted, i.e., if the metadata is well-formed. Otherwise, it outputs $a = 0$. The output is sent to the bulletin board.*

- $\texttt{VSID.genQuery}(1^\lambda, aux, k, pp) \to c := (\boldsymbol{q}, \boldsymbol{w}_q)$. *A probabilistic algorithm run by the client. It takes as input the security parameter* 1^λ*, auxiliary information aux, the key pair* k*, and public parameters pp. It outputs a pair* c *containing a query vector,* $\boldsymbol{q} = Q(aux, k, pp)$*, and proofs,* $\boldsymbol{w}_q$*, proving the queries are well-structured. Depending on service types,* c *might be empty or contain only random strings. It sends* c *to the bulletin board.*
- $\texttt{VSID.checkQuery}(c, pk, pp) \to b \in \{0,1\}$. *It is run by the server. It takes as input a pair* $c := (\boldsymbol{q}, \boldsymbol{w}_q)$ *including queries and their proofs, as well as public key* pk*, and public parameters pp. It outputs* $b = 1$ *if the proofs* $\boldsymbol{w}_\sigma$ *are accepted, i.e., the queries are well-structured. Otherwise, it outputs* $b = 0$*.*
- $\texttt{VSID.prove}(u^*, \sigma, c, pk, pp) \to \boldsymbol{\pi}$. *It is run by the server. It takes as input the encoded service input* u^**, metadata* σ*, a pair* $c := (\boldsymbol{q}, \boldsymbol{w}_q)$*, public key* pk*, and public parameters pp. It outputs a proof,* $\boldsymbol{\pi} = [F(u^*, \boldsymbol{q}, pp), \delta)]$ *containing the function evaluation, i.e.,* $h = F(u^*, \boldsymbol{q}, pp)$*, and a proof* δ *asserting the evaluation is performed correctly, where computing* h *may involve* pk *and computing* δ *may involve* σ*. It sends* $\boldsymbol{\pi}$ *to the board.*
- $\texttt{VSID.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \to d \in \{0,1\}$. *It is run by the client. It takes as input the proof* $\boldsymbol{\pi}$*, queries* $\boldsymbol{q}$*, key pair* k*, and public parameters pp. If the proof is accepted, it outputs* $d = 1$*; otherwise, it outputs* $d = 0$*.*
- $\texttt{VSID.identify}(\boldsymbol{\pi}, c, k, e, u^*, pp) \to I \in \{\mathcal{C}, \mathcal{S}, \perp\}$. *It is run by a third party arbiter. It takes as input the proof* $\boldsymbol{\pi}$*, query pair* $c := (\boldsymbol{q}, \boldsymbol{w}_q)$*, key pair* k*, metadata pair* $e := (\sigma, w_\sigma)$*,* u^**, and public parameters pp. If proof* w_σ *or* $\boldsymbol{w}_q$ *is rejected, then it outputs* $I = \mathcal{C}$*; otherwise, if proof* $\boldsymbol{\pi}$ *is rejected it outputs* $I = \mathcal{S}$*. Otherwise, if* $w_\sigma, \boldsymbol{w}_q$*, and* $\boldsymbol{\pi}$ *are accepted, it outputs* $I = \perp$*.*

A VSID 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. In the following, 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. It is formally stated as follows.

Definition 6 (VSID Correctness). *A verifiable service with identifiable abort scheme with functions* F, M, Q *is correct for an auxiliary information aux, if for any service input* u *it holds that:*

$$\Pr\left[\begin{array}{l} \texttt{VSID.keyGen}(1^\lambda) \to k := (sk, pk) \\ \texttt{VSID.setup}(1^\lambda, u, k) \to (u^*, pp, e) \\ \texttt{VSID.serve}(u^*, e, pk, pp) \to a \\ \texttt{VSID.genQuery}(1^\lambda, aux, k, pp) \to c \\ \texttt{VSID.checkQuery}(c, pk, pp) \to b \\ \texttt{VSID.prove}(u^*, \sigma, c, pk, pp) \to \boldsymbol{\pi} \\ \texttt{VSID.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \to d \\ \hline \texttt{VSID.identify}(\boldsymbol{\pi}, c, k, e, u^*, pp) \to I = \perp \ \wedge \\ a = 1 \ \wedge \ b = 1 \ \wedge \ d = 1 \end{array}\right] = 1.$$

Intuitively, a VSID is sound if a malicious server cannot convince the client to accept an incorrect output of F except with negligible probability. It is formally stated as follows.

Definition 7 (VSID Soundness). *A VSID with functions* F, M, Q *is sound for an auxiliary information aux, if for any probabilistic polynomial time adversary* $\mathcal{A}$*, it holds that:*

$$\Pr\left[\begin{array}{l} \texttt{VSID.keyGen}(1^\lambda) \to k := (sk, pk) \\ \mathcal{A}(1^\lambda, pk, F, M, Q) \to u \\ \texttt{VSID.setup}(1^\lambda, u, k) \to (u^*, pp, e) \\ \texttt{VSID.genQuery}(1^\lambda, aux, k, pp) \to c := (\boldsymbol{q}, \boldsymbol{w}_q) \\ \mathcal{A}(c, e, u^*, pp) \to \boldsymbol{\pi} = [h, \delta] \\ \texttt{VSID.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \to d \\ \hline F(u^*, \boldsymbol{q}, pp) \neq h \wedge d = 1 \end{array}\right] = \mathsf{negl}(\lambda).$$

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

Definition 8 (VSID Inputs Well-formedness). *A VSID with functions F, M, Q has well-formed inputs for an auxiliary information aux, if for any probabilistic polynomial time adversary $\mathcal{A}$, it holds that:*

$$\Pr\left[\begin{array}{l}\mathcal{A}(1^\lambda, F, M, Q) \to \Big(u^*, k := (sk, pk),\\ e := (\sigma, w_\sigma), pp\Big)\\ \texttt{VSID.serve}(u^*, e, pk, pp) \to a\\ \mathcal{A}(1^\lambda, aux, k, pp) \to c := (\boldsymbol{q}, \boldsymbol{w}_q)\\ \texttt{VSID.checkQuery}(c, pk, pp) \to b\\ \hline (M(u^*, k, pp) \neq \sigma \wedge a = 1) \vee\\ (Q(aux, k, pp) \neq \boldsymbol{q} \wedge b = 1)\end{array}\right] = \mathsf{negl}(\lambda).$$

The above property ensures an honest server can detect a malicious client if the client provides ill-structured inputs. 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 VSID supports detectable abort if a corrupt party can escape from being identified, by the arbiter, with only negligible probability. Formally:

Definition 9 (VSID Detectable Abort). *A VSID with functions F, M, Q supports detectable abort for an auxiliary information aux, if the following hold:*

1. *For any PPT adversary $\mathcal{A}_1$:*

$$\Pr\left[\begin{array}{l}\texttt{VSID.keyGen}(1^\lambda) \to k := (sk, pk)\\ \mathcal{A}_1(1^\lambda, pk, F, M, Q) \to u\\ \texttt{VSID.setup}(1^\lambda, u, k) \to (u^*, pp, e)\\ \texttt{VSID.genQuery}(1^\lambda, aux, k, pp) \to c := (\boldsymbol{q}, \boldsymbol{w}_q)\\ \mathcal{A}_1(c, e, u^*, pp) \to \boldsymbol{\pi} = [h, \delta]\\ \texttt{VSID.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \to d\\ \texttt{VSID.identify}(\boldsymbol{\pi}, c, k, e, u^*, pp) \to I\\ \hline d = 0 \wedge I \neq \mathcal{S}\end{array}\right] = \mathsf{negl}(\lambda).$$

2. *For any PPT adversary $\mathcal{A}_2$:*

$$\Pr\left[\begin{array}{l}\mathcal{A}_2(1^\lambda, F, M, Q) \to \Big(u^*, k := (sk, pk),\\ e := (\sigma, w_\sigma), pp\Big)\\ \texttt{VSID.serve}(u^*, e, pk, pp) \to a\\ \mathcal{A}_2(aux, k) \to c := (\boldsymbol{q}, \boldsymbol{w}_q)\\ \texttt{VSID.checkQuery}(c, pk, pp) \to b\\ \texttt{VSID.prove}(u^*, \sigma, c, pk, pp) \to \boldsymbol{\pi}\\ \texttt{VSID.identify}(\boldsymbol{\pi}, c, k, e, u^*, pp) \to I\\ \hline (a = 0 \vee b = 0) \wedge I \neq \mathcal{C}\end{array}\right] = \mathsf{negl}(\lambda).$$

Lighter VSID Scheme ($\text{VSID}_{\text{light}}$) In the VSID definition, algorithm $\texttt{VSID.identify}(\cdot)$ allows an arbiter to identify a misbehaving client even in the setup phase. Nevertheless, it is often sufficient to let the arbiter pinpoint a corrupt party *after* the client and server agree to deal with each other, i.e., after the setup when the server runs $\texttt{VSID.serve}(\cdot)$ and outputs 1. A VSID protocol that meets the latter (lighter) requirements, denoted by $\text{VSID}_{\text{light}}$, would impose lower costs especially when u and elements of e are of large size. Because the arbiter is not required to identify a misbehaving client in setup; therefore, it does not need to have access to the entire file u^* and metadata e. This means (a) the server or client does not need to send u^* and e to the arbiter that leads to lower communication cost, and (b) the arbiter skips checking the correctness of metadata in $\texttt{VSID.identify}(\cdot)$, which ultimately saves it computation cost too. In $\text{VSID}_{\text{light}}$, algorithm $\texttt{VSID.identify}(\cdot)$ needs to take only (π, c, k, e', pp) as input, where $e' \subset e$. So, this requires two changes to the VSID definition, (a) the arbiter algorithm would be $\texttt{VSID.identify}(\pi, c, k, e', pp) \to I$, and (b) in case 2, in Definition 9 we would have $b = 0 \wedge I \neq \mathcal{C}$, so event $a = 0$ is excluded. In this paper, any time we refer to $\text{VSID}_{\text{light}}$, we assume the above minor adjustments are applied to the VSID definition.

6.2 VSID Protocol

In this section, we present the VSID protocol. We show how it can be built upon a protocol that satisfies the VS definition. As stated previously, a VS scheme inherently protects an honest client from a malicious server. Therefore, at a high-level, VSID needs to have two added features; namely, it protects an honest server from a malicious client and allows an arbiter to detect a corrupt party. VSID can be built upon VS using the following standard techniques; Briefly, (a) all parties sign their outgoing messages, (b) they post the signed messages on a bulletin board, and (c) the client, using a publicly verifiable NIZK scheme, proves to the server that its inputs have been correctly constructed. In particular, like VS, the client first generates its secret and public parameters. Then, in the setup, it processes its input, u, to generate encoded input and metadata using the metadata generation function, M. Also, the client utilizes a publicly verifiable NIZK scheme to prove to the server that the metadata has been constructed correctly. The client posts the encoded input, metadata and the proofs along with their signatures to a bulletin board. Next, the server verifies the signatures and proofs. It agrees to serve the client, if they are accepted. Like VS, when the client wants the server to run function F on its input, it uses function Q to generate a query. However, it uses the zero-knowledge scheme to prove to the server that the query has been constructed correctly. The client posts the query, proofs, and their signatures to the board. After that, the server verifies the signatures and proofs. The server-side proves and client-side verifies algorithms remain unchanged with a difference that the server posts its proofs (i.e., the output of the prove algorithm) and their signatures to the board and the client first verifies the signatures before checking the proofs. In the case of any dispute/abort, either party invokes the arbiter which, given the signed posted messages, checks the signatures and proofs in turn to identify a corrupt party. Below, we present the VSID protocol in which we assume all parties sign their outgoing messages and their counter-party first verifies the signature on the messages, before they feed them to their local algorithms.

1. **Key Generation**. $\texttt{VSID.keyGen}(1^\lambda)$
   (a) Calls $\texttt{VS.keyGen}(1^\lambda)$ to generate a pair of secret and public keys, $k : (sk, pk)$.
   (b) Commits to the secret key and appends the commitment: $\texttt{Com}_{sk}$ to pk.
   (c) Posts pk to a bulletin board.
2. **Client-side Setup**. $\texttt{VSID.setup}(1^\lambda, u, k)$
   (a) Calls $\texttt{VS.setup}(1^\lambda, u, k) \rightarrow (\sigma, u^*)$, to generate a metadata: $\sigma = M(u^*, k, pp)$, encoded file service input and (input dependent) parameters pp.
   (b) Generates non-interactive publicly verifiable zero-knowledge proofs asserting σ has been generated correctly, i.e., σ is the output of M that is evaluated on u^*, pk, sk, and pp without revealing sk. Let w_σ contain the proofs.
   (c) Posts $e := (\sigma, w_\sigma)$, pp, and u^* to the bulletin board.
3. **Server-side Setup**. $\texttt{VSID.serve}(u^*, e, pk, pp)$
   Ensures the metadata σ has been constructed correctly, by verifying the proofs in w_σ (where $\sigma, w_\sigma \in e$). If the proofs are accepted, then it outputs $a = 1$ and proceeds to the next step; otherwise, it outputs $a = 0$ and halts.
4. **Client-side Query Generation**. $\texttt{VSID.genQuery}(1^\lambda, \text{aux}, k, pp)$.
   (a) Calls $\texttt{VS.genQuery}(1^\lambda, \text{aux}, k, pp) \rightarrow \boldsymbol{q}$, to generate a query vector, $\boldsymbol{q} = Q(\text{aux}, k, pp)$. If aux is a private input, then it also commits to it, that yields $\texttt{Com}_{\text{aux}}$
   (b) Generates non-interactive publicly verifiable zero-knowledge proofs proving $\boldsymbol{q}$ has been generated correctly, i.e., $\boldsymbol{q}$ is the output of Q which is evaluated on aux, pk, sk, and pp without revealing sk (and aux, if it is a private input). Let $\boldsymbol{w}_q$ contain the proofs and aux (or $\texttt{Com}_{\text{aux}}$ if aux is a private input).
   (c) Posts $c : (\boldsymbol{q}, \boldsymbol{w}_q)$ to the board.
5. **Server-side Query Verification**. $\texttt{VSID.checkQuery}(c, pk, pp)$
   Checks if the query: $\boldsymbol{q} \in c$ has been constructed correctly by verifying the proofs $\boldsymbol{w}_q \in c$. If the check passes, then it outputs $b = 1$; otherwise, it outputs $b = 0$.
6. **Server-side Service Proof Generation**. $\texttt{VSID.prove}(u^*, \sigma, c, pk, pp)$ This phase starts only if the query was accepted, i.e., $b = 1$.
   (a) Calls $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \boldsymbol{\pi}$, to generate $\boldsymbol{\pi} = [F(u^*, \boldsymbol{q}, pp), \delta]$. Recall that $\boldsymbol{q} \in c$.

(b) Posts $\boldsymbol{\pi}$ to the board.

7. **Client-side Proof Verification**. $\texttt{VSID.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp)$
   Calls $\texttt{VS.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \rightarrow d$, to verify proof $\boldsymbol{\pi}$. It accepts the proof if $d = 1$; otherwise, it rejects it.
8. **Arbiter-side Identification**. $\texttt{VSID.identify}(\boldsymbol{\pi}, c, k, e, u^*, pp)$
   (a) Calls $\texttt{VSID.serve}(u^*, e, pk, pp) \rightarrow a$. If $a = 1$, then it proceeds to the next step. Otherwise, it outputs $I = \mathcal{C}$ and halts.
   (b) Calls $\texttt{VSID.checkQuery}(c, pk, pp) \rightarrow b$. If $b = 1$, then it proceeds to the next step. Otherwise, it outputs $I = \mathcal{C}$ and halts.
   (c) If $\boldsymbol{\pi}$ is privately verifiable, then the arbiter first checks if $sk \in k$ (provided by the client along with other opening information) matches $\texttt{Com}_{sk} \in pk$. If they do not match, then the arbiter outputs $I = \mathcal{C}$. Otherwise, it calls $\texttt{VS.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \rightarrow d$. If $d = 1$, then it outputs $I = \perp$; otherwise, it outputs $I = \mathcal{S}$.

Theorem 1. *The VSID protocol with functions F, M, Q satisfies the correctness, soundness, inputs well-formedness, and detectable abort properties for auxiliary information aux, (cf. Definitions 6-9), if the underlying VS protocol with functions F, M, Q is correct and sound for aux and the underlying commitment, publicly verifiable non-interactive zero-knowledge, and signature schemes are correct/complete and secure.*

Proof (sketch). Correctness is implied by the correctness/completeness of the underlying primitives. The soundness of VSID stems from the hiding property of the commitment, zero-knowledge property of the publicly verifiable NIZK proofs, and soundness of the verifiable service (VS) schemes. In particular, in VSID, the verifier (i.e., in this case, the client) makes block-box calls to the algorithms of VS to ensure soundness. However, the prover (i.e., the server) is given additional messages, i.e., $\texttt{Com}_{sk}$, $\texttt{Com}_{\text{aux}}$, w_σ and w_q. The hiding property of the commitment scheme and zero-knowledge property of the zero-knowledge system ensure, given the messages, the prover learns nothing about the verification key and auxiliary information, except with negligible probability. Moreover, the soundness of VS scheme ensures a corrupt prover cannot convince an honest verifier, except with a negligible probability. Inputs well-formedness property boils down to the security of the commitment and publicly verifiable NIZK proofs schemes that are used in steps 1, 2 and 4 in VSID protocol. Specifically, the binding property of the commitment and the soundness of the publicly verifiable NIZK proofs schemes ensure that a corrupt prover (i.e., in this case the client) cannot convince a verifier (i.e., the server) to accept metadata proofs, w_σ and $\texttt{Com}_{sk} \in pk$, while $M(u^*, k, pp) \neq \sigma$ or to accept query proofs, $\boldsymbol{w}_q$ and $\texttt{Com}_{\text{aux}}$, while $Q(\text{aux}, k, pp) \neq \boldsymbol{q}$, except with negligible probability.

Moreover, the detectable abort property holds as long as both previous properties (i.e., soundness and inputs well-formedness) hold, the commitment is secure, the zero-knowledge proofs are publicly verifiable and the signature scheme is secure. The reason is that the algorithm $\texttt{VSID.identify}(.)$, which ensures detectable abort, is a wrapper function that is invoked by the arbiter, and sequentially makes subroutine calls to algorithms $\texttt{VSID.serve}(.)$, $\texttt{VSID.checkQuery}(.)$ and $\texttt{VS.verify}(.)$, where the first two ensure input well-formedness, and the last one ensures soundness. Also, due to the security of the commitment (i.e., binding), the malicious client cannot provide the arbiter with another secret verification key than what was initially committed. Moreover, due to the public verifiability of the zero-knowledge proofs, the arbiter can verify all proofs input to $\texttt{VSID.serve}(.)$ and $\texttt{VSID.checkQuery}(.)$. The signature's security ensures if a proof is not signed correctly, then it can also be rejected by the arbiter; on the other hand, if a proof is signed correctly, then it cannot be repudiated by the signer later on (due to signature's unforgeability); this guarantees that the signer is held accountable for a rejected proof it provides. □

Remark 2. As we mentioned before, it is often sufficient to let the arbiter pinpoint a corrupt party *after* the client and server agree to deal with each other. We denoted a VSID protocol that meets the latter (lighter) requirement, by $\text{VSID}_{\text{light}}$. This version would impose lower costs, when u and elements of e are of large size. In $\text{VSID}_{\text{light}}$ protocol, the client and server run phases 1-3 of the VSID protocol as before, with a difference that the client does not post e and u^* to the board; instead, it sends them directly to the server. In $\text{VSID}_{\text{light}}$ the arbiter algorithm, i.e., $\texttt{VSID.identify}(\cdot)$, needs to take only $(\boldsymbol{\pi}, c, k, e', pp)$ as input, where e' contains the opening of $\texttt{Com}_{sk}$ if $\texttt{VSID.verify}(\cdot)$ is privately verifiable or $e' = \perp$ if it is publicly verifiable. In this light version, the arbiter skips step 8a. Thus, $\text{VSID}_{\text{light}}$ saves (a) communication cost, as u^* and e are never sent to the board and arbiter, and (b) computation cost as the arbiter does not need to run $\texttt{VSID.serve}(\cdot)$ anymore.

7 Recurring Contingent Service Payment (RC-S-P)

In this section, first, we provide a formal definition of RC-S-P. Then, we present the Statement Agreement Protocol (SAP) that will be used as a subroutine in the construction of RC-S-P. After that, we present a generic construction that realises the RC-S-P's definition. After that, we prove the construction's security.

7.1 RC-S-P Definition

In this section, we introduce a formal definition of recurring contingent service payment (RC-S-P).

Definition 10 (RC-S-P Scheme). *A recurring contingent service payment scheme RC-S-P involves four parties; namely, client, server, arbiter, and smart contract (which represents a bulletin board). The scheme is parameterized by five functions:*

- *A function F that will be run on the client's input by the server as a part of the service it provides.*
- *A metadata generator function M.*
- *A pair of encoding/decoding functions (E, D).*
- *A query generator function Q.*

Also, the scheme consists of eight algorithms defined as follows.

