Recurring Contingent Service Payment

Aydin Abadi* and Thomas Zacharias**

University of Gloucestershire, University of Edinburgh

1 Related Work

Fair exchange is an interesting problem in which two mutually distrusted parties want to swap digital items such that neither is able to cheat the other. It captures various real-world scenarios; for instance, when two parties want to exchange digital items or when a seller wants to sell a piece of secret digital information, s, in exchange for fixed amounts of digital coin. Solutions to the problem are usually certain cryptographic protocols and have been studied for decades. It has been shown that fairness is unachievable without the aid of a trusted third-party [4]. By the advent of decentralised cryptocurrencies and in general blockchain technology, it seemed fair exchange protocols can be designed in a trustless manner. 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. So, the trusted third-party's involvement can be eliminated. This ultimately results in a stronger security guarantee, as there would be no need to trust a single entity, anymore. Ethereum is the most predominant generic smart contract platform. Although the arbiter's role can be directly encoded in 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 Bitcoin's contract.

Zero-knowledge Contingent Payment. In [15] for the first time it is shown how to construct a fair exchange protocol, called "zero-knowledge contingent payment", that utilises Bitcoin's smart contract. The protocol allows the fair exchange of digital goods and payments over Bitcoin's network. Its primary 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 who can provide its preimage k, i.e $\mathtt{H}(k) = y$, to claim the coin amounts specified in the transaction. In the following, we elaborate on the contingent payment protocol proposed in [15]. The seller first picks a secret key, k, of 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 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 fixed coin amounts if the seller provides, to the blockchain, a value k such that $\mathtt{H}(k) = y$. Next, the server 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 on , after the advancement of "succinct non-interactive argument of knowledge" (zk-SNARK), that results in 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 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.

Zero-knowledge Contingent Service Payment. Later on, Campanelli *et al.* in [3] 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 (who generates 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, i.e. supporting contingent payment for *digital*

* aydinabadi@glos.ac.uk

** thomas.zacharias@ed.ac.uk

(verifiable) services. 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 who first ensures π is valid. Then, the buyer sends to the blockchain a hash-lock transaction that would send n coins to the party who 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 ensures that it has been served honestly by the server, as the server 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 a concrete instantiation of zkCSP they offer a scheme in which the (verifiable) service is "proof of retrievability" (PoR) [21]. In the proposed scheme, 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. The zkCSP is an interesting protocol; however, as we will show in Section 3, we observed that it suffers from serious issues. Specifically, (a) it allows a malicious client to waste the server's resources, (b) it leaks non-trivial information in *real-time* to the public, and (c) when the payment is recurring a malicious client can get a free ride from the server.

Known Zero-knowledge Contingent (Service) Payment's Flaw in the Literature. Later on, Fuchsbauer in [5] identifies a flaw in the above zkCSP. The author shows that the minimal efficient check that the seller performs in 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 CRS is inevitable to address the issue. Very recently, Nguyen *et al.* in [19] show that by relying on a slightly stronger notion of WI (i.e. trapdoor subversion witness indistinguishability), zkCSP can remain secure and would not be susceptible to the issues Fuchsbauer pointed out. Moreover, they propose an efficient scheme that relies on *interactive* ZK proof system based garbled circuits and oblivious transfer.

2 Preliminaries and Notations

2.1 Smart Contract

Cryptocurrencies, such as Bitcoin and Ethereum, in addition to offering a decentralised currency, support computations on transactions. In this setting, often a certain computation logic is encoded in a computer program, called *"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 (i.e. miners) 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"*, depending on the complexity of the contract running on it. Nonetheless, Ethereum smart contracts suffer from an important issue; namely, the *lack of privacy*, as it requires every contract's data to be public, which is a major impediment to the broad adoption of smart contracts when a certain level of privacy is desired. To address the issue, researchers/users may either (a) utilise existing decentralised frameworks which support privacy-preserving smart contracts, e.g. [14]. But, due to the use of generic and computationally expensive cryptographic tools, they impose a significant cost to their users. Or (b) design efficient tailored cryptographic protocols that preserve (contracts) data privacy, even though non-private smart contracts are used. We take the latter approach in this work.

2.2 Pseudorandom Function

Informally, a pseudorandom function (PRF) is a deterministic function that takes a key and an input; and outputs a value indistinguishable from that of a truly random function with the same input. A PRF is formally defined as follows [12].

Definition 1. *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. We let public parameters $\zeta : (\psi, \eta, \iota)$ be the description of PRF

2.3 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*: 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*: 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 [20], 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 resistance hash function, i.e. the probability to find x and x' such that $\texttt{H}(x) = \texttt{H}(x')$ is negligible, $\mu(\lambda)$.

2.4 Publicly Verifiable Non-interactive Zero-knowledge Proofs

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 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.

A (publicly verifiable) NIZK should satisfy the following properties:

- *Completeness*: $\mathcal{V}$ always accepts the proof generated by the honest prover $\mathcal{P}$.
- *Soundness*: if $x \notin L$, a potentially malicious prover $\mathcal{P}^*$ cannot convince $\mathcal{V}$ except with negligible probability.
- *Zero-knowledge*: a potentially malicious verifier $\mathcal{V}^*$ cannot learn anything beyond the validity of $x \in L$. Namely, there exists a simulator $\mathcal{S}$ that (without having the witness w) produces transcripts that are indistinguishable from the actual transcripts generated by $\mathcal{P}$.

For a formal definition of NIZKs we refer the reader to [8].

2.5 Digital Signatures

A digital signature scheme $\mathbf{DS}$ is a triple of algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ as follows:

- The key generation algorithm Gen takes as input the security parameter 1^{λ} and outputs a pair (sk, vk) where sk is the signing (or private) key and vk is the verification (or public) key.
- The signing algorithm Sign takes as input sk and a message M and outputs a signature σ on M.
- The verification algorithm Verify takes as input vk, M and σ and outputs 1 or 0 meaning valid or invalid, respectively.

The properties that $\mathbf{DS}$ should satisfy are the following:

– *Correctness*: for every message M, it holds that
$$\Pr[(sk, vk) \leftarrow \mathsf{Gen}(1^\lambda) : \mathsf{Verify}(vk, M, \mathsf{Sign}(sk, M)) = 1] = 1$$
– *Existential unforgeability under chosen message attacks (EUF-CMA)*: a PPT adversary that obtains vk and has access to a signing oracle for messages of its choice cannot create a valid pair (M^*, σ^*) for a new message M^* (that was never a query to the signing oracle), except with negligible probability in λ.

For a formal definition of digital signatures and their security we refer the reader to [13].

2.6 Merkle Tree

A Merkle tree scheme introduced by Merkle [16, 17] allows committing to data blocks, such that it is possible later to open the commitment and verify individual blocks of the file without the need to have the entire file to verify the opening. To construct a Merkle tree a file is split into blocks, then the blocks are grouped in pairs. Next, a collision-resistant hash function 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 remains. This results in a tree with the leaves corresponding to the blocks of the input file and the root corresponding to the last remaining hash value. Add the Merkle tree algorithms, e.g. build, prove, verify

2.7 Proofs of Retrievability (PoR)

In general, a PoR scheme considers the case where an honest client wants to store its file(s) on a potentially malicious server, i.e 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 (and security property) originally provided in [11, 21]. PoR scheme comprises five algorithms:

- $\texttt{PoR.keyGen}(1^\lambda) \rightarrow k := (sk, pk)$. A probabilistic algorithm, run by a client. It takes as input the security parameter 1^λ and outputs private and public verification keys $k := (sk, pk)$.

- $\texttt{PoR.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma, pp)$. A probabilistic algorithm, run by the client. It takes as input the security parameter 1^λ, a file u, and key k. It encodes u, denoted by u^* and generates a metadata, σ. The client outputs encoded file u^*, metadata σ, and (possibly file dependent) public parameters pp. It sends to the server u^*, σ, and pp.

- $\texttt{PoR.genQuery}(1^\lambda, k, pp) \rightarrow \boldsymbol{q}$. A probabilistic algorithm, run by the client. It takes as input the security parameter 1^λ, key k, and public parameters pp. It outputs a query vector $\boldsymbol{q}$, possibly picked uniformly at random. The query is given to the server.

- $\texttt{PoR.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \pi$. It is run by a server. 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, given to the client.

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

Informally, a PoR scheme has two main properties: *correctness* and *soundness*. Correctness requires that the verification algorithm accepts proofs generated by an honest verifier. Formally, it 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 [21], 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 $\mathcal{E}$ the execution of $\texttt{PoR.verify}(\pi, \boldsymbol{q}, k, pp)$ for any u used to query $\texttt{PoR.setup}(.)$. Accordingly, $\mathcal{E}$ informs $\mathcal{A}$ about the verification output. The adversary can send a polynomial number of queries to $\mathcal{E}$. Finally, $\mathcal{A}$ outputs metadata σ returned from a setup query and the description of a prover, $\mathcal{A}'$, for any file it has already chosen above.

It is said that a cheating prover, $\mathcal{A}'$, is *ϵ-admissible* if it convincingly answers ϵ fraction of verification challenges. Informally, a PoR scheme supports extractability, if there is an extractor algorithm $\texttt{Ext}(k, \sigma, \texttt{P}')$, that takes as input the key k, metadata σ, and the description $\texttt{P}'$ of the machine implementing the prover's role $\mathcal{A}'$ 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 $\texttt{P}'$.

Definition 2 (ϵ-soundness). *A PoR scheme is ϵ-sound if there exists an extraction algorithm $\texttt{Ext}(.)$ such that, for every adversary $\mathcal{A}$ who plays the experiment above, and outputs an ϵ-admissible cheating prover $\mathcal{A}'$ for a file u, the extraction algorithm recovers u from $\mathcal{A}'$, given honest parties private key, public parameters, metadata and the description of $\mathcal{A}'$, i.e. $\texttt{Ext}(k, pp, \sigma, \texttt{P}') \rightarrow u$, except with a negligible probability.*

In contrast to the PoR definition in [11, 21] where $\texttt{PoR.genQuery}(.)$ is implicit, in the above we have explicitly stated it, as it plays an important role in this paper. Also, there are PoR protocols, e.g. [18], that do not involve $\texttt{PoR.keyGen}(.)$. Instead a set of public parameters/keys (e.g. file size or a root of Merkle tree) are output by $\texttt{PoR.setup}(.)$. To make the PoR definition generic to capture both cases, we have explicitly included the public parameters pp in the algorithms definitions too.

2.8 Notations

In the formal definitions in this paper, we often use bar symbol, i.e. " | ", for the sake of readability and to separate events from an experiment. It should not be confused with conditional probability's symbol.

3 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 the limitations.

3.1 Limitations of zkCSP

As stated in Section 1, the main purpose of zero-knowledge contingent service payment (zkCSP) protocol [3] is to minimise the role of smart contract as much as possible, so (a) the verification's cost would be much lower, if a Turing-complete smart contract framework (e.g. Ethereum) is used, or (b) it can be implemented on a non-Turing-complete contract framework (e.g. Bitcoin).

Nevertheless, as we will show, zkCSP suffers from serious issues; namely, it allows a malicious client to waste the server resources and it leaks non-trivial information in real-time to the public. Also, when the payment is recurring (i.e. the server interacts with a client multiple times and/or the server interact with multiple clients), a malicious client can get a free ride from the server, in the sense that it can collect enough fresh information convincing him that the server is behaving honestly, without paying the server. In the following, we elaborate on the above issues and explain where the issues stem from.

- *Issue 1: Discrepancies between the Security Guarantees of Verifiable Service and Fair Exchange Schemes*. zkCSP combines a certain verifiable service scheme (i.e. PoR) secure against only a malicious server (where a client is assumed to be fully trusted) with a fair exchange protocol that takes into the consideration that either server or

client might be malicious. This mismatch allows the client to avoid paying the server in different ways. The client can simply avoid participating in the payment phase despite it has been using the server, e.g. using the server storage. Moreover, the client may engage in the payment protocol but falsely accuses the server for behaving maliciously, or makes the server generate invalid proofs. At a first glance it seems the client can only waste the server's resource without gaining anything. However, as we will show shortly, in the recurring payment (when the server deals with multiple clients) the client can collect convincing background information about an honest server. The information allows the client to conclude that it has been served honestly; even though, it does not pay the server and does not check the proof. Thus, it can get a free ride from the server.

- *Issue 2: Real-time Leakage of Verification Outputs and Deposit Amount*. zkCSP leaks in real-time non-trivial fresh information, about the server and clients, to the public. The leakage includes:

  - *proofs' status*: it is visible in the real-time to everyone that a proof has been accepted or reject, that reflects whether the server has successfully delivered the agreed-upon service or failed to do so. This can have serious immediate consequences for both the server and clients, e.g. lost revenue, negative press, stock value drop, or opening doors for attackers to exploit such incident. As an example, observing proof's verification outputs (when a server deals with multiple clients) allows a malicious client to immediately construct a comprehensive background knowledge on the server's current behaviour and status, e.g. the server has been acting honestly. Such auxiliary information can assist the client to more wisely exploit the above deposit issue (that can avoid sending the deposit); for instance, when the sever always acts honestly towards its clients, the client refuses to send the deposit and still has high confidence that the server has delivered the service. As another example, in the case of PoR, a malicious observer can simply find out that the service is suffering hardware/software failure and exploit such vulnerability to mount social engineering attacks on clients or penetrate to the system.

  - *deposit amount*: the amount of deposit placed on the contract, swiftly leaks non-trivial information about the client to the public. For instance, in the case of PoR, an observer can learn the size of data outsourced to the server, service type or level of data sensitivity. The situation gets even worse if the client updates its data (e.g. delete or append) or asks the server for additional service (e.g. S3 Glacier or S3 Glacier Deep Archive[1]), as an observer can learn such changes immediately by just observing the amount of deposit put by the client for each payment.

Strawman Solutions for the Two Problems. To address Problem 1, one may slightly adjust the zkCSP protocol such that it would require the client to deposit coins (long) 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/transaction to the contract. But a malicious client can avoid doing so or make the server compute invalid proofs, that ultimately allows the client to get its deposit back. Alternatively, one may let a smart contract perform the verification on the client's behalf, such that the client deposits its coins in the contract when it starts using the service. Then, the server sends its proof to the contract who performs the verification and pays the server if the proof is accepted. Even though this approach would solve (only) problem 1 above, it imposes high costs and defeats the purpose of zkCSP design. The reason is that the contract has to *always* be involved to run the verification algorithm that has to be a publicly verifiable one, which usually imposes high computation cost. Moreover, one might want to let the server pick a fresh address for each verifier/verification, to preserver its pseudonymity with the hope that an observer cannot link clients to a server (so both problems can be addressed). However, for this to work, we have to assume multiple service providers use the same protocol on the blockchain and all of them are pseudonymous. But, this is a strong assumption and may not be always feasible.

3.2 Overview of Our Solution

Addressing Issue 1. To address issue 1, we use a combination of the following techniques, that are oversimplified. First, we upgrade a verifiable service scheme to a "verifiable service with identifiable abort" (VSID). This guarantees

[1] https://aws.amazon.com/s3/pricing/

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; otherwise, its deposit is sent to the server. Third, we allow the party who 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 sever 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 who 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 setting, the arbiter's role can be efficiently played by a smart contract, so its involvement is not needed in that case.