$\underline{\texttt{RCSP.keyGen}(1^\lambda) \rightarrow \boldsymbol{k}}$*: A probabilistic algorithm run by client $\mathcal{C}$. It takes as input security parameter 1^λ. It outputs $\boldsymbol{k} := (k, k')$ 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 contract.*

$\underline{\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, z, pl) \rightarrow (u^*, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin^*_{\mathcal{C}})}$*: It is run by $\mathcal{C}$. It takes as input 1^λ, the service input u, $\boldsymbol{k} := (k, k')$, the total number of verifications z, and price list pl containing pairs of actual coin amount for each accepting service proof and the amount for covering each potential dispute resolution's cost. It represents u as an input of M, let u^* be this representation. It sets pp as (possibly) input dependent parameters, e.g., file size. It computes metadata $\sigma = M(u^*, k, pp)$ and a proof w_σ asserting the metadata is well-structured. It sets the value of $p_{\mathcal{S}}$ 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. 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 σ) and parameters (in sk') that will be used to encode the service 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 $\boldsymbol{y}$ that includes binary vectors $[\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, \boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{S}}]$ each of which is set to 0 and its length is z. Note $\boldsymbol{y}$ may contain other public parameters, e.g., the contract's address. It outputs u^*, $e := (\sigma, w_\sigma)$, $T := (T_{cp}, T_{qp})$, $p_{\mathcal{S}}$, $\boldsymbol{y}$, and the encoded coins amount $coin^*_{\mathcal{C}}$ (that contains o and l coins in an encoded form). $\mathcal{C}$ sends u^*, z, e, $T_{cp} \setminus \{g_{cp}\}$, and $T_{qp} \setminus \{g_{qp}\}$ to the server $\mathcal{S}$ and sends $g_{cp}, g_{qp}, p_{\mathcal{S}}$, $\boldsymbol{y}$, and $coin^*_{\mathcal{C}}$ coins to the contract.*

$\underline{\texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \rightarrow (coin^*_{\mathcal{S}}, a)}$*: It is run by $\mathcal{S}$. It takes as input u^*, metadata-proof pair $e := (\sigma, w_\sigma)$, pk (read from the contract), z, and $T := (T_{cp}, T_{qp})$, where $\{g_{cp}, g_{qp}\}$ are read from the smart contract. It reads $p_{\mathcal{S}}$, and $\boldsymbol{y}$ from the smart contract. It checks the validity of e and T elements. It checks elements of $\boldsymbol{y}$ and ensures each element of $\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, \boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{S}} \in \boldsymbol{y}$ has been set to 0. If all checks pass, then it encodes the amount of its coins that yields $coin^*_{\mathcal{S}}$, and sets $a = 1$. Otherwise, it sets $coin^*_{\mathcal{S}} = \perp$ and $a = 0$. It outputs $coin^*_{\mathcal{S}}$ and a. The smart contract is given $coin^*_{\mathcal{S}}$ coins and a.*

$\underline{\texttt{RCSP.genQuery}(1^\lambda, aux, k, T_{qp}) \rightarrow c^*_j}$*: A probabilistic algorithm run by $\mathcal{C}$. It takes as input 1^λ, auxiliary information aux, the key pair k, and encoding token T_{qp}. It computes a pair c_j containing a query vector $\boldsymbol{q}_j = Q(aux, k, pp)$, and proof $\boldsymbol{w}_{q_j}$ proving the query is well-structured, where $pp \in T_{qp}$. It outputs the encoding of the pair, $c^*_j = E(c_j, T_{qp})$, and sends the output to the contract.*

$\underline{\texttt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp}) \rightarrow (b_j, m_{\mathcal{S},j}, \boldsymbol{\pi}^*_j)}$*: It is run by $\mathcal{S}$. It takes as input u^*, metadata σ, c^*_j, pk, and T_{qp}. It checks the validity of decoded query pair $c_j = D(c^*_j, T_{qp})$. If it is rejected, then it sets $b_j = 0$ and constructs a complaint $m_{\mathcal{S},j}$. Otherwise, it sets $b_j = 1$ and $m_{\mathcal{S},j} = \perp$. It outputs $b_j, m_{\mathcal{S},j}$, and encoded proof*

$\boldsymbol{\pi}_j^* = E(\pi_j, T_{qp})$, *where* $\boldsymbol{\pi}_j$ *contains* $h_j = F(u^*, \boldsymbol{q}_j, pp)$ *and a proof* δ_j *asserting the evaluation is performed correctly (*$\boldsymbol{\pi}_j$ *may contain dummy values if* $b_j = 0$*). The smart contract is given* $\boldsymbol{\pi}_j^*$*.*

$\underline{\texttt{RCSP.verify}(\boldsymbol{\pi}_j^*, c_j^*, k, T_{qp}) \rightarrow (d_j, \boldsymbol{m}_{\mathcal{C},j})}$*: A deterministic algorithm run by* $\mathcal{C}$*. It takes as input* $\boldsymbol{\pi}_j^*$*, query vector* $\boldsymbol{q}_j \in c_j^*$*,* k*, and* T_{qp}*. It checks the decoded proof* $\boldsymbol{\pi}_j = D(\boldsymbol{\pi}_j^*, T_{qp})$*, if it is rejected, it outputs* $d_j = 0$ *and a complaint* $\boldsymbol{m}_{\mathcal{C},j}$*. Else, it outputs* $d_j = 1$ *and* $\boldsymbol{m}_{\mathcal{C},j} = \perp$*.*

$\underline{\texttt{RCSP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp}) \rightarrow \boldsymbol{y}}$*: It is run by the arbiter* $\mathcal{R}$*. It takes as input* $\mathcal{C}$*'s complaints* $\boldsymbol{m}_{\mathcal{C}}$*,* $\mathcal{S}$*'s complaints* $\boldsymbol{m}_{\mathcal{S}}$*,* z*, all encoded proofs* $\boldsymbol{\pi}^*$*, all encoded query pairs* $\boldsymbol{c}^*$*,* pk*, and encoding token* T_{qp}*. It verifies the token, decoded queries, and proofs. It reads the binary vectors* $[\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, \boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{S}}]$ *from the smart contract. It updates* $\boldsymbol{y}_{\mathcal{P}}$ *by setting an element of it to one, i.e.,* $y_{\mathcal{P},j} = 1$*, if party* $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}\}$ *has misbehaved in the* j*-th verification (i.e., provided invalid query or service proof). It also updates* $\boldsymbol{y}'_{\mathcal{P}}$ *(by setting an element of it to one) if party* $\mathcal{P}$ *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.*

$\underline{\texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin_{\mathcal{C}}^*, coin_{\mathcal{S}}^*) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{R}})}$*: It is run by the smart contract. It takes as input the binary vectors* $[\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, \boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{S}}] \in \boldsymbol{y}$ *that indicate which party misbehaved, or sent invalid complaint in each verification,* $T_{cp} := \{cp, g_{cp}\}$*,* a*, the total coins the server should deposit* $p_{\mathcal{S}}$*,* $coin_{\mathcal{C}}^*$*, and* $coin_{\mathcal{S}}^*$*. If* $a = 1$ *and* $coin_{\mathcal{S}}^* = p_{\mathcal{S}}$*, then it verifies the validity of* T_{cp}*. If* T_{cp} *is rejected, then it aborts. If it is accepted, then it constructs vector* $\boldsymbol{coin}_{\mathcal{P}}$*, where* $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}, \mathcal{R}\}$*; It sends* $coin_{\mathcal{P},j} \in \boldsymbol{coin}_{\mathcal{P}}$ *coins to party* $\mathcal{P}$ *for each* j*-th verification. Otherwise (i.e.,* $a = 0$ *or* $coin_{\mathcal{S}}^* \neq p_{\mathcal{S}}$*) it sends* $coin_{\mathcal{C}}^*$ *and* $coin_{\mathcal{S}}^*$ *coins to* $\mathcal{C}$ *and* $\mathcal{S}$ *respectively.*

In the above definition, algorithms RCSP.genQuery, RCSP.prove, RCSP.verify, and RCSP.resolve implicitly take $(a, coin_{\mathcal{S}}^*, p_{\mathcal{S}})$ as other inputs and execute only if $a = 1$ and $coin_{\mathcal{S}}^* = p_{\mathcal{S}}$; however, for simplicity we avoided explicitly stating it in the definition.

An RC-S-P scheme must meet correctness and security. Correctness requires that by the end of the protocol's execution (that involves honest client and server) the client receives all z valid service proofs 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 service proof (and no one is identified as misbehaving party). Moreover, 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. Correctness is formally stated below.

Definition 11 (Correctness). *An RC-S-P scheme with functions* F, M, E, D, Q *is* correct *for auxiliary information* aux *if for any* z *polynomial in* λ*, any price list* pl*, and any service input* u*, it holds that the following probability is equal to* 1*:*

$$
\Pr\left[\begin{array}{l}
\texttt{RCSP.keyGen}(1^\lambda) \rightarrow \boldsymbol{k} \\
\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, z, pl) \rightarrow (u^*, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin_{\mathcal{C}}^*) \\
\texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \rightarrow (coin_{\mathcal{S}}^*, a) \\
\textsf{For} j = 1, \ldots, z \textsf{ do :} \\
\quad \texttt{RCSP.genQuery}(1^\lambda, aux, k, T_{qp}) \rightarrow c_j^* \\
\quad \texttt{RCSP.prove}(u^*, \sigma, c_j^*, pk, T_{qp}) \rightarrow (b_j, m_{\mathcal{S},j}, \boldsymbol{\pi}_j^*) \\
\quad \texttt{RCSP.verify}(\boldsymbol{\pi}_j^*, c_j^*, k, T_{qp}) \rightarrow (d_j, \boldsymbol{m}_{\mathcal{C},j}) \\
\texttt{RCSP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp}) \rightarrow \boldsymbol{y} \\
\underline{\texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin_{\mathcal{C}}^*, coin_{\mathcal{S}}^*) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{R}})} \\
\hline
(a = 1) \wedge (\bigwedge_{j=1}^{z} b_j = \bigwedge_{j=1}^{z} d_j = 1) \wedge (\boldsymbol{y}_{\mathcal{C}} = \boldsymbol{y}_{\mathcal{S}} = \boldsymbol{y}'_{\mathcal{C}} = \boldsymbol{y}'_{\mathcal{S}} = 0) \wedge \\
(\sum_{j=1}^{z} coin_{\mathcal{C},j} = coin_{\mathcal{C}}^* - o \cdot z) \ \wedge (\sum_{j=1}^{z} coin_{\mathcal{S},j} = coin_{\mathcal{S}}^* + o \cdot z) \wedge \\
(\sum_{j=1}^{z} coin_{\mathcal{R},j} = 0)
\end{array}\right]
$$

where $\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, \boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{S}} \in \boldsymbol{y}$.

An RC-S-P scheme is said to be secure if it satisfies three main properties: (a) security against a malicious server, (b) security against a malicious client, and (c) privacy. In the following, we formally define each of

them. Intuitively, security against a malicious server states that (at the end of the protocol execution) either (i) for each verification the client gets a valid proof and gets back its deposit minus the service payment, or (ii) the client gets its deposit back (for the j-th verification) and the arbiter receives l coins, or (iii) if a malicious server unnecessarily invokes the arbiter, then it has to pay the arbiter. In particular, for each j-th verification, the security requires that only with a negligible probability the adversary wins, if it provides either (a) correct evaluation of the function on the service input but it either makes the client withdraw an incorrect amount of coins (i.e., something other than its deposit minus service payment) or makes the arbiter withdraw an incorrect amount of coins if it unnecessarily invokes the arbiter, or (b) incorrect evaluation of the function on the service input, but either persuades the client or the arbiter to accept it or makes them withdraw an incorrect amount of coins (i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z}$ or $coin_{\mathcal{R},j} \neq l$ coins). Below, we formalize this intuition.

Definition 12 (Security Against Malicious Server). *An RC-S-P scheme with functions F, M, E, D, Q is* secure against a malicious server *for auxiliary information aux, if for any z polynomial in λ, any price list pl, every j (where $1 \leq j \leq z$), and any PPT adversary $\mathcal{A}$, it holds that the following probability is* $\mathsf{negl}(\lambda)$*:*

$$\Pr\left[\begin{array}{l}
\texttt{RCSP.keyGen}(1^\lambda) \rightarrow \boldsymbol{k} \\
\mathcal{A}(1^\lambda, pk, F, M, E, D, Q) \rightarrow u \\
\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, z, pl) \rightarrow (u^*, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin^*_{\mathcal{C}}) \\
\mathcal{A}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \rightarrow (coin^*_{\mathcal{S}}, a) \\
\texttt{RCSP.genQuery}(1^\lambda, aux, k, T_{qp}) \rightarrow c^*_j \\
\mathcal{A}(c^*_j, \sigma, u^*, a) \rightarrow (b_j, m_{\mathcal{S},j}, h^*_j, \delta^*_j) \\
\texttt{RCSP.verify}(\boldsymbol{\pi}^*_j, c^*_j, k, T_{qp}) \rightarrow (d_j, \boldsymbol{m}_{\mathcal{C},j}) \\
\texttt{RCSP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp}) \rightarrow \boldsymbol{y} \\
\texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}}) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{R}}) \\
\hline
\Big(F(u^*, \boldsymbol{q}_j, pp) = h_j \ \wedge \\
(coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o \vee (coin_{\mathcal{R},j} \neq l \ \wedge \ y'_{\mathcal{S},j} = 1))\Big) \ \vee \\
\Big(F(u^*, \boldsymbol{q}_j, pp) \neq h_j \ \wedge \\
(d_j = 1 \vee y_{\mathcal{S},j} = 0 \vee \ coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} \ \vee coin_{\mathcal{R},j} \neq l)\Big)
\end{array}\right]$$

where $\boldsymbol{q}_j \in D(c^*_j, T_{qp})$, $\boldsymbol{\pi}^*_j = [h^*_j, \delta^*_j], h_j = D(h^*_j, T_{qp})$, $\sigma \in e$, $\boldsymbol{m}_{\mathcal{C},j} \in \boldsymbol{m}_{\mathcal{C}}, m_{\mathcal{S},j} \in \boldsymbol{m}_{\mathcal{S}}, y'_{\mathcal{S},j} \in \boldsymbol{y}'_{\mathcal{S}} \in \boldsymbol{y}$, $y_{\mathcal{S},j} \in \boldsymbol{y}_{\mathcal{S}} \in \boldsymbol{y}$, *and* $pp \in T_{qp}$.

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

Definition 13 (Security Against Malicious Client). *An RC-S-P scheme with functions F, M, E, D, Q is* secure against a malicious client *for an auxiliary information aux, if for any z polynomial in λ, every j (where $1 \leq j \leq z$), and any PPT adversary $\mathcal{A}$, it holds that the following probability is* $\mathsf{negl}(\lambda)$*:*

$$\Pr\left[\begin{array}{l}\mathcal{A}(1^\lambda, F, M, E, D, Q) \to (u^*, z, \boldsymbol{k}, e, T, pl, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, aux, \boldsymbol{y}, pk)\\ \texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \to (coin^*_{\mathcal{S}}, a)\\ \mathcal{A}(coin^*_{\mathcal{S}}, a, 1^\lambda, aux, k, T_{qp}) \to c^*_j\\ \texttt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp}) \to (b_j, m_{\mathcal{S},j}, \boldsymbol{\pi}^*_j)\\ \mathcal{A}(\boldsymbol{\pi}^*_j, c^*_j, k, T_{qp}) \to (d_j, \boldsymbol{m}_{\mathcal{C},j})\\ \texttt{RCSP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp}) \to \boldsymbol{y}\\ \texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}}) \to (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{R}})\\ \hline \Big((M(u^*, k, pp) = \sigma \ \wedge Q(aux, k, pp) = \boldsymbol{q}_j) \wedge\\ (coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o \ \vee coin_{\mathcal{R},j} \neq l \ \wedge \ y'_{\mathcal{C},j} = 1)\Big) \vee\\ \Big(M(u^*, k, pp) \neq \sigma \wedge a = 1\Big) \ \vee \Big(Q(aux, k, pp) \neq \boldsymbol{q}_j \ \wedge (b_j = 1 \vee\\ \ y_{\mathcal{C},j} = 0 \ \vee coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o \ \vee coin_{\mathcal{R},j} \neq l)\Big)\end{array}\right]$$

where $\boldsymbol{q}_j \in D(c^*_j, t_{qp})$, $\sigma \in e$, $y'_{\mathcal{C},j} \in \boldsymbol{y}'_{\mathcal{C}} \in \boldsymbol{y}$, $y_{\mathcal{C},j} \in \boldsymbol{y}_{\mathcal{C}} \in \boldsymbol{y}$, *and* $pp \in T_{qp}$.

Informally, RC-S-P is privacy-preserving if it guarantees the privacy of (1) the service input (e.g., outsourced file) and (2) the service proof's status during the private time bubble. In the following, we formally define privacy.

Definition 14 (Privacy). *An RC-S-P scheme with functions* F, M, E, D, Q *preserves* privacy *for auxiliary information aux if for any z polynomial in* λ *and any price list pl, the following hold:*

1. *For any PPT adversary* $\mathcal{A}_1$, *it holds that the following probability is no more than* $\frac{1}{2} + \mathsf{negl}(\lambda)$.

$$\Pr\left[\begin{array}{l}\texttt{RCSP.keyGen}(1^\lambda) \to \boldsymbol{k}\\ \mathcal{A}_1(1^\lambda, pk, F, M, E, D, Q) \to (u_0, u_1)\\ \beta \xleftarrow{\$} \{0,1\}\\ \texttt{RCSP.cInit}(1^\lambda, u_\beta, \boldsymbol{k}, z, pl) \to (u^*_\beta, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin^*_{\mathcal{C}})\\ \texttt{RCSP.sInit}(u^*_\beta, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \to (coin^*_{\mathcal{S}}, a)\\ \mathsf{For} j = 1, \ldots, z \ \mathsf{do}:\\ \quad \texttt{RCSP.genQuery}(1^\lambda, aux, k, T_{qp}) \to c^*_j\\ \quad \texttt{RCSP.prove}(u^*_\beta, \sigma, c^*_j, pk, T_{qp}) \to (b_j, m_{\mathcal{S},j}, \boldsymbol{\pi}^*_j)\\ \quad \texttt{RCSP.verify}(\boldsymbol{\pi}^*_j, c^*_j, k, T_{qp}) \to (d_j, \boldsymbol{m}_{\mathcal{C},j})\\ \hline \mathcal{A}_1(\boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl, a) \to \beta\end{array}\right]$$

   where $\boldsymbol{c}^* = [c^*_1, ..., c^*_z]$ *and* $\boldsymbol{\pi}^* = [\boldsymbol{\pi}^*_1, ..., \boldsymbol{\pi}^*_z]$.
2. *For any PPT adversaries* $\mathcal{A}_2$ *and* $\mathcal{A}_3$, *it holds that the following probability is no more than* $Pr_{\max} + \mathsf{negl}(\lambda)$*:*

$$\Pr\left[\begin{array}{l}\texttt{RCSP.keyGen}(1^\lambda) \to \boldsymbol{k}\\ \mathcal{A}_2(1^\lambda, pk, F, M, E, D, Q) \to u\\ \texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, M, z, pl, enc) \to (u^*, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin^*_{\mathcal{C}})\\ \texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \to (coin^*_{\mathcal{S}}, a)\\ \mathsf{For} j = 1, \ldots, z \ \mathsf{do}:\\ \quad \mathcal{A}_2(1^\lambda, aux, k, T_{qp}) \to c^*_j\\ \quad \texttt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp}) \to (b_j, m_{\mathcal{S},j}, \boldsymbol{\pi}^*_j)\\ \quad \texttt{RCSP.verify}(\boldsymbol{\pi}^*_j, c^*_j, k, T_{qp}) \to (d_j, \boldsymbol{m}_{\mathcal{C},j})\\ \hline \mathcal{A}_3(F, M, E, D, Q, \boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl, a) \to (d_j, j)\end{array}\right]$$

   where $Pr_{\max}$ *is defined as follows. Let* $Exp^{\mathcal{A}_2}_{\mathsf{priv}}(1^\lambda)$ *be the above experiment. Let* $\boldsymbol{q}_j \in D(c^*_j, T_{qp})$, $pp \in T_{qp}$. *We define the events* $Con^{(1)}_{0,j} : Q(aux, k, pp) \neq \boldsymbol{q}_j$, $Con^{(2)}_{0,j} : b_j = 0, Con^{(1)}_{1,j} : Q(aux, k, pp) = \boldsymbol{q}_j$, *and* $Con^{(2)}_{1,j} : b_j = 1$. *For* $i \in \{0,1\}$ *and* $j \in [z]$, *we define:*

$$Pr_{i,j} := \Pr\left[\frac{Exp^{\mathcal{A}_2}_{\mathsf{priv}}(1^\lambda)}{Con^{(1)}_{i,j} \ \wedge \ Con^{(2)}_{i,j}}\right].$$

   Then, we have $Pr_{\max} := \max\{Pr_{0,1}, Pr_{1,1}, ..., Pr_{0,z}, Pr_{1,z}\}$.

In the above definition, for each j-th verification, the adversary $\mathcal{A}_2$ produces an invalid query with probability $Pr_{0,j}$ and a valid query with probability $Pr_{1,j}$. It is required that privacy is preserved regardless of the queries and proofs status, i.e., whether they are valid/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 $\texttt{RCSP.keyGen}(\cdot)$ is executed up to the time when $\texttt{RCSP.resolve}(\cdot)$ is run. In other words, the privacy holds up to the point where $\texttt{RCSP.resolve}(.)$ is run. This is why the latter algorithm is excluded from the experiments in Definition 14.

Definition 15 (RC-S-P Security). *An RC-S-P with functions F, M, E, D, Q is* secure *for auxiliary information aux, if it satisfies security against malicious server, security against malicious client, and preserves privacy for aux, w.r.t. Definitions 12, 13 , 14, respectively.*

7.2 Statement Agreement Protocol (SAP)

As we stated in Section 4.2, RC-S-P relies on the idea that the server and client can efficiently agree on private statements at the beginning of the protocol. Therefore, in this section, we present a protocol, called statement agreement protocol (SAP), that satisfies the above requirement. Informally, an SAP is secure if it meets four security properties:

1. Neither party can persuade a third party verifier that it has agreed with its counter-party on an invalid statement, i.e., a statement that both parties have not agreed on.
2. After they agree on the statement, an honest party can (almost) always prove to the verifier that it has the agreement.
3. The privacy of the statement should be preserved (from the public) before either of the two parties attempts to prove the agreement on the statement.
4. After both parties reach an agreement neither can later deny the agreement.

To that end, we use a combination of a smart contract (including digital signatures involved) and a commitment scheme. The idea is as follows. Let x be the statement. The client picks a random value and uses it to commit to x. It sends the commitment to the contract and the commitment opening (i.e., statement and the random value) to the server. The server checks if the opening matches the commitment and if so, it commits to the statement using the same random value and sends its commitment to the contract. Later, for a party to prove to the contract/verifier that it has agreed on the statement with the other party, it only sends the opening of the commitment. The contract/verifier checks if the opening matches both commitments and accepts if it matches. The SAP protocol is provided below. It assumes that each party $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}\}$ already has a blockchain public address $adr_{\mathcal{P}}$ (via creating an account).

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

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

7.3 Recurring Contingent Service Payment (RC-S-P) Protocol

In this section, we present the "recurring contingent service payment" (RC-S-P) protocol for a generic service. It utilises a novel combination of $\text{VSID}_{\text{light}}$, SAP, the private time bubble notion, and symmetric-key encryption schemes along with the coin masking and padding techniques. At a high level, the protocol works as follows. The client and server use SAP to provably agree on two private statements; the first statement includes payment details, while another one specifies a secret key, k, and the pads' length. They also agree on public parameters such as (a) the private time bubble's length, that is the total number of billing cycles, z, plus a waiting period, $\mathtt{J}$, and (b) a smart contract which specifies z and the total amount of masked coins each party should deposit. The client deploys the contract. Each party deposits its masked coins in the contract. If either party does not deposit enough coins on time, later each party has a chance to withdraw its coins and terminate the contract. To start using/providing the service, they invoke $\text{VSID}_{\text{light}}$ protocol. In particular, they engage in the $\mathtt{VSID.keyGen}(\cdot)$, $\mathtt{VSID.setup}(\cdot)$, and $\mathtt{VSID.serve}(\cdot)$ algorithms. If the server decides not to serve, e.g., it detects the client's misbehaviour, it sends 0 within a fixed time; in this case, the parties can withdraw their deposit and terminate the contract. Otherwise, the server sends 1 to the contract.

At the end of each billing cycle, the client generates an encrypted query, by calling $\mathtt{VSID.genQuery}(\cdot)$ and encrypting its output using the key, k. It pads the encrypted query and sends the result to the contract. The encryption and pads ensure nothing about the client's input (e.g., outsourced file) is revealed to the public within the private time bubble. In the same cycle, the server retrieves the query, removes the pads and decrypts the result. Then, it locally checks its validity, by calling $\mathtt{VSID.checkQuery}(\cdot)$. If the query is rejected, the server locally stores the index of the billing cycle and then generates a dummy proof. Otherwise, if the server accepts the query, it generates a proof of service by calling $\mathtt{VSID.prove}(\cdot)$. In either case, the server encrypts the proof, pads it and sends the result to the contract. Note that sending (padded encrypted) dummy proofs ensures that the public, during the private time bubble, does not learn if the client generates invalid queries. After the server sends the messages to the contract, the client removes the pads, decrypts the proof and locally verifies it, by calling $\mathtt{VSID.verify}(\cdot)$. If the verification is passed, then the client knows the server has delivered the service honestly. But, if the proof is rejected, it waits until the private time bubble passes and dispute resolution time arrives. During the dispute resolution period, in the case the client or server rejects any proofs, it invokes the arbiter, refers it to the invalid encrypted proofs in the contract, and sends to it the decryption key and the pads' detail. The arbiter checks the validity of the key and pads, by using SAP. If they are accepted, then the arbiter locally removes the pads from the encrypted proofs, decrypts the related proofs, and runs $\mathtt{VSID.identify}(\cdot)$ to check the validity of the party's claim. The arbiter sends to the contract a report of its findings that includes the total number of times the server and client provided invalid proofs. In the next phase, to distribute the coins, either client or server sends: (a) "pay" message, (b) the agreed statement that specifies the payment details, and (c) the statement's proof to the contract which verifies the statement and if approved it distributes the coins according to the statement's detail, and the arbiter's report.

Now we outline why RC-S-P addresses the issues, raised in Section 4. In the setup, if the client provides ill-formed inputs (so later it can accuse the server) then the server can detect and avoid serving it. After the setup, if the client avoids sending any input, then the server still gets paid for the service it provided. Also, in the case of a dispute between the parties, their claim is checked, and the corrupt party is identified. The corrupt party has to pay the arbiter and if that is the client, then it has to pay the server as well. These features not only do guarantee the server's resource is not wasted, but also ensures fairness (i.e., if a potentially malicious server is paid, then it must have provided the service and if a potentially malicious client does not pay, then it will learn nothing). Furthermore, as during the private time bubble (a) no plaintext proof is given to the contract, and (b) no dispute resolution and coin transfer take place on contract, the public cannot figure out the outcome of each verification. This preserves the server's privacy. Also, because the deposited coins are masked and the agreed statement is kept private, nothing about the detail of the service is leaked to the public before the bubble bursts. This preserves the client's privacy. Also, as either party can prove to the contract the validity of the agreed statement, and ask the contract to distribute the coins, the coins will be not be locked forever.

Protocol description The RC-S-P protocol is parameterized by the functions F, M, Q of the underlying VSID and encoding/decoding functions (E, D) that refer to "encrypt then pad"/"remove pad then decrypt" procedures, respectively. It is assumed that (a) each party $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}, \mathcal{R}\}$ already has a blockchain public address, $adr_{\mathcal{P}}$, which is known to all parties, (b) it uses that (authorised) address to send transactions to the smart contract, (c) the contract before recording a transaction, ensures the transaction is originated from an authorised address, and (d) there is a public price list pl known to everyone. The protocol is presented below.

1. **Key Generation**. $\texttt{RCSP.keyGen}(1^\lambda)$
   (a) $\mathcal{C}$ runs $\texttt{VSID.keyGen}(1^\lambda) \rightarrow k := (sk, pk)$.
   (b) $\mathcal{C}$ picks a random secret key $\bar{k}$ for a symmetric-key encryption. Also, it sets two parameters: pad_π and pad_q, where pad_π and pad_q refer to the number of dummy values that will be used to pad encrypted proofs and encrypted queries respectively[5], determined by the security parameter and description of F. Let $sk' := (pad_\pi, pad_q, \bar{k})$. The keys' size is part of the security parameter. Let $\boldsymbol{k} = [k, k']$, where $k' := (sk', pk')$ and $pk' := (adr_{\mathcal{C}}, adr_{\mathcal{S}})$.

2. **Client-side Initiation**. $\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, z, pl)$
   (a) Calls $\texttt{VSID.setup}(1^\lambda, u, k) \rightarrow (u^*, pp, e)$, to encode service input, and generate metadata. It sets $qp = sk'$ and appends pp to qp.
   (b) Calls $\texttt{SAP.init}(1^\lambda, adr_{\mathcal{C}}, adr_{\mathcal{S}}, qp) \rightarrow (r_{qp}, g_{qp}, adr_{\text{SAP}_1})$, to initiate an agreement (with $\mathcal{S}$) on qp. Let $T_{qp} := (\ddot{x}_{qp}, g_{qp})$ be proof/query encoding token, where $\ddot{x}_{qp} := (qp, r_{qp})$ is the opening and g_{qp} is the commitment stored on the contract as a result of running SAP.
   (c) Sets coin parameters as follows, o: the amount of coins for each accepting proof, and l: the amount of coins to cover the cost of each potential dispute resolution, given price list pl.
   (d) Sets $cp := (o, o_{max}, l, l_{max}, z)$, where o_{max} is the maximum amount of coins for an accepting service proof, l_{max} is the maximum amount of coins to resolve a potential dispute, and z is the number of service proofs/verifications. Then, $\mathcal{C}$ calls $\texttt{SAP.init}(1^\lambda, adr_{\mathcal{C}}, adr_{\mathcal{S}}, cp) \rightarrow (r_{cp}, g_{cp}, adr_{\text{SAP}_2})$, to initiate an agreement (with $\mathcal{S}$) on cp. Let $T_{cp} := (\ddot{x}_{cp}, g_{cp})$ be coin encoding token, where $\ddot{x}_{cp} := (cp, r_{cp})$ is the opening and g_{cp} is the commitment stored on the contract as a result of executing SAP. Let $T := \{T_{qp}, T_{cp}\}$.
   (e) Set parameters $coin^*_{\mathcal{C}} = z \cdot (o_{max} + l_{max})$ and $p_{\mathcal{S}} = z \cdot l_{max}$, where $coin^*_{\mathcal{C}}$ and $p_{\mathcal{S}}$ are the total number of masked coins $\mathcal{C}$ and $\mathcal{S}$ should deposit respectively. It also designs a smart contract, SC, that explicitly specifies parameters z, $coin^*_{\mathcal{C}}$, $p_{\mathcal{S}}$, adr_{SAP_1}, adr_{SAP_2}, pk, and pk'. It sets a set of time points/windows, $\texttt{Time} : \{\texttt{T}_0, ..., \texttt{T}_2, \texttt{G}_{1,1}, ..., \texttt{G}_{z,2}, \texttt{J}, \texttt{K}_1, ..., \texttt{K}_3, \texttt{L}\}$, that are explicitly specified in the contract which will accept a certain party's message only in a specified time point/window. The time allocation will become clear in the next phases.
   (f) Sets also four counters $[y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{S}}]$ in SC, where their initial value is 0. It signs and deploys SC to the blockchain. Let adr_{SC} be the address of the deployed SC, and $\boldsymbol{y} : [y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{S}}, adr_{\text{SC}}]$.
   (g) Deposits $coin^*_{\mathcal{C}}$ coins in the contract. It sends $u^*, z, e, \ddot{x}_{qp}$, and $\ddot{x}_{cp}$ (along with adr_{SC}) to $\mathcal{S}$. Let $\texttt{T}_0$ be the time that the above process finishes.

3. **Server-side Initiation**. $\texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y})$
   (a) Checks the parameters in T (e.g., qp and cp) and in SC (e.g., $p_{\mathcal{S}}, \boldsymbol{y}$) and ensures a sufficient amount of coins has been deposited by $\mathcal{C}$.
   (b) Calls $\texttt{SAP.agree}(qp, r_{qp}, g_{qp}, adr_{\mathcal{C}}, adr_{\text{SAP}_1}) \rightarrow (g'_{qp}, b_1)$ and $\texttt{SAP.agree}(cp, r_{cp}, g_{cp}, adr_{\mathcal{C}}, adr_{\text{SAP}_2}) \rightarrow (g'_{cp}, b_2)$, to verify the correctness of tokens in T and to agree on the tokens' parameters, where $qp, r_{qp} \in \ddot{x}_{qp}$, and $cp, r_{cp} \in \ddot{x}_{cp}$. Recall that if both $\mathcal{C}$ and $\mathcal{S}$ are honest, then $g_{qp} = g'_{qp}$ and $g_{cp} = g'_{cp}$.
   (c) If any above check is rejected, then it sets $a = 0$. Otherwise, it calls $\texttt{VSID.serve}(u^*, e, pk, pp) \rightarrow a$.

[5] The values of pad_π and pad_q is determined as follows, $pad_\pi = \pi_{max} - \pi_{act}$ and $pad_q = q_{max} - q_{act}$, where π_{max} and π_{act} refer to the maximum and actual the service's proof size while q_{max} and q_{act} refer to the maximum and actual the service's query size, respectively.

(d) Sends a and $coin^*_{\mathcal{S}} = p_{\mathcal{S}}$ coins to SC at time $\mathtt{T}_1$, where $coin^*_{\mathcal{S}} = \perp$ if $a = 0$

Note that, $\mathcal{S}$ and $\mathcal{C}$ can withdraw their coins at time $\mathtt{T}_2$, if $\mathcal{S}$ sends $a = 0$, fewer coins than $p_{\mathcal{S}}$, or nothing to the SC. To withdraw, $\mathcal{S}$ or $\mathcal{C}$ simply sends a "pay" message to $\mathtt{RCSP.pay}(\cdot)$ algorithm (only) at time $\mathtt{T}_2$.

Billing-cycles Onset. $\mathcal{C}$ and $\mathcal{S}$ engage in the following three phases, i.e., 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, $\mathtt{G}_{j,1}$ and $\mathtt{G}_{j,2}$, where $\mathtt{G}_{j,2} > \mathtt{G}_{j,1}$, and $\mathtt{G}_{1,1} > \mathtt{T}_2$.

4. **Client-side Query Generation**. $\mathtt{RCSP.genQuery}(1^\lambda, \text{aux}, k, T_{qp})$
   (a) Calls $\mathtt{VSID.genQuery}(1^\lambda, \text{aux}, k, pp) \rightarrow c_j := (\boldsymbol{q}_j, \boldsymbol{w}_{q_j})$, to generate a query-proof pair.
   (b) Encodes c_j, by first encrypting it, $\mathtt{Enc}(\bar{k}, c_j) = c'_j$, where $\bar{k} \in T_{qp}$; and then, padding (each element of) the result with $pad_q \in T_{qp}$ random values that are picked uniformly at random from the encryption's output range, U. Let c^*_j be the result.
   (c) Sends the padded encrypted query-proof pair, c^*_j, to SC at time $\mathtt{G}_{j,1}$.

5. **Server-side Proof Generation**. $\mathtt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp})$
   (a) Constructs an empty vector, $\boldsymbol{m}_{\mathcal{S}} = \perp$, if $j = 1$.
   (b) Removes the pads from c^*_j, using parameters of T_{qp}. Let c'_j be the result. Next, it decrypts the result, $\mathtt{Dec}(\bar{k}, c'_j) = c_j$. Then, it runs $\mathtt{VSID.checkQuery}(c_j, pk, pp) \rightarrow b_j$, to check the correctness of the queries.
      - If $\mathcal{S}$ accepts the query, i.e., $b_j = 1$, then calls $\mathtt{VSID.prove}(u^*, \sigma, c_j, pk, pp) \rightarrow \boldsymbol{\pi}_j$, to generate the service proof. In this case, $\mathcal{S}$ encrypts it, $\mathtt{Enc}(\bar{k}, \boldsymbol{\pi}_j) = \boldsymbol{\pi}'_j$. Next, it pads (every element of) the encrypted proof with $pad_\pi \in T_{qp}$ random values picked uniformly at random from U. Let $\boldsymbol{\pi}^*_j$ be the result. It sends the padded encrypted proof to SC at time $\mathtt{G}_{j,2}$.
      - Otherwise (if $\mathcal{S}$ rejects the query), it appends j to $\boldsymbol{m}_{\mathcal{S}}$, constructs a dummy proof $\boldsymbol{\pi}'_j$, picked uniformly at random from U, pads the result as above, and sends the padded dummy proof, $\boldsymbol{\pi}^*_j$, to SC at time $\mathtt{G}_{j,2}$.

   When $j = z$ and $\boldsymbol{m}_{\mathcal{S}} \neq \perp$, $\mathcal{S}$ sets $m_{\mathcal{S}} := (\boldsymbol{m}_{\mathcal{S}}, adr_{\text{SC}})$.

6. **Client-side Proof Verification**. $\mathtt{RCSP.verify}(\boldsymbol{\pi}^*_j, c^*_j, k, T_{qp})$
   (a) Constructs an empty vector, $\boldsymbol{m}_{\mathcal{C}} = \perp$, if $j = 1$.
   (b) Removes the pads from $\boldsymbol{\pi}^*_j$, utilising parameters of T_{qp}. Let $\boldsymbol{\pi}'_j$ be the result. It decrypts the service proof: $\mathtt{Dec}(\bar{k}, \boldsymbol{\pi}'_j) = \boldsymbol{\pi}''_j$ and then calls $\mathtt{VSID.verify}(\boldsymbol{\pi}''_j, \boldsymbol{q}_j, k, pp) \rightarrow d_j$, to verify the proof, where $\boldsymbol{q}_j \in c_j$ (and c_j is the result of removing pads from c^*_j and then decrypting the result). Note that if $\boldsymbol{\pi}'_j = \mathtt{Enc}(\bar{k}, \boldsymbol{\pi}_j)$, then $\boldsymbol{\pi}''_j = \boldsymbol{\pi}_j$.
      - If $\boldsymbol{\pi}''_j$ passes the verification (i.e., $d_j = 1$), then $\mathcal{C}$ concludes that the service for this verification has been delivered successfully.
      - Otherwise (when $\boldsymbol{\pi}''_j$ is rejected), $\mathcal{C}$ appends j to $\boldsymbol{m}_{\mathcal{C}}$.

   When $j = z$ and $\boldsymbol{m}_{\mathcal{C}} \neq \perp$, $\mathcal{C}$ sets $m_{\mathcal{C}} := (\boldsymbol{m}_{\mathcal{C}}, adr_{\text{SC}}, e')$, where e' contains the opening of $\mathtt{Com}_{sk}$ or $\perp$, as stated in Remark 2.

7. **Dispute Resolution**. $\mathtt{RCSP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp})$
   The phase takes place only in case of dispute, e.g., when $\mathcal{C}$ and/or $\mathcal{S}$ reject any proofs in the previous phases.
   (a) The arbiter sets counters: $y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}$ and $y'_{\mathcal{S}}$, that are initially set to 0, before time $\mathtt{K}_1$, where $\mathtt{K}_1 > \mathtt{G}_{z,2} + \mathtt{J}$.
   (b) $\mathcal{C}$ sends $m_{\mathcal{C}}$ and $\ddot{x}_{qp}$ to the arbiter at time $\mathtt{K}_1$. Or, $\mathcal{S}$ sends $m_{\mathcal{S}}$ and $\ddot{x}_{qp}$ to the arbiter at time $\mathtt{K}_1$.

(c) At time $\mathtt{K}_2$, the arbiter checks the validity of statement $\ddot{x}_{qp}$ sent by each party $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}\}$. To do so, it sends each $\ddot{x}_{qp}$ to SAP contract which returns either 1 or 0. The arbiter constructs an empty vector, $\boldsymbol{v}$. If party $\mathcal{P}$'s statement is accepted, then it appends every element of $\boldsymbol{m}_{\mathcal{P}}$ to $\boldsymbol{v}$. It ensures $\boldsymbol{v}$ contains only distinct elements which are in the range $[1, z]$. Otherwise (if the party's statement is rejected) it discards the party's request, $\boldsymbol{m}_{\mathcal{P}}$. It proceeds to the next step if $\boldsymbol{v}$ is not empty, otherwise it halts.

(d) The arbiter for every element $i \in \boldsymbol{v}$:

i. removes the pads from the related encrypted query-proof pair and from encrypted service proof. Let c'_i and $\boldsymbol{\pi}'_i$ be the result.

ii. decrypts the encrypted query-proof pair and encrypted service proof as follows, $\mathtt{Dec}(\bar{k}, c'_i) = c_i$ and $\mathtt{Dec}(\bar{k}, \boldsymbol{\pi}'_i) = \boldsymbol{\pi}''_i$.

iii. calls $\mathtt{VSID.identify}(\boldsymbol{\pi}''_i, c_i, k, e', pp) \rightarrow I_i$
- if $I_i = \mathcal{C}$, then it increments $y_{\mathcal{C}}$ by 1.
- if $I_i = \mathcal{S}$, then it increments $y_{\mathcal{S}}$ by 1.
- if $I_i = \perp$, then it increments $y'_{\mathcal{C}}$ or $y'_{\mathcal{S}}$ by 1, if i is in the complaint of $\mathcal{C}$ or $\mathcal{S}$ respectively.

Let $\mathtt{K}_3$ be the time that the arbiter finishes the above checks.

(e) The arbiter at time $\mathtt{K}_3$ sends $[y_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{C}}, y'_{\mathcal{S}}]$ to SC that accordingly overwrites the elements it holds (i.e., elements of $\boldsymbol{y}$) by the related vectors elements the arbiter sent.

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

(a) If SC receives "pay" message at time $\mathtt{T}_2$, where $a = 0$ or $coins^*_{\mathcal{S}} < p_{\mathcal{S}}$, then it sends $coin^*_{\mathcal{C}}$ coins to $\mathcal{C}$ and $coin^*_{\mathcal{S}}$ coins to $\mathcal{S}$. In other words, the parties can withdraw their coins if they do not reach to an agreement in the end of phase 3, i.e., server-side initiation. Otherwise (i.e., they reach to an agreement), they take the following steps.

(b) Either $\mathcal{C}$ or $\mathcal{S}$ sends "pay" message and the statement, $\ddot{x}_{cp} \in T_{cp}$, to SC at time $\mathtt{L} > \mathtt{K}_3$.

(c) SC checks the validity of the statement by sending $\ddot{x}_{cp}$ to the SAP contract which returns either 1 or 0. SC only proceeds to the next step if the output is 1.

(d) 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 $\mathcal{S}$.
- $coin_{\mathcal{R}} = l \cdot (y_{\mathcal{S}} + y_{\mathcal{C}} + y'_{\mathcal{S}} + y'_{\mathcal{C}})$ coins to the arbiter.

Discussion on the RC-S-P protocol description We conclude Subsection 7.3 with the following remarks:

- The length of a private time bubble can be agreed between the server and client to be of any size that suits them and can exceed the point where the z-th verifications is completed.
- For the sake of simplicity, in the RC-S-P protocol, we let each $y \in \{y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{S}}\}$ be a counter; instead of a binary vector, $\boldsymbol{y} \in \{\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, \boldsymbol{y}'_{\mathcal{S}}\}$, defined in the RC-S-P definition. However, it is not hard to see that the sum of all elements $\boldsymbol{y}$ of equal y, i.e., $y = \sum_{j=1}^{z} \boldsymbol{y}_j$. The same holds for the amounts of coin each party receives, $coin \in \{coin_{\mathcal{C}}, coin_{\mathcal{S}}, coin_{\mathcal{R}}\}$, in the protocol and the coin vector used in the definition, $\boldsymbol{coin} \in \{\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{R}}\}$.
- 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.
- As stated in Section 7.1, $\mathtt{RCSP.genQuery}(\cdot), \mathtt{RCSP.prove}(\cdot)$, $\mathtt{RCSP.verify}(\cdot)$ and $\mathtt{RCSP.resolve}(\cdot)$ implicitly take $a, coin^*_{\mathcal{S}}$, $p_{\mathcal{S}}$ as another inputs and execute only if $a = 1$ and $coin^*_{\mathcal{S}} = p_{\mathcal{S}}$. For the sake of simplicity, we avoided explicitly stating it in the protocol. Also, keeping track of $(y'_{\mathcal{C}}, y'_{\mathcal{S}})$ enables the arbiter to make malicious parties, that *unnecessarily* invoke it for accepting proofs in step 7(d)iii, pay for the verifications it performs.

- The total coin amounts the client receives is as follows; its initial deposit, i.e., $coin_{\mathcal{C}}^{*}$, minus the total coin amounts that the server should be paid for those verifications that it has acted honestly towards the client, i.e., $o \cdot (z - y_{\mathcal{S}})$, minus the total coin amounts the client has to pay to the arbiter when it misbehaved towards the server and the arbiter, i.e., $l \cdot (y_c + y'_c)$. The total coin amounts the server receives is as follows. Its initial deposit, i.e., $coin_{\mathcal{S}}^{*}$, plus the total coin amounts that it should get paid for those verifications that it acted honestly towards the client, i.e., $o \cdot (z - y_{\mathcal{S}})$, minus the total coin amounts it has to pay to the arbiter when it misbehaved towards the client and the arbiter, i.e., $l \cdot (y_{\mathcal{S}} + y'_{\mathcal{S}})$. Moreover the arbiter receives in total $l \cdot (y_{\mathcal{S}} + y_c + y'_{\mathcal{S}} + y'_c)$ coins to cover its cost of resolving disputes, i.e., $l \cdot (y_{\mathcal{S}} + y_c)$, plus the cost imposed to it when it is unnecessarily invoked, i.e., $l \cdot (y'_{\mathcal{S}} + y'_c)$. If all parties behave honestly, then the server receives all its deposit back plus the coin amounts they initially agreed to pay the server if it delivers accepting proofs for all z cycles, i.e., in total it receives $coin_{\mathcal{S}}^{*} + o \cdot z$ coins. Also, in this case an honest client receives all coins minus the coin amounts paid to the server for delivering accepting proofs for z cycles, i.e., in total it receives $coin_{\mathcal{C}}^{*} - o \cdot z$ coins. However, the arbiter receives no coins, as it is never invoked.
- The VSID scheme does not (need to) preserve the privacy of the proofs. However, in RC-S-P protocol each proof's privacy must be preserved, for a certain time; otherwise, the proof itself can leak its status, e.g., when it can be publicly verified. This is the reason why in the RC-S-P protocol, *encrypted* proofs are sent to the contract. Moreover, for the sake of simplicity, in the above protocol, we assumed that each arbiter's invocation has a fixed cost regardless of the number of steps it takes. To define a fine-grained costing, one can simply allocate to each step the arbiter takes a certain rate and also separate counter for the client and server.
- In the case where `VSID.verify`$(\cdot)$ is privately verifiable and the server invokes the arbiter, the client needs to provide inputs to the arbiter too. Otherwise (when it is publicly verifiable and the server invokes the arbiter), the client's involvement is not required in the dispute resolution phase. In contrast, if the client invokes the arbiter, the server's involvement is not required in that phase, regardless of the type of verifiability `VSID.verify`$(\cdot)$ supports. Furthermore, with a minor adjustment to the RC-S-P protocol, we can let the client and server be compensated (by a misbehaving party) for the transaction they send to the contract. To do so, briefly, we can let the parties, in initiation phases, agree on and include in *cp* parameters, l' and l'', that cover the client's and server's cost of sending a transaction, respectively. The parameters are encoded the same way as l is encoded. In this setting, in the coin transfer phase, the client and server receive $coin_{\mathcal{C}}^{*} - o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_c + y'_c) + l' \cdot y_{\mathcal{S}} - l'' \cdot y_c$ and $coin_{\mathcal{S}}^{*} + o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_{\mathcal{S}} + y'_{\mathcal{S}}) - l' \cdot y_{\mathcal{S}} + l'' \cdot y_c$, coins respectively. The amount of coins the arbiter receives remains unchanged.
- The server or client, even during the private time bubble, can spend (or more accurately promise to a third party) the amount of coins kept in the contract and will ultimately be transferred to it. With slight adjustments to the RC-S-P, they can do so in a privacy-preserving manner. We briefly explain how it can be done. For the sake of simplicity, we assume the server will receive $coin_{\mathcal{S}}$ coins after the bubble bursts and wants to promise $\hat{coin}_{\mathcal{S}}$ coins (where $\hat{coin}_{\mathcal{S}} \leq coin_{\mathcal{S}}$) to the third party $\mathcal{D}$ within the bubble. First, the server proves to $\mathcal{D}$ that it will receive $coin_{\mathcal{S}}$ coins after the bubble bursts. To do that, it sends the RC-S-P transcripts (that includes all proofs) to $\mathcal{D}$ which can verify the server's claim, as all proofs are publicly verifiable. Next, if $\mathcal{D}$ is convinced, the server and $\mathcal{D}$ invoke a new instance of the SAP and insert the value $\hat{coin}_{\mathcal{S}}$ into the SAP's private statement. This results in a smart contract, $\mathrm{SC}_{\mathrm{SAP}_3}$. Next, if both parties agree on the parameters of $\mathrm{SC}_{\mathrm{SAP}_3}$, then the server sends the address of $\mathrm{SC}_{\mathrm{SAP}_3}$ to the main contract of RC-S-P, i.e., SC. When the bubble bursts, SC transfers the client's share of coins to the client as before. But, SC distributes the server's coins if the server or $\mathcal{D}$ sends to it a valid proof for the above private statement (in addition to the proofs required in the Phase 8 of the original RC-S-P). Upon receiving that proof, SC invokes $\mathrm{SC}_{\mathrm{SAP}_3}$ to check the validity of the proof. If the proof is accepted, then SC sends $\hat{coin}_{\mathcal{S}}$ to $\mathcal{D}$ and $coin_{\mathcal{S}} - \hat{coin}_{\mathcal{S}}$ to the server. It is evident that this approach leaks no information about the coins amount (including $\hat{coin}_{\mathcal{S}}$) during the bubble to the public, due to the security of the SAP. The above idea can be further extended to support multiple parties. For instance, if the server wants to promise $coin_{\mathcal{S}} - \hat{coin}_{\mathcal{S}}$ coins to $\mathcal{D}'$ (after its promise to $\mathcal{D}$), it needs to send to $\mathcal{D}'$ all the proofs, including the one related to the above private statement.

- As stated previously, the proofs are sent to the contract to avoid running into the deniability issue, i.e., a malicious client wrongly claims the server never sent a proof for a certain verification or a malicious server wrongly claims it sent its proof to the client. However, in the case where the proof size is large and posting it to the smart contract would impose a high cost, the parties can use the following technique to directly communicate with each other to send and receive the proof. The server sends a signed proof directly to the client which needs to send back to the server a signed acknowledgment stating that it received the proof, within a fixed time period. If the server does not receive a valid acknowledgment on time, it sends the signed proof to the arbiter. Moreover, if the client 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 the client which allows the client to perform the rest of the computation. On the other hand, if the arbiter does not have the proof, it asks the server to send to it the client's acknowledgment. If the server provides a valid acknowledgment, then the arbiter considers the client as a misbehaving party; otherwise (if the server could not provide the acknowledgment), it considers the server as a misbehaving one. However, if both the server and client 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.

7.4 Security Analysis of RC-S-P Protocol

In this section, we analyse the security of RC-S-P protocol, presented in Section 7.3. First, we present the protocol's primary security theorem.

Theorem 2. *The RC-S-P protocol with functions F, M, E, D, Q presented in Section 7.3 is secure for auxiliary information aux, (cf. Definition 15), if the underlying VSID protocol with functions F, M, Q satisfies correctness, soundness, inputs well-formedness, and detectable abort for auxj, the SAP is secure, the signature scheme is secure, and the symmetric-key encryption scheme is IND-CPA secure.*

To prove Theorem 2, we show that RC-S-P meets all security properties defined in Section 7.1. We start by proving that RC-S-P satisfies security against a malicious server.