Addressing Issue 2. We use the following ideas to address issue 2. Instead of trying to hide the information from the public *forever*, we let it become *stale*, to lose its sensitivity, and then it will become publicly accessible. In particular, the client and server agree on the period in which the data should remain hidden, "private time bubble". During that period, all messages sent to the contract are encrypted and the parties do not raise any dispute. They postpone raising any dispute to the time when the private time bubble ends (or the bubble bursts). Nevertheless, the client/server can still find out if a proof is valid as soon as it is provided by its counter-party, because it can locally verify the proof. To further hide the amount of deposit, we let each party mask its coins such that no one other than the two parties knows the amount of masking coins. But this raises another challenge: *how can the (mutually untrusted) parties claim back their masking coins after the bubble bursts, while hiding the coins amount from the public in the private time bubble?* One may want to explicitly encode 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. To address the challenge, we let the client and server, at the beginning of the protocol, (efficiently) agree on a private statement specifying the deposit details, e.g. parties' 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 first checks the validity of the statement and if it is accepted, it distributes coins according to the statement (and status of the contract). We will show how they can efficiently agree on such statement, by using a statement agreement protocol (SAP).

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

4 Definitions

In this section, first we present a generic formal definition of a verifiable service (VS) scheme. Second, we provide a formal definition of an enhanced VS scheme that allows: (a) either party to be malicious and (b) a trusted third-party arbiter, to identify a corrupt party. We call the scheme that offers the above features verifiable service with identifiable abort (VSID). Third, we present a formal definition of an upgraded VSID scheme that addresses the issues we identified in Section 3, i.e. wasting server's resources and real-time leakage of service input and proof's status. We call the latter scheme recurring contingent service payment (RC-S-P).

4.1 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, and its input u, and a query $\boldsymbol{q}$ to a server. The server is expected to evaluate F on u and q 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 3 (VS Scheme). *A verifiable service scheme VS* $:= (\texttt{VS.keyGen}, \texttt{VS.setup}, \texttt{VS.genQuery}, \texttt{VS.prove}, \texttt{VS.verify})$ *consists of five algorithms defined as follows.*

- $\texttt{VS.keyGen}(1^\lambda, F) \rightarrow k := (sk, pk)$. *A probabilistic algorithm run by the client. It takes as input the security parameter* 1^λ *and a function,* F*, that will be run on the client's input by the server. It outputs a secret/public verification key pair* k*.*

- $\texttt{VS.setup}(1^\lambda, u, k, M) \rightarrow (u^*, \sigma, pp)$. *It is run by the client. It takes as input the security parameter* 1^λ*, the service input* u*, key pair* k *and metadata generator deterministic function* M*, where* M *is publicly known. If an encoding is needed, then it encodes* u*, that results* u^**; otherwise,* $u^* = u$*. It outputs encoded input* u^**, (possibly input dependent) public parameters* pp*, metadata* $\sigma = M(u^*, k, pp)$*. Right after that, the server is given* u^**,* σ*,* pp*, and* pk*.*

- $\texttt{VS.genQuery}(1^\lambda, aux, k, Q, pp) \rightarrow \boldsymbol{q}$. *A probabilistic algorithm run by the client. It takes as input the security parameter* 1^λ*, auxiliary information aux, the key pair* k*, query generator deterministic function* Q *(where* Q *is publicly known) 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.*

- $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \pi$. *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 pair,* $\pi := (F(u^*, \boldsymbol{q}, pp), \delta)$ *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.*

- $\texttt{VS.verify}(\pi, \boldsymbol{q}, k, pp) \rightarrow d \in \{0, 1\}$. *It is run by the client. It takes as input the proof* π*, query vector* $\boldsymbol{q}$*, key* k*, and public parameters* pp*. In the case where* $\texttt{VS.verify}(.)$ *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 4 (VS Correctness). *A verifiable service scheme, VS, is correct, if for functions* F, Q, M*, and an auxiliary information* aux*, the key generation algorithm produces keys* $\texttt{VS.keyGen}(1^\lambda, F) \rightarrow k := (sk, pk)$ *s.t. for any service input* u*, if* $\texttt{VS.setup}(1^\lambda, u, k, M) \rightarrow (u^*, \sigma, pp)$*,* $\texttt{VS.genQuery}(1^\lambda, aux, k, Q, pp) \rightarrow \boldsymbol{q}$ *and* $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \pi$*, then* $\texttt{VS.verify}(\pi, \boldsymbol{q}, k, pp) \rightarrow 1$

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 5 (VS Soundness). *A verifiable service VS is sound for functions* F, Q, M*, and an auxiliary information* aux*, if for any probabilistic polynomial time adversaries* $\mathcal{A}$*, there exists a negligible function* $\mu(\cdot)$*, such that for any security parameter* λ*:*

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

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 [11, 21] or Provable Data Possession [1, 22], verifiable computation, verifiable searchable encryption, and verifiable information retrieval, to name a few. Other additional security properties mandated by certain services can be added to the above definition. Depending on the properties, they can be

plugged into the above definition with minimal adjustment to the definition. Privacy is an example. 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 2.7) captures VS definition. In particular, PoR's ϵ-soundness captures VS's soundness. Because in the ϵ-soundness, the extractor algorithm interacts (many times) with the cheating prover who 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. Thus, any protocol that realises PoR definition, realises VS definition as well.

4.2 Verifiable Service with Identifiable Abort (VSID) Definition

A protocol that realises only VS's definition, would be merely secure against a malicious server and assumes the client is honest. Although this assumption would suffice in certain settings and has been used before (e.g. in []), 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 VS's definition to allow (a) either party to be malicious and (b) a trusted third-party, *arbiter*, to identify a corrupt party. We call an upgraded verifiable service scheme with that features "verifiable service with identifiable abort" (VSID), inspired by the notion of secure multi-party computation with identifiable abort [10].

Definition 6 (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`) *consists of eight algorithms defined below.*

- `VSID.keyGen`$(1^\lambda, F) \rightarrow k := (sk, pk)$. *A probabilistic algorithm run by the client. It takes as input the security parameter* 1^λ *and a function,* F*, that will be run on the client's input by the server. It outputs a secret/public verification key pair* k*.*

- `VSID.setup`$(1^\lambda, u, k, M) \rightarrow (u^*, pp, e)$. *It is run by the client. It takes as input the security parameter* 1^λ*, the service input* u*, the key pair* k*, and metadata generator deterministic function* M*, where* M *is publicly known. 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 a metadata and* w_σ *is a proof asserting the metadata is well-structured.*

- `VSID.serve`$(u^*, e, pk, pp) \rightarrow a \in \{0, 1\}$. *It is run by the server. 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* $a = 0$*.*

- `VSID.genQuery`$(1^\lambda, aux, k, Q, pp) \rightarrow c := (\boldsymbol{q}, \boldsymbol{w}_q)$. *A probabilistic algorithm run by the client. It takes as input the security parameter* 1^λ*, auxiliary information aux, the key pair* k*, query generator deterministic function* Q *(where* Q *is publicly known), 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.*

- `VSID.checkQuery`$(c, pk, pp) \rightarrow b \in \{0, 1\}$. *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* $b = 0$*.*

- `VSID.prove`$(u^*, \sigma, c, pk, pp) \rightarrow \pi$. *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 pair,* $\pi := (F(u^*, \boldsymbol{q}, pp), \delta)$

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.

- $\texttt{VSID.verify}(\pi, \boldsymbol{q}, k, pp) \rightarrow d \in \{0,1\}$. *It is run by the client. It takes as input the proof π, queries $\boldsymbol{q}$, key pair k, and public parameters pp. If the proof is accepted, it outputs $d = 1$; otherwise, it outputs $d = 0$.*

- $\texttt{VSID.identify}(\pi, c, k, e, u^*, pp) \rightarrow I \in \{\mathcal{C}, \mathcal{S}, \perp\}$. *It is run by a third-party arbiter. It takes as input the proof π, 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 π is rejected it outputs $I = \mathcal{S}$. Otherwise, if $w_\sigma, \boldsymbol{w}_q$, and π 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. 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 7 (VSID Correctness). *A verifiable service with identifiable abort scheme is correct if for any functions F, M, Q, and any auxiliary information aux, the key generation algorithm produces keys $\texttt{VSID.keyGen}(1^\lambda, F) \rightarrow k := (sk, pk)$ such that for any service input u, if $\texttt{VSID.setup}(1^\lambda, u, k, M) \rightarrow (u^*, pp, e)$, $\texttt{VSID.serve}(u^*, e, pk, pp) \rightarrow a$, $\texttt{VSID.genQuery}(1^\lambda, aux, k, Q, pp) \rightarrow c$, $\texttt{VSID.checkQuery}\ (c, pk, pp) \rightarrow b$, $\texttt{VSID.prove}(u^*, \sigma, c, pk, pp) \rightarrow \pi$, and $\texttt{VSID.verify}(\pi, \boldsymbol{q}, k, pp) \rightarrow d$, then $\texttt{VSID.identify}(\pi, c, k, e, u^*, pp) \rightarrow I = \perp\ \wedge\ a = 1\ \wedge\ b = 1\ \wedge\ d = 1$*

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 8 (VSID Soundness). *A VSID is sound for functions F, Q, M, and an auxiliary information aux, if for any probabilistic polynomial time adversary $\mathcal{A}_1$, there exists a negligible function $\mu(\cdot)$, such that for any security parameter λ:*

$$\Pr\left[F(u^*, \boldsymbol{q}, pp) \neq h \wedge d = 1 \;\middle|\; \begin{array}{l} \texttt{VSID.keyGen}(1^\lambda, F) \rightarrow k := (sk, pk) \\ \mathcal{A}_1(1^\lambda, pk, F) \rightarrow u \\ \texttt{VSID.setup}(1^\lambda, u, k, M) \rightarrow (u^*, e, pp) \\ \texttt{VSID.genQuery}(1^\lambda, aux, k, Q, pp) \rightarrow c := (\boldsymbol{q}, \boldsymbol{w}_q) \\ \mathcal{A}_1(c, e, u^*, pp) \rightarrow \pi := (h, \delta) \\ \texttt{VSID.verify}(\pi, \boldsymbol{q}, k, pp) \rightarrow d \end{array} \right] \leq \mu(\lambda).$$

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 9 (VSID Inputs Well-formedness). *A VSID has well-formed inputs for functions F, Q, M, and an auxiliary information aux, if for any probabilistic polynomial time adversary $\mathcal{A}_2$, there exists a negligible function $\mu(\cdot)$, such that for any security parameter λ:*

$$\Pr\left[\begin{array}{l} (M(u^*, k, pp) \neq \sigma \wedge a = 1) \vee \\ (Q(aux, k, pp) \neq \boldsymbol{q}) \wedge b = 1) \end{array} \;\middle|\; \begin{array}{l} \mathcal{A}_2(1^\lambda, F, M, Q) \rightarrow (u^*, k := (sk, pk), e := (\sigma, w_\sigma), pp) \\ \texttt{VSID.serve}(u^*, e, pk, pp) \rightarrow a \\ \mathcal{A}_2(aux, k) \rightarrow c := (\boldsymbol{q}, \boldsymbol{w}_q) \\ \texttt{VSID.checkQuery}(c, pk, pp) \rightarrow b \end{array} \right] \leq \mu(\lambda).$$

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 to 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 10 (VSID Detectable Abort). *A VSID supports detectable for functions F, Q, M, and an auxiliary information aux, if the following hold:*

1. *For any PPT adversary $\mathcal{A}_1$ there exists a negligible function $\mu_1(\cdot)$ such that for any security parameter λ:*

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

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

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

Lighter VSID Scheme (VSID$_{\text{light}}$) In the VSID definition, algorithm $\texttt{VSID.identify}(.)$ allows an arbiter to identify a misbehaving party even in the setup phase. Nevertheless, often it is 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 $\texttt{VSID.serve}(.)$ and outputs 1. A VSID protocol that meets the latter (lighter) requirements, denoted by VSID$_{\text{light}}$, would impose lower costs especially when u and elements of e are of large size. In VSID$_{\text{light}}$ the arbiter algorithm, i.e. $\texttt{VSID.identify}(.)$, needs to take only (π, c, k, e', pp) as input, where $e' \subset e$. Note also u^* is not given to the arbiter. In the light version, the arbiter skips checking the correctness of metadata. So, this requires two changes to the VSID definition, (a) the arbiter algorithm would be $\texttt{VSID.identify}(\pi, c, k, e', pp) \rightarrow I$, and (b) in case 2, in Definition 10 we would have $b = 0 \wedge I \neq \mathcal{C}$, so event $a = 0$ is excluded. In this paper, any time we refer to VSID$_{\text{light}}$, we assume the above minor adjustments are applied to the VSID definition.

4.3 Recurring Contingent Service Payment (RC-S-P) Definition

Even though a VSID scheme offers two appealing features, it is not sufficient yet to address all the issues we identified in Section 3. In particular, it does not take the privacy of service input and proofs' status into consideration and does not take into account a secure and fair payment (so it cannot deal with the issue related to wasting the server's resources). Thus, in the following, we present an upgraded VSID's definition that takes the above points into account. We call the enhanced VSID, which offers the features, recurring contingent service payment (RC-S-P).

Definition 11 (RC-S-P Scheme). *A recurring contingent service payment scheme RC-S-P =* $(\texttt{RCSP.keyGen}, \texttt{RCSP.cInit}, \texttt{RCSP.sInit}, \texttt{RCSP.genQuery}, \texttt{RCSP.prove}, \texttt{RCSP.verify}, \texttt{RCSP.resolve}, \texttt{RCSP.pay})$ *involves four parties; namely, client, server, arbiter and smart contract, and consists of eight algorithms defined as follows.*

- $\texttt{RCSP.keyGen}(1^\lambda, F) \rightarrow \boldsymbol{k}$. *A probabilistic algorithm run by the client. It takes as input security parameter 1^λ, and function F that will be run on the client's input by the server as a part of the service it provides. It outputs $\boldsymbol{k}$ that contains a secret and public verification key pair $k := (sk, pk)$ and a set of secret and public parameters, $k' := (sk', pk')$. It sends pk and pk' to the smart contract.*

- $\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, M, z, pl, enc) \rightarrow (u^*, e, T, p_S, \boldsymbol{y}, coin_c^*)$. *It is run by the client. It takes as input 1^λ, the service input u, key pair $\boldsymbol{k} := (k, k')$, metadata generator function M, 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 also takes as input encoding/decoding functions $enc := (E, D)$ used*

to encode/decode the service queries/proofs. It encodes u, that yields u^*. It sets pp as (possibly) input dependent parameters, e.g. file size. It computes metadata $\sigma = M(u^*, k, pp)$ and a proof w_σ asserting the metadata is well-structured. It sets 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, e.g. its maximum values. It constructs coin encoding token T_{cp} containing cp and cp's witness, g_{cp}. It constructs encoding token T_{qp} that contains secret parameters qp including pp, (a representation of σ) and parameters (in sk') that will be used to encode the service queries/proofs. Also 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. 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). The client sends u^*, z, e, $T_{cp} \setminus \{g_{cp}\}$ and $T_{qp} \setminus \{g_{qp}\}$ to the server and sends $g_{cp}, g_{qp}, p_{\mathcal{S}}$, and $\boldsymbol{y}$, and $coin^*_{\mathcal{C}}$ coins to the contract.