Lemma 1. *If the SAP and signature scheme are secure and the VSID protocol satisfies correctness, soundness, and detectable abort for auxiliary information aux, then the RC-S-P protocol presented in Section 7.3 is secure against malicious server for aux.(cf. Definition 12).*

Proof. We first consider event

$$\Big(F(u^*, \boldsymbol{q}_j, pp) = h_j\Big) \ \wedge \Big((coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o) \vee (coin_{\mathcal{R},j} \neq l \ \wedge \ y'_{\mathcal{S},j} = 1)\Big)$$

that captures the case where the server provides an accepting service proof but makes an honest client withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o$, or it makes the arbiter withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{R},j} \neq l$, if it unnecessarily invokes the arbiter. As the service proof is valid, an honest client accepts it and does not raise any dispute. However, the server would be able to make the client withdraw incorrect amounts of coins, if it manages to either

1. convince the arbiter that the client has misbehaved, by making the arbiter output $y_{\mathcal{C},j} = 1$ through the dispute resolution phase, or
2. submit to the contract, in the coin transfer phase, an accepting statement $\ddot{x}'_{cp}$ other than what was agreed in the initiation phase, i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$, so it can change the payments' parameters (e.g., l or o) or send a message on the client's behalf to invoke the arbiter unnecessarily.

Nevertheless, the server cannot falsely accuse the client of misbehaviour. This is because, due to the security of SAP, it cannot convince the arbiter to accept different decryption key or pads other than what was agreed with the client in the initiation phase. Specifically, it cannot persuade the arbiter to accept $\ddot{x}'_{qp}$, where $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$, except with a negligible probability. This ensures that the honest client's message is accessed by

the arbiter with a high probability, as the arbiter can extract the client's message using valid pad information and decryption key. On the other hand, if the adversary provides a valid statement, i.e., $\ddot{x}_{qp}$, then due to the correctness of VSID, algorithm $\texttt{VSID.identify}(\cdot)$ outputs $I_j = \perp$. Therefore, due to the security of SAP and correctness of VSID, $y_{\mathcal{C}}$ and $y_{\mathcal{S}}$ are not incremented by 1 in the j-th verification, i.e., $y_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$. Also, due to the security of SAP, the server cannot change the payment parameters by persuading the contract to accept any statement $\ddot{x}'_{cp}$ other than what was agreed initially between the client and server, except with a negligible probability when it finds the hash function's collision (in the SAP scheme). Moreover, since the proof is valid the client never raises a dispute, also due to the digital signature's unforgeability, the server cannot send a message on behalf of the client (to unnecessarily invoke the arbiter), and make the arbiter output $y'_{\mathcal{C},j} = 1$ for the j-th verification, except with a negligible probability. So with a high probability $y'_{\mathcal{C},j} = 0$. Recall, in the protocol, the total coins the client should receive after z verifications is $coin^*_{\mathcal{C}} - o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_{\mathcal{C}} + y'_{\mathcal{C}})$. Since we focus on the j-th verification, the amount of coins that should be credited to the client for that verification is

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

As shown above $y_{\mathcal{C},j} = y'_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$. So, according to Equation 1, the client is credited $\frac{coin^*_{\mathcal{C}}}{z} - o$ coins for j-th verification, with a high probability. On the other hand, as stated above, if the adversary invokes the arbiter, the arbiter with a high probability outputs $I_j = \perp$ which results in $y'_{\mathcal{S},j} = 1$. Recall, in the RC-S-P protocol, the total coins the arbiter should receive for z verifications is $l \cdot (y_{\mathcal{S}} + y_{\mathcal{C}} + y'_{\mathcal{S}} + y'_{\mathcal{C}})$, so for the j-th the credited coins should be:

$$coin_{\mathcal{R},j} = l \cdot (y_{\mathcal{S},j} + y_{\mathcal{C},j} + y'_{\mathcal{S},j} + y'_{\mathcal{C},j}) \tag{2}$$

As already shown, in the case where arbiter is unnecessarily invoked by the server, it holds that $y'_{\mathcal{S},j} = 1$; So, according to Equation 2, l coins is credited to the arbiter for the j-th verification. For the server to make the arbiter withdraw other than that amount (for the j-th verification), in the coin transfer phase, it has to send to the contract an accepting statement $\ddot{x}'_{cp}$ other than what was agreed in the initiation phase, i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$, so it can change the payments' parameters, e.g., l or o. But, as argued above, it cannot succeed with probability significantly greater than negligible. We now move on to the following event

$$\Big(F(u^*, \boldsymbol{q}_j, pp) \neq h_j\Big) \ \wedge \Big(d_j = 1 \vee y_{\mathcal{S},j} = 0 \ \vee coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} \ \vee coin_{\mathcal{R},j} \neq l\Big)$$

This event captures the case where the server provides an invalid service proof but either persuades the client to accept the proof, or persuades the arbiter to accept the proof (e.g., when the client raises a dispute) or makes the client or arbiter withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z}$ or $coin_{\mathcal{R},j} \neq l$ respectively. Nevertheless, due to the soundness of VSID, the probability that a corrupt server can convince an honest client to accept invalid proof (i.e., outputs $d_j = 1$) is negligible. So, the client detects it with a high probability and raises a dispute. On the other hand, the server may try to convince the arbiter, and make it output $y_{\mathcal{S},j} = 0$, e.g., by sending a complaint. For $y_{\mathcal{S},j} = 0$ to happen, the server has to either provide a different accepting statement $\ddot{x}'_{qp}$, than what was initially agreed with the client (i.e., $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$) and passes the verification, which requires finding the hash function's collision (in the SAP scheme), and its probability of success is negligible. Or it makes the arbiter accept an invalid proof, but due to the detectable abort property of VSID, its probability of success is also negligible. Also, as we discussed above, the probability that the adversary makes the arbiter to recognise the client as misbehaving, and output $y_{\mathcal{C},j} = 1$ is negligible too. Therefore, the arbiter outputs $y_{\mathcal{S},j} = 1$ and $y_{\mathcal{C},j} = 0$ with a high probability, in both events when it is invoked by the client or server. Also, in this case, $y'_{\mathcal{C},j} = y'_{\mathcal{S},j} = 0$ as the arbiter has already identified a misbehaving party. So, according to Equation 1, the client is credited $\frac{coin^*_{\mathcal{C}}}{z}$ coins for that verification, with a high probability. Moreover, according to Equation 2, the arbiter is credited l coins for that verification, with a high probability. The adversary may try to make them withdraw an incorrect amount of coins, e.g., in the case where it does not succeed in convincing the client or arbiter. To this end, in the coin transfer

phase, it has to send a different accepting statement than what was initially agreed with the client. But, it would succeed only with a negligible probability, due to the security of SAP.

Lemma 2. *If the SAP and signature scheme are secure and the VSID scheme satisfies correctness, inputs well-formedness, and detectable abort for auxiliary information aux, then the RC-S-P protocol presented in Section 7.3 is secure against malicious client for aux (cf. Definition 13).*

Proof. First, we consider event

$$\left(\left(M(u^*,k,pp)=\sigma \ \wedge \ Q(\text{aux},k,pp)=\boldsymbol{q}_j\right) \ \wedge \left((coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z}+o) \ \vee \ (coin_{\mathcal{R},j} \neq l \ \wedge \ y'_{\mathcal{C},j}=1)\right)\right)$$

This event captures the case where the client provides accepting metadata and query but makes the server withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z}+o$, or makes the arbiter withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{R},j} \neq l$, if it unnecessarily invokes the arbiter. Since the metadata and query's proofs are valid, an honest server accepts them and does not raise any dispute, so we have $y_{\mathcal{C},j}=0$. The client could make the server withdraw incorrect amount of coins, if it manages to either convince the arbiter, in phase 7, that the server has misbehaved, i.e., makes the arbiter output $y_{\mathcal{S},j}=1$, or submit to the contract an accepting statement $\ddot{x}'_{cp}$ other than what was agreed at the initiation phase, i.e., $\ddot{x}_{cp}$, in phase 8, or send a message on the server's behalf to invoke the arbiter unnecessarily. However, it cannot falsely accuse the server of misbehaviour, as, due to the security of SAP, it cannot convince the arbiter to accept different decryption key and pads' detail, by providing a different accepting statement $\ddot{x}'_{qp}$ (where $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$), than what was initially agreed with the server, except with negligible probability. This ensures the arbiter is given the honest server's messages, with a high probability. So, with a high probability $y_{\mathcal{S},j}=0$. On the other hand, if the adversary provides a valid statement, i.e., $\ddot{x}_{qp}$, then due to the correctness of VSID, algorithm $\texttt{VSID.identify}(\cdot)$ outputs $I_j=\perp$. So, due to the security of SAP and correctness of VSID, we would have $y_{\mathcal{C},j}=y_{\mathcal{S},j}=0$ with a high probability. Moreover, due to the security of SAP, the client cannot convince the contract to accept any statement $\ddot{x}'_{cp}$ other than what was initially agreed between the client and server (i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$), except with negligible probability. Also, it holds that $y'_{\mathcal{S},j}=0$ because an honest server never invokes the arbiter when the client's messages are well-structured and due to the signature's unforgeability, the client cannot send a signed message on the server's behalf to unnecessarily invoke the arbiter. According to RC-S-P protocol, the total coins the server should receive after z verifications is $coin^*_{\mathcal{S}}+o\cdot(z-y_{\mathcal{S}})-l\cdot(y_{\mathcal{S}}+y'_{\mathcal{S}})$. 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

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

As shown above, the following holds $y_{\mathcal{S},j}=y'_{\mathcal{S},j}=0$, which means, according to Equation 3, the server is credited $\frac{coin^*_{\mathcal{S}}}{z}+o$ coins for the j-th verification, with a high probability. Furthermore, if the adversary invokes the arbiter, the arbiter with a high probability outputs $I_j=\perp$ which yields $y'_{\mathcal{C},j}=1$. Also, as stated above, $y'_{\mathcal{S},j}=0$. Hence, according to Equation 2, the arbiter for the j-th verification is credited l coins, if it is unnecessarily invoked. As previously stated, due to the security of SAP, the client cannot make the arbiter withdraw incorrect amounts of coin by changing the payment parameters and persuading the contract to accept any statement $\ddot{x}'_{cp}$ other than what was agreed initially between the client and server, except with negligible probability. We now turn our attention to

$$\left(M(u^*,k,pp)\neq\sigma \ \wedge \ a=1\right)$$

that captures the case where the server accepts an ill-formed metadata. However, due to inputs well-formedness of VSID, the probability that event happens is negligible. So, with a high probability $a=0$. Note, in the case where $a=0$, the server does not raise any dispute, instead it avoids serving the client. Next, we move on to

$$\left(\left(Q(\text{aux}, k, pp) \neq \boldsymbol{q}_j\right) \ \wedge \left(b_j = 1 \ \vee \ y_{\mathcal{C},j} = 0 \ \vee \ coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o \ \vee \ coin_{\mathcal{R},j} \neq l\right)\right)$$

This event 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 coins, i.e., $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o$ or $coin_{\mathcal{R},j} \neq l$ respectively. Nevertheless, due to inputs well-formedness of VSID, the probability that the server outputs $b_j = 1$ in this case is negligible. When the server rejects the query and raises a dispute, the client may try to convince the arbiter and make it output $y_{\mathcal{C},j} = 0$, e.g., by sending a complaint. However, for the adversary to win, either

1. it has to provide a different accepting statement $\ddot{x}'_{qp}$, than what was initially agreed with the server (i.e., $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$) and passes the verification. Due to the security of SAP, its probability of success is negligible. Or,
2. it has to make the arbiter accept an invalid query, i.e., makes the arbiter output $y_{\mathcal{C},j} = 0$. Due to the detectable abort property of VSID, its probability of success is negligible too.

Therefore, with a high probability, we have $y_{\mathcal{C},j} = 1$. Also, as discussed above, the client cannot make the arbiter recognise the honest server as a misbehaving party with a probability significantly greater than negligible. That means with a high probability $y_{\mathcal{S},j} = 0$. Furthermore, as we already discussed, since the arbiter has identified a misbehaving party, the following holds $y'_{\mathcal{C},j} = y'_{\mathcal{S},j} = 0$. Hence, according to Equation 3 the server is credited $\frac{coin^*_{\mathcal{S}}}{z} + o$ coins for this verification. Also, the arbiter is credited l coins, according to Equation 2. Note that the adversary may still try to make them withdraw an incorrect amount of coins (e.g., if the adversary does not succeed in convincing the server or arbiter). To this end, at the coin transfer phase, it has to send a different accepting statement than what was initially agreed with the server. However, due to the security of SAP, its success probability is negligible.

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

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

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

Lemma 4. *If SAP is secure and the symmetric-key encryption scheme is IND-CPA secure, then the RC-S-P protocol presented in Section 7.3 preserves privacy for auxiliary information aux, (cf. Definition 14).*

Proof. We start with case 1, i.e., the privacy of service input. Due to the privacy property of SAP, that stems from the hiding property of the commitment scheme, given the commitments g_{qp} and g_{cp}, (that are stored in the blockchain as a result of running SAP) the adversary learns no information about the committed values (e.g., o, l, pad_π, pad_q, and $\bar{k}$), except with a negligible probability. Also, given price list pl, encoded coins $coin^*_{\mathcal{C}} = z \cdot (o_{max} + l_{max})$ and $coin^*_{\mathcal{S}} = z \cdot l_{max}$, the adversary learns nothing about the actual price that was agreed between the server and client, (o, l), for each verification, due to Lemma 3. Next we analyse the privacy of padded encrypted query vector $\boldsymbol{c}^*$. For the sake of simplicity, we focus on $\boldsymbol{q}^*_j \in c^*_j \in \boldsymbol{c}^*$, that is a padded encrypted query vector for j-th verification. Let $\boldsymbol{q}_{j,0}$ and $\boldsymbol{q}_{j,1}$ be query vectors, for j-th verification, related to the service inputs u_0 and u_1 that are picked by the adversary according to Definition 14 which

lets the environment pick $\beta \xleftarrow{\$} \{0,1\}$. Also, let $\{\boldsymbol{q}_{j,0}, ..., \boldsymbol{q}_{j,\bar{m}}\}$ be a list of all queries of different sizes. In the experiment, if $\boldsymbol{q}_{j,\beta}$ is only encrypted (but not padded), then given the ciphertext, due to semantical security of the encryption, an adversary cannot tell if the ciphertext corresponds to $\boldsymbol{q}_{j,0}$ or $\boldsymbol{q}_{j,1}$ (accordingly to u_0 or u_1) with probability significantly greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$, under the assumption that the size of $\boldsymbol{q}_{j,\beta}$ is equal to the size of largest query size [6], i.e., $Max(|\boldsymbol{q}_{j,0}|, ..., |\boldsymbol{q}_{j,\bar{m}}|) = |\boldsymbol{q}_{j,\beta}|$. The above assumption is relaxed with the use of a pad; as each encrypted query is padded to the queries' maximum size, i.e., $Max(|\boldsymbol{q}_{j,0}|, ..., |\boldsymbol{q}_{j,\bar{m}}|)$, the adversary cannot tell with a probability greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$ if the padded encrypted proof corresponds to $\boldsymbol{q}_{j,0}$ or $\boldsymbol{q}_{j,1}$, as the padded encrypted query *always has the same size* and the pad values are picked from the same range as the encryption's ciphertext are defined. The same argument holds for $\boldsymbol{w}^*_{q_j} \in c^*_j \in \boldsymbol{c}^*$. Next we analyse the privacy of padded encrypted proof vector $\boldsymbol{\pi}^*$. The argument is similar to the one presented above, however, we provide it for the sake of completeness. We focus on an element of the vector, $\pi^*_j \in \boldsymbol{\pi}^*$, that is a padded encrypted proof for j-th verification. Let $\pi_{j,0}$ and $\pi_{j,1}$ be proofs, for j-th verification, related to the service inputs u_0 and u_1, where the inputs are picked by the adversary, w.r.t. Definition 14 in which the environment picks $\beta \xleftarrow{\$} \{0,1\}$. Let $\{\pi_{j,0}, ..., \pi_{j,\bar{m}}\}$ be proof list including all proofs of different sizes. If we assume $\pi_{j,\beta}$ is only encrypted, then given the ciphertext, due to 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 $\frac{1}{2} + \mathsf{negl}(\lambda)$, if $Max(|\pi_{j,0}|, ..., |\pi_{j,\bar{m}}|) = |\pi_{j,\beta}|$. However, the assumption is relaxed with the use of a pad. In particular, since each encrypted proof is padded to the proofs' maximum size, the adversary cannot tell with a probability greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$ if the padded encrypted proof corresponds to $\pi_{j,0}$ or $\pi_{j,1}$. Also, since the value of a is independent of u_0 or u_1, and only depends on whether the metadata is well-formed, it leaks nothing about the service input u_β, β, the query-proof pair and service proof. Thus (given $\boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl$, and a) the probability that the adversary can tell the value of β is at most $\frac{1}{2} + \mathsf{negl}(\lambda)$.

Now we move on to case 2, i.e., the privacy of proof's status. Recall that in the experiment, an *invalid* query-proof pair is generated with probability $Pr_{0,j}$ and a *valid* query-proof pair is generated with probability $Pr_{1,j}$. As stated above, each encoded query-proof pair $c^*_j \in \boldsymbol{c}^*$ has a fixed size and contains random elements of U, i.e., they are uniformly random elements in the symmetric-key encryption scheme's output range. Also, it is assumed that for each j-th verification, an encoded query-proof is always provided to the contract. Therefore, each encoded pair leaks nothing, not even the query's status to the adversary. So, given only a vector of c^*_j (i.e., $\boldsymbol{c}^*$) it can learn a query-proof's status with probability at most $Pr' + \mu(\lambda)$, where $Pr' := Max\{Pr_{0,1}, Pr_{1,1}, ..., Pr_{0,z}, Pr_{1,z}\}$. On the other hand, for each j-th verification, an encoded service proof $\pi^*_j \in \boldsymbol{\pi}^*$ is always provided to the contract, regardless of the query's status. As stated above, each π^*_j has a fixed size and contains random element of U too. As we showed above, g_{cp}, g_{qp}, pl, and a leak no information about the service input, except with a negligible probability, $\mu(\lambda)$. They are also independent of the query-proof pair and service proof, so they leak no information about the pair and service proof too. So, given $\boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl$, and a, an adversary has to learn a proof's status from the aforementioned values or by correctly guessing a query's status. In other words, its probability of learning a proof' status is at most $Pr' + \mu(\lambda)$.

8 Recurring Contingent PoR Payment (RC-PoR-P) Protocol

In this section, we present recurring contingent PoR payment (RC-PoR-P) that is a concrete instantiation of the RC-S-P, when the verifiable service is PoR. Now, instead of the function F, we have F_{PoR} which is an algorithm that takes as input $\mathcal{C}$'s encoded file u^* and $\mathcal{C}$'s query $\mathbf{q}$ and outputs a proof asserting the outsourced data u is retrievable. For instance, if a PoR utilises a Merkle tree, then F_{PoR} is the algorithm that generates the Merkle tree's proofs. As a concrete instantiation, RC-PoR-P offers two primary added features. Specifically, unlike the generic RC-S-P construction (cf. Appendix 7), it (a) does not use any zk proofs (even though either $\mathcal{C}$ or $\mathcal{S}$ can be malicious) which significantly improves costs, and (b) has a much lower arbiter-side computation cost; as we will show later, this also allows for a smart contract to efficiently play the arbiter's role. Below, we first explain how the features are satisfied.

[6] The assumption that all queries have the same size is subsumed under the above assumption.

Avoiding the use of zk proofs. The majority of PoRs assume that only $\mathcal{S}$ is potentially malicious while $\mathcal{C}$ is honest. To ensure a file's availability, they rely on metadata that is either a set of tags (e.g., MACs or signatures) or a root of a Merkle tree, built on the file blocks. In the case where $\mathcal{C}$ can also be malicious, if tags are used then using zk proofs seem an obvious choice, as it allows $\mathcal{C}$ to guarantee to $\mathcal{S}$ that the tags have been constructed correctly (similar to the PoR in [8]). But, this imposes significant computation and communication costs. We observed that using a Merkle tree would benefit our protocol from a couple of perspectives; in short, it removes the need for zk proofs and it supports proof of misbehaviour. Our first observation is that if a Merkle tree is used, then $\mathcal{S}$ can efficiently check the metadata's correctness by reconstructing this tree on the file blocks, without involving zk proofs.
Low arbiter-side cost. In a Merkle tree-based PoR, in each verification, the number of proofs (or paths) are linear with the number of blocks that are probed, say ϕ. In this scheme, given the proofs, the verifier checks all proofs and rejects them if only one of them is invalid. We observed that if this scheme is used in the RC-PoR-P, then once $\mathcal{C}$ finds an invalid proof, it can send only that single invalid proof as a *proof of misbehaviour* to the arbiter.[7] This technique significantly reduces the arbiter computation cost, i.e., from $\phi \log_2(n)$ to $\log_2(n)$, where n is the number of file blocks.

The RC-PoR-P scheme (cf. Subsection 8.2) deploys the following two building blocks:

1. A PoR scheme, presented in Subsection 8.1, that can be seen as a variant of the standard Merkle tree-based PoR [35,52,39]. The security of the construction relies on the security of the underlying Merkle tree and pseudorandom function (cf. Subsection 3.3).
2. A *statement agreement protocol* (SAP), introduced in Subsection 7.2, that lets $\mathcal{S}$ and $\mathcal{C}$ efficiently agree on private statements at the beginning of the RC-PoR-P scheme. The SAP is built upon a binding and hiding commitment scheme, a smart contract, and a secure digital signature scheme used to sign transactions on the blockchain (cf. Subsections 3.3 and 3.2).

8.1 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, $\mathcal{C}$ encodes its input file using an error-correcting code, splits the result into blocks, and builds a Merkle tree on the blocks. Then, it locally stores the tree's root and sends the blocks to $\mathcal{S}$ which rebuilds the tree. At the verification time, $\mathcal{C}$ sends a PRF's key to $\mathcal{S}$ which derives a number of blocks' indices showing which blocks are probed. $\mathcal{S}$ for each probed block generates a proof. It sends all proofs to $\mathcal{C}$ which checks them. If it accepts all proofs, then it concludes that its file is retrievable. Otherwise, if it rejects some proofs, it stores only one index of the blocks whose proofs were rejected. Below, we present the PoR protocol.

1. **Client-side Setup**. $\texttt{PoR.setup}(1^\lambda, u)$
   (a) $\mathcal{C}$ uses an error-correcting code, to encode the input file, u. Let u' be the encoded file. Then, it splits u' into m blocks as follows, $u^* = u'_1||1, ..., u'_m||m$.
   (b) $\mathcal{C}$ constructs a Merkle tree on u^*'s blocks, i.e., $\texttt{MT.genTree}(u^*)$. Let σ be the root of the tree, and ϕ be the number of blocks that will be probed. It sets public parameters as $pp := (\sigma, \phi, m, \zeta)$, where ζ is a PRF's description, as defined in Subsection 3.3 . It sends pp and u^* to $\mathcal{S}$.
2. **Client-side Query Generation**. $\texttt{PoR.genQuery}(1^\lambda, pp)$
   (a) $\mathcal{C}$ picks a key $\hat{k}$ for PRF.
   (b) $\mathcal{C}$ sends $\hat{k}$ to $\mathcal{S}$.
3. **Server-side Proof Generation**. $\texttt{PoR.prove}(u^*, \hat{k}, pp)$
   (a) $\mathcal{S}$ derives ϕ pseudorandom indices from $\hat{k}$ as follows.
   $\forall i, 1 \leq i \leq \phi : q_i = \big(\texttt{PRF}(\hat{k}, i) \bmod m\big) + 1$. Note that $1 \leq q_i \leq m$. Let $\boldsymbol{q} = [q_1, ..., q_\phi]$.
   (b) $\mathcal{S}$ generates a proof $\pi_{q_i} = \texttt{MT.prove}(u^*, q_i)$, for each random index q_i. Let the final result be $\boldsymbol{\pi} = [(u^*_{q_i}, \pi_{q_i})]_{q_i \in \boldsymbol{q}}$, where i-th element in $\boldsymbol{\pi}$ corresponds to q_i, and the probed block is $u^*_{q_i}$. It sends $\boldsymbol{\pi}$ to $\mathcal{C}$.

[7] This idea is akin to the proof of misbehaviour proposed in [18].

4. **Client-side Proof Verification**. $\texttt{PoR.verify}(\boldsymbol{\pi}, \boldsymbol{q}, pp)$
   (a) If $|\boldsymbol{\pi}| = |\boldsymbol{q}| = 1$, then $\mathcal{C}$ sets $\phi = 1$. This step is only for the case where a single proof and query is provided (e.g., in the proof of misbehaviour).
   (b) $\mathcal{C}$ checks if $\mathcal{S}$ sent all proofs, by parsing each element of $\boldsymbol{\pi}$ as: $\texttt{parse}(u^*_{q_i}) = u'_{q_i}||q_i$, and checking if its index q_i equals to $\boldsymbol{q}$'s i-th element. If all checks pass, it takes the next step. Otherwise, it outputs $\boldsymbol{d} = [0, i]$, where i is the index of $\boldsymbol{\pi}$'s element that did not pass the check.
   (c) $\mathcal{C}$ checks if every path in $\boldsymbol{\pi}$ is valid, by calling $\texttt{MT.verify}(u^*_{q_i}, \pi_{q_i}, \sigma)$. If all checks pass, it outputs $\boldsymbol{d} = [1, \perp]$; otherwise, it outputs $\boldsymbol{d} = [0, i]$, where i refers to the index of the first element in $\boldsymbol{\pi}$ that does not pass the check.

The above protocol differs from the standard Merkle tree-based PoR from two perspectives; First, in step 4, $\mathcal{C}$ also outputs one of the rejected proofs' indices. Given that index (and vectors of proofs and challenges), this will let a third party *efficiently* verify that $\mathcal{S}$ did not pass the verification. Second, in step 2, instead of sending ϕ challenges, we let $\mathcal{C}$ send only a key/seed of the PRF to $\mathcal{S}$ which can derive a set of challenges from it, such a technique has been used before, e.g., in [32,20,22]. This will lead to a decrease in the $\mathcal{C}$'s communication and smart contract's storage costs.

8.2 Recurring Contingent PoR Payment (RC-PoR-P) Protocol

In this section, we present our RC-PoR-P construction. The RC-PoR-P and the generic RC-S-P design share some ideas, yet as already mentioned, the two constructions have several differences. We provide the overview of the RC-PoR-P scheme and its detailed description below.

In the beginning, $\mathcal{C}$ generates a symmetric encryption key $\bar{k}$ and sets the number of dummy values to pad encrypted proofs, pad_π. In its setup step, $\mathcal{C}$ runs $\texttt{PoR.setup}(1^\lambda, u)$ to obtain the encoding u^* and the parameters $pp := (\sigma, \phi, m, \zeta)$. The query/proof secret parameters qp include $(\bar{k}, pad_\pi, pp)$. $\mathcal{C}$ sets the coin secret parameters $cp := (o, o_{max}, l, l_{max}, z)$ (cf. Subsection 3.1) that determine $coin^*_{\mathcal{C}}$ and $p_{\mathcal{S}}$, i.e. the total number of masked coins $\mathcal{C}$ and $\mathcal{S}$ must deposit. It initiates two SAP sessions for agreements on qp and cp with $\mathcal{S}$ and deploys a smart contract, SC. It completes setup by providing $\mathcal{S}$ with u^*, the SAP parameters (including qp and cp), and the number of verifications, z, and depositing $coin^*_{\mathcal{C}}$ coins in SC. In server setup, $\mathcal{S}$ checks whether a sufficient amount of coins has been deposited by $\mathcal{C}$ and runs the agreement step of the two SAP sessions initiated by $\mathcal{C}$. If agreement is successful and the public parameters (σ, ϕ, m) verify, it sends $coin^*_{\mathcal{S}} = p_{\mathcal{S}}$ coins to SC.

After their setup is complete, $\mathcal{C}$ and $\mathcal{S}$ engage in the billing cycles phase for a number of z verifications as follows. During the j-th verification, $\mathcal{C}$ runs $\texttt{PoR.genQuery}$ and sends the output query, $\hat{k}_j$, encrypted to SC. In turn, $\mathcal{S}$ reads SC and decrypts the encrypted query. If $\hat{k}_j$ is invalid, it creates a complaint $m_{\mathcal{S},j}$. Else, it runs $\texttt{PoR.prove}$ to generate a proof $\boldsymbol{\pi}_j$ for $\hat{k}_j$. Next, it sends $\boldsymbol{\pi}_j$ encrypted and padded to SC. In order to verify, $\mathcal{C}$ removes the pads and decrypts as $\boldsymbol{\pi}_j$ and runs $\texttt{PoR.verify}$ for $\boldsymbol{\pi}_j$ and $\hat{k}_j$. If $\boldsymbol{\pi}_j$ does not pass the verification, it creates a complaint $\boldsymbol{m}_{\mathcal{C},j}$.

Dispute resolution takes place when $\mathcal{C}$ rejects service proofs or $\mathcal{S}$ rejects the queries. The arbiter $\mathcal{R}$ receives the complaint vectors $\boldsymbol{m}_{\mathcal{C}}$ and $\boldsymbol{m}_{\mathcal{S}}$ from $\mathcal{C}$ and $\mathcal{S}$ along with each party's "views" of the two SAP sessions. Given $\boldsymbol{m}_{\mathcal{S}}$ and the view of $\mathcal{S}$, if $\mathcal{S}$'s view is valid, then $\mathcal{R}$ decides for every complaint in $\boldsymbol{m}_{\mathcal{S}}$ by decrypting the corresponding query and executing $\mathcal{S}$'s steps for that query in the billing cycles phase described above. Given $\boldsymbol{m}_{\mathcal{C}}$ and the view of $\mathcal{C}$, if $\mathcal{C}$'s view is valid, then $\mathcal{R}$ decides for every complaint in $\boldsymbol{m}_{\mathcal{C}}$ by retrieving the rejected proof's details (included in the complaint), decrypting the related query and (i) executing $\mathcal{S}$'s steps for that query, (ii) executing $\mathcal{C}$'s verification for the rejected proof and the related query. The arbiter updates SC's state based upon its decisions. Finally, coin transfer is carried out according to the state of SC, as updated by $\mathcal{R}$.

Before we present the protocol, we discuss how metadata generator function M_{PoR}, the pair of encoding/decoding functions (E_{PoR}, D_{PoR}) and the query generator function Q_{PoR} (involved in the RC-S-P Definition 10) are defined in the PoR context, as they are often implicit in the original definition of PoR. Briefly, M_{PoR} is a function that processes a file and generates metadata. For instance, when PoR uses a Merkle tree, then

M_{PoR} refers to $\texttt{MT.genTree}(w) \rightarrow (tr, \sigma)$, where tr is the tree constructed on in file w and σ is the root of the tree. Encoding by E_{PoR} refers to encrypting with a symmetric key and then adding an appropriate number of pads, while decoding by D_{PoR} refers to removing the pads and then decrypting with the symmetric key. Furthermore, Q_{PoR} can be a $\texttt{PRF}$ that generates a set of pseudorandom strings in a certain range, e.g., file block's indices.

1. **Key Generation**. $\texttt{RCPoRP.keyGen}(1^\lambda)$
(a) $\mathcal{C}$ picks a fresh symmetric encryption key $\bar{k} \leftarrow \texttt{SKE.keyGen}(1^\lambda)$.
(b) $\mathcal{C}$ sets parameter pad_π: the number of dummy values to pad encrypted proofs. Let $sk' := (pad_\pi, \bar{k})$. The key's size is part of the security parameter. Let $k' := (sk', pk')$, where $pk' := (adr_\mathcal{C}, adr_\mathcal{S})$.
2. **Client-side Initiation**. $\texttt{RCPoRP.cInit}(1^\lambda, u, k', z, pl)$
   (a) Calls $\texttt{PoR.setup}(1^\lambda, u) \rightarrow (u^*, pp)$ to encode u. It appends $pp := (\sigma, \phi, m, \zeta)$ and sk' to secret parameters qp.
   (b) Sets coin secret parameters $cp := (o, o_{max}, l, l_{max}, z)$, then $coin^*_\mathcal{C} = z \cdot (o_{max} + l_{max})$ and $p_\mathcal{S} = z \cdot l_{max}$, given the price list pl, where $coin^*_\mathcal{C}$ and $p_\mathcal{S}$ are the total number of masked coins $\mathcal{C}$ and $\mathcal{S}$ should deposit. Section 3.1 defines the parameters.
   (c) Calls $\texttt{SAP.init}(1^\lambda, adr_\mathcal{C}, adr_\mathcal{S}, qp) \rightarrow (r_{qp}, g_{qp}, adr_{\text{SAP}_1})$ and $\texttt{SAP.init}(1^\lambda, adr_\mathcal{C}, adr_\mathcal{S}, cp) \rightarrow (r_{cp}, g_{cp}, adr_{\text{SAP}_2})$ to initiate agreements on qp and cp with $\mathcal{S}$. Let $T_{qp} := (\ddot{x}_{qp}, g_{qp})$ and $T_{cp} := (\ddot{x}_{cp}, g_{cp})$, s.t. $\ddot{x}_{qp} := (qp, r_{qp})$ and $\ddot{x}_{cp} := (cp, r_{cp})$ 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, $coin^*_\mathcal{C}$, $p_\mathcal{S}$, adr_{SAP_1}, adr_{SAP_2}, pk', including time values $\texttt{Time} := \{\texttt{T}_0, ..., \texttt{T}_2, \texttt{G}_{1,1}, ..., \texttt{G}_{z,2}, \texttt{J}, \texttt{K}_1, ..., \texttt{K}_6, \texttt{L}\}$ and a vector $[y_\mathcal{C}, y'_\mathcal{C}, y_\mathcal{S}, y'_\mathcal{S}]$ initialized as $[0, 0, 0, 0]$. It deploys SC. Let adr_{SC} be the address of the deployed SC and $\boldsymbol{y} := [y_\mathcal{C}, y'_\mathcal{C}, y_\mathcal{S}, y'_\mathcal{S}, adr_{\text{SC}}]$.
   (e) Deposits $coin^*_\mathcal{C}$ coins in the contract. It sends $u^*, z, \ddot{x}_{qp}$, and $\ddot{x}_{cp}$ (along with adr_{SC}) to $\mathcal{S}$. Let $\texttt{T}_0$ be the time that the above process finishes.
3. **Server-side Initiation**. $\texttt{RCPoRP.sInit}(u^*, z, T, p_\mathcal{S}, \boldsymbol{y})$
   (a) Checks the parameters in T (e.g., qp and cp) and in SC (e.g., $p_\mathcal{S}, \boldsymbol{y}$) and ensures sufficient amount of coins has been deposited by $\mathcal{C}$.
   (b) Calls $\texttt{SAP.agree}(qp, r_{qp}, g_{qp}, adr_\mathcal{C}, adr_{\text{SAP}_1}) \rightarrow (g'_{qp}, b_1)$ and $\texttt{SAP.agree}(cp, r_{cp}, g_{cp}, adr_\mathcal{C}, adr_{\text{SAP}_2}) \rightarrow (g'_{cp}, b_2)$, to check and agree on qp and cp.
   (c) If $b_1 = 0$ or $b_2 = 0$, it sets $a = 0$. Otherwise, it verifies the public parameters correctness as follows (i) rebuilds the Merkle tree on u^* and checks the resulting root equals σ, and (ii) checks $|u^*| = m$ and $\phi \leq m$, where $(m, \phi) \in T$, and $\sigma \in pp \in T$. If the checks pass, it sets $a = 1$; else, it sets $a = 0$. It sends a and $coin^*_\mathcal{S} = p_\mathcal{S}$ coins to SC at time $\texttt{T}_1$, where $coin^*_\mathcal{S} = \perp$ if $a = 0$.

   $\mathcal{S}$ and $\mathcal{C}$ can withdraw their coins at time $\texttt{T}_2$, if $\mathcal{S}$ sends $a = 0$, fewer coins than $p_\mathcal{S}$, or nothing to the SC. To withdraw, $\mathcal{S}$ or $\mathcal{C}$ sends a "pay" message to $\texttt{RCPoRP.pay}(.)$ at time $\texttt{T}_2$.
   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, $\texttt{G}_{j,1}$ and $\texttt{G}_{j,2}$, where $\texttt{G}_{j,2} > \texttt{G}_{j,1}$, and $\texttt{G}_{1,1} > \texttt{T}_2$.
4. **Client-side Query Generation**. $\texttt{RCPoRP.genQuery}(1^\lambda, T_{qp})$
   (a) Calls $\texttt{PoR.genQuery}(1^\lambda, pp) \rightarrow \hat{k}_j$, where $pp \in T_{qp}$.
   (b) Sends encrypted query $c^*_j = \texttt{Enc}(\bar{k}, \hat{k}_j)$ to SC at time $\texttt{G}_{j,1}$.
5. **Server-side Proof Generation**. $\texttt{RCPoRP.prove}(u^*, c^*_j, T_{qp})$
   (a) Decrypts the query, $\hat{k}_j = \texttt{Dec}(\bar{k}, c^*_j)$, where $\bar{k} \in T_{qp}$.
   (b) Checks the query's correctness by ensuring $\hat{k}_j$ is not empty, and is in the key's universe, i.e., $\hat{k}_j \in \{0,1\}^\psi$. If the checks pass, it sets $b_j = 1$; otherwise, it sets $b_j = 0$.
      - if $b_j = 1$, it sets $m_{\mathcal{S},j} = \perp$. Also, it generates proofs vector by calling $\texttt{PoR.prove}(u^*, \hat{k}_j, pp) \rightarrow \boldsymbol{\pi}_j$. Then, it encrypts the proofs, i.e., for $1 \leq g \leq |\boldsymbol{\pi}_j| : \texttt{Enc}(\bar{k}, \boldsymbol{\pi}_j[g]) = \boldsymbol{\pi}'_j[g]$. Let $\boldsymbol{\pi}'_j$ contain the encrypted proofs. It pads every encrypted proof in $\boldsymbol{\pi}'_j$ with $pad_\pi \in T_{qp}$ random values picked from the encryption's output range, U. Let $\boldsymbol{\pi}^*_j$ be the result. It sends $\boldsymbol{\pi}^*_j$ to SC at time $\texttt{G}_{j,2}$.
      - if $b_j = 0$, it sets the complaint $m_{\mathcal{S},j} = j$. It constructs a dummy proof $\boldsymbol{\pi}'_j$ whose elements are randomly picked from U, pads the result as above, and sends the result, $\boldsymbol{\pi}^*_j$, to SC at time $\texttt{G}_{j,2}$.

It outputs b_j and $m_{\mathcal{S},j}$.

6. **Client-side Proof Verification**. $\texttt{RCPoRP.verify}(\boldsymbol{\pi}^*_j, c^*_j, T_{qp})$
   (a) Removes the pads from $\boldsymbol{\pi}^*_j$, yielding $\boldsymbol{\pi}'_j$. It decrypts the service proofs $\texttt{Dec}(\bar{k}, \boldsymbol{\pi}'_j) = \boldsymbol{\pi}_j$ and then verifies the proof by calling $\texttt{PoR.verify}(\boldsymbol{\pi}_j, \hat{k}_j, pp) \rightarrow \boldsymbol{d}_j$, where $\hat{k}_j = \texttt{Dec}(\bar{k}, c^*_j)$.
      - if $\boldsymbol{\pi}_j$ passes the verification, i.e., $\boldsymbol{d}_j[0] = 1$, it sets $\boldsymbol{m}_{\mathcal{C},j} = \perp$ and concludes that the service for this verification was delivered.
      - otherwise (i.e., $\boldsymbol{d}_j[0] = 0$), it sets $g = \boldsymbol{d}_j[1]$ and the complaint $\boldsymbol{m}_{\mathcal{C},j} = [j, g]$. Recall, $\boldsymbol{d}_j[1]$ refers to a rejected proof's index in proof vector $\boldsymbol{\pi}_j$.
   (b) It outputs $\boldsymbol{d}_j$ and $\boldsymbol{m}_{\mathcal{C},j}$.
7. **Dispute Resolution**. $\texttt{RCPoRP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, T_{qp})$
   This phase takes place only in case of dispute, i.e., when $\mathcal{C}$ rejects service proofs or $\mathcal{S}$ rejects the queries.
   (a) The arbiter $\mathcal{R}$ ensures counters: $y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}$ and $y'_{\mathcal{S}}$ are set to 0, before time $\texttt{K}_1$, where $\texttt{K}_1 > \texttt{G}_{z,2} + \texttt{J}$.
   (b) $\mathcal{S}$ sends complaints $\boldsymbol{m}_{\mathcal{S}}$ and $\ddot{x}_{qp}$ to the arbiter, at time $\texttt{K}_1$.
   (c) Upon receiving $\boldsymbol{m}_{\mathcal{S}}$ and $\ddot{x}_{qp}$, the arbiter takes the following steps at time $\texttt{K}_2$.
      i. checks $\ddot{x}_{qp}$'s validity, by calling the SAP's verification which returns d. If the output is $d = 0$, it discards $\boldsymbol{m}_{\mathcal{S}}$ and does not take steps 7(c)ii and 7(c)iii. Otherwise, it proceeds to the next step.
      ii. removes from $\boldsymbol{m}_{\mathcal{S}}$ any element that is duplicated or is not in the range $[1, z]$. It constructs an empty vector $\boldsymbol{v}$.
      iii. for any element $i \in \boldsymbol{m}_{\mathcal{S}}$: fetches the related encrypted query $c^*_i \in \boldsymbol{c}^*$ from SC and decrypts it as $\hat{k}_i = \texttt{Dec}(\bar{k}, c^*_i)$; it checks the query by doing the same checks performed in step 5b. If the query is rejected, it increments $y_{\mathcal{C}}$ by 1 and appends i to $\boldsymbol{v}$. If the query is accepted, it increments $y'_{\mathcal{S}}$ by 1. Let $\texttt{K}_3$ be the time the above checks are complete.
   (d) $\mathcal{C}$ sends complaints $\boldsymbol{m}_{\mathcal{C}}$ and $\ddot{x}_{qp}$ to the arbiter, at time $\texttt{K}_4$.
   (e) Upon receiving $\boldsymbol{m}_{\mathcal{C}}$ and $\ddot{x}_{qp}$, $\mathcal{R}$ takes the below steps at $\texttt{K}_5$.
      i. checks $\ddot{x}_{qp}$'s validity, by calling the SAP's verification which returns d'. If $d' = 0$, it discards $\boldsymbol{m}_{\mathcal{C}}$, and does not take steps 7(e)ii-7(e)iii. Otherwise, it proceeds to the next step.
      ii. ensures each vector $\boldsymbol{m} \in \boldsymbol{m}_{\mathcal{C}}$ is well-formed. Specifically, it checks there exist no two vectors: $\boldsymbol{m}, \boldsymbol{m}' \in \boldsymbol{m}_{\mathcal{C}}$ such that $\boldsymbol{m}[0] = \boldsymbol{m}'[0]$. If such vectors exist, it deletes the redundant ones from $\boldsymbol{m}_{\mathcal{C}}$. This ensures no two claims refer to the same verification. It removes any vector $\boldsymbol{m}$ from $\boldsymbol{m}_{\mathcal{C}}$ if $\boldsymbol{m}[0]$ is not in the range $[1, z]$ or if $\boldsymbol{m}[0] \in \boldsymbol{v}$. The latter check ensures $\mathcal{C}$ cannot hold $\mathcal{S}$ accountable if $\mathcal{C}$ generated an invalid query for the same verification.
      iii. for every vector $\boldsymbol{m} \in \boldsymbol{m}_{\mathcal{C}}$:
         A. retrieves a rejected proof's details by setting $j = \boldsymbol{m}[0]$ and $g = \boldsymbol{m}[1]$. Recall that g refers to the index of a rejected proof in the proof vector which was generated for j-th verification, i.e., $\boldsymbol{\pi}_j$.
         B. fetches the related encrypted query $c^*_j \in \boldsymbol{c}^*$ from SC and decrypts it: $\hat{k}_j = \texttt{Dec}(\bar{k}, c^*_j)$. It removes the pads only from g-th padded encrypted proof. Let $\boldsymbol{\pi}'_j[g]$ be the result. Next, it decrypts the encrypted proof, $\texttt{Dec}(\bar{k}, \boldsymbol{\pi}'_j[g])) = \boldsymbol{\pi}_j[g]$.
         C. identifies the misbehaving party as follows.
            - verifies $\hat{k}_j$ by doing the same checks done in step 5b. If the checks do not pass, it sets $I_j = \mathcal{C}$ and skips the next two steps; otherwise, it proceeds to the next step.
            - derives the related challenged block's index from $\hat{k}_j$, by computing $q_g = \big(\texttt{PRF}(\hat{k}_j, g) \bmod m\big)+1$.
            - verifies only g-th proof, by calling $\texttt{PoR.verify}(\boldsymbol{\pi}_j[g],\ q_g,\ pp) \rightarrow \boldsymbol{d}'$. If $\boldsymbol{d}'[0] = 0$, then it sets $I_j = \mathcal{S}$. Otherwise, it outputs $I_j = \perp$.
            - if $I_j = \mathcal{C}$, it increments $y_{\mathcal{C}}$ by 1. If $I_j = \mathcal{S}$, it increments $y_{\mathcal{S}}$ by 1. If $I_j = \perp$, it increments $y'_{\mathcal{C}}$ by 1.
   (f) The arbiter at time $\texttt{K}_6$ sends $[y_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{C}}, y'_{\mathcal{S}}]$ to SC which accordingly adds them to $\boldsymbol{y}$.
8. **Coin Transfer**. $\texttt{RCPoRP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}})$
   (a) If SC receives "pay" message at time $\texttt{T}_2$, where $a = 0$ or $coins^*_{\mathcal{S}} < p_{\mathcal{S}}$, then it sends $coin^*_{\mathcal{C}}$ coins to $\mathcal{C}$ and $coin^*_{\mathcal{S}}$ coins to $\mathcal{S}$. Otherwise (i.e., they reach an agreement), the following step is executed.
   (b) If SC receives "pay" message and $\ddot{x}_{cp} \in T_{cp}$ at time $\texttt{L} > \texttt{K}_6$, it checks $\ddot{x}_{cp}$'s validity by calling the SAP's verification which returns d''.

(c) If $d'' = 1$, SC distributes the coins to the parties as follows:
   i. $coin_{\mathcal{C}} = coin^*_{\mathcal{C}} - o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_{\mathcal{C}} + y'_{\mathcal{C}})$ coins to $\mathcal{C}$.
   ii. $coin_{\mathcal{S}} = coin^*_{\mathcal{S}} + o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_{\mathcal{S}} + y'_{\mathcal{S}})$ coins to $\mathcal{S}$.
   iii. $coin_{\mathcal{R}} = l \cdot (y_{\mathcal{S}} + y_{\mathcal{C}} + y'_{\mathcal{S}} + y'_{\mathcal{C}})$ coins to the arbiter.

Briefly, the RC-PoR-P protocol's correctness holds dues to the correctness of PoR, symmetric key encryption, SAP, and smart contract. Appendix E presents a more detailed discussion. Below, we state our main theorem on the security of the RC-PoR-P scheme. Appendix F presents the theorem's proof.

Theorem 3. *The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ described in Subsections 8.1 and 8.2 is secure (cf. Definition 15), if the underlying Merkle tree, pseudorandom function, commitment scheme, and digital signature scheme are secure, and the underlying symmetric-key encryption scheme is IND-CPA secure.*

Due to the efficiency of the arbiter-side algorithm, in the above protocol, we can delegate the arbiter's role to the smart contract, SC. In Appendix G, we explain how the RC-PoR-P's protocol (and definition) can be adjusted to support such a delegation.

9 Evaluation of RC-PoR-P

In this section, we provide an analysis of the RC-PoR-P protocol. Table 2 summarises the protocol's concrete cost (we also provide a table for its asymptotic cost in Appendix H). Also, we compare RC-PoR-P with (a) the zero-knowledge contingent (publicly verifiable) PoR payment in [17] and the fair PoR payment scheme in [3] that are more efficient than the state-of-the-art and closest to our work. Table 3 summarises the comparison. The analysis of RC-PoR-P covers both asymptotic and concrete overheads. To conduct the concrete cost study, we have implemented RC-PoR-P. The protocol's off-chain and on-chain parts have been implemented in C++ and Solidity programming languages respectively. To conduct the off-chain experiment, we used a server with dual Intel Xeon Gold 5118, 2.30 GHz CPU and 256 GB RAM. To carry out 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 ran the experiment 10 times. In the experiment, we used the SHA-2 hash function and set its output length and the security parameter to 128 bits. We set the size of every block to 128 bits, as in [58]. We used a random file whose size is in the range [64 MB, 4 GB]. 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 allow the parties to use the technique explained in Section 7.3, which lets the server and client exchange the (PoR) proofs off-chain in an irrefutable fashion[8]. The prototype implementation utilises the Cryptopp [21] and GMP [56] libraries. The protocol's off-chain and on-chain source code are publicly available in [1] and [2] respectively.