- $\texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}, enc) \rightarrow (coin^*_{\mathcal{S}}, a)$. *It is run by the server. It takes as input the service encoded input u^*, metadata-proof pair $e := (\sigma, w_\sigma)$, public key pk (read from the contract), the total number of verifications z, and $T := (T_{cp}, T_{qp})$ (where $\{g_{cp}, g_{qp}\}$ are read from the smart contract). Also, it reads $p_{\mathcal{S}}$, and $\boldsymbol{y}$ from the smart contract and takes as input the encoding/decoding functions $enc := (E, D)$. It verifies the validity of e and T elements. Also, it checks elements of $\boldsymbol{y}$ and ensures element of $\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, \boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{S}} \in \boldsymbol{y}$ have been set to 0. If all checks are successful, then it encodes the amount of its coins $coin^*_{\mathcal{S}}$ 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.*

- $\texttt{RCSP.genQuery}(1^\lambda, aux, k, Q, T_{qp}, enc) \rightarrow c^*_j$. *A probabilistic algorithm run by the client. It takes as input 1^λ, auxiliary information aux, the key pair k, query generator function Q, encoding token T_{qp} and $enc := (E, D)$. 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.*

- $\texttt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp}, enc) \rightarrow (b_j, m_{\mathcal{S},j}, \pi^*_j)$. *It is run by the server. It takes as input the encoded service input u^*, metadata σ, encoded query pair c^*_j, public key pk, the encoding token T_{qp}, and $enc := (E, D)$. It checks the validity of decoded query $c_j = D(c^*_j, T_{qp})$. 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 $\pi^*_j = E(\pi_j, T_{qp})$, where π_j contains $h_j = F(u^*, \boldsymbol{q}_j, pp)$ and a proof δ_j asserting the evaluation is performed correctly (π_j may contain dummy values if $b_j = 0$). The smart contract is given π^*_j.*

- $\texttt{RCSP.verify}(\pi^*_j, c^*_j, k, T_{qp}, enc) \rightarrow (d_j, m_{\mathcal{C},j})$. *A deterministic algorithm run by the client. It takes as input the encoded proof π^*_j, query vector $\boldsymbol{q}_j \in c^*_j$, key pair k, the encoding token T_{qp} and $enc := (E, D)$. If the decoded proof $\pi_j = D(\pi^*_j, T_{qp})$ is rejected, it outputs $d_j = 0$ and a complaint $m_{\mathcal{C},j}$. Otherwise, it outputs $d_j = 1$ and $m_{\mathcal{C},j} = \perp$.*

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

- $\texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}}) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{A}r})$. *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, coins' token $T_{cp} := \{cp, g_{cp}\}$, the output of the checks that server-side initiation algorithm performed a, the total coins the server should deposit $p_{\mathcal{S}}$, and the total coins amount the client and server deposited, i.e. $coin^*_{\mathcal{C}}$ and $coin^*_{\mathcal{S}}$ respectively. 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 $\boldsymbol{coin}_{\mathcal{I}}$, where $\mathcal{I} \in \{\mathcal{C}, \mathcal{S}, \mathcal{A}r\}$; It sends $coin_{\mathcal{I},j} \in \boldsymbol{coin}_{\mathcal{I}}$*

coins to party $\mathcal{I}$ 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 $\texttt{RCSP.genQuery}(.)$, $\texttt{RCSP.prove}(.)$, $\texttt{RCSP.verify}(.)$ and $\texttt{RCSP.resolve}(.)$ implicitly take $a, coin^*_{\mathcal{S}}, p_{\mathcal{S}}$ as another inputs and execute only if $a = 1$ and $coin^*_{\mathcal{S}} = p_{\mathcal{S}}$; however, for the sake of simplicity we avoided explicitly stating it in the definition.

A recurring contingent service payment (RC-S-P) scheme satisfies correctness and security. At a high level, correctness requires that by the end of the protocol's execution (that involves honest client and server) the client receives all z valid 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. It is formally stated as below.

Definition 12 (RC-S-P Correctness). *A recurring contingent service payment scheme is correct for functions F, Q, M, E, D, an auxiliary information $aux_1, ..., aux_z$, if for any price list pl, the key generation algorithm produces keys $\texttt{RCSP.keyGen}(1^\lambda, F) \rightarrow \boldsymbol{k}$, such that for any service input u, if $\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, M, z, pl, enc) \rightarrow (u^*, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin^*_{\mathcal{C}})$, $\texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}, enc) \rightarrow (coin^*_{\mathcal{S}}, a)$, $\forall j : \Big(\texttt{RCSP.genQuery}(1^\lambda, aux_j, k, Q, T_{qp}, enc) \rightarrow c^*_j$, $\texttt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp}, enc) \rightarrow (b_j, m_{\mathcal{S},j}, \pi^*_j)$, $\texttt{RCSP.verify}(\pi^*_j, c^*_j, k, T_{qp}, enc) \rightarrow (d_j, m_{\mathcal{C},j})\Big)$, $\texttt{RCSP.resolve}(m_{\mathcal{C}}, m_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp}, enc) \rightarrow \boldsymbol{y}$, $\texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}}) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{A}r})$, then $(a = 1) \wedge (\bigwedge_{j=1}^{z} b_j = \bigwedge_{j=1}^{z} d_j = 1) \wedge (\boldsymbol{y}_{\mathcal{C}} = \boldsymbol{y}_{\mathcal{S}} = \boldsymbol{y}'_{\mathcal{C}} = \boldsymbol{y}'_{\mathcal{S}} = 0) \wedge (\sum_{j=1}^{z} coin_{\mathcal{C},j} = coin^*_{\mathcal{C}} - o \cdot z) \wedge (\sum_{j=1}^{z} coin_{\mathcal{S},j} = coin^*_{\mathcal{S}} + o \cdot z) \wedge (\sum_{j=1}^{z} coin_{\mathcal{A}r,j} = 0)$, where $\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, \boldsymbol{y}'_{\mathcal{C}}, \boldsymbol{y}'_{\mathcal{S}} \in \boldsymbol{y}$*

A RC-S-P scheme is said to be secure if it satisfies three main properties: (a) security against malicious server, (b) security against 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 j-th verification) and the arbiter receives l coins, or (iii) if it 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 incorrect amounts of coin 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 (i.e., $b_j = 1$ or $y_{\mathcal{S},j} = 0$) or makes them withdraw incorrect amounts of coin (i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z}$ or $coin_{\mathcal{A}r,j} \neq l$ coins). Below, we formalize this intuition.

Definition 13 (RC-S-P Security Against Malicious Server). *A RC-S-P is secure against a malicious server, for functions F, Q, M, D, E, and an auxiliary information aux, if for any price list pl, every j (where $1 \leq j \leq z$), and any PPT adversary $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that for any security parameter λ:*

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

where $\boldsymbol{q}_j \in D(c^*_j, t_{qp})$, $\pi^*_j := (h^*_j, \delta^*_j), h_j = D(h^*_j, T_{qp})$, $D \in enc$, $\sigma \in e$, $m_{\mathcal{C},j} \in m_{\mathcal{C}}, m_{\mathcal{S},j} \in m_{\mathcal{S}}, y'_{\mathcal{S},j} \in \boldsymbol{y}'_{\mathcal{S}} \in \boldsymbol{y}$, $y_{\mathcal{S},j} \in \boldsymbol{y}_{\mathcal{S}} \in \boldsymbol{y}$, 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 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 $coin_{\mathcal{A}r,j} \neq l$ coins). Below, we formally state the property.

Definition 14 (RC-S-P Security Against Malicious Client). *A RC-S-P is secure against a malicious client for functions F, Q, M, D, E, and an auxiliary information aux, if for every j (where $1 \leq j \leq z$), and any probabilistic polynomial time adversary $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that for any security parameter λ:*

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

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

Note, in the above definition, an honest server either does not deposit (e.g. when $a = 0$) or if it deposits (i.e. agrees to serve) ultimately receives its deposit *plus the service payment* (with a high probability). Informally, RC-S-P is privacy preserving if it guarantees the privacy of (a) the service input (e.g. outsourced file) and (b) the service proof's status during the private time bubble. In the following, we formally define privacy.

Definition 15 (RC-S-P Privacy). *A RC-S-P preserves privacy, for functions F, Q, M, E, D, an auxiliary information aux, if for any number of verifications z, any price list pl, the following hold:*

1. *For any PPT adversary $\mathcal{A}_1$ there exists a negligible function $\mu(\cdot)$, such that for any security parameter λ:*

$$\Pr\left[\begin{array}{l|l} \mathcal{A}_1(\boldsymbol{c}^*,coin^*_{\mathcal{S}},coin^*_{\mathcal{C}},g_{cp}, & \texttt{RCSP.keyGen}(1^\lambda,F)\rightarrow\boldsymbol{k} \\ g_{qp},\boldsymbol{\pi}^*,pl,a)\rightarrow\beta & \mathcal{A}_1(1^\lambda,pk,F)\rightarrow(u_0,u_1) \\ & \beta\xleftarrow{\$}\{0,1\} \\ & \texttt{RCSP.cInit}(1^\lambda,u_\beta,\boldsymbol{k},M,z,pl,enc)\rightarrow(u^*_\beta,e,T,p_{\mathcal{S}},\boldsymbol{y},coin^*_{\mathcal{C}}) \\ & \texttt{RCSP.sInit}(u^*_\beta,e,pk,z,T,p_{\mathcal{S}},\boldsymbol{y},enc)\rightarrow(coin^*_{\mathcal{S}},a) \\ & \forall j\in[z]: \\ & \Big(\texttt{RCSP.genQuery}(1^\lambda,aux,k,Q,T_{qp},enc)\rightarrow c^*_j \\ & \texttt{RCSP.prove}(u^*_\beta,\sigma,c^*_j,pk,T_{qp},enc)\rightarrow(b_j,m_{\mathcal{S},j},\pi^*_j) \\ & \texttt{RCSP.verify}(\pi^*_j,c^*_j,k,T_{qp},enc)\rightarrow(d_j,m_{\mathcal{C},j})\Big) \end{array}\right]\leq\frac{1}{2}+\mu(\lambda).$$

2. *For any PPT adversaries $\mathcal{A}_2$ and $\mathcal{A}_3$ there exists a negligible function $\mu(\cdot)$ such that for any security parameter λ:*

$$\Pr\left[\begin{array}{l|l} \mathcal{A}_3(\boldsymbol{c}^*,coin^*_{\mathcal{S}},coin^*_{\mathcal{C}},g_{cp}, & \texttt{RCSP.keyGen}(1^\lambda,F)\rightarrow\boldsymbol{k} \\ g_{qp},\boldsymbol{\pi}^*,pl,a)\rightarrow(d_j,j) & \mathcal{A}_2(1^\lambda,pk,F)\rightarrow u \\ & \texttt{RCSP.cInit}(1^\lambda,u,\boldsymbol{k},M,z,pl,enc)\rightarrow(u^*,e,T,p_{\mathcal{S}},\boldsymbol{y},coin^*_{\mathcal{C}}) \\ & \texttt{RCSP.sInit}(u^*,e,pk,z,T,p_{\mathcal{S}},\boldsymbol{y},enc)\rightarrow(coin^*_{\mathcal{S}},a) \\ & \forall j\in[z]: \\ & \Big(\mathcal{A}_2(1^\lambda,aux,k,Q,T_{qp},enc)\rightarrow c^*_j \\ & \texttt{RCSP.prove}(u^*,\sigma,c^*_j,pk,T_{qp},enc)\rightarrow(b_j,m_{\mathcal{S},j},\pi^*_j) \\ & \texttt{RCSP.verify}(\pi^*_j,c^*_j,k,T_{qp},enc)\rightarrow(d_j,m_{\mathcal{C},j})\Big) \end{array}\right]\leq Pr'+\mu(\lambda)$$

where $\boldsymbol{c}^* = [c_1^*, ..., c_z^*], \boldsymbol{\pi}^* = [\pi_1^*, ..., \pi_z^*]$, $pp \in T_{qp}$, *and* $Pr' = Max(Pr_0, Pr_1)$, *and* Pr_i *for every* j, i ($i \in \{0,1\}$, *and* $1 \leq j \leq z$) *is defined as:*

$$Pr_i = \Pr\left[\begin{array}{l|l} & \texttt{RCSP.keyGen}(1^\lambda, F) \rightarrow \boldsymbol{k} \\ & \mathcal{A}_2(1^\lambda, pk, F) \rightarrow u \\ & \texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, M, z, pl, enc) \rightarrow (u^*, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin_{\mathcal{C}}^*) \\ Con_i^{(1)} \wedge Con_i^{(2)} & \texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}, enc) \rightarrow (coin_{\mathcal{S}}^*, a) \\ & \mathcal{A}_2(1^\lambda, aux, k, Q, T_{qp}, enc) \rightarrow c_j^* \\ & \texttt{RCSP.prove}(u^*, \sigma, c_j^*, pk, T_{qp}, enc) \rightarrow (b_j, m_{\mathcal{S},j}, \pi_j^*) \\ & \texttt{RCSP.verify}(\pi_j^*, c_j^*, k, T_{qp}, enc) \rightarrow (d_j, m_{\mathcal{C},j}) \end{array}\right]$$

such that $\boldsymbol{q}_j \in D(c_j^*, t_{qp}), Con_0^{(1)} : Q(aux, k, pp) \neq \boldsymbol{q}_j, Con_0^{(2)} : b_j = 0, Con_1^{(1)} : Q(aux, k, pp) = \boldsymbol{q}_j$, *and* $Con_1^{(2)} : b_j = 1$.

Note that in the above definition, for each j-th verification, adversary $\mathcal{A}_2$ with probability Pr_0 produces an invalid query and with probability Pr_1 produces a valid query. It is required that the privacy holds regardless of the queries and proofs status, i.e. whether they are valid or invalid, as long as they are correctly encoded and provided. In the above definitions, the private time bubble is a time period from the point when $\texttt{RCSP.keyGen}(.)$ is executed up to the time when $\texttt{RCSP.resolve}(.)$ is run. In other words, the privacy holds up to the point where $\texttt{RCSP.resolve}(.)$ is run. That is why the latter algorithm is excluded from the experiments in Definition 15.

Definition 16 (RC-S-P Security). *A RC-S-P scheme is secure if it satisfies security against malicious server, security against malicious client, and preserves privacy, w.r.t. Definitions 13-15.*

5 VSID Protocol

In this section, we present the VSID protocol. We show how it can be built upon a protocol that realises the VS definition. As stated previously, 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 non-interactive publicly verifiable zero-knowledge 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 and then, in the setup, it encodes its input: u using the encoding function M. This results in a metadata. Also, the client utilises a non-interactive publicly verifiable zero-knowledge scheme to prove to the server that the metadata has been constructed correctly. The client posts u, metadata and the poofs 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 u, it uses function Q to generate a query. But, 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 prove and client-side verify algorithms remain unchanged with a difference that the server posts its proofs (i.e. output of prove algorithm) and thier 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 who, given the signed posted messages, checks the signatures and proofs in turn to identify a corrupt party. Below, we present VSID protocol in which we assume all parties sign their outgoing messages and their counter-party first verify the signature on the messages, before they fed them to their local algorithms.