9.1 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 RC-PoR-P. $\mathcal{C}$'s cost is as follows. In phase 2, its cost in step 2a involves $m \cdot \sum_{i=1}^{\log_2(m)} \frac{1}{2^i}$ invocations of a hash function. So its complexity in this step is $O(m)$. Its total cost in steps 2b and 2c involves two invocations of the hash function. Therefore, the client-side total complexity in this phase is $O(m)$. In this phase, its off-chain *run-time* increases about 2× (i.e., from 23.1 to 45.5, ..., from 732.1 to 1596.6 seconds) when m increases (i.e., from 2^{22} to 2^{23}, ..., from 2^{27} to 2^{28} blocks). This phase also costs it $123 \cdot 10^{-5}$ ether. In phase 4, $\mathcal{C}$ invokes PRF and symmetric-key encryption ϕ times and

[8] For each j-th verification, $\mathcal{S}$ sends each related path to $\mathcal{C}$, via an authenticated channel. If $\mathcal{C}$ rejects a path, then it inserts into its complaint $\mathcal{S}$'s message that includes one of the invalid paths for j-th verification. Our analysis excludes signature generation and verification processes as they can be efficiently incorporated by using standard authenticated channels (e.g., PKI-based XML signatures).

Table 2: RC-PoR-P off-chain run-time (in seconds) and on-chain cost, of z verifications; breakdown by phases. In the table, z' is the maximum number of complaints the client and server send to the arbiter, and m is the number of blocks in a file.

Phase	Off-chain cost							On-chain cost	
	$m: 2^{22}$	$m: 2^{23}$	$m: 2^{24}$	$m: 2^{25}$	$m: 2^{26}$	$m: 2^{27}$	$m: 2^{28}$	Ether	US Dollar
Client-side Init.	23.1	45.5	89.7	185.8	413	732.1	1596.6	$123 \cdot 10^{-5}$	3.42
Server-side Init.	8.9	16.5	33.2	134.6	149.4	248.8	548.8	$9 \cdot 10^{-5}$	0.22
Client-side Query Gen.	-	-	-	-	-	-	-	$6 \cdot 10^{-5} \cdot z$	$0.17 \cdot z$
Server-side Proof Gen.	$22.4 \cdot z$	$30.4 \cdot z$	$57.4 \cdot z$	$166.8 \cdot z$	$376.1 \cdot z$	$793.1 \cdot z$	$1820.7 \cdot z$	-	-
Client-side Proof Ver.	$0.09 \cdot z$	$0.11 \cdot z$	$0.12 \cdot z$	$0.16 \cdot z$	$0.18 \cdot z$	$0.21 \cdot z$	$0.24 \cdot z$	-	-
Arbiter-side Dispute Res.	$2 \cdot 10^{-5} \cdot z'$	$4 \cdot 10^{-5} \cdot z'$	$8 \cdot 10^{-5} \cdot z'$	$8 \cdot 10^{-5} \cdot z'$	$9 \cdot 10^{-5} \cdot z'$	$9 \cdot 10^{-5} \cdot z'$	$10^{-4} \cdot z'$	10^{-4}	0.27
Coin Transfer	-	-	-	-	-	-	-	$6 \cdot 10^{-5}$	0.17

Table 3: Contingent PoRs comparison. In the table, m is the number of blocks in a file, T is a time parameter, and ϕ is the number of challenged blocks.

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

once respectively. So, for z verifications its total computation cost is $O(z \cdot \phi)$. Its off-chain run-time in this phase is negligibly small. This phase also costs it $6 \cdot 10^{-5} \cdot z$ ether. In phase 6, $\mathcal{C}$ for each verification decrypts and verifies proofs 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, its total complexity in this phase is $O(z \cdot \phi \cdot \log_2(m))$. Its off-chain run-time in this phase is very low and grows almost 1.1× (i.e., from $0.09 \cdot z$ to $0.11 \cdot z$, ..., from $0.21 \cdot z$ to $0.24 \cdot z$ seconds) when m increases.

Now, we analyse $\mathcal{S}$'s computation cost. In phase 3, $\mathcal{S}$'s complexity is $O(m)$. Its off-chain run-time in this phase grows 2× (i.e., from 8.9 to 16.5, ..., from 248.8 to 548.8 seconds) when m increases. This phase costs it $9 \cdot 10^{-5}$ ether. In phase 5, $\mathcal{S}$ decrypts a value for each verification, generates and encrypts proofs that require $\phi \cdot \log_2(m)$ invocations of the hash function and $\phi \cdot (\log_2(m) + 1)$ invocations of symmetric key encryption, for each verification. So, its total complexity in phase 5 is $O(z \cdot \phi \cdot \log_2(m))$. In this phase, its off-chain run-time grows about 2.1× (i.e., from $22.4 \cdot z$ to $30.4 \cdot z$, ..., from $793.1 \cdot z$ to $1820.7 \cdot z$ seconds) when m increases. [9]

Next, we analyse $\mathcal{R}$'s cost in phase 7. First, we evaluate $\mathcal{R}$'s cost when it is invoked by an honest $\mathcal{S}$. In this case, it invokes the hash function twice and decrypts $|\boldsymbol{v}_{\mathcal{S}}|$ queries, where $|\boldsymbol{v}_{\mathcal{S}}|$ is the total number of verifications that $\mathcal{S}$ complained about and $|\boldsymbol{v}_{\mathcal{S}}| \leq z$. Now, we evaluate its cost when it is invoked by an honest $\mathcal{C}$. It invokes the hash function twice to check the correctness of the statement, $\ddot{x}_{qp}$, sent by the client. It invokes the hash function $|\boldsymbol{v}_{\mathcal{C}}| \cdot (\log_2(m) + 2)$ times and the symmetric key encryption $|\boldsymbol{v}_{\mathcal{C}}| \cdot (\log_2(m) + 2)$ times, where $|\boldsymbol{v}_{\mathcal{C}}|$ is the total number of verifications that $\mathcal{C}$ complained about. Thus, its cost, in phase 7 is at most $O(z' \cdot \log_2(m))$, where $z' = Max(|\boldsymbol{v}_{\mathcal{C}}|, |\boldsymbol{v}_{\mathcal{S}}|)$ and $z' \leq z$. Note that due to the use of the proof of misbehaviour in the protocol, $\mathcal{R}$'s cost is about $\frac{1}{\phi} = \frac{1}{460}$ of its computation cost in the absence of such

[9] To determine $\mathcal{S}$'s cost for generating a proof, we considered the case where $\mathcal{S}$ does not store the Merkle tree nodes (to save storage space), instead it generates the tree's paths every time a challenge is given to it. If we let $\mathcal{S}$ store the tree, then it would have a lower computation overhead.

technique where it has to check all ϕ proofs for each verification.[10] Its off-chain run-time is very low and increases about 1.3× (i.e., from $2 \cdot 10^{-5} \cdot z'$ to $4 \cdot 10^{-5} \cdot z'$, ..., from $9 \cdot 10^{-5} \cdot z'$ to $10^{-4} \cdot z'$ seconds) when m increases. Phase 7 also imposes 10^{-4} ether to $\mathcal{R}$. In phase 8, SC invokes the hash function only twice, so its computation complexity is constant. This phase imposes $6 \cdot 10^{-5}$ ether to the party that calls `RCPoRP.pay`.

9.2 Communication Cost

We first analyse $\mathcal{C}$'s communication cost. In phase 2, $\mathcal{C}$ sends $||u^*|| + 384$ bits. So, in this phase, its complexity is $O(||u^*||)$. In phase 7, it sends $(\ddot{x}_{qp}, m_c)$, where $\ddot{x}_{qp}$ contains (a) padding information whose size is a few bits and (b) the symmetric-key encryption's key whose size is 128 bits. Also, m_c contains at most z invalid paths of the Merkle tree. Thus, in this phase, its communication cost is $z \cdot \log_2(||u^*||) + 128$ bits or $O(z \cdot \log_2(||u^*||))$. $\mathcal{S}$'s complexity for z verifications is $O(z \cdot ||\boldsymbol{\pi}_j^*||)$, as in phase 5, for each verification, it sends out a proof vector $\boldsymbol{\pi}_j^*$. $\mathcal{R}$'s communication cost is constant, as it only sends a transaction containing four values in phase 7.

9.3 Comparison

The fair PoR scheme in [3] assumes that $\mathcal{C}$ is trusted. The initiation phase involves $O(z)$ modular exponentiations and $O(m + z\phi)$ modular multiplications to generate puzzles and MACs respectively. Given the puzzles, $\mathcal{S}$ has to *continuously* solve them sequentially until all z verifications end, which requires $\mathcal{S}$ to perform the exponentiations even between two consecutive verifications. This requires $\mathcal{S}$ to perform $O(Tz)$ exponentiations and z modular multiplications, where T is a time parameter. For z verifications, $\mathcal{S}$ performs $O(z\phi)$ multiplications to generate z proofs. A verifier performs $O(z\phi)$ multiplications to verify all proofs. Now we focus on the scheme in [17]. As we showed in Section 4.1, this scheme is not secure against a malicious $\mathcal{C}$. In the initiation phase, $\mathcal{C}$ generates 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 (i) performs $O(z\phi)$ exponentiations to combine the signatures, (ii) invokes the hash function at least $O(1)$ times, and (iii) invokes zk proof system $O(z\phi)$ times. The scheme imposes the same computation complexity on the verifier as it does on the prover. Campanelli *et al.* [17] provide an implementation of zkCSP for publicly and privately verifiable PoRs. We were informed by Campanelli that the total size of the outsourced file used in their experiment is at most 256 bits, which is very small. In contrast, in our experiment, we used a much large file size, i.e., 4-GB.

Since both schemes in [3] and [17] use homomorphic tags, proofs for each verification can be combined resulting in constant proof size, i.e., $O(1)$. These schemes do not address the privacy issue we highlighted in Section 4.1. However, RC-PoR-P is secure against a malicious $\mathcal{C}$ and rectifies the privacy issue. Similar to the other two schemes, its initiation complexity is $O(m)$; but, unlike them, it does not require any modular exponentiations. Instead, it involves only invocations of the 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), it is only 28. RC-PoR-P's prove and verify algorithms, similar to the ones [3], involve only symmetric key operations; whereas, the ones in [17] need asymmetric key operations. Also, RC-PoR-P's the proof size complexity is larger than the other two schemes; However, each message length in RC-PoR-P is much shorter than the one in [17], i.e., 128-bit vs 2048-bit.

Thus, overall RC-PoR-P is computationally more efficient than [17] and [3] while offering stronger security guarantees (i.e., security against a malicious client and privacy).

10 Conclusion

Fair exchange is an interesting problem. It captures the scenario in which two mutually distrusted parties want to swap digital items such that either each party gets the other's item, or neither party does. Solutions to the

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

problem are often certain cryptographic protocols. They have numerous real-world applications; especially, nowadays where the use of the Internet for conducting business is swiftly increasing. Many years ago, it was shown that fairness is unachievable without the aid of a trusted third party. However, the advent of blockchain offered a possibility to eliminate the trusted third party's involvement. Therefore, various blockchain-based fair exchange protocols were proposed. In this work, we showed that the blockchain-based fair exchange protocols are not suitable for the exchange of digital coins and digital verifiable services. If they are used directly, in this setting, then serious issues would arise, i.e., real-time information leakage and waste of seller's resources. We formally defined and proposed a generic construction called "recurring contingent service payment" (RC-S-P) that addresses the issues. RC-S-P lets a fair exchange of digital coins and verifiable service's proof, while ensuring that the buyer cannot waste the seller's resources, and the parties' privacy is preserved. RC-S-P uses smart contracts; however, most of the computation is performed off-chain which makes the smart contracts-side computation very low. RC-S-P is the first fair service payment scheme that offers a combination of the above appealing features. We also presented concrete instantiation of the RC-S-P, when the verifiable service is the "proofs of retrievability" (PoR). The instantiation is called "recurring contingent PoR payment" (RC-PoR-P). We implemented the RC-PoR-P and analysed its costs. Our cost evaluation indicated that the RC-PoR-P is efficient. RC-PoR-P is the first PoR scheme that offers all the above features at once.

Acknowledgments

Aydin Abadi and Steven J. Murdoch were supported by REPHRAIN: The National Research Centre on Privacy, Harm Reduction and Adversarial Influence Online, under UKRI grant: EP/V011189/1. Steven J. Murdoch was also 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-PoR-P-Source-code/blob/master/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-PoR-P-Source-code/blob/master/RC-PoR-P-Smart-Contract.sol
3. Abadi, A., Kiayias, A.: Multi-instance publicly verifiable time-lock puzzle and its applications. In: Financial Cryptography and Data Security Conference, FC (2021)
4. Abadi, A., Murdoch, S.J.: Payment with dispute resolution: A protocol for reimbursing frauds' victims. IACR Cryptol. ePrint Arch. p. 107 (2022), https://eprint.iacr.org/2022/107
5. Abadi, A., Murdoch, S.J., Zacharias, T.: Recurring contingent payment for proofs of retrievability. IACR Cryptol. ePrint Arch. (2021), https://eprint.iacr.org/2021/1145
6. Amazon: Amazon s3 pricing (2021), https://aws.amazon.com/s3/pricing/
7. Androulaki, E., Karame, G., Roeschlin, M., Scherer, T., Capkun, S.: Evaluating user privacy in bitcoin. In: Financial Cryptography and Data Security Conference FC (2013)
8. Armknecht, F., Bohli, J.M., Karame, G.O., Liu, Z., Reuter, C.A.: Outsourced proofs of retrievability. In: CCS'14
9. Asokan, N., Schunter, M., Waidner, M.: Optimistic protocols for fair exchange. In: Graveman, R., Janson, P.A., Neuman, C., Gong, L. (eds.) Conference on Computer and Communications Security CCS (1997)
10. Asokan, N., Shoup, V., Waidner, M.: Optimistic fair exchange of digital signatures (extended abstract). In: Nyberg, K. (ed.) International Conference on the Theory and Application of Cryptographic Techniques EUROCRYPT (1998)
11. 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'07
12. Bao, F., Deng, R.H., Mao, W.: Efficient and practical fair exchange protocols with off-line TTP. In: IEEE Symposium on Security and Privacy (1998)
13. Barber, S., Boyen, X., Shi, E., Uzun, E.: Bitter to better - how to make bitcoin a better currency. In: Financial Cryptography and Data Security Conference, FC (2012)
14. Bary, E.: Zoom stock falls after service outage (2020), https://www.marketwatch.com/story/zoom-stock-falls-amid-service-ou

15. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: IEEE Symposium on Security and Privacy, SP (2014)
16. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000
17. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS'17
18. Canetti, R., Riva, B., Rothblum, G.N.: Practical delegation of computation using multiple servers. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011
19. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: Proceedings of the 18th Annual ACM Symposium on Theory of Computing. pp. 364–369. ACM (1986)
20. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Kilian, J. (ed.) TCC. Springer (2005)
21. Dai, W.: Crypto++ library (2014), https://cryptopp.com
22. Damgård, I., Ishai, Y.: Constant-round multiparty computation using a black-box pseudorandom generator. In: Shoup, V. (ed.) CRYPTO, 2005, Proceedings (2005)
23. Dong, C., Chen, L., Camenisch, J., Russello, G.: Fair private set intersection with a semi-trusted arbiter. In: Data and Applications Security and Privacy (2013)
24. Dropbox: Choose the right dropbox for you (2021), https://www.dropbox.com/plans?tab=personal
25. Du, Y., Duan, H., Zhou, A., Wang, C., Au, M.H., Wang, Q.: Enabling secure and efficient decentralized storage auditing with blockchain. IEEE Transactions on Dependable and Secure Computing (2021)
26. Dziembowski, S., Eckey, L., Faust, S.: Fairswap: How to fairly exchange digital goods. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018
27. Eckey, L., Faust, S., Schlosser, B.: Optiswap: Fast optimistic fair exchange. In: Sun, H., Shieh, S., Gu, G., Ateniese, G. (eds.) ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security (2020)
28. Fuchsbauer, G.: WI is not enough: Zero-knowledge contingent (service) payments revisited. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019
29. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02
30. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference (2010)
31. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: Advances in Cryptology - EUROCRYPT 2013 (2013)
32. Ghosh, S., Nilges, T.: An algebraic approach to maliciously secure private set intersection. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT. Lecture Notes in Computer Science, Springer (2019)
33. Goldreich, O.: The Foundations of Cryptography - Volume 1: Basic Techniques. Cambridge University Press (2001), http://www.wisdom.weizmann.ac.il/%7Eoded/foc-vol1.html
34. GoogleOne: Upgrade to a plan that works for you (2021), https://one.google.com/about/plans?hl=en_GB
35. Halevi, S., Harnik, D., Pinkas, B., Shulman-Peleg, A.: Proofs of ownership in remote storage systems. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011 (2011)
36. Haselton, T.: Slack service goes down for more than three hours (2021), https://www.cnbc.com/2021/01/04/slack-outage-on-first-monday-of-2021.html
37. 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)
38. Ishai, Y., Ostrovsky, R., Zikas, V.: Secure multi-party computation with identifiable abort. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference (2014)
39. Juels, A., Jr., B.S.K.: Pors: Proofs of retrievability for large files. IACR Cryptology ePrint Archive 2007, 243 (2007)
40. Kalodner, H.A., Goldfeder, S., Chen, X., Weinberg, S.M., Felten, E.W.: Arbitrum: Scalable, private smart contracts. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security. USENIX Association (2018)
41. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)
42. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press (2014), https://www.crcpress.com/Introduction-to-Modern-Cryptography-Second-Edition/Katz-Lindell/p/book/9781466570269
43. 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'16

44. 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
45. Lai, J., Deng, R.H., Pang, H., Weng, J.: Verifiable computation on outsourced encrypted data. In: Kutylowski, M., Vaidya, J. (eds.) ESORICS 2014 (2014)
46. Liu, Z., Li, T., Li, P., Jia, C., Li, J.: Verifiable searchable encryption with aggregate keys for data sharing system. Future Gener. Comput. Syst. (2018)
47. Maxwell, G.: Zero knowledge contingent payment (2011)
48. Merkle, R.C.: Protocols for public key cryptosystems. In: Proceedings of the 1980 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 14-16, 1980. pp. 122–134. IEEE Computer Society (1980)
49. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference (1989)
50. Merryweather, L.: Three in five people have received a scam delivery text in the past year (2021), https://www.which.co.uk/news/2021/06/three-in-five-people-have-received-a-scam-delivery-text-in-the-last-year/
51. 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)
52. Miller, A., Juels, A., Shi, E., Parno, B., Katz, J.: Permacoin: Repurposing bitcoin work for data preservation. In: S&P'14
53. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2019)
54. Nguyen, K., Ambrona, M., Abe, M.: WI is almost enough: Contingent payment all over again. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020
55. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91
56. Project, G.: The gnu multiple precision arithmetic library (1991), https://gmplib.org/
57. Reid, F., Harrigan, M.: An analysis of anonymity in the bitcoin system. In: Privacy, Security, Risk and Trust (PASSAT) (2011)
58. Shacham, H., Waters, B.: Compact proofs of retrievability. In: ASIACRYPT. pp. 90–107 (2008)
59. Shen, S., Tzeng, W.: Delegable provable data possession for remote data in the clouds. In: ICICS 2011
60. Tramèr, F., Zhang, F., Lin, H., Hubaux, J., Juels, A., Shi, E.: Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017 (2017)
61. UK Finance: 2021 half year fraud update (2021), https://www.ukfinance.org.uk/system/files/Half-year-fraud-update-2021-
62. Wang, X., Zhao, L.: Verifiable single-server private information retrieval. In: Information and Communications Security - 20th International Conference, ICICS 2018 (2018)
63. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2014) (2014)
64. Zhang, L.F., Safavi-Naini, R.: Verifiable multi-server private information retrieval. In: Boureanu, I., Owesarski, P., Vaudenay, S. (eds.) Applied Cryptography and Network Security - 12th International Conference (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 [47] 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 $\mathtt{H}(k) = y$, to claim the amount of coin specified in the transaction. The contingent payment protocol in [47] 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, $y = \mathtt{H}(k)$. 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 $\mathtt{H}(k) = y$.

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 $\mathtt{H}(k) = y$. 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) [31], 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.* [17] 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 $\mathtt{v}(.)$ be the verification algorithm for a certain service and s be the service's proof, where if the proof is valid it holds that $\mathtt{v}(s) = 1$. The parties agree on two claw-free hash functions, e.g., $\mathtt{H}_1(.)$ and $\mathtt{H}_2(.)$. The seller picks a random value, r. Then, it computes either $y = \mathtt{H}_1(r)$ if it knows s such that $\mathtt{v}(s) = 1$, or $y = \mathtt{H}_2(r)$ otherwise. The seller also generates a witness indistinguishable proof of knowledge (WIPoK), π, using a compound sigma protocol to prove that it knows either the preimage of $y = \mathtt{H}_1(r)$ if it knows a valid s, i.e., $\mathtt{v}(s) = 1$, or the preimage of $y = \mathtt{H}_2(r)$. Note, due to the witness indistinguishability of π 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 π 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 $y = \mathtt{H}_1(r)$. 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) [58]. 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 [28] 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.* [54] 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 [28,54].

A.4 Using Ethereum Smart Contracts in Contingent Payment

Tramer *et al.* [60] 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.* [26] 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 [18]. 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.* [27] 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 [40]. 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 [26,27,60] 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 [3,25]. 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 [25] 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 [3,25] 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 Preliminaries

B.1 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 [41].

Definition 16. *Let $W : \{0,1\}^{\psi} \times \{0,1\}^{\eta} \rightarrow \{0,1\}^{\iota}$ 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, $\mu(.)$, such that:*

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

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

B.2 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 $\texttt{Com}(x, r) = \texttt{Com}_x$, that involves a secret value: $r \xleftarrow{\$} \{0,1\}^{\lambda}$. In the end of the commit phase, the commitment $\texttt{Com}_x$ is sent to the receiver. In the open phase, the sender sends the opening $\ddot{x} := (x, r)$ to the receiver who verifies its correctness: $\texttt{Ver}(\texttt{Com}_x, \ddot{x}) \stackrel{?}{=} 1$ 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 $\texttt{Com}_x$ is opened, and (b) *binding*: it is infeasible for an adversary (i.e., the sender) to open a commitment $\texttt{Com}_x$ to different values $\ddot{x}' := (x', r')$ than that was used in the commit phase, i.e., infeasible to find $\ddot{x}'$, *s.t.* $\texttt{Ver}(\texttt{Com}_x, \ddot{x}) = \texttt{Ver}(\texttt{Com}_x, \ddot{x}') = 1$, where $\ddot{x} \neq \ddot{x}'$. There exist efficient non-interactive commitment schemes both in (a) the standard model, e.g., Pedersen scheme [55], and (b) the random oracle model using the well-known hash-based scheme such that committing is : $\texttt{H}(x||r) = \texttt{Com}_x$ and $\texttt{Ver}(\texttt{Com}_x, \ddot{x})$ requires checking: $\texttt{H}(x||r) \stackrel{?}{=} \texttt{Com}_x$, where $\texttt{H} : \{0,1\}^* \rightarrow \{0,1\}^{\lambda}$ is a collision resistant hash function; i.e., the probability to find x and x' such that $\texttt{H}(x) = \texttt{H}(x')$ is negligible in the security parameter λ.

B.3 Publicly Verifiable Non-interactive Zero-knowledge Proof

In a non-interactive zero-knowledge proof (NIZK), a prover $\mathcal{P}$, given a witness w for some statement x in an NP language L, wants to convince a verifier $\mathcal{V}$ of the validity of $x \in L$. The main security property of the scheme is *Zero-knowledge*; meaning, a potentially malicious verifier cannot learn anything beyond the validity of the statement. The procedure is non-interactive, i.e., $\mathcal{P}$ generates a proof π and provides $\mathcal{V}$ with

π, who accepts (or rejects) verification. A NIZK is publicly verifiable when any party by obtaining π can verify the validity of $x \in L$. Publicly verifiable NIZKs have been constructed under trust assumptions such as the presence of a common reference string, or setup assumptions such as the existence of a random oracle which is used in this work. For a formal definition of NIZKs we refer the reader to [33].

B.4 Symmetric-key Encryption Scheme

A symmetric-key encryption scheme consists of three algorithms ($\texttt{SKE.keyGen}, \texttt{Enc}, \texttt{Dec}$), defined as follows. (1) $\texttt{SKE.keyGen}(1^\lambda) \to k$ is a probabilistic algorithm that outputs a symmetric key k. (2) $\texttt{Enc}(k, m) \to c$ takes as input k and a message m in some message space and outputs a ciphertext c. (3) $\texttt{Enc}(k, c) \to m$ takes as input k and a ciphertext c and outputs a message m.