1. **Key Generation**. $\texttt{VSID.keyGen}(1^\lambda, F)$
   (a) Calls $\texttt{VS.keyGen}(1^\lambda, F)$ 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**. $\texttt{VSID.setup}(1^\lambda, u, k, M)$
   (a) Calls $\texttt{VS.setup}(1^\lambda, u, k, M) \rightarrow (\sigma, u^*)$, 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 contains 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**. $\texttt{VSID.genQuery}(1^\lambda, \text{aux}, k, Q, pp)$
   (a) Calls $\texttt{VS.genQuery}(1^\lambda, \text{aux}, k, Q, pp) \rightarrow \boldsymbol{q}$, 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 $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \pi$, to generate $\pi := (F(u^*, \boldsymbol{q}, pp), \delta)$. Recall that $\boldsymbol{q} \in c$
   (b) Posts π to the board.

7. **Client-side Proof Verification**. $\texttt{VSID.verify}(\pi, \boldsymbol{q}, k, pp)$
   Calls $\texttt{VS.verify}(\pi, \boldsymbol{q}, k, pp) \rightarrow d$, to verify proof π. It accepts the proof if $d = 1$; otherwise, it rejects it.

8. **Arbiter-side Identification**. $\texttt{VSID.identify}(\pi, c, k, e, u^*, pp)$
   (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 π 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 $\texttt{VS.verify}(\pi, \boldsymbol{q}, k, pp) \rightarrow d$. If $d = 1$, then it outputs $I = \perp$; otherwise, it outputs $I = \mathcal{S}$

Theorem 1. *The VSID protocol satisfies the soundness, inputs well-formedness, and detectable abort properties, w.r.t. Definitions 8-10, if the commitment, non-interactive publicly verifiable zero-knowledge, VS, and signature schemes are secure.*

Proof (sketch). The soundness of VSID stems from the hiding property of the commitment, zero-knowledge property of the publicly verifiable zero-knowledge 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 a negligible probability $\mu(\lambda)$. Moreover, the soundness of VS scheme ensures a corrupt prover cannot convince an honest verifier, except with

probability $\mu(\lambda)$. Inputs well-formedness property boils down to the security of the commitment and publicly verifiable non-interactive zero-knowledge 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 non-interactive zero-knowledge 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 probability $\mu(\lambda)$. 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 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 to the arbiter other 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, often it is 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. $\texttt{VSID.identify}(.)$, needs to take only (π, c, k, e', pp) as input, where e' contains the opening of $\texttt{Com}_{sk}$ if $\texttt{VSID.verify}(.)$ 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 $\texttt{VSID.serve}(.)$ anymore.

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

In this section, we present RC-S-P Protocol. As we explained in Section 3.2, the protocol relies on the idea that the server and client can efficiently agree on private statements at the begging of the protocol. Therefore, in the following, we first present a protocol (called SAP) that satisfies that requirements and then present RC-S-P Protocol.

6.1 Statement Agreement Protocol (SAP)

In this section, we explain how a client and server, mutually distrusted, can efficiently agree on a private statement, e.g. a string or tuple, that will be used to reclaim parties' masking coins or utilised by a party to prove it has an agreement with its counter-party on secret parameters, when the private bubble bursts. Informally, a statement agreement protocol (SAP) is secure if it meets four security properties. First, neither party should be able to persuade a third-party verifier that it has agreed with its counter-party on an invalid statement, i.e. the statement that both parties have not agreed on. Second, after they successfully agree on the statement, an honest party should be able to successfully prove it to the verifier, i.e. an adversary cannot prevent an honest party from successfully proving it. Third, the privacy of the statement should be preserved, (against other parties than the client and server before either of them attempt to prove the agreement on the statement). Forth, after both parties reach an agreement, neither can later deny the agreement. To that end, we use a combination of smart contract and 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 on, for a party to prove to the contract, i.e. the verifier, that it has an agreed on the statement with the other party, it only sends the opening of the commitment. The contract checks if the opening matches both

commitments and accepts if it matches. The SAP protocol is provided below. It assumes that each party $\mathcal{R} \in \{\mathcal{C}, \mathcal{S}\}$ already has a blockchain public address $adr_{\mathcal{R}}$ (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) Designs and deploys a smart contract, SAP, that explicitly states both parties' addresses, $adr_{\mathcal{C}}$ and $adr_{\mathcal{S}}$. Let adr_{SAP} be the deployed contract's address.
   (b) Picks a random value r, and commits to the statement, $\texttt{Com}(x, r) = g_{\mathcal{C}}$
   (c) Sends adr_{SAP} and $\ddot{x} := (x, r)$ to $\mathcal{S}$.
   (d) Sends $g_{\mathcal{C}}$ to the contract, using its account
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 $g_{\mathcal{C}}$ has been sent to the contract from $adr_{\mathcal{C}}$, and checks $\texttt{Ver}(g_{\mathcal{C}}, \ddot{x}) = 1$. If the checks pass, it sets $b = 1$ and it computes $\texttt{Com}(x, r) = g_{\mathcal{S}}$. Otherwise, it sets $b = 0$ and $g_{\mathcal{S}} = \perp$.
   (b) If $b = 1$, then sends $g_{\mathcal{S}}$ to the contract using its account.
3. **Prove**. For either $\mathcal{C}$ or $\mathcal{S}$ to prove, it has agreement on x with its counter-party, 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 $adr_{\mathcal{S}}$ respectively.
   (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 the two previous steps (i.e. steps 4a and 4b) were passed. Otherwise, it outputs $d = 0$.

Intuitively, the first property is guaranteed due to binding property of the (hash-based) commitment scheme, while the second property is satisfied due to the security of the blockchain and smart contract, i.e. due to blockchain's liveness property an honestly generated transaction, containing the opening, eventually gets into chains of honest miners [7], and due to security and correctness of smart contracts a valid opening is always accepted by the contract. The third property is met due to the hiding property of the commitment, while the forth property is satisfied due to the security of the signature scheme that is used to sign transactions originated from the account holders.

Remark 3. The verification algorithm can also 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 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 relives the verifier from the transaction and contract's execution costs.

Remark 4. One may simply let each party sign the statement and send it to the other party, so later on each party can send both signatures to the contract who verifies them. But, this would not work, as the party who first receives the other party's signature may refuse to send its signature, that prevents the other party to prove that it has an agreed on the statement with its counter-party. Alternatively, one may want to use a protocol for a fair exchange of digital signature (or fair contract signing) such as [2, 6]. 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; who first checks the validity of both signatures. Although this satisfies the above security requirements, it yields two main efficiency and practical issues: (a) it imposes very high computation costs, as protocols for fair exchange of signature involve generic zero-knowledge proofs and require a high number of modular exponentiations. And (b) it is impractical, because protocols for fair exchange of signature protocol support only certain signature schemes (e.g. RSA, Rabin, or Schnorr) that are not directly supported by the most predominant smart contract framework, Ethereum, as it only supports Elliptic Curve Digital Signature Algorithm (EDCSA).

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

In this section, we present "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; first statement includes payment details, while another one specifies a secret key, k, and a pad's 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, $\mathtt{H}$, and (b) a smart contract which specifies z and the total amount of masked coins each party should deposit. They deploy 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 $\mathtt{VSID.keyGen}(.)$, $\mathtt{VSID.setup}(.)$, and $\mathtt{VSID.serve}(.)$ 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.

In the end of each billing cycle, the client generates an encrypted query, by calling $\mathtt{VSID.genQuery}(.)$ 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 $\mathtt{VSID.checkQuery}(.)$. 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 $\mathtt{VSID.prove}(.)$. In either case, the server encrypts the proof, pads it and sends the result to the contract. Note that sending (padded encrypted) dummy proof 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 $\mathtt{VSID.verify}(.)$. If the verification is passed, then the client knows the server has delivered the service honestly. However, 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 sends a "dispute" message to the contract. The party also 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 $\mathtt{VSID.identify}(.)$ 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. 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.

The RC-S-P protocol is presented below. It is assumed that (a) each party $\mathcal{R} \in \{\mathcal{C}, \mathcal{S}, \mathcal{A}r\}$ already has a blockchain public address, $adr_{\mathcal{R}}$, 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 eveyone.

1. **Key Generation**. $\mathtt{RCSP.keyGen}(1^{\lambda}, F)$

(a) $\mathcal{C}$ runs $\texttt{VSID.keyGen}(1^\lambda, F) \rightarrow k := (sk, pk)$. It 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[2], determined by the security parameter and description of F. Let $qp := (pad_\pi, pad_q, \bar{k})$. The keys' size is part of the security parameter. Let $\boldsymbol{k} = [k, k']$, where $k' := (sk', pk')$, $sk' = qp$ and $pk' := (adr_{\mathcal{C}}, adr_{\mathcal{S}})$.

2. **Client-side Initiation**. $\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, M, z, pl, enc)$
   (a) Calls $\texttt{VSID.setup}(1^\lambda, u, k, M) \rightarrow (u^*, pp, e)$, to encode service input, and generate metadata. It 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 designs a smart contract, SC, that explicitly specifies parameters z, $coin^*_{\mathcal{C}}$, $p_{\mathcal{S}}$, adr_{SAP_1}, and adr_{SAP_2}. It sets a set of time points/windows, $\texttt{Time} : \{\texttt{T}_0, ..., \texttt{T}_3, \texttt{G}_{1,1}, ..., \texttt{G}_{z,2}, \texttt{H}, \texttt{K}_1, ..., \texttt{K}_3, \texttt{L}\}$, that are explicitly specified in the contract who 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 $\boldsymbol{y} : [y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{S}}, \texttt{Time}, adr_{\text{SC}}]$
   (g) Deposits $coin^*_{\mathcal{C}}$ coins in the contract. It sends $u^*, e, \ddot{x}_{qp}, \ddot{x}_{cp}$, and $p_{\mathcal{S}}$ (along with adr_{SC}) to $\mathcal{S}$. It sends (pk, pk') to SC. Let $\texttt{T}_0$ be the time that the above process finishes.

3. **Server-side Initiation**. $\texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}, enc)$
   (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 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 $\texttt{VSID.serve}(u^*, e, pk, pp) \rightarrow a$.
   (d) Sends a and $coin^*_{\mathcal{S}} = p_{\mathcal{S}}$ coins to SC at time $\texttt{T}_1$, where $coin^*_{\mathcal{S}} = \perp$ if $a = 0$

   Note, $\mathcal{S}$ 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 $\mathcal{C}$ simply sends a "pay" message to $\texttt{RCSP.pay}(.)$ algorithm (only) at time $\texttt{T}_2$.

Billing-cycles Onset. $\mathcal{C}$ and $\mathcal{S}$ engage in the following three phases, i.e. phase 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**. $\texttt{RCSP.genQuery}(1^\lambda, \text{aux}, k, Q, T_{qp}, enc)$
   (a) Calls $\texttt{VSID.genQuery}(1^\lambda, \text{aux}, k, Q, pp) \rightarrow c_j := (\boldsymbol{q}_j, \boldsymbol{w}_{q_j})$, to generate a query-proof pair.
   (b) Encrypts the pair, $\texttt{Enc}(\bar{k}, c_j) = c'_j$, where $\bar{k} \in T_{qp}$. Then, it pads (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 $\texttt{G}_{j,1}$

[2] 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.

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

   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**. $\texttt{RCSP.verify}(\pi_j^*, c_j^*, k, T_{qp}, enc)$
   (a) Constructs an empty vector, $\boldsymbol{m}_{\mathcal{C}} = \perp$, if $j = 1$.
   (b) Removes the pads from π_j^*, utilising parameters of T_{qp}. Let π_j' be the result. It decrypts the service proof: $\texttt{Dec}(\bar{k}, \pi_j') = \pi_j''$ and then calls $\texttt{VSID.verify}(\pi_j'', \boldsymbol{q}_j, k, pp) \rightarrow d_j$, to verify the proof, where $\boldsymbol{q}_j \in D(c_j^*, t_{qp})$. Note that if $\pi_j' = \texttt{Enc}(\bar{k}, \pi_j)$, then $\pi_j'' = \pi_j$.
      - If π_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 π_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 $\texttt{Com}_{sk}$ or $\perp$, as stated in Remark 2.

7. **Dispute Resolution**. $\texttt{RCSP.resolve}(m_{\mathcal{C}}, m_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp}, enc)$
   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 $\texttt{K}_1$, where $\texttt{K}_1 > \texttt{G}_{z,2} + \texttt{H}$.
   (b) $\mathcal{C}$ sends $m_{\mathcal{C}}$ and $\ddot{x}_{qp}$ to the arbiter at time $\texttt{K}_1$. Or, $\mathcal{S}$ sends $m_{\mathcal{S}}$ and $\ddot{x}_{qp}$ to the arbiter at time $\texttt{K}_1$
   (c) At time $\texttt{K}_2$, the arbiter checks the validity of statement $\ddot{x}_{qp}$ sent by each party $\forall \mathcal{R} \in \{\mathcal{C}, \mathcal{S}\}$. 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 $\mathcal{R}$'s statement is accepted, then it appends every element of $\boldsymbol{m}_{\mathcal{R}}$ 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, $m_{\mathcal{R}}$. 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 π_i' be the result.
      ii. decrypts the encrypted query-proof pair and encrypted service proof as follows, $\texttt{Dec}(\bar{k}, c_i') = c_i$ and $\texttt{Dec}(\bar{k}, \pi_i') = \pi_i''$
      iii. calls $\texttt{VSID.identify}(\pi_i'', c_i, k, e', pp) \rightarrow I_i$
         - 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 $\texttt{K}_3$ be the time that the arbiter finishes the above checks.
   (e) The arbiter at time $\texttt{K}_3$ sends $[y_{\mathcal{C}}, y_{\mathcal{S}}, y_{\mathcal{C}}', y_{\mathcal{S}}']$ to SC who accordingly overwrites the elements it holds (i.e. elements of $\boldsymbol{y}$) by the related vectors elements the arbiter sent.

8. **Coin Transfer**. $\texttt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_{\mathcal{S}}, coin_{\mathcal{C}}^*, coin_{\mathcal{S}}^*)$
   (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 $\mathcal{S}$. In other words, the parties can withdraw their coins if they do not reach an agreement in the end of phase 3, i.e. server-side initiation. Otherwise (i.e. they reach an agreement), they take the following step.
   (b) Either $\mathcal{C}$ or $\mathcal{S}$ send "pay" message and the statement, $\ddot{x}_{cp} \in T_{cp}$, to SC at time $\texttt{L} > \texttt{K}_3$

(c) SC checks the validity of the statement by sending $\ddot{x}_{cp}$ to 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:
- $coin^*_{\mathcal{C}} - o(z - y_{\mathcal{S}}) - l(y_{\mathcal{C}} + y'_{\mathcal{C}})$ coins to $\mathcal{C}$
- $coin^*_{\mathcal{S}} + o(z - y_{\mathcal{S}}) - l(y_{\mathcal{S}} + y'_{\mathcal{S}})$ coins to $\mathcal{S}$
- $l(y_{\mathcal{S}} + y_{\mathcal{C}} + y'_{\mathcal{S}} + y'_{\mathcal{C}})$ coins to the arbiter.

Remark 5. If all parties behave honestly, then the server receives all its deposit back plus the amount of coins they initially agreed to pay the sever 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 amount of coins 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.