The correctness requirement is that for all messages m in the message space, it holds that

$$\Pr\Big[\ \texttt{Dec}(k, \texttt{Enc}(k, m)) = m : \texttt{SKE.keyGen}(1^\lambda) \to k\Big] = 1.$$

The symmetric-key encryption scheme satisfies *indistinguishability against chosen-plaintext attacks (IND-CPA)*, if any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ has no more than $\frac{1}{2} + \texttt{negl}(\lambda)$ probability in winning the following game: the challenger generates a symmetric key $\texttt{SKE.keyGen}(1^\lambda) \to k$. The adversary $\mathcal{A}$ is given access to an encryption oracle $\texttt{Enc}(k, \cdot)$ and eventually sends to the challenger a pair of messages m_0, m_1 of equal length. In turn, the challenger chooses a random bit b and provides $\mathcal{A}$ with a ciphertext $\texttt{Enc}(k, m_b) \to c_b$. Upon receiving c_b, $\mathcal{A}$ continues to have access to $\texttt{Enc}(k, \cdot)$ and wins if its guess b' is equal to b.

B.5 Digital Signature Scheme

A digital signature is a scheme for verifying the authenticity of digital messages. It involves three algorithms, ($\texttt{Sig.keyGen}, \texttt{Sig.sign}$, $\texttt{Sig.ver}$), defined as follows. (1) $\texttt{Sig.keyGen}(1^\lambda) \to (sk, pk)$ is probabilistic algorithm run by a signer that outputs a key pair (sk, pk), consisting of secret key sk, and public key pk. (2) $\texttt{Sig.sign}(sk, pk, u) \to sig$ is an algorithm run by the signer. It takes as input key pair (sk, pk) and a message u. It outputs a signature sig. (3) $\texttt{Sig.ver}(pk, u, sig) \to h \in \{0, 1\}$ is an algorithm run by a verifier. It takes as input public key pk, message u, and signature sig. It checks the signature's validity. If the verification passes, then it outputs 1; otherwise, it outputs 0.

A digital signature scheme should meet two properties. (1) *Correctness:* for every input u it holds that:

$$\Pr\Big[\texttt{Sig.ver}(pk, u, \texttt{Sig.sign}(sk, pk, u)) = 1 : \texttt{Sig.keyGen}(1^\lambda) \to (sk, pk)\Big] = 1$$

(2) *Existential unforgeability under chosen message attacks (EUF-CMA)*: a probabilistic polynomial time PPT adversary that obtains pk and has access to a signing oracle for messages of its choice, cannot create a valid pair (u^*, sig^*) for a new message u^*, except with a negligible probability, σ. For a formal definition of digital signatures, we refer readers to [42].

B.6 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 ($\texttt{MT.genTree}, \texttt{MT.prove}, \texttt{MT.verify}$) as follows:

- The algorithm that constructs a Merkle tree, $\texttt{MT.genTree}$, is run by $\mathcal{V}$. It takes file blocks, $u := u_1, ..., u_m$, as input. Then, it groups the blocks in pairs. Next, a collision-resistant hash function, $\texttt{H}(.)$, 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.

C Further Discussion on Proof Status Leakage

As we already highlighted, the leakage of proof status (which reveals whether the server is suffering from hardware/software failure) might be problematic in certain circumstances. In Section 4.1, we have described a case where such leakage could benefit adversaries.

However, this is not the only case. An adversary may directly target business and/or individual clients by mounting social engineering attacks on them. It can exploit such leakage to increase the chance of success in its attack. A survey conducted by Kaspersky lab suggests that 33% of attacks that affect business clients of cloud servers are of type social engineering [44] which is a high rate. Moreover, social engineering attacks on individuals are still prevalent. In the UK, scammers have been impersonating the Post Office and sending "extremely convincing" text messages to the citizens and asking them to use the link provided in the message to schedule redelivery and pay for the redelivery using the given online form. The attackers would steal victims' payment details once they use the form and insert their card details. This message may seem relevant and convincing to a certain percentage of recipients as they are indeed waiting for parcel delivery. A survey conducted by a UK-based consumer protection organisation suggests that 61% of surveyed people had received such a text message [50].

The same scam can be simply adjusted to target users of the cloud servers which are suffering a failure. Given real-time evidence of servers' failure (that is also known to the clients), an adversary can provide more convincing evidence to their victims as a part of social engineering. This ultimately increases the adversary's chance of success. The adversary can target a large set of people a subset of which is likely to be the cloud's clients (akin to the above parcel delivery scam), or it can target specific cloud clients by using the techniques used to link the blockchain's addresses to certain parties [7,13,57].

D Further Discussion of the SAP

In this section, first we outline why the SAP satisfies all four security properties set out in Section 8.2 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.

D.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.

D.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 [16,29]. 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).

D.3 Off-chain Verification in the SAP

The SAP's verification algorithm can be executed *off-chain*. In particular, given statement $\ddot{x}$, anyone can read $(g_{\mathcal{C}}, g_{\mathcal{S}}, adr_{\mathcal{C}}, adr_{\mathcal{S}})$ from the SAP smart contract and locally run $\texttt{SAP.verify}(\ddot{x}, g_{\mathcal{C}}, g_{\mathcal{S}}, adr_{\mathcal{C}}, adr_{\mathcal{S}})$ to check the statement's correctness. This relieves the verifier from the transaction and smart contract's execution costs.

E RC-PoR-P's Correctness

In this section, we briefly discuss why the correctness of the RC-PoR-P protocol holds, w.r.t. Definition 11. Recall, correctness requires that $\mathcal{S}$ accepts an honest $\mathcal{C}$'s encoded data and query while honest $\mathcal{C}$ accepts $\mathcal{S}$'s valid service proof. Also, honest $\mathcal{C}$ gets back all its deposited coins minus the service payment, the honest $\mathcal{S}$ gets back all its deposited coins plus the service payment and the arbiter receives nothing. In short, this protocol's correctness holds dues to the correctness of PoR, symmetric key encryption, SAP, and smart contract. Below, we elaborate on that:

1. due to the correctness of smart contracts, any message sent by a party (i.e., $\mathcal{C}$ or $\mathcal{S}$) to a contract is kept intact by the contract.
2. due to the correctness of symmetric key encryption, the encrypted messages sent by a party to another one can be correctly decrypted by its counterparty who knows the correct key.
3. due to points 1 and 2 above, an encrypted message sent to a contract by a party can be correctly decrypted by its counterparty who knows the correct key.
4. due to the correctness of the underlying PoR scheme, proofs generated by honest $\mathcal{S}$ are always accepted by honest $\mathcal{C}$. Also, for the same reason, a query generated by honest $\mathcal{C}$ is always accepted by honest $\mathcal{S}$.
5. due to points 1-4, the counters remain 0, i.e., $y_{\mathcal{S}} = y_{\mathcal{C}} = y'_{\mathcal{S}} = y'_{\mathcal{C}} = 0$.
6. due to the correctness of SAP and point 1, a private statement's proof (e.g., $\ddot{x}_{cp} \in T_{cp}$) sent by an honest $\mathcal{C}$ or $\mathcal{S}$ will always be accepted by the smart contract.

Therefore, $\mathcal{C}$ receives $coin_{\mathcal{C}} = coin^*_{\mathcal{C}} - o \cdot z$ coins, $\mathcal{S}$ receives $coin_{\mathcal{S}} = coin^*_{\mathcal{S}} + o \cdot z$, and arbiter receives 0 coins.

F Proof of Theorem 3

This section contains the security analysis of the RC-PoR-P consrtuction presented in Section 8. First, we prove the security of the PoR scheme in Subsection 8.1 in the following lemma.

Lemma 5. *Let ϵ be non-negligible in the security paramenter λ. Then, the PoR scheme presented in Subsection 8.1 is ϵ-sound w.r.t. Definition 1, if the underlying 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., [35,39,52]. Alternatively, our protocol can be proven based on the security analysis of the PoR schemes that use MACs or BLS signatures, e.g., [58]. 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 ϵ 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 [58]. 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.

By applying Lemma 5, we prove the main theorem of Section 8.

THEOREM 3. *The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ described in Subsections 8.1 and 8.2 is secure (cf. Definition 15), if the underlying Merkle tree, pseudorandom function, commitment scheme, digital signature scheme are secure, and the underlying symmetric-key encryption scheme is IND-CPA secure.*

Proof. We show that the RC-PoR-P scheme meets all security properties defined in Section 7.1 by proving a series of claims. First, we recall that $coin_{\mathcal{P},j}$ denotes the coins that are credited to the party $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}, \mathcal{R}\}$ for the j-th verification and h_j is a value included in the decoded proof π_j that should match $F_{\mathsf{PoR}}(u^*, \hat{k}_j, pp)$. In addition, $y_{\mathcal{C},j} = 1$ (resp. $y_{\mathcal{S},j} = 1$) if $\mathcal{C}$ (resp. $\mathcal{S}$) misbehaved in the j-th billing cycle, and $y'_{\mathcal{C},j} = 1$ (resp. $y'_{\mathcal{S},j} = 1$) if $\mathcal{C}$ (resp. $\mathcal{S}$) has provided a complaint that does not allow $\mathcal{R}$ to identify a misbehaved party in the j-th verification .

Claim. The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ is secure against a malicious server (cf. Definition 12), if the SAP and signature scheme are secure, and the PoR scheme satisfies correctness and soundness.

PROOF OF CLAIM F. First, we consider event $\Big(F_{\mathsf{PoR}}(u^*, \hat{k}_j, pp) = h_j \ \wedge \Big((coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o) \vee (coin_{\mathcal{R},j} \neq l \ \wedge \ y'_{\mathcal{S},j} = 1)\Big)\Big)$ that captures the case where the server provides an accepting proof, but makes an honest client withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o$, or makes the arbiter withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{R},j} \neq l$, if it unnecessarily invokes the arbiter. Because the proof is valid, an honest client accepts it and does not raise a dispute. However, the server could make the client withdraw incorrect amount of coins, if it manages to

1. either convince the arbiter that the client has misbehaved, by making the arbiter output $y_{\mathcal{C},j} = 1$ through the dispute resolution phase, or
2. submit an accepting statement $\ddot{x}'_{cp}$ to SC which is other than what was agreed in the initiation phase, i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$, so it can change the payments' parameters, or
3. send a message on the client's behalf to unnecessarily invoke the arbiter.

In any of the above cases, the server cannot falsely accuse the client of misbehaviour; due to the binding property of the SAP commitment scheme, 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 SAP initiation phase. In particular, it cannot persuade the arbiter to accept $\ddot{x}'_{qp}$, where $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$, except with $\mathsf{negl}(\lambda)$ probability. 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., $\ddot{x}_{qp}$, then due to the correctness of the PoR and query-checking process (specified in step 5b), no one is identified as a misbehaving party in the dispute resolution phase, i.e., so we would have $I_j = \perp$. Therefore, due to the binding property of SAP and correctness of PoR and query-checking process, the following holds $y_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$.

Moreover, due to the binding property of the SAP commitment scheme, the server cannot change the payment parameters by convincing the contract to accept any statement $\ddot{x}'_{cp}$ other than what was agreed initially between the client and server, except with $\mathsf{negl}(\lambda)$ probability. Also, due to the signature's security, the adversary cannot send a message on behalf of the client to unnecessarily invoke the arbiter and make it output $y'_{\mathcal{C},j} = 1$, except with $\mathsf{negl}(\lambda)$ probability; so with high probability, it holds that $y'_{\mathcal{C},j} = 0$. Recall, in RC-PoR-P or RC-S-P protocol, according to Equation (1), the amount of coins that should be credited to the client for the j-th verification is $coin_{\mathcal{C},j} = \frac{coin^*_{\mathcal{C}}}{z} - o \cdot (1 - y_{\mathcal{S},j}) - l \cdot (y_{\mathcal{C},j} + y'_{\mathcal{C},j})$. Since it holds that $y_{\mathcal{C},j} = y_{\mathcal{S},j} = y'_{\mathcal{C},j} = 0$, the client is credited $\frac{coin^*_{\mathcal{C}}}{z} - o$ coins for the j-th verification, with high probability.

As stated above, if the adversary invokes the arbiter, the arbiter with a high probability outputs $I_j = \perp$ that yields $y'_{\mathcal{S},j} = 1$. In RC-PoR-P or RC-S-P protocol, according to Equation 2, the amount of coins the arbiter should be credited for j-th verification is $coin_{\mathcal{R},j} = l \cdot (y_{\mathcal{S},j} + y_{\mathcal{C},j} + y'_{\mathcal{S},j} + y'_{\mathcal{C},j})$. As shown above $y_{\mathcal{C},j} = y_{\mathcal{S},j} = y'_{\mathcal{C},j} = 0$ and $y'_{\mathcal{S},j} = 1$, 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 this amount, it has to send to SC (in the coin transfer phase) an accepting statement $\ddot{x}'_{cp}$ other than what was agreed in the initiation phase, i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$, so it can change the payments' parameters. However, as stated above, it cannot succeed with a probability significantly greater than $\mathsf{negl}(\lambda)$.

We now study the event $\Big(\Big(F_{\mathsf{PoR}}(u^*, \hat{k}_j, pp) \neq h_j\Big) \wedge \Big(d_j = 1 \vee y_{\mathcal{S},j} = 0 \vee coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} \vee coin_{\mathcal{R},j} \neq l\Big)\Big)$ which captures the case where the server provides an invalid proof and 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 coins, i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z}$ or $coin_{\mathcal{R},j} \neq l$ respectively. Due to the security of the Merkle tree and soundness of PoR (Lemma 5), the probability that the adversary can convince an honest client to accept invalid proof is $\mathsf{negl}(\lambda)$ and the file is extractable within a polynomial number of interactions with a ϵ-admissible adversary. 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 $y_{\mathcal{S},j} = 0$. For the adversary to succeed, it has to convince the arbiter that the client has misbehaved, and output $y_{\mathcal{C},j} = 1$. In this case, according to RC-PoR-P protocol, the client's complaint (for the j-th verification) would not be processed by the arbiter. This allows $y_{\mathcal{S},j}$ to remain 0. However, as we argued in the study of the previous event, the probability that the adversary makes the arbiter recognise the client as misbehaving is $\mathsf{negl}(\lambda)$. So, with high probability $y_{\mathcal{S},j} = 1$ and $y_{\mathcal{C},j} = 0$, after the arbiter is invoked by the client or server. It also holds that $y'_{\mathcal{C},j} = y'_{\mathcal{S},j} = 0$, because the arbiter has already identified a misbehaving party; specifically, recall if a malicious server invokes the arbiter, the arbiter discards its complaint without carrying out any investigation/computation as a malicious party has already been identified, thus $y'_{\mathcal{S},j}$ remains 0. Moreover, due to SAP's security, the probability that the adversary succeeds in changing the payment parameters to make the client or arbiter withdraw an incorrect amount of coins is $\mathsf{negl}(\lambda)$ too. So, according to Equations (1) and 2 the client and arbiter are credited $\frac{coin^*_{\mathcal{C}}}{z}$ and l coins for the j-th verification respectively. Also, due to the security of SAP (i.e., after the parties agree on the statement, an honest party can almost always prove to the verifier that it has the agreement), the adversary cannot block an honest client's messages, "pay" and $\ddot{x}_{cp}$, to the contract in the coin transfer phase. ⊣

Claim. The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ is secure against a malicious client (cf. Definition 13), if SAP and signature scheme are secure and PoRID scheme supports correctness, inputs well-formedness, and detectable abort.

Proof of Claim F. We first consider event $\Big(\Big(M(u^*,k,pp)=\sigma \ \wedge \ Q(\text{aux},k,\ pp)=\boldsymbol{q}_j\Big) \ \wedge \ \Big((coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z}+o) \ \vee \ (coin_{\mathcal{R},j}\neq l \ \wedge \ y'_{\mathcal{C},j}=1)\Big)\Big)$. It captures the case where the client provides accepting metadata (i.e., a Merkle tree and its root) and query but makes the server withdraw incorrect amounts of coin, i.e., $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z}+o$, or makes the arbiter withdraw incorrect amounts of coin, i.e. $coin_{\mathcal{R},j}\neq l$, if it unnecessarily invokes the arbiter. Since the metadata and queries are valid and correctly structured, an honest server accepts them and does not raise a dispute, so $y_{\mathcal{C},j}=0$. However, the client could make the server withdraw an incorrect amount of coins if it manages to either persuade the arbiter to recognise the server as misbehaving, i.e., makes the arbiter output $y_{\mathcal{S},j}=1$, or submit to the contract an accepting statement $\ddot{x}'_{cp}$ other than what was agreed at the initiation phase, i.e., $\ddot{x}_{cp}$ or send a message on the client's behalf to unnecessarily invoke the arbiter. Nevertheless, it cannot falsely accuse the server of misbehaviour. Because, due to SAP's security, it cannot convince the arbiter to accept different decryption key and pads' detail, by providing a different accepting statement $\ddot{x}'_{qp}$ (where $\ddot{x}'_{qp}\neq\ddot{x}_{qp}$), than what was initially agreed with the server, 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 $y_{\mathcal{S},j}=0$.

Also, if the adversary provides a valid statement, i.e., $\ddot{x}_{qp}$, then due to the correctness of the PoR and query-checking process (explained in step 5b), we would have $I_j=\perp$. So, due to the security of SAP and correctness of the PoR and query-checking process the following holds $y_{\mathcal{C},j}=y_{\mathcal{S},j}=0$ with a high probability. Moreover, it holds that $y'_{\mathcal{S}}=0$ because the honest server never invokes the arbiter when the client's queries are well-structured and due to the signature scheme's security, the client cannot send a message on the server's behalf to unnecessarily invoke the arbiter. Note, due to SAP's security, the client cannot change the payment parameters by convincing the contract to accept any statement $\ddot{x}'_{cp}$ other than what was initially agreed between the client and server (i.e., $\ddot{x}'_{cp}\neq\ddot{x}_{cp}$) except with a negligible probability, $\mu(\lambda)$. Recall, according to Equation 3, in RC-PoR-P or RC-S-P protocol, the total coins the server should be credited for j-th verification is $coin_{\mathcal{S},j}=\frac{coin^*_{\mathcal{S}}}{z}+o\cdot(1-y_{\mathcal{S},j})-l\cdot(y_{\mathcal{S},j}+y'_{\mathcal{S},j})$. Therefore, given $y_{\mathcal{S},j}=y'_{\mathcal{S},j}=0$, the server is credited $\frac{coin^*_{\mathcal{S}}}{z}+o$ 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 $y'_{\mathcal{C},j}=1$. 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 security of SAP, the client cannot make the arbiter withdraw incorrect coin amounts by changing the payment parameters and persuading the contract to accept any statement $\ddot{x}'_{cp}$ other than what was agreed initially between the client and server, except with a negligible probability $\mu(\lambda)$. We now turn our attention to $\Big(M(u^*,k,pp)\neq\sigma \ \wedge \ a=1\Big)$ which captures the case where the server accepts ill-formed metadata. But, due to the security of the Merkle tree scheme, the probability 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.

Next, we move on to $\Big((Q(\text{aux},k,pp)\neq\boldsymbol{q}_j)\wedge(b_j=1\vee y_{\mathcal{C},j}=0\vee coin_{\mathcal{S},j}\neq\frac{coin^*_{\mathcal{S}}}{z}+o\vee coin_{\mathcal{R},j}\neq l)\Big)$ 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 coins, i.e., $coin_{\mathcal{S},j}\neq\frac{coin^*_{\mathcal{S}}}{z}+o$ or $coin_{\mathcal{R},j}\neq l$ respectively. Due to the correctness of the query-checking process, the probability that the server outputs $b_j=1$ is 0. Note, when the honest server rejects the query and raises a dispute, the arbiter checks the query and sets $y_{\mathcal{C},j}=1$. After that, due to RC-PoR-P design, the client cannot make the arbiter set $y_{\mathcal{C},j}=0$ (unless it manages to modify the blockchain's content later on, 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 $y_{\mathcal{S},j}=0$. Furthermore, since the arbiter has identified a misbehaving party, the following holds $y'_{\mathcal{C},j}=y'_{\mathcal{S},j}=0$. The adversary may still try to make them withdraw incorrect amounts of coin. 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 SAP's security, its success probability is $\mu(\lambda)$. Hence, according to Equations 3 and 2 the server and arbiter are credited $\frac{coin^*_{\mathcal{S}}}{z}+o$ and l coins respectively for the j-th verification, with a high

probability. Furthermore, due to SAP's security, the adversary cannot block an honest server's messages, "pay" and $\ddot{x}_{cp}$, to the contract in the coin transfer phase. ⊣

Claim. The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ preserves privacy (cf. Definition 14), if SAP is secure and the symmetric-key encryption scheme is IND-CPA secure.

Proof of Claim F. Briefly, due to SAP's privacy property, given commitments g_{qp} and g_{cp} (stored in the blockchain as a result of running SAP) the adversary learns no information about the committed values (e.g. o, l, pad_{π} and $\bar{k}$), except with negligible probability $\mu(\lambda)$. Moreover, given price list pl, and the parties' encoded coins $coin^*_{\mathcal{C}}$ and $coin^*_{\mathcal{S}}$, the adversary learns nothing about the actual price agreed between the server and client, i.e., (o, l), for each verification, due to Lemma 3. Also, since each proof $\boldsymbol{\pi}^*_j$ is encrypted and then padded, given $\boldsymbol{\pi}^*_j$ the adversary cannot tell whether $\boldsymbol{\pi}^*_j$ is associated with u_0 or with u_1 (i.e., where u_0 and u_1 are the adversary's choice of files), with probability significantly greater than $\frac{1}{2} + \mu(\lambda)$. As each $\hat{k}^*_j$ is an output of semantically secure symmetric-key encryption and its size is fixed, it leaks noting to the adversary. The value of a is also independent of u_0 or u_1, and only depends on whether the metadata is well-formed, so it leaks nothing about the choice of input file u_β and $\beta \in \{0, 1\}$. 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.

Also, in the experiment, an invalid query-proof pair is computed with probability Pr_0 and a valid query-proof pair is generated with probability Pr_1. We know each encoded query-proof pair $c^*_j \in \boldsymbol{c}^*$ has a fixed size and contains random elements of U. It is also assumed that for each j-th verification, an encoded query-proof is always provided to the contract. So, each encoded pair leaks nothing, not even the query's status to the adversary; which means given only a vector of c^*_j , the adversary can learn a query-proof's status with a probability at most $Pr' + \mu(\lambda)$, where $Pr' = Max(Pr_0, Pr_1)$. Furthermore, each padded encrypted proof leak no information and always contain a fixed number of elements. Thus, for the adversary to the tell the status of proof for each j-th verification it has to learn information from $\hat{k}^*_j, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}$, $\boldsymbol{\pi}^*, pl$, and a but its success probability is at most $\mu(\lambda)$ or correctly guess a query's status but it has at most Pr' probability of success; this means it cannot tell a proof's status with a probability significantly greater than $Pr' + \mu(\lambda)$. ⊣

The security of the construction follows from Claims F, F, and F.

G RC-PoR-P Without Arbiter's Involvement

In the RC-PoR protocol, due to the efficiency of the arbiter-side algorithm, i.e., `RCSPoR.resolve`$(\cdot)$, we can delegate the arbiter's role to the smart contract, SC. In this case, the involvement of the third-party arbiter is not needed anymore. However, to have the new variant of RC-PoR-P, some adjustments need to be applied to the original RC-PoR-P's protocol and definition, primarily from two perspectives. First, the way a party pays to resolve a dispute would change, that ultimately affects the amount of coins each party receives in the coin distribution phase. Recall, in the RC-PoR-P and RC-S-P (presented in sections 8.2 and 7.3 respectively) the party which raises a dispute does not pay the arbiter when it sends to it the dispute query. Instead, loosely speaking, the arbiter in the coin distribution phase is paid 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 amounts of coin it sent to the contract to resolve the dispute. Therefore, the amount of coins each party receives in the coin distribution phase would change, compare to the original RC-PoR-P protocol. 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. In Appendix G.1, we provide a generic definition for RC-S-P for the case where the arbiter's role can be played by a smart contract. The generic definition also captures the new variant of RC-PoR-P. Moreover, in Appendix G.2, we elaborate on how the new variant of RC-PoR-P can be constructed and we prove its security.

G.1 Definition of RC-S-P Without Arbiter's Involvement

There are cases, in RC-S-P schemes, where the third party arbiter's role can be efficiently delegated to a smart contract. In this variant of the RC-S-P scheme, denoted by RC-$\overline{\text{S}}$-P, the arbiter's involvement is not needed anymore. The primary difference between RC-S-P and RC-$\overline{\text{S}}$-P is the way a party pays to resolve a dispute. In particular, in RC-S-P, the party which raises a dispute does not pay the arbiter when it sends to it a dispute query. Instead, loosely speaking, the arbiter at coin distribution is paid by a misbehaving party. Whereas, in RC-$\overline{\text{S}}$-P, the party which raises a dispute and sends a 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. In this section, we show how the RC-S-P definition (presented in Section 7.1) can be adjusted to capture RC-$\overline{\text{S}}$-P. In the following, we highlight the main changes that should be applied to the RC-S-P definition.