Remark 6. As stated in Section 4.3, algorithms $\texttt{RCSP.genQuery}(.), \texttt{RCSP.prove}(.), \texttt{RCSP.verify}(.)$ and $\texttt{RCSP.resolve}(.)$ 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.

Remark 7. Keeping track of $(y'_{\mathcal{C}}, y'_{\mathcal{S}})$ enables the arbiter to make malicious parties, who *unnecessarily* invoke it for accepting proofs in step 7(d)iii, pay for the verifications it performs.

Remark 8. 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 in RC-S-P protocol, *encrypted* proofs are sent to the contract.

Remark 9. 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.

Remark 10. In the case where $\texttt{VSID.verify}(.)$ 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 $\texttt{VSID.verify}(.)$ supports.

6.3 Security Analysis of RC-S-P Protocol

In this section, we analyse the security of RC-S-P protocol, presented in Section 6.2. First, we present the protocol's primary security theorem.

Theorem 2. *The RC-S-P protocol is secure, w.r.t. Definition 16, if VSID, SAP, and signature scheme are secure and the encryption scheme is semantically secure.*

To prove the above theorem, we show that RC-S-P meets all security properties defined in Section 4.3. We start by proving that RC-S-P satisfies security against a malicious server.

Lemma 1. *If SAP and signature scheme are secure and VSID scheme supports correctness, soundness, and detectable abort, then RC-S-P is secure against malicious server, w.r.t. Definition 13.*

Proof (sketch). We first consider event

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

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. $coin_{\mathcal{A}r,j} \neq l$, 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 convince the arbiter that the client has misbehaved, by making the arbiter 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) or send a message on the client's behalf to invoke the arbiter unnecessarily. Nevertheless, it cannot falsely accuse the client of misbehaviour. 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, $\mu(\lambda)$. 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 `VSID.identify`(.) 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 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 $\mu(\lambda)$ when it finds the hash function's collision. 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 j-th verification, except with a negligible probability $\mu(\lambda)$. 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 $coin^*_{\mathcal{C}} - o(z - y_{\mathcal{S}}) - l(y_{\mathcal{C}} + y'_{\mathcal{C}})$. Since we focus on j-th verification, the amount of coins that should be credited to the client for that verification is

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

As shown above $y_{\mathcal{C},j} = y'_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$. Therefore, 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 $l(y_{\mathcal{S}} + y_{\mathcal{C}} + y'_{\mathcal{S}} + y'_{\mathcal{C}})$, so for the j-th the credited coins should be:

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

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 j-th verification. For the server to make the arbiter to withdraw other than that amounts (for 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. However, as argued above, it cannot succeed with probability significantly greater than $\mu(\lambda)$. We now move on to event

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

It 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 $coin_{\mathcal{A}r,j} \neq l$ 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, $\mu(\lambda)$. 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, it 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, and its probability of success is $\mu(\lambda)$. Or it makes the arbiter accept an invalid proof, but due to the detectable abort property of VSID, its probability of success is also $\mu(\lambda)$. 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 $\mu(\lambda)$ 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, $\mu(\lambda)$, due to the security of SAP. Furthermore, in general, due to the security of SAP, the adversary cannot block an honest client's messages, "pay" and $\ddot{x}_{cp}$, to the contract in the coin transfer phase. □

Lemma 2. *If SAP and signature scheme are secure and VSID scheme supports correctness, inputs well-formedness, and detectable abort, then RC-S-P is secure against malicious client, w.r.t. Definition 14.*

Proof (sketch). First, we consider event

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

It 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 incorrect amount of coins, i.e. $coin_{\mathcal{A}r,j} \neq l$, 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 amounts 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 probability $\mu(\lambda)$. 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 $\texttt{VSID.identify}(.)$ 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 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 probability $\mu(\lambda)$. 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 $coin^*_{\mathcal{S}} + o(z - y_{\mathcal{S}}) - l(y_{\mathcal{S}} + y'_{\mathcal{S}})$. Since we focus on j-th verification, the amounts of coin that should be credited to the server for j-th verification is

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

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 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 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 probability $\mu(\lambda)$. We now turn our attention to

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

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, $\mu(\lambda)$. 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

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

It 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 $coin_{\mathcal{A}r,j} \neq l$ respectively. Nevertheless, due to inputs well-formedness of VSID, the probability that the server outputs $b_j = 1$, in this case is negligible, $\mu(\lambda)$. 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 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. But, due to the security of SAP, its probability of success is negligible, $\mu(\lambda)$. Or 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 $\mu(\lambda)$ 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 $\mu(\lambda)$. 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; and also the arbiter is credited l coins, according to Equation 2. Note, 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 $\mu(\lambda)$. Also, due to the security of SAP, the adversary cannot block an honest server's messages, "pay" and $\ddot{x}_{cp}$, to the contract in the coin transfer phase. □

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 $coin^*_{\mathcal{C}} = z \cdot (Max(o_\beta, o_{|\beta-1|}) + Max(l_\beta, l_{|\beta-1|}))$ and $coin^*_{\mathcal{S}} = z \cdot (Max(l_\beta, l_{|\beta-1|}))$. Then, given the price list, z, $coin^*_{\mathcal{C}}$, and $coin^*_{\mathcal{S}}$, an adversary $\mathcal{A}$ cannot tell the value of β with 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 $coin'^*_{\mathcal{C}} = \frac{coin^*_{\mathcal{C}}}{z} = Max(o_\beta, o_{|\beta-1|}) + Max(l_\beta, l_{|\beta-1|})$. 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 encryption scheme is semantically secure, then RC-S-P preserves privacy, w.r.t. Definition 15.*

Proof (sketch). 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, $\mu(\lambda)$. 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 15 which lets the environment pick $\beta \xleftarrow{\$} \{0,1\}$. Also, let $\{\boldsymbol{q}_{j,0}, ..., \boldsymbol{q}_{j,m}\}$ 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 $\frac{1}{2} + \mu(\lambda)$, under the assumption that the size

of $\boldsymbol{q}_{j,\beta}$ is equal to the size of largest query size [3], i.e. $Max(|\boldsymbol{q}_{j,0}|, ..., |\boldsymbol{q}_{j,m}|) = |\boldsymbol{q}_{j,\beta}|$. The above assumption is relaxed with the use of a pad; as each encrypted query is padded to the queries' maximum size, i.e. $Max(|\boldsymbol{q}_{j,0}|, ..., |\boldsymbol{q}_{j,m}|)$, the adversary cannot tell with a probability greater than $\frac{1}{2} + \mu(\lambda)$ 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, for the sake of completeness we provide it. 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 15 in which the environment picks $\beta \xleftarrow{\$} \{0, 1\}$. Let $\{\pi_{j,0}, ..., \pi_{j,m}\}$ be proof list including all proofs of different sizes. In the experiment, if $\pi_{j,\beta}$ 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 $\frac{1}{2} + \mu(\lambda)$, if $Max(|\pi_{j,0}|, ..., |\pi_{j,m}|) = |\pi_{j,\beta}|$. 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 $\frac{1}{2} + \mu(\lambda)$ 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 adversary can tell the value of β is at most $\frac{1}{2} + \mu(\lambda)$.

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 and a *valid* query-proof pair is generated with probability Pr_1. 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 $Pr' = Max(Pr_0, Pr_1)$. 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. Therefore, 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 $Pr' + \mu(\lambda)$. □

7 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 generic recurring contingent service payment (RC-S-P), when the verifiable service is PoR. Nevertheless, RC-PoR-P offers two primary added features. Specifically, unlike RC-S-P, it (a) does not use any zero-knowledge proofs (even though either client or server can still be malicious) which significantly improves costs, and (b) has a much lower arbiter-side computation cost; as we will show later, this also allows a smart contract efficiently plays the arbiter's role. In the following, first we explain how the features are satisfied.

Avoiding the Use of Zero-knowledge Proofs. In general, the majority of PoR's are in the security model where a client is honest while the server is potentially malicious. They rely on metadata that is either a set of tags (e.g. MAC's or signatures) or a root of a Merkle tree, constructed on file blocks to ensure the file's availability. In the case where a client can also be malicious, if tags are used then using zero-knowledge proofs seem an obvious choice, as it allows the client to guarantee to the server that the tags have been constructed correctly without leaking verification keys. However, 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 zero-knowledge proofs and it supports proof of misbehaviour. Our first observation is that if a Merkle tree is used to generate a metadata, then there

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

would be no need for the client to use zero-knowledge proofs to prove the correctness of the metadata to the server. Instead, the server can efficiently check the metadata's correctness, by reconstructing the Merkle tree on top of the file blocks.

Reducing Arbiter-side Cost. As stated above, RC-PoR-P uses a Merkle tree-based PoR. In this case, each j-th proof contains a set of Merkle tree paths that are encoded and stored on a smart contract. For each verification, the client decoded and then verifies all paths. In a naive approach, when the client rejects proofs of j-th verification, it raises a dispute and let the arbiter verify the proofs, i.e. *all paths* in the proofs [4]. Instead, we use the idea of proof of misbehaviour, put forth in []. In particular, in j-th verification, if the client detects invalid proofs, then it sends details of only one invalid proof/path to the arbiter who decodes that single proof and checks its validity (as apposed to decoding and checking all proofs). This significantly reduces the arbiter computation cost.

To present RC-PoR-P protocol, we will use the same approach used to present RC-S-P. In particular, first we present the verifiable service, that is a (modified) Merkle tree-based PoR. Then, we upgrade it to the one that supports identifiable abort, denoted by PoRID. Next, we use PoRID to build RC-PoR-P.

7.1 Modified Merkle tree-based PoR

In this section, we present a modified version of the standard Merkle tree-based PoR, detonated by PoR. At a high level, the protocol works as follows. The client encodes its file using an error-correcting code, splits the encoded file into blocks, and constructs a Merkle tree on top of the blocks. It locally keeps the tree's root and sends the blocks to the server who rebuilds the tree on the blocks. At a verification time, the client sends a pseudorandom function's key to the server who derives a predetermined number of pseudorandom indices of the blocks, that indicates which blocks have been challenged. The server for each challenged block generates a Merkle tree proof and sends all proofs to the client. The client, given the root and key, verifies all proofs. If all proofs are accepted, then the client outputs 1 and concludes that its file is retrievable (with a high probability). However, if it rejects a set of proofs, it outputs 0 along with an index of the challenged block whose proof was rejected. In the following, we first present the protocol and then elaborate on the modifications we have applied.

1. **Client-side Setup**. $\texttt{PoR.setup}(1^\lambda, u)$
   (a) Uses an error correcting code, e.g. Reed-Solomon codes, to encode the file: u. Let u' be the encoded file. It splits u' into blocks as follows, $u^* = u'_0||0, ..., u'_m||m$
   (b) Generates metadata: σ, by constructing Merkle tree on blocks of u^*, i.e. $\texttt{MT.genTree}(u^*)$. Let σ be the root of the resulting tree, and β be a security parameter. It sets public parameters as $pp := (\sigma, \beta, m, \zeta)$, where $\zeta := (\psi, \eta, \iota)$ is a PRF's description, as it was defined in Section 2
   (c) Sends pp and u^* to $\mathcal{S}$

2. **Client-side Query Generation**. $\texttt{PoR.genQuery}(1^\lambda, pp)$
   (a) Picks a random key $\hat{k}$ of a pseudorandom function PRF, i.e. $\hat{k} \xleftarrow{\$} \{0,1\}^\psi$. It ensures the function outputs distinct values, i.e. $\forall i, j \in [0, m] : (\texttt{PRF}(\hat{k}, i) \bmod m + 1) \neq (\texttt{PRF}(\hat{k}, j) \bmod m + 1)$, where $i \neq j$
   (b) It sends $\hat{k}$ to $\mathcal{S}$

3. **Server-side Proof Generation**. $\texttt{PoR.prove}(u^*, \hat{k}, pp)$
   (a) Derives β pseudorandom indices from $\hat{k}$ as follows. $\forall i, 1 \leq i \leq \beta : q_i = \texttt{PRF}(\hat{k}, i) \bmod m + 1$. Let $\boldsymbol{q} = [q_1, ..., q_\beta]$
   (b) For each random index q_i, generates a Merkle tree proof: π_{q_i}, by running Merkle tree proof generator function on u^*, i.e. $\texttt{MT.prove}(u^*, q_i)$. The final result is $\boldsymbol{\pi} = [(u^*_{q_i}, \pi_{q_i})]_{q_i \in \boldsymbol{q}}$, where i-th element in $\boldsymbol{\pi}$ corresponds to i-th pseudorandom value: q_i and each π_{q_i} is path in the tree that proves its corresponding block: $u^*_{q_i}$ is a leaf node of the tree.

[4] In a Merkle tree-based PoR, the number of proofs that are sent to a verifier for each verification is linear with the number of challenges, e.g. 460 challenges to ensure 99% of file blocks is retrievable. In contrast, in a tag-based PoR, in each verification, the verifier receives only a few proofs.

(c) Sends $\boldsymbol{\pi}$ to $\mathcal{C}$

4. **Client-side Proof Verification**. $\texttt{PoR.verify}(\boldsymbol{\pi}, \boldsymbol{q}, pp)$
   (a) If $|\boldsymbol{\pi}| = |\boldsymbol{q}| = 1$, then set $\beta = 1$. This step is taken only in the case where single proof and query is provided to a third-party verifier (e.g. in the case of proof of misbehaviour).
   (b) Checks if the server has sent proofs related to all challenged file blocks. To do that, for all i (where $1 \leq i \leq \beta$), it first parses every element of π as follows, $\texttt{parse}(u^*_{q_i}) = u'_{q_i}||q_i$, and then checks if its index: q_i equals i-th element of $\boldsymbol{q}$. If all checks pass, then it proceeds to the next step. Otherwise, it outputs $\boldsymbol{d} : [0, i]$, where i refers to the index of the element in $\boldsymbol{\pi}$ that does not pass the check.
   (c) Checks if every path in $\boldsymbol{\pi}$ is valid and corresponds to the root, by calling $\texttt{MT.verify}(u^*_{q_i}, \pi_{q_i}, \sigma)$. If all checks pass, it outputs $\boldsymbol{d} = [1, \perp]$ (where $\perp$ denotes empty); otherwise, it outputs $\boldsymbol{d} : [0, i]$, where i refers to the index of the element in $\boldsymbol{\pi}$ that does not pass the check.

Theorem 3. *The PoR scheme, presented in Section 7.1, is ϵ-sound, w.r.t. Definitions 2, if Merkle tree and pseudorandom function* PRF*, are secure.*

The above protocol differs from the standard Merkle tree-based PoR from two perspectives; First and far most, in step 4 in addition to outputting a binary value, the client outputs only one index of a rejected proof. This will enable any third-party who is given that index (and vectors of proofs and challenges) to verify the client's claim by checking only that proof, i.e. proof of misbehaviour. Second, in step 2a instead of sending β challenges, we allow the client to send only a key of a pseudorandom function to the server who can derive a set of challenges form it. This will ultimately lead to a decrease in costs too, i.e. the client's communication and a smart contract's storage costs.

Proof (sketch). 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 and its security proof is similar to the existing Merkle tree-based PoR schemes, e.g. [9, 18, 11]. Alternatively, our protocol can be proven based on the security analysis of the PoR schemes that use MACs or BLS signatures, e.g. [21]. In this case, the extractor design (in the Merkle tree-based PoR) would be simpler as it does not need to extract blocks from a linear combination of MAC's or signatures, as the blocks are included in PoR proofs, i.e. they are part of the Merkle tree proofs.

Intuitively, in either case, the extractor interacts with any adversarial prover that passes 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. Due to the security, i.e. authenticity, of the Merkle tree, the retrieved values are the valid and correct file blocks and due to security of the pseudorandom function, the challenges (or the function's outputs) are not predictable. After collecting sufficient number of blocks, the extractor can use the error correcting code to decode and recover the entire file blocks, given the retrieved ones. □

Remark 11. Recall, the generic definition of a verifiable service scheme (i.e. Definition 6) involves three algorithms: F, M, and Q. However, the three algorithms are implicit in the original definition of PoR and accordingly in PoR protocols. In the following, we explain how each algorithm is defined in PoR context. M is an algorithm that processes a file and generates metadata. For instance, when PoR uses a Merkle tree (to ensure the file's integrity and availability), then M refers to the Merkle tree's algorithm that constructs a tree on top of the file blocks. Also, F is an algorithm that, during generating a PoR proof, processes a subset of the outsourced file, given the client's query (or challenged file blocks). For instance, if a PoR utilises a Merkle tree, then F refers to the algorithm that generates Merkle tree's proofs, i.e. membership of the challenged file blocks. Furthermore, Q can be a pseudorandom function that generates a set of pseudorandom strings in a certain range, e.g. file block's indices.

7.2 PoRID Protocol

In this section, we propose "PoR with identifiable abort" (PoRID) that is a concrete instantiation of $\text{VSID}_{\text{light}}$. It is built upon the PoR protocol, presented in the previous section and is in the same security model as $\text{VSID}_{\text{light}}$ is, i.e. either $\mathcal{C}$ or $\mathcal{S}$ can be malicious. In PoRID similar to $\text{VSID}_{\text{light}}$, $\mathcal{C}$ and $\mathcal{S}$ use a bulletin board to exchange signed messages. In the protocol, at setup $\mathcal{C}$ encodes its file and generates public parameters and metadata. It posts the public parameters and metadata to the bulletin board and sends the encoded file to $\mathcal{S}$ who runs a few lightweight checks to ensure the correctness of the public parameters and metadata. It agrees to serve, if it is convinced of their correctness. Later, when $\mathcal{C}$ wants to ensure the availability of its outsourced file, it generates and posts a query to the board. $\mathcal{S}$ checks the correctness of the query, by performing a couple of highly efficient verifications.

The server-side prove and client-side verify algorithms are similar to those in PoR with a difference that $\mathcal{S}$ posts the PoR proofs (i.e. output of prove algorithm) to the board. In case of any dispute, $\mathcal{C}$ or $\mathcal{S}$ invokes the arbiter who, given the signed posted messages, checks the proofs to identify a corrupt party. In particular, it first checks the validity of the query (regardless of the party who invokes it). However, if is invoked by $\mathcal{C}$, it also checks only one of the PoR proofs that the client claims it is invalid. Thus, it is much more efficient than $\text{VSID}_{\text{light}}$ as it does not need any zero-knowledge proofs (mainly due to the use of Merkle tree) and requires the arbiter to check only one of the proofs (due to the idea of proof of misbehaviour). PoRID protocol is presented below.

1. **Client-side Setup**. $\texttt{PoRID.setup}(1^\lambda, u)$
   (a) Calls $\texttt{PoR.setup}(1^\lambda, u) \rightarrow (u^*, pp)$, that results in public parameters $pp := (\sigma, \beta, m, \zeta)$ and encoded file: $u^* = u'_0||0, ..., u'_m||m$. Recall, $\zeta := (\psi, \eta, \iota)$ is the PRF's description.
   (b) Posts pp to the bulletin board and sends u^* to $\mathcal{S}$

2. **Server-side Setup**. $\texttt{PoRID.serve}(u^*, pp)$
   Verifies the correctness of public parameters:
   (a) rebuilds the Merkle tree on u^* and checks the resulting root equals σ
   (b) checks $|u^*| = m$ and $\beta \leq m$
   If the proofs are accepted, then it outputs $a = 1$ and proceeds to the next step; otherwise, it outputs $a = 0$ and halts.

3. **Client-side Query Generation**. $\texttt{PoRID.genQuery}(1^\lambda, pp)$
   (a) Calls $\texttt{PoR.genQuery}(1^\lambda, pp) \rightarrow \hat{k}$, to generate a key, $\hat{k}$
   (b) Posts $\hat{k}$ to the board.

4. **Server-side Query Verification**. $\texttt{PoRID.checkQuery}(\hat{k}, pp)$
   (a) Checks if $\hat{k}$ is not empty, i.e. $\hat{k} \neq \perp$, and is in the key's universe, i.e. $\hat{k} \in \{0,1\}^\psi$
   (b) If the checks pass, then it outputs $b = 1$; otherwise, it outputs $b = 0$

5. **Server-side Service Proof Generation**. $\texttt{PoRID.prove}(u^*, \hat{k}, pp)$
   (a) Calls $\texttt{PoR.prove}(u^*, \hat{k}, pp) \rightarrow \boldsymbol{\pi}$, to generate proof vector: $\boldsymbol{\pi}$
   (b) Posts $\boldsymbol{\pi}$ to the board.

6. **Client-side Proof Verification**. $\texttt{PoRID.verify}(\boldsymbol{\pi}, \hat{k}, pp)$
   Calls $\texttt{PoR.verify}(\boldsymbol{\pi}, \hat{k}, pp) \rightarrow \boldsymbol{d}$, to verify the proof. If $\boldsymbol{d}[0] = 1$, it accepts the proof; otherwise, it rejects it.

7. **Arbiter-side Identification**. $\texttt{PoRID.identify}(\boldsymbol{\pi}, g, \hat{k}, pp)$
   This algorithm can be invoked by $\mathcal{C}$ or $\mathcal{S}$, in the case of dispute. If it is invoked by $\mathcal{C}$, then g refers to a rejected proof's index; however, if it is invoked by $\mathcal{S}$, then g is null, i.e. $g = \perp$. The arbiter performs as follows.
   (a) Ensures query $\hat{k}$ is well-structured by calling $\texttt{PoRID.checkQuery}(\hat{k}, pp)$. If it returns $b = 0$, then it outputs $I = \mathcal{C}$ and halts; otherwise, it proceeds to the next step.
   (b) Derives the related challenged block's index from $\hat{k}$, by computing $q_g = \texttt{PRF}(\hat{k}, g) \bmod m + 1$

(c) If $g \neq \perp$, then verifies only g-th proof, by setting $\hat{\boldsymbol{\pi}} = \boldsymbol{\pi}[g], \hat{\boldsymbol{q}} = q_g$ and then calling $\texttt{PoR.verify}(\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{q}}, pp) \rightarrow \boldsymbol{d}'$. If $\boldsymbol{d}'[0] = 0$, then it outputs $I = \mathcal{S}$. Otherwise, it outputs $I = \perp$

Theorem 4. *The PoRID protocol satisfies the ϵ-soundness, inputs well-formedness, and detectable abort properties, w.r.t. Definitions 2, 9, and 10, if PoR is ϵ-sound and the signature scheme is secure.*

Proof (sketch). The ϵ-soundness of PoRID directly stems from the security of PoR scheme, i.e. ϵ-soundness. Specifically, in PoRID the (honest) client makes black-box calls to the algorithms of PoR, to ensure the soundness. The latter scheme's soundness ensures that an extractor can recover the entire file interacting with a corrupt server who passes ϵ fraction of challenges. On the other hand, the inputs well-formedness holds for the following reasons. The metadata generation algorithm, i.e. the Merkle tree algorithm that builds a tree and computes a root, is deterministic and involves only public parameters. Thus, given the tree's leaves (i.e. file blocks), its parameters, and the root, anyone can reconstruct it, check if it yields the same root, and verify the tree's parameters. Also, a query contains a single random key, $\hat{k}$, whose correctness can be checked deterministically, i.e. by checking $\hat{k} \neq \perp$ and $\hat{k} \in \{0,1\}^{\psi}$. The detectable abort property holds as long as the soundness and inputs well-formedness hold and the signature scheme is secure. The reason is that algorithm $\texttt{PoRID.identify}(.)$, which ensures detected abort, is a wrapper function that makes black-box calls to algorithms $\texttt{PoRID.checkQuery}(.)$ and $\texttt{PoR.verify}(.)$, where the former ensures input (i.e. query) well-formedness, and the latter ensures soundness. Although the number of proofs that are passed to $\texttt{PoR.verify}(.)$ in phase 6, differs from the number of proofs passed to the same algorithm in phase 7, the difference does not affect the security, as due to the security of PoR (i.e. Merkle tree) an invalid proof is detected with the same probability in both phases. The signature's security ensures if a proof is not signed correctly, then it can also be rejected by the arbiter and the signer is held accountable for providing an ill-formed message; on the other hand, if a proof is signed correctly, then it cannot be repudiated by the signer later on that guarantees the signer is held accountable for a rejected proof it provides. □

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

In this section, we present RC-PoR-P, a concrete instantiation of the generic RC-S-P. Although RC-PoR-P and RC-S-P have some overlaps, they have many differences too. Therefore, we provide the protocol's overview and its detailed description below. At a high level the protocol works as follows. The client and server utilise SAP to provably agree on two private statements, one statement includes payment details, and another one specifies a secret key, $\bar{k}$, and a pad's details, that will be used to encode sensitive messages they send to the contract. Moreover, they agree on public parameters such as the private time bubble's length (that is the total number of billing cycles: z, plus a waiting period, $\texttt{H}$) and a smart contract that specifies z and the total amount of masked coins each party should deposit. They deploy the contract. Each party deposits its masked coins in the contract within a fixed time. If a party does not deposit enough coins on time, then the parties have a chance to withdraw their coins and terminate the contract after a certain time.

To start using/providing the service, the client invokes $\texttt{PoRID.setup}(.)$ to encode the file and generate metadata and public parameters. It sends encryption of the metadata and public parameters to the smart contract. Also, it sends the encoded file to the server who decrypts them and using the encoded file checks their correctness by calling $\texttt{PoRID.server}(.)$. If the server decides not to serve, it sends to the contract 0 within a fixed time; in this case, the parties can withdraw their deposit and terminate the contract. At the end of each billing cycle, the client generates a query, by calling $\texttt{PoRID.genQuery}(.)$ and sends the query's encryption to the contract. In the same cycle, the server retrieves the query from the contract, decrypts and locally checks its correctness, by calling $\texttt{PoRID.checkQuery}(.)$. If the query is rejected, the server locally stores the index of that billing cycle and generates dummy PoR proofs. However, if the server accepts the query, it generates PoR proofs by calling $\texttt{PoRID.prove}(.)$. Then, in either case, the server encrypts the proofs, pads them and sends the result to the contract. Next, the client removes the pads, decrypts the proofs and locally verifies them, by calling $\texttt{PoRID.verify}(.)$. If the verification is passed, then the client knows the file is retrievable with a high probability. But, if the proof is rejected, then it locally stores the index of that billing cycle and details of one of the invalid proofs (in the same cycle). It waits until the private time bubble passes and dispute resolution time arrives.

During the dispute resolution period, in case the server rejects the query or the client rejects the PoR proofs, that party sends to the arbiter (a) details of invalid proofs and (b) the statement that contains the decryption key and padding

detail. The arbiter checks the validity of the statement first. If it accepts the statement, then it removes the pads and decrypts the values whose indices were provided by the parties. Then, the arbiter checks the party's claim by calling $\texttt{PoRID.checkQuery}(.)$ and $\texttt{PoRID.identify}(.)$ if the server or client calls the contract respectively. The arbiter also keeps track of the number of times each party misbehaved. After the arbiter processes the parties' claim, it tells to the contract how many times each party misbehaved. In the next phase, to distribute the coins, either client or server sends to the contract: (a) "pay" message, (b) the agreed statement that specifies the payment details, and (c) the statement's proof. The contract verifies the statement and if it is approved, then it distributes the coins according to the statement's detail, and the number of times each party misbehaved.

1. **Key Generation**. $\texttt{RCPoRP.keyGen}(1^\lambda)$
   (a) $\mathcal{C}$ picks a random secret key $\bar{k}$ for a symmetric key encryption. It also sets parameter pad_π which is the number of dummy values that will be used to pad encrypted proofs, let $qp := (pad_\pi, \bar{k})$. The key's size is part of the security parameter. Let $k := (sk', pk')$, where $sk' = qp$ and $pk' := (adr_\mathcal{C}, adr_\mathcal{S})$.

2. **Client-side Initiation**. $\texttt{RCPoRP.cInit}(1^\lambda, u, k, z, pl)$
   (a) Calls $\texttt{PoRID.setup}(1^\lambda, u) \rightarrow (u^*, pp)$ to encode service input. It 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 (given price list pl) as follows, o: the amount of coins for each accepting proof, and l the amount of coin $\mathcal{C}$ or $\mathcal{S}$ needs to send to a smart contract to resolve a potential dispute.
   (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 designs a smart contract, SC, that explicitly specifies parameters z, $coin^*_\mathcal{C}$, $p_\mathcal{S}$, adr_{SAP_1}, adr_{SAP_2}, and pk'. It sets a set of time points/windows, $\texttt{Time} : \{\texttt{T}_0, ..., \texttt{T}_3, \texttt{G}_{1,1}, ..., \texttt{G}_{z,2}, \texttt{H}, \texttt{K}_1, ..., \texttt{K}_6, \texttt{L}\}$, that are explicitly specified in SC who will accept a certain party's message only in a specified time point/window. Time allocations 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 the 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 $\boldsymbol{y} : [y_\mathcal{C}, y'_\mathcal{C}, y_\mathcal{S}, y'_\mathcal{S}, \texttt{Time}, adr_{\text{SC}}]$
   (g) Deposits $coin^*_\mathcal{C}$ coins in the contract. It sends $u^*, \ddot{x}_{qp}, \ddot{x}_{cp}$, and $p_\mathcal{S}$ (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 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, 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 $\texttt{PoRID.serve}(u^*, pp) \rightarrow a$.
   (d) Sends a and $coin^*_\mathcal{S} = p_\mathcal{S}$ coins to SC at time $\texttt{T}_1$, where $coin^*_\mathcal{S} = \perp$ if $a = 0$

   Note, $\mathcal{S}$ 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 $\mathcal{C}$ simply sends a "pay" message to $\texttt{RCPoRP.pay}(.)$ algorithm only at time $\texttt{T}_2$.