- In Definition 10:
  - Three parties are involved; namely, client, server, and smart contract (so an arbiter is not involved anymore).
  - Vectors $\boldsymbol{y}'_{\mathcal{C}}$ and $\boldsymbol{y}'_{\mathcal{S}}$ are not needed anymore. Because a misbehaving party, which unnecessarily invokes the contract, pays the contract ahead of time. Therefore, there is no need to keep track of unnecessary contract's invocation.
  - $\texttt{RCSP.resolve}(\cdot)$ is run by a smart contract.
  - $\texttt{RCSP.pay}(\cdot)$ outputs $(\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}})$, so $\boldsymbol{coin}_{\mathcal{R}}$ is excluded from the output, as a third party arbiter plays no role anymore.
- In Definition 11: only the above changes are applied to it.
- In Definition 12: the above changes are applied to the algorithms' syntax in the experiment. Moreover, the events are slightly modified, i.e. the amount of coins each party receives. For the sake of clarity and completeness, we state the entire modified definition below.

  Definition 17 (RC-$\overline{\text{S}}$-P Security Against Malicious Server). *An RC-$\overline{\text{S}}$-P scheme with functions F, M, E, D, Q is secure against a malicious server for an auxiliary information aux, if for any z polynomial in λ, any price list pl, every j (where $1 \leq j \leq z$), and any PPT adversary $\mathcal{A}$, the following probability is* $\mathsf{negl}(\lambda)$*:*

$$\Pr\left[\begin{array}{l}
\texttt{RCSP.keyGen}(1^\lambda,) \rightarrow \boldsymbol{k} \\
\mathcal{A}(1^\lambda, pk, F, M, E, D, Q) \rightarrow u \\
\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, z, pl) \rightarrow (u^*, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin^*_{\mathcal{C}}) \\
\mathcal{A}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \rightarrow (coin^*_{\mathcal{S}}, a) \\
\texttt{RCSP.genQuery}(1^\lambda, aux, k, T_{qp}) \rightarrow c^*_j \\
\mathcal{A}(c^*_j, \sigma, u^*, a) \rightarrow (b_j, m_{\mathcal{S},j}, h^*_j, \delta^*_j) \\
\texttt{RCSP.verify}(\pi^*_j, c^*_j, k, T_{qp}) \rightarrow (d_j, m_{\mathcal{C},j}) \\
\texttt{RCSP.resolve}(m_{\mathcal{C}}, m_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp}) \rightarrow \boldsymbol{y} \\
\texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}}) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}) \\
\hline
\Big(F(u^*, \boldsymbol{q}_j, pp) = h_j \ \wedge (coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o)\Big) \ \vee \\
\Big(F(u^*, \boldsymbol{q}_j, pp) \neq h_j \ \wedge \\
(d_j = 1 \vee y_{\mathcal{S},j} = 0 \ \vee \ coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} + l)\Big)
\end{array}\right]$$

  where $\boldsymbol{q}_j \in D(c^*_j, t_{qp})$, $\pi^*_j := (h^*_j, \delta^*_j), h_j = D(h^*_j, T_{qp})$, $\sigma \in e$, $m_{\mathcal{C},j} \in m_{\mathcal{C}}, m_{\mathcal{S},j} \in m_{\mathcal{S}}$, $y_{\mathcal{S},j} \in \boldsymbol{y}_{\mathcal{S}} \in \boldsymbol{y}$, and $pp \in T_{qp}$.

- In Definition 13: similar to the previous point, only the algorithms' syntax (in the experiment) and the amount of coins each party receives changes. Below, we state the entire modified definition.

  Definition 18 (RC-$\overline{\text{S}}$-P Security Against Malicious Client). *An RC-$\overline{\text{S}}$-P scheme wih functions F, M, E, D, Q is secure against a malicious client for an auxiliary information aux, if for any z polynomial in λ, every j (where $1 \leq j \leq z$), and any PPT adversary $\mathcal{A}$, the following probability is* $\mathsf{negl}(\lambda)$*:*

$$\Pr\left[\begin{array}{l}\mathcal{A}(1^\lambda, F, M, E, Q, D) \rightarrow (u^*, z, \boldsymbol{k}, e, T, pl, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, enc, aux, \\ \boldsymbol{y}, enc, pk) \\ \texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \rightarrow (coin^*_{\mathcal{S}}, a) \\ \mathcal{A}(coin^*_{\mathcal{S}}, a, 1^\lambda, aux, k, T_{qp}) \rightarrow c^*_j \\ \texttt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp}) \rightarrow (b_j, m_{\mathcal{S},j}, \pi^*_j) \\ \mathcal{A}(\pi^*_j, \boldsymbol{q}_j, k, T_{qp}) \rightarrow (d_j, m_{\mathcal{S},j}) \\ \texttt{RCSP.resolve}(m_{\mathcal{C}}, m_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp}) \rightarrow \boldsymbol{y} \\ \texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}}) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}) \\ \hline \Big((M(u^*, k, pp) = \sigma \wedge Q(aux, k, pp) = \boldsymbol{q}_j) \wedge \\ (coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o)\Big) \vee \\ \Big(M(u^*, k, pp) \neq \sigma \wedge a = 1\Big) \vee \\ \Big(Q(aux, k, pp) \neq \boldsymbol{q}_j \wedge \\ (b_j = 1 \ \vee \ y_{\mathcal{C},j} = 0 \ \vee coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o + l)\Big)\end{array}\right]$$

where $\boldsymbol{q}_j \in D(c^*_j, t_{qp})$, $D \in enc$, $\sigma \in e$, $y_{\mathcal{C},j} \in \boldsymbol{y}_{\mathcal{C}} \in \boldsymbol{y}$, *and* $pp \in T_{qp}$.

Note that Definition 14 remains almost the same with a minor change, that is vectors $(\boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{S}})$ are excluded from the related algorithms input/output.

Definition 19. *An RC-$\overline{S}$-P scheme is secure if it satisfies security against malicious server, security against malicious client, and preserves privacy (cf. . Definitions 17,18, and 14).*

G.2 Protocol For RC-PoR-P Without Arbiter's Involvement

In this section, we elaborate on how the original recurring contingent PoR payment (RC-PoR-P) protocol, presented in Section 8.2, can be adjusted such that the third party arbiter's role, i.e., resolving disputes, is totally delegated to the smart contract, SC. The new variant is denoted by RC-$\overline{\text{PoR}}$-P. Briefly, Phases 1-6 remain unchanged, with an exception. Namely, in step 2d, only two counters $y_{\mathcal{C}}$ and $y_{\mathcal{S}}$ are created, instead of four counters; accordingly, in the same step, vector $\boldsymbol{y}$ is now $\boldsymbol{y} : [y_{\mathcal{C}}, y_{\mathcal{S}}, adr_{\text{SC}}]$, so counters $y'_{\mathcal{C}}$ and $y'_{\mathcal{S}}$ are excluded from the 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 $y'_{\mathcal{C}}$ and $y'_{\mathcal{S}}$ anymore, SC takes the related steps (on the arbiter's behalf), and it reads its internal state any time it needs to read data already stored on the contract. Moreover, the main adjustment to phase 8 is that the amounts 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**. $\texttt{RCPoRP.resolve}(m_{\mathcal{C}}, m_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{q}^*, T_{qp})$
   The phase takes place only in case of dispute, i.e., when $\mathcal{C}$ rejects service proofs or $\mathcal{S}$ rejects the queries.
   (a) $\mathcal{S}$ sends $m_{\mathcal{S}}$ and $\ddot{x}_{qp}$ to SC, at time $\texttt{K}_1$, where $\texttt{K}_1 > \texttt{G}_{z,2} + \texttt{J}$
   (b) SC upon receiving $m_{\mathcal{S}}$ does the following a time $\texttt{K}_2$.
      i. Checks the validity of statement $\ddot{x}_{qp}$, by sending it to the SAP contract which returns 1 or 0. If the output is 0, then SC discards the server's complaint, $m_{\mathcal{S}}$, and does not take steps 7(c)ii and 7(c)iii. Otherwise, it proceeds to the next step.
      ii. Removes from $\boldsymbol{v}_{\mathcal{S}}$ any element that is duplicated or not in the range $[1, z]$. It also constructs an empty vector $\boldsymbol{v}$.
      iii. For any element $i \in \boldsymbol{v}_{\mathcal{S}}$:
         - Fetches the related encrypted query $\hat{k}^*_i \in \boldsymbol{q}^*$, and decrypts it, $\hat{k}_i = \texttt{Dec}(\bar{k}, \hat{k}^*_i)$.
         - Checks if the query is well-formed, by doing the same checks performed in step 5b (of the RC-PoR-P). If the query is rejected, then it increments $y_{\mathcal{C}}$ by 1 and appends i to $\boldsymbol{v}$.

      Let $\texttt{K}_3$ be the time SC finishes the above checks.
   (c) $\mathcal{C}$ sends $m_{\mathcal{C}}$ and $\ddot{x}_{qp}$ to SC, at time $\texttt{K}_4$.
   (d) SC upon receiving $m_{\mathcal{C}}$, does the following at time $\texttt{K}_5$.

i. Checks the validity of statement $\ddot{x}_{qp}$, by sending $\ddot{x}_{qp}$ to the SAP contract which returns either 1 or 0. If the output is 0, then SC discards the client's complaint, $m_{\mathcal{C}}$, and does not take steps 7(e)ii-7(e)iii. Otherwise, it proceeds to the next step.

ii. Ensures each vector $\boldsymbol{m} \in \boldsymbol{m}_{\mathcal{C}}$ is well-formed. In particular, it verifies there exist no two vectors: $\boldsymbol{m}, \boldsymbol{m}' \in \boldsymbol{m}_{\mathcal{C}}$ such that $\boldsymbol{m}[0] = \boldsymbol{m}'[0]$. If such vectors exist, it deletes the redundant ones from $\boldsymbol{m}_{\mathcal{C}}$. This ensures no two claims refer to the same verification. Also, it removes any vector $\boldsymbol{m}$ from $\boldsymbol{m}_{\mathcal{C}}$ if $\boldsymbol{m}[0]$ is not in the range $[1, z]$ or if $\boldsymbol{m}[0] \in \boldsymbol{v}$. Note the latter check (i.e., $\boldsymbol{m}[0] \in \boldsymbol{v}$) ensures $\mathcal{C}$ cannot hold $\mathcal{S}$ accountable if $\mathcal{C}$ has generated an ill-formed query for the same verification.

iii. For every vector $\boldsymbol{m} \in \boldsymbol{m}_{\mathcal{C}}$:

A. retrieves a rejected proof's details by setting $j = \boldsymbol{m}[0]$ and $g = \boldsymbol{m}[1]$. Recall that g refers to the index of a rejected proof in the proof vector which was generated for j-th verification, i.e., $\boldsymbol{\pi}_j$.

B. fetches the related encrypted query $c_j^* \in \boldsymbol{c}^*$ from SC and decrypts it as $\hat{k}_j = \mathtt{Dec}(\bar{k}, c_j^*)$. It removes the pads only from g-th padded encrypted proof. Let $\boldsymbol{\pi}'_j[g]$ be the result. Next, it decrypts the encrypted proof, $\mathtt{Dec}(\bar{k}, \boldsymbol{\pi}'_j[g])) = \boldsymbol{\pi}_j[g]$.

C. identifies the misbehaving party as follows.

- verifies $\hat{k}_j$ by doing the same checks done in step 5b (of the RC-PoR-P). If the checks do not pass, it sets $I_j = \mathcal{C}$ and skips the next two steps; otherwise, it proceeds to the next step.
- derives the related challenged block's index from $\hat{k}_j$, by computing $q_g = \big(\mathtt{PRF}(\hat{k}_j, g) \bmod m\big) + 1$.
- verifies only g-th proof, by calling $\mathtt{PoR.verify}(\boldsymbol{\pi}_j[g], q_g, pp) \rightarrow \boldsymbol{d}'$. If $\boldsymbol{d}'[0] = 0$, then it sets $I_j = \mathcal{S}$. Otherwise, it outputs $I_j = \perp$.
- if $I_j = \mathcal{C}$, it increments $y_{\mathcal{C}}$ by 1. If $I_j = \mathcal{S}$, it increments $y_{\mathcal{S}}$ by 1.

Let $\mathtt{K}_6$ be the time that SC finishes all the above checks.

8. **Coin Transfer**. $\mathtt{RCPoRP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin_{\mathcal{C}}^*, coin_{\mathcal{S}}^*)$

(a) If SC receives "pay" message at time $\mathtt{T}_2$, where $a = 0$ or $coins_{\mathcal{S}}^* < p_{\mathcal{S}}$, then it sends $coin_{\mathcal{C}}^*$ coins to $\mathcal{C}$ and $coin_{\mathcal{S}}^*$ coins to $\mathcal{S}$. Otherwise (i.e., they reach an agreement), they take the following step.

(b) Either $\mathcal{C}$ or $\mathcal{S}$ sends "pay" message and statement $\ddot{x}_{cp} \in T_{cp}$ to SC at time $\mathtt{L} > \mathtt{K}_6$.

(c) SC checks the validity of the statement by sending it to the SAP contract that returns either 1 or 0. SC only proceeds to the next step if the output is 1.

(d) SC distributes the coins to the parties as follows:

- $coin_{\mathcal{C}}^* - o \cdot (z - y_{\mathcal{S}}) + l \cdot (y_{\mathcal{S}} - y_{\mathcal{C}})$ coins to $\mathcal{C}$.
- $coin_{\mathcal{S}}^* + o \cdot (z - y_{\mathcal{S}}) + l \cdot (y_{\mathcal{C}} - y_{\mathcal{S}})$ coins to $\mathcal{S}$.

Theorem 4. *The RC-$\overline{PoR}$-P protocol is secure, w.r.t. Definition 19, if PoRID, SAP, and blockchain are secure and the encryption scheme is semantically secure.*

G.3 Proof of the RC-PoR-P Without Arbiter

To prove Theorem 4, we show that RC-$\overline{\text{PoR}}$-P meets all security properties defined in Appendix G.1. We start by proving that RC-$\overline{\text{PoR}}$-P meets security against a malicious server. The proof to some extent is simpler to that of RC-PoR-P against a malicious server (i.e., proof of Claim F) as it does not involve any third party arbiter.

Lemma 6. *If SAP and blockchain are secure and PoRID scheme supports correctness, soundness, and detectable abort, then RC-$\overline{PoR}$-P is secure against malicious server, w.r.t. Definition 17.*

Proof. First, we consider event

$$\Big(F(u^*, \boldsymbol{q}_j, pp) = h_j \ \wedge (coin_{\mathcal{C},j} \neq \frac{coin_{\mathcal{C}}^*}{z} - o)\Big)$$

that captures the case where the server provides an accepting proof, i.e. PoR, but makes an honest client withdraw incorrect amounts of coin, i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o$. Note, in RC-$\overline{\text{PoR}}$-P protocol, the total coins the client should receive after z verifications is $coin^*_{\mathcal{C}} - o \cdot (z - y_{\mathcal{S}}) + l \cdot (y_{\mathcal{S}} - y_{\mathcal{C}})$. Since we focus on j-th verification, the amounts of coin that should be credited to the client for the j-th verification is

$$coin_{\mathcal{C},j} = \frac{coin^*_{\mathcal{C}}}{z} - o \cdot (1 - y_{\mathcal{S},j}) + l \cdot (y_{\mathcal{S},j} - y_{\mathcal{C},j}) \tag{4}$$

As the proof is valid, an honest client accepts it and does not raise any dispute. But, the server would be able to make the client withdraw incorrect amounts of coin, if it manages to either convince the contract that the client has misbehaved, by making the contract output $y_{\mathcal{C},j} = 1$ through the dispute resolution phase, or submit to the contract, in the coin transfer phase, an accepting statement $\ddot{x}'_{cp}$ other than what was agreed in the initiation phase, i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$, so it can change the payments' parameters, e.g., l or o. Nevertheless, it cannot falsely accuse the client of misbehaviour. As, due to the security of SAP, it cannot convince the contract to accept different query's parameters other than what was agreed with the client in the initiation phase. In particular, it cannot persuade the contract to accept $\ddot{x}'_{qp}$ such that $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$, except with a negligible probability, $\mu(\lambda)$. Furthermore, if the adversary provides a valid statement then, then due to the correctness of the PoR and query-checking process, values $y_{\mathcal{C}}$ and $y_{\mathcal{S}}$ are not incremented by 1 in the j-th verification, i.e., $y_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$. Also, due to the security of SAP, the server cannot change the payment parameters by persuading the contract to accept any statement $\ddot{x}'_{cp}$ other than what was agreed initially between the client and server, except with a negligible probability $\mu(\lambda)$. Therefore, according to Equation 4, the client is credited $\frac{coin^*_{\mathcal{C}}}{z} - o$ coins for the j-th verification, with a high probability. We now move on to event

$$\Big(F(u^*, \boldsymbol{q}_j, pp) \neq h_j \ \wedge (d_j = 1 \vee y_{\mathcal{S},j} = 0 \ \vee \ coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} + l)\Big)$$

It captures the case where the server provides an invalid proof however either persuades the client to accept the proof, or persuades the contract to set $y_{\mathcal{S},j} = 0$ or makes the client withdraw incorrect amounts of coin, i.e. $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} + l$. Nevertheless, due to the soundness of PoR, the probability that a corrupt server can convince an honest client to accept invalid proof, i.e. outputs $d_j = 1$, is negligible, $\mu(\lambda)$. So, the client detects it with a high probability and raises a dispute. Also, the server may try to make the contract keep $y_{\mathcal{S},j} = 0$. For $y_{\mathcal{S},j} = 0$ to happen, it has to make the contract recognise the client as the misbehaving party, i.e., makes the contract output $y_{\mathcal{C},j} = 1$. In this case, the client's complaint would not be processed by the contract; therefore, $y_{\mathcal{S},j}$ remains 0. Nevertheless, as we discussed above, the probability that the adversary makes the contract recognise the client as misbehaving is negligible, $\mu(\lambda)$. Therefore, with a high probability $y_{\mathcal{S},j} = 1$ and $y_{\mathcal{C},j} = 0$, after the contract is invoked by the client or server. The adversary may try to make the client withdraw incorrect amounts of coin, e.g., in the case where it does not succeed in convincing the client or contract. To do so, in the coin transfer phase, it has to send a different accepting statement than what was initially agreed with the client. But, it would succeed only with a negligible probability, $\mu(\lambda)$, due to the security of SAP. So, according to Equation 4, the client is credited $\frac{coin^*_{\mathcal{C}}}{z} + l$ coins for the j-th verification, with a high probability.

Next, we prove that RC-$\overline{\text{PoR}}$-P satisfies security against a malicious client. The proof is also slightly simpler than that of RC-PoR-P against a malicious client (i.e., proof of Claim F) as it does not involve any third party arbiter.

Lemma 7. *If SAP and blockchain are secure and PoRID scheme supports correctness, inputs well-formedness, and detectable abort, then RC-$\overline{PoR}$-P is secure against malicious client, w.r.t. Definition 18.*

Proof. First, we consider event

$$\Big((M(u^*, k, pp) = \sigma \ \wedge \ Q(\text{aux}, k, pp) = \boldsymbol{q}_j) \ \wedge \ (coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o)\Big)$$

It captures the case where the client provides accepting metadata and query but makes the server withdraw an incorrect amounts of coin, i.e. $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o$. According to RC-$\overline{\text{PoR}}$-P protocol, the total coins the server should receive after z verifications is $coin^*_{\mathcal{S}} + o \cdot (z - y_{\mathcal{S}}) + l \cdot (y_{\mathcal{C}} - y_{\mathcal{S}})$. As we focus on j-th verification, the amount of coins that should be credited to the server for the j-th verification is

$$coin_{\mathcal{S},j} = \frac{coin^*_{\mathcal{S}}}{z} + o \cdot (1 - y_{\mathcal{S},j}) + l \cdot (y_{\mathcal{C},j} - y_{\mathcal{S},j}) \tag{5}$$

Since the metadata and query are valid, an honest server accepts them and does not raise any dispute, so we have $y_{\mathcal{C},j} = 0$. The client however could make the server withdraw incorrect amounts of coin, if it manages to either convince the contract, in the dispute resolution phase, that the server has misbehaved, i.e., makes the contract output $y_{\mathcal{S},j} = 1$, or submit to the contract an accepting statement $\ddot{x}'_{cp}$ other than what was agreed at the initiation phase, i.e. $\ddot{x}_{cp}$, in the coin transfer phase. But, it cannot falsely accuse the server of misbehaviour, because due to the security of SAP, it cannot convince the contract to accept different decryption key and pads' detail, by providing a different accepting statement $\ddot{x}'_{qp}$ (where $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$), than what was initially agreed with the server, except with a negligible probability, $\mu(\lambda)$. So, with a high probability $y_{\mathcal{S},j} = 0$. On the other hand, if the adversary provides a valid statement, i.e. $\ddot{x}_{qp}$, then due to the correctness of the PoR and query-checking process, we would have $I_j = \perp$. Thus, due to the security of SAP and the correctness of the PoR and query-checking process, we would have $y_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$ with a high probability. Also, due to the security of SAP, the client cannot change the payment parameters by convincing the contract to accept any accepting statement $\ddot{x}'_{cp}$ other than what was initially agreed between the client and server (i.e. $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$), except with a negligible probability, $\mu(\lambda)$. That means, according to Equation 5, the server is credited $\frac{coin^*_{\mathcal{S}}}{z} + o$ coins for that verification, with a high probability. We now move on to

$$\Big(M(u^*, k, pp) \neq \sigma \ \wedge \ a = 1\Big)$$

It captures the case where the server accepts ill-formed metadata. But, due to the security of the Merkle tree scheme, the probability of the event happening is negligible, $\mu(\lambda)$. So, with a high probability $a = 0$; in this case, the server does not raise any dispute, instead it avoids serving the client. Next, we turn our attention to

$$\Big(Q(\text{aux}, k, pp) \neq \boldsymbol{q}_j) \ \wedge \ (b_j = 1 \ \vee \ y_{\mathcal{C},j} = 0 \ \vee \ coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o + l)\Big)$$

It considers the case where the client provides an invalid query, but either convinces the server or contract to accept it, or makes the server withdraw incorrect amounts of coin, i.e. $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o + l$. Due to the correctness of the query-checking process, the probability that the server outputs $b_j = 1$ is 0. When the honest server rejects the query and raises a dispute, the contract checks the server's query and sets $y_{\mathcal{C},j} = 1$. After that, due to the security of blockchain the client cannot make the contract to set $y_{\mathcal{C},j} = 0$ except with probability $\mu(\lambda)$. Also, as discussed above, the client cannot make the contract recognise the honest server as a misbehaving party with a probability significantly greater than $\mu(\lambda)$. That means with a high probability $y_{\mathcal{S},j} = 0$. The adversary may still try to make the server withdraw incorrect amounts of coin (e.g., if the adversary does not succeed in convincing the server). To this end, at the coin transfer phase, it has to convince the contract to accept a different statement than what was initially agreed with the server. However, due to the security of SAP, its success probability is negligible, $\mu(\lambda)$. Hence, according to Equation 5, the server is credited $\frac{coin^*_{\mathcal{S}}}{z} + o + l$ coins for the j-th verification.

In the following, we provide a lemma for RC-$\overline{\text{PoR}}$-P's privacy. For the lemma's proof, we refer readers to the proof of Claim F.

Lemma 8. *If SAP is secure and the encryption scheme is semantically secure, then RC-$\overline{\text{PoR}}$-P preserves privacy, w.r.t. Definition 14.*

H A Table Summarizing RC-PoR-P Assymptotic Costs

Table 4 summarizes the RC-PoR-P's asymptotic costs of z verifications, breakdown by parties. In the table, ϕ is the number of challenged blocks, z' is the maximum number of complaints the client and server send to the arbiter, m is the number of blocks in a file, $||u^*||$ is the file bit-size, and $||\boldsymbol{\pi}^*||$ is the number of elements in the padded encrypted proof vector.

Table 4: RC-PoR-P asymptotic complexity, of z verifications, breakdown by parties. In the table, ϕ is the number of challenged blocks, z' is the maximum number of complaints the client and server send to the arbiter, m is the number of blocks in a file, $||u^*||$ is the file bit-size, and $||\boldsymbol{\pi}^*||$ is the number of elements in the padded encrypted proof vector.

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