   Billing-cycles Onset. $\mathcal{C}$ and $\mathcal{S}$ engage in the following three phases, i.e. phase 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 $\texttt{PoRID.genQuery}(1^\lambda, pp) \rightarrow \hat{k}_j$ to generate a query, where $pp \in T_{qp}$

(b) Sends encrypted query $\hat{k}_j^* = \texttt{Enc}(\bar{k}, \hat{k}_j)$ to SC at time $\texttt{G}_{j,1}$

5. **Server-side Proof Generation**. $\texttt{RCPoRP.prove}(u^*, \hat{k}_j^*, T_{qp})$
   (a) Constructs an empty vector, $\boldsymbol{m}_{\mathcal{S}} = \perp$, if $j = 1$.
   (b) Decrypts the query, $\hat{k}_j = \texttt{Dec}(\bar{k}, \hat{k}_j^*)$
   (c) Calls $\texttt{PoRID.checkQuery}(\hat{k}_j, pp) \rightarrow b_j$ to check the query's correctness.
   - If it accepts the query, then it calls $\texttt{PoRID.prove}(u^*, \hat{k}_j, pp) \rightarrow \boldsymbol{\pi}_j$, to generate a PoR proof. In this case, $\mathcal{S}$ encrypts every proof in the proof vector, i.e. $\forall g, 1 \leq g \leq |\boldsymbol{\pi}_j| : \texttt{Enc}(\bar{k}, \boldsymbol{\pi}_j[g]) = \boldsymbol{\pi}'_j[g]$, where $\bar{k} \in T_{qp}$. Let vector $\boldsymbol{\pi}'_j$ contain the encryption of all proofs. It pads every encrypted proof in $\boldsymbol{\pi}'_j$ with $pad_{\pi} \in T_{qp}$ random values that are picked from the encryption's output range U, (by appending the random values to the encrypted proofs vector). Let $\boldsymbol{\pi}^*_j$ be the result. It sends the padded encrypted proofs to SC at time $\texttt{G}_{j,2}$
   - Otherwise (if $\mathcal{S}$ rejects the query), it appends j to $\boldsymbol{m}_{\mathcal{S}}$, constructs a dummy proof $\boldsymbol{\pi}'_j$ whose elements are randomly picked from U, pads the result as above, and sends the result, $\boldsymbol{\pi}^*_j$, to SC at time $\texttt{G}_{j,2}$

   When $j = z$ and $\boldsymbol{m}_{\mathcal{S}} \neq \perp$, it sets $m_{\mathcal{S}} := \boldsymbol{m}_{\mathcal{S}}$.

6. **Client-side Proof Verification**. $\texttt{RCPoRP.verify}(\boldsymbol{\pi}^*_j, \hat{k}^*_j, T_{qp})$
   (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 proofs $\texttt{Dec}(\bar{k}, \boldsymbol{\pi}'_j) = \boldsymbol{\pi}_j$ and then calls $\texttt{PoRID.verify}(\boldsymbol{\pi}_j, \hat{k}_j, pp) \rightarrow \boldsymbol{d}_j$, to verify the proof, where $\hat{k}_j = \texttt{Dec}(\bar{k}, \hat{k}^*_j)$.
   - If $\boldsymbol{\pi}_j$ passes the verification, i.e. $\boldsymbol{d}_j[0] = 1$, then $\mathcal{C}$ concludes that the service for this verification has been delivered successfully.
   - Otherwise (if proof $\boldsymbol{\pi}_j$ is rejected, i.e. $\boldsymbol{d}_j[0] = 0$), it sets $g = \boldsymbol{d}_j[1]$ and appends vector $[j, g]$ to $\boldsymbol{m}_{\mathcal{C}}$. Recall, $\boldsymbol{d}_j[1]$ refers to a rejected proof's index in proof vector $\boldsymbol{\pi}_j$.

   When $j = z$ and $\boldsymbol{m}_{\mathcal{C}} \neq \perp$, $\mathcal{C}$ sets $m_{\mathcal{C}} := \boldsymbol{m}_{\mathcal{C}}$.

7. **Dispute Resolution**. $\texttt{RCPoRP.resolve}(m_{\mathcal{C}}, m_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{q}^*, T_{qp})$
   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 sets counters: $y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}$ and $y'_{\mathcal{S}}$, that are initially set to 0, before time $\texttt{K}_1$, where $\texttt{K}_1 > \texttt{G}_{z,2} + \texttt{H}$.
   (b) $\mathcal{S}$ sends $m_{\mathcal{S}}$ and $\ddot{x}_{qp}$ to the arbiter, at time $\texttt{K}_1$.
   (c) The arbiter after receiving $m_{\mathcal{S}}$, does the following at time $\texttt{K}_2$.
      i. Checks the validity of statement $\ddot{x}_{qp}$, by sending it to SAP contract which returns 1 or 0. If the output is 0, then it 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 is 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 $\hat{k}^*_i \in \boldsymbol{q}^*$ from SC, and decrypts it, $\hat{k}_i = \texttt{Dec}(\bar{k}, \hat{k}^*_i)$
         - Checks if the query is well-formed, by calling $\texttt{PoRID.checkQuery}(\hat{k}_i, pp) \rightarrow b_i$. If the query is rejected, i.e. $b_i = 0$, then it increments $y_{\mathcal{C}}$ by 1 and appends i to $\boldsymbol{v}$. Otherwise (if the query is accepted) it increments $y'_{\mathcal{S}}$ by 1.

      Let $\texttt{K}_3$ be the time the arbiter finishes the above checks.
   (d) $\mathcal{C}$ sends $m_{\mathcal{C}}$ and $\ddot{x}_{qp}$ to the arbiter, at time $\texttt{K}_4$
   (e) The arbiter after 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 SAP contract which returns either 1 or 0. If the output is 0, then it 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 ensures 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 $\boldsymbol{m}[0] \in \boldsymbol{v}$. Note the latter check (i.e. $\boldsymbol{m}[0] \in \boldsymbol{v}$) ensures $\mathcal{C}$ cannot hold $\mathcal{S}$ accountable if $\mathcal{C}$ generated an ill-formed query for the same verification.

iii. For every vector $\boldsymbol{m} \in \boldsymbol{m}_{\mathcal{C}}$:
- Retrieves details of a proof that was rejected in each i-th verification. In particular, it sets $i = \boldsymbol{m}[0]$ and $g = \boldsymbol{m}[1]$. Recall that g refers to the index of a rejected proof in the proof vector which was generated for i-th verification, i.e. $\boldsymbol{\pi}_i$
- Fetches the related encrypted query $\hat{k}_i^* \in \boldsymbol{q}^*$ from SC, and decrypts it, $\hat{k}_i = \texttt{Dec}(\bar{k}, \hat{k}_i^*)$
- Removes the pads only from g-th padded encrypted proof. Let $\boldsymbol{\pi}_i'[g]$ be the result. Next, it decrypts the encrypted proof, $\texttt{Dec}(\bar{k}, \boldsymbol{\pi}_i'[g])) = \boldsymbol{\pi}_i[g]$
- Constructs a fresh vector: $\boldsymbol{\pi}_i''$, such that its g-th element equals $\boldsymbol{\pi}_i[g]$ (i.e. $\boldsymbol{\pi}_i''[g] = \boldsymbol{\pi}_i[g]$ and $|\boldsymbol{\pi}_i''| = |\boldsymbol{\pi}_i|$) and the rest of its elements are dummy values.
- Calls $\texttt{PoRID.identify}(\boldsymbol{\pi}_i'', g, \hat{k}_i, pp) \rightarrow I_i$. If $I_i = \mathcal{S}$, then it increments $y_{\mathcal{S}}$ by 1. If $I_i = \perp$, then 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 who accordingly overwrites the elements it holds (i.e. elements of $\boldsymbol{y}$) by the related vectors elements the arbiter sent.

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), 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 $\texttt{L} > \texttt{K}_6$
   (c) SC checks the validity of the statement by sending it to 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:
   - $coin_{\mathcal{C}}^* - o(z - y_{\mathcal{S}}) - l(y_{\mathcal{C}} + y_{\mathcal{C}}')$ coins to $\mathcal{C}$
   - $coin_{\mathcal{S}}^* + o(z - y_{\mathcal{S}}) - l(y_{\mathcal{S}} + y_{\mathcal{S}}')$ coins to $\mathcal{S}$
   - $l(y_{\mathcal{S}} + y_{\mathcal{C}} + y_{\mathcal{S}}' + y_{\mathcal{C}}')$ coins to the arbiter.

Remark 12. The reason in step 7(e)iii vector $\boldsymbol{\pi}_i''$ is constructed is to let SC make *black-box* use of $\texttt{PoRID.identify}(.)$. Alternatively, SC could decrypt all proofs in $\texttt{Enc}(\bar{k}, \boldsymbol{\pi}_i)$ and pass them to $\texttt{PoRID.identify}(.)$. However, this approach would impose a high cost, as all proofs have to be decrypted.

Remark 13. In general, a transaction that is sent to a smart contract should cover the cost of the contract's execution. Therefore, in the above protocol, if a party unnecessarily invokes a contract for an accepting proof, it has to pay the execution cost in advance. This is the reason the above protocol (unlike RC-S-P protocol) does not need to track the number of times a party unnecessarily invokes the contract.

Remark 14. In the protocol, for the sake of simplicity, it is assumed that the cost imposed by a malicious client to the arbiter (to resolve a dispute) is the same as the cost imposed by a malicious server. To relax the assumption, we can simply introduce another parameter l'. We let l and l' be the amounts of coin a malicious client and malicious server must pay to the arbiter respectively. In this case, (a) in step 2d, the client appends l' to cp and (b) in the coin transfer phase, the amounts of coin each party receives would be as follow: $coin_{\mathcal{C}}^* - o(z - y_{\mathcal{S}}) - l(y_{\mathcal{C}} + y_{\mathcal{C}}')$ coins to $\mathcal{C}$, $coin_{\mathcal{S}}^* + o(z - y_{\mathcal{S}}) - l'(y_{\mathcal{S}} + y_{\mathcal{S}}')$ coins to $\mathcal{S}$, and $l(y_{\mathcal{C}} + y_{\mathcal{C}}') + l'(y_{\mathcal{S}} + y_{\mathcal{S}}')$ coins to the arbiter.

Delegating the Arbiter's Role to a Smart Contract In the above protocol, due to the efficiency of arbiter-side algorithm, i.e. $\texttt{RCSPoR.resolve}(.)$, we can totally 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 amounts of coin each party receives in the coin distribution phase. Recall, in the RC-PoR-P and RC-S-P (presented in sections 7.3 and 6.2 respectively) the party who raises dispute does not pay the arbiter when it sends to it a 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 who raises dispute and sends a dispute query to the contract (due to the nature of smart contracts' platform) has to pay the contract before the contract processes its query. This means, an honest party who 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 amounts of coin 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 A, 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 B, we elaborate on how the new variant of RC-PoR-P can be constructed and we prove its security.

7.4 Security Analysis of RC-PoR-P

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

Theorem 5. *The RC-PoR-P protocol is secure, w.r.t. Definition 16, if PoRID, SAP, and signature scheme are secure and the encryption scheme is semantically secure.*

To prove the above theorem, we show that RC-PoR-P meets all security properties defined in Section 4.3. Since RC-PoR-P is instantiation of RC-S-P, its proof has similarities with the latter one. Nevertheless, for the sake of completeness we provide its proof. We start by proving that RC-PoR-P satisfies security against a malicious server.

Lemma 5. *If SAP and signature scheme are secure and PoRID scheme supports correctness, soundness, and detectable abort, then RC-PoR-P is secure against malicious server, w.r.t. Definition 13.*

Proof (sketch). First, we consider event $\Big(F(u^*, \boldsymbol{q}_j, pp) = h_j \wedge \Big((coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o) \vee (coin_{\mathcal{A}r,j} \neq l \wedge y'_{\mathcal{S},j} = 1)\Big)\Big)$ 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. $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o$, or it makes the arbiter withdraw an incorrect amount of coins, i.e. $coin_{\mathcal{A}r,j} \neq l$, if it unnecessarily invokes the arbiter. Because the proof is valid, an honest client accepts it and does not raise a dispute. But, the server could make the client withdraw incorrect amounts of coins, if it manages to 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 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, or send a message on the client's behalf to unnecessarily invoke the arbiter. Nevertheless, it cannot falsely accuse the client of misbehaviour. Because, due to the security of SAP, it cannot convince the arbiter to accept a different decryption key (that will be used to decrypt queries) other than what was agreed with the client in the initiation phase. In particular, it cannot persuade the arbiter to accept $\ddot{x}'_{qp}$, where $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$, except with a negligible probability, $\mu(\lambda)$. This ensures that the honest client's queries are accessed by the arbiter with a high probability. Furthermore, if the adversary provides a valid statement, i.e. $\ddot{x}_{qp}$, then due to PoRID's correctness, algorithm `PoRID.identify`(.) outputs $I_j = \perp$. Therefore, due to the security of SAP and correctness of PoRID, the following holds $y_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$. Furthermore, because of SAP's security, 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 a negligible probability, $\mu(\lambda)$. 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 a negligible probability $\mu(\lambda)$; so with a high probability $y'_{\mathcal{C},j} = 0$. Recall, in RC-PoR-S or RC-S-P protocol, according to Equation 1, the amounts of coin that should be credited to the client for j-th verification is $coin_{\mathcal{C},j} = \frac{coin^*_{\mathcal{C}}}{z} - o(1 - y_{\mathcal{S},j}) - l(y_{\mathcal{C},j} + y'_{\mathcal{C},j})$. 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 j-th verification, with a high probability. Furthermore, as stated above, if the adversary invokes the arbiter, the arbiter with a high probability outputs $I_j = \perp$ that yields $y'_{\mathcal{S},j} = 1$. In RC-PoR-P or RC-S-P protocol, according to Equation 2, the amounts of coin the arbiter should be credited for j-th verification is $coin_{\mathcal{A}r,j} = l(y_{\mathcal{S},j} + y_{\mathcal{C},j} + y'_{\mathcal{S},j} + y'_{\mathcal{C},j})$. 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 j-th verification if it is unnecessarily invoked by the adversary. In this case, for the server to make the arbiter withdraw other than that amounts, it has to send 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. But, as stated above, it cannot succeed with a probability significantly greater than $\mu(\lambda)$.

We now move on to event $\Big(\Big(F(u^*, \boldsymbol{q}_j, pp) \neq h_j\Big) \wedge \Big(d_j = 1 \vee y_{\mathcal{S},j} = 0 \vee coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} \vee coin_{\mathcal{A}r,j} \neq l\Big)\Big)$ which captures the case where the server provides an invalid proof (i.e. PoR) but it either convinces the client to accept the proof, or persuades the arbiter to accept it or makes the client or arbiter withdraw incorrect amounts of coin, i.e. $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z}$ or $coin_{\mathcal{A}r,j} \neq l$ respectively. Due to the soundness of PoRID, the probability that the adversary can convince an honest client to accept invalid proof is negligible, $\mu(\lambda)$. Therefore, the client outputs $d_j = 0$ with a high probability and raises a dispute. Furthermore, the server may try to make the arbiter keep $y_{\mathcal{S},j} = 0$. For the adversary to succeed, it has to make the arbiter identify the client as the misbehaving party, and output $y_{\mathcal{C},j} = 1$. In this case, according to RC-PoR-P protocol, the client's complaint (for j-th verification) would not be processed by the arbiter. This allows $y_{\mathcal{S},j}$ to remain 0. But, as we argued above, the probability that the adversary makes the arbiter recognise the client as misbehaving is at most $\mu(\lambda)$. So, with a high probability $y_{\mathcal{S},j} = 1$ 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. Moreover, due to SAP's security, the probability that the adversary succeeds in changing the payment parameters to make the client or arbiter withdraw incorrect amounts of coin is negligible too. So, according to Equations 1 and 2 the client and arbiter are credited $\frac{coin^*_{\mathcal{C}}}{z}$ and l coins for j-th verification respectively. Also, due to the security of SAP, the adversary cannot block an honest client's messages, "pay" and $\ddot{x}_{cp}$, to the contract in the coin transfer phase. □

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

Lemma 6. *If SAP and signature scheme are secure and PoRID scheme supports correctness, inputs well-formedness, and detectable abort, then RC-PoR-P is secure against malicious client, w.r.t. Definition 14.*

Proof (sketch). We first consider event $\Big(\Big(M(u^*, k, pp) = \sigma \wedge Q(\text{aux}, k, pp) = \boldsymbol{q}_j\Big) \wedge \Big((coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o) \vee (coin_{\mathcal{A}r,j} \neq l \wedge y'_{\mathcal{C},j} = 1)\Big)\Big)$. 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. $coin_{\mathcal{A}r,j} \neq l$, 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 incorrect amounts of coin, 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 PoRID, algorithm `PoRID.identify`(.) outputs $I_j = \perp$. So, due to the security of SAP and correctness of PoRID, 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 $coin_{\mathcal{S},j} = \frac{coin^*_{\mathcal{S}}}{z} + o(1 - y_{\mathcal{S},j}) - l(y_{\mathcal{S},j} + y'_{\mathcal{S},j})$. Therefore, given $y_{\mathcal{S},j} = y'_{\mathcal{S},j} = 0$, the server is credited $\frac{coin^*_{\mathcal{S}}}{z} + o$ coins for 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 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 an ill-formed metadata. But, due to PoRID's inputs well-formedness, 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 $\Big((Q(\text{aux}, k, pp) \neq \boldsymbol{q}_j) \wedge (b_j = 1 \vee y_{\mathcal{C},j} = 0 \vee coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o \vee coin_{\mathcal{A}r,j} \neq l)\Big)$ 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 $coin_{\mathcal{A}r,j} \neq l$ respectively. But, due to PoRID's inputs well-formedness, the probability that the server outputs $b_j = 1$ is negligible $\mu(\lambda)$. Note, when the honest server rejects the query and raises a dispute, the arbiter checks the query and sets $y_{\mathcal{C},j} = 1$. 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 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. □

In the following, we prove RC-PoR-P's privacy. Since the proof has many similarities with the proof of RC-S-P's privacy (i.e. proof of Lemma 4) in the following we provide only the proof's outline.

Lemma 7. *If SAP is secure and the encryption scheme is semantically secure, then RC-PoR-P preserves privacy, w.r.t. Definition 15.*

Proof (sketch). 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 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 probability significantly greater than $Pr' + \mu(\lambda)$. □

7.5 Evaluation of RC-PoR-P

In this section, we provide a full analysis of RC-PoR-P, presented in Section 7.3. Table 1 summarises the protocol's costs, breakdown by party.

Computation Cost. In our analysis, the cost of erasure-coding a file is not taken into consideration, as it is identical in all schemes. We first analyse the computation cost of RC-PoR-P. A client's cost is as follows. In the client-side initiation phase (i.e. phase 2), a client's cost in step 2a involves $O(|u^*|)$ invocations of a hash function to construct a

Table 1: RC-PoR-P Costs, breakdown by parties. In the table, z is the total number of verifications, max is the maximum number of complaints the client and server send to the arbiter, $|u^*|$ is a file bit-size, $||u^*||$ is the number of file's blocks, and $||\boldsymbol{\pi}^*||$ is the number of elements in the padded encrypted proof vector.

Phase	Party	Computation Cost	Communication Cost
2 and 3 (i.e. Outsourcing)	Client	$O(\|u^*\|)$	$O(\|\|u^*\|\|)$
	Server	$O(\|u^*\|)$	$O(1)$
4- 8	Client	$O(z \cdot \beta \cdot \log_2(\|u^*\|))$	$O(z)$
	Server	$O(z \cdot \beta \cdot \log_2(\|u^*\|))$	$O(z \cdot \|\|\boldsymbol{\pi}^*_j\|\|)$
	Arbiter	$O(max \cdot \log_2(\|u^*\|))$	$O(1)$
	Contract	$O(1)$	$-$

Merkle tree on the encoded file, u^*, while its total cost in steps 2b and 2d involves two invocations of the hash function when it invokes $\texttt{SAP.init}(.)$ twice, one for qp and the other for cp. Therefore, the client-side total complexity, to initiate the protocol's parameters and encode the file, is $O(|u^*|)$ which is a one-off cost. In the client-side query generation phase (i.e. phase 4), in step 4a, the client call $\texttt{PoRID.genQuery}(.)$ that involves β invocations of $\texttt{PRF}(.)$ to generate a query; also, in step 4b, the client uses the symmetric key encryption to encrypt a single query. So, for z verifications its total computation cost, for this phase, involves $z \cdot \beta$ invocations of $\texttt{PRF}(.)$ and z encryptions, or $O(z \cdot \beta)$. In the client-side proof verification phase (i.e. phase 6), in step 6b, the client for each verification decrypts β proofs; also, in the same step, it verifies the PoR proofs (i.e. the Merkle tree paths) which involves $\beta \cdot \log_2(|u^*|)$ invocations of the hash function. So, for z verifications its total cost, for this phase, involves $z \cdot \beta$ decryption and $z \cdot \beta \cdot \log_2(|u^*|)$ invocations of the hash function. In other words, the client's total complexity in phase 6 is $O(z \cdot \beta \cdot \log_2(|u^*|))$.

Now, we analyse a server's computation cost in RC-PoR-P. In the server-side initiation phase (i.e. phase 3), in step 3b, the server calls $\texttt{SAP.agree}(.)$ twice, to check and agree on parameters of cp and qp, that results in four invocations of the hash function, in total. Also, in step 3c, the server calls $\texttt{PoRID.serve}(.)$ that requires it to construct a Merkle tree on the file. This results in $O(|u^*|)$ invocations of the hash function. So, the server-side total complexity to check and agree on the protocol's parameters is $O(|u^*|)$, that is one-off. In the server-side proof generation phase (i.e. phase 5), in step 5b, the server decrypts a single value for each verification. Also, in step 5c, checks a query's correctness that imposes a negligible computation cost. In the same step, it generates proofs (i.e. PoR) that require $\beta \cdot \log_2(|u^*|)$ invocations of hash function (to generate Merkle tree's paths), for each verification. In the same step, it encrypts each proof, for each verification. So, for z verifications its total cost, for phase 5, involves $z \cdot \beta$ encryptions and $z \cdot \beta \cdot \log_2(|u^*|)$ invocations of the hash function. In other words, the server's total complexity in phase 5 is $O(z \cdot \beta \cdot \log_2(|u^*|))$. Next, we analyse an arbiter's cost in RC-PoR-P in the dispute resolution phase (i.e. phase 7). If both the client and server are honest, then the arbiter is not invoked, accordingly it performs no computation. So, in the following, we consider the case where one of the parties complains about its counter-party's behaviour. First, we evaluate the arbiter's cost when it is invoked by an honest server. In step 7(c)i, the arbiter invokes the hash function twice to check the correctness of the statement, $\ddot{x}_{qp}$. In step 7(c)iii, it decrypts $|\boldsymbol{v}_{\mathcal{S}}|$ queries, where $|\boldsymbol{v}_{\mathcal{S}}|$ is the total number of verifications that the server has complained about and $|\boldsymbol{v}_{\mathcal{S}}| \leq z$. Note, in the same step the arbiter calls $\texttt{PoRID.checkQuery}(.)$ to check the queries; however, its cost is negligibly low. Now, we evaluate the arbiter's cost when it is invoked by a honest client. In step 7(e)i, the arbiter invokes the hash function twice to check the correctness of the statement, $\ddot{x}_{qp}$, sent by the client. Also, in step 7(e)iii, it decrypts $|\boldsymbol{v}_{\mathcal{C}}|$ queries and $|\boldsymbol{v}_{\mathcal{C}}|$ proofs (i.e. only one Merkle tree path for each complaint), where $|\boldsymbol{v}_{\mathcal{C}}|$ is the total number of verifications that the client complained about. In the same step, the arbiter also invokes the hash function $|\boldsymbol{v}_{\mathcal{C}}| \cdot \log_2(|u^*|)$ times in total, to process the client's complaints. Thus, the arbiter's cost, in phase 7, involves at most $2 \cdot max$ decryptions and $max \cdot \log_2(|u^*|)$ invocations of the hash function, where $max = Max(|\boldsymbol{v}_{\mathcal{C}}|, |\boldsymbol{v}_{\mathcal{S}}|)$ and $max \leq z$; its complexity, in this phase, is $O(max \cdot \log_2(|u^*|))$. Note that due to the use of the proof of misbehaviour in the protocol, the arbiter's computation cost is $\frac{1}{\beta} = \frac{1}{460}$ of it's computation cost in the absence of such technique where it has to check all β proofs for each verification. The smart contract performs computations only in the coin transfer phase (i.e. phase 8) that involves two invocations of the hash function, in step 8c, to check the correctness of the statement, $\ddot{x}_{cp}$, so its computation complexity is constant, $O(1)$.

Communication Cost. We first analyse the communication cost of RC-PoR-P. Since in the protocol parties deploy smart contracts (i.e. SAP_1, SAP_2 and SC) and interact with them, we have implemented the smart contracts, and deployed them to Ethereum's test-net blockchain, to determine (a) the deployment cost, and (b) the communication cost of interacting with them. The contract's source code includes about 200 lines of code and is available on a public repository [5]. A client's cost is as follows. In the client-side initiation phase (i.e. phase 2), in steps 2b, 2d, and 2f, in total it deploys (to the blockchain) three smart contracts, i.e. SAP_1, SAP_2 and SC. Our analysis indicates that the contracts' deployment imposes only 4 kilobytes to the client, in total. The client also sends two transactions (that each includes a commitment) to SAP_1 and SAP_2 in steps 2b and 2d, which imposes 72 bytes, in total. Also, in step 2g, it sends a transaction to SC to deposits $coin_{\mathcal{C}}^*$ coins, where the transaction size is only 4 bytes. In the same step, it sends its encoded file u^*, $p_{\mathcal{S}}$, and $\ddot{x}_{cp}$, and $\ddot{x}_{qp}$ to the server, where (a) $\ddot{x}_{qp}$ contains padding information whose size is negligible, the symmetric-key encryption's key of size 256-bit and a random value of size 256-bit, and (b) $\ddot{x}_{cp}$ contains five small-sized values whose size is negligible, and a random value of size 256-bit. Therefore, the client's communication cost in the initiation phase (that includes the file outsourcing) is $\frac{||u^*||}{8} + 4172$ bytes or $O(||u^*||)$. Also, in the client-side query generation phase (i.e. Phase 4), in step 4b, the client for each verification sends to SC a transaction that contains an encrypted query; where the transaction size is 36 bytes. Therefore, for z verifications its total cost in this step is $z \cdot 36$ bytes. In the dispute resolution phase (i.e. Phase 7), the client's communication cost in step 7d is low; because, in this step, it sends statement $\ddot{x}_{qp}$ and its complaint $m_{\mathcal{C}}$ to the arbiter, such that $\ddot{x}_{qp}$ contains (a) padding information whose size is at most a few bits and (b) the symmetric-key encryption's key whose size is at most 256 bits. Moreover, $m_{\mathcal{C}}$ contains at most $2z$ negligible size values/indices (i.e. each value is at most a few bits). In the coin transfer phase (i.e. Phase 8), the client takes either step 8a or step 8b, in either step the client sends to SC a transaction whose maximum size is 196 bytes. Thus, the client's total communication cost (excluding setup and initiation phases) is only $36z + 228$ bytes or $O(z)$.

Now, we determine a server's communication cost. In the server-side initiation phase (i.e. Phase 3), the server sends two transactions (where each includes a commitment) to SC in step 3b, where the total transactions' size is 72 bytes. Also, the server in step 3d sends a transaction (that includes a and $coin_{\mathcal{S}}^*$) to SC, where the transaction's size is only 4 bytes. In the server-side proof generation phase (i.e. Phase 5), the server in step 5c, for each verification sends an encrypted padded proof vector $\boldsymbol{\pi}_j^*$ to SC. Therefore, its complexity for z verifications is $O(z \cdot ||\boldsymbol{\pi}_j^*||)$. In the following, we provide a concrete communication cost of the server, for each verification, in Phase 5, where the size of encoded file u^* is 1-GB. Similar to [21], we let each file block be of size 128 bits which results in $\frac{\text{1-GB}}{\text{128-bit}} = 625 \times 10^5$ blocks. In general, a proof (or path) in a Merkle tree, contains $\log_2(X)$ elements, where X is the number of leaf nodes. Therefore, for the above u^*, a proof would contain $\log_2(625 \times 10^5) = 25$ elements. If we set the hash function output's length to 128 bits, then a proof's total bit-size would be 3200 bits. Moreover, if we set the number of challenged blocks to 460, then the total size of proofs (related to the challenged blocks) would be 184×10^3 bytes [6]. Note, since each element's size in the proof is of a sufficient length, i.e. 128-bit, if it is encrypted using a symmetric key encryption whose output size is also 128-bit, then the ciphertext would have the same length, i.e. 128-bit. Let the padding extend the proofs' length by factor of two, that means the total padded encrypted proofs' size (for each verification) is 368×10^3 bytes. If the server sends a transaction, containing the padded encrypted proofs, $\boldsymbol{\pi}_j^*$, to the contract, then the transaction's size is $368,068$ bytes or 0.3 megabytes (MB). So, in Phase 5, for z verifications its cost is $0.3 \cdot z$ MB. The server's communication cost in the dispute resolution phase (i.e. Phase 7) is low. Because it sends statement $\ddot{x}_{qp}$ and its complaint $m_{\mathcal{S}}$ to the arbiter, where $\ddot{x}_{qp}$ contains padding information whose size is negligible and the symmetric-key encryption's key, whose size is at most 256 bits. Also, $m_{\mathcal{S}}$ contains at most z negligible size indices, where each index is at most a few bits. In the coin transfer phase (i.e. Phase 8), the server's communication cost is identical to the client's cost in the same phase, i.e. 196 bytes. Thus, the server's total communication cost is $304 + \frac{z \cdot ||\boldsymbol{\pi}_j^*||}{8}$ bytes or $O(z \cdot ||\boldsymbol{\pi}_j^*||)$. The arbiter's communication cost is constant, as it only sends a transaction containing four values to SC in step 7f, in the dispute resolution phase (i.e. Phase 7); where the transaction's size is 132 bytes.

[5] The contracts' source code is accessible via the following link: t.ly/3oNa

[6] As shown in [1], to ensure 99% of file blocks is retrievable, it would be sufficient to set the number of challenged blocks to 460.

References

1. 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
2. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000
3. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS'17
4. 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)
5. 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. ACM (2019)
6. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02
7. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9057, pp. 281–310. Springer (2015)
8. Goldreich, O.: The Foundations of Cryptography - Volume 1: Basic Techniques. Cambridge University Press (2001), http://www.wisdom.weizmann.ac.il/%7Eoded/foc-vol1.html
9. Halevi, S., Harnik, D., Pinkas, B., Shulman-Peleg, A.: Proofs of ownership in remote storage systems. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011. pp. 491–500. ACM (2011)
10. 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, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II. Springer (2014)
11. Juels, A., Jr., B.S.K.: Pors: Proofs of retrievability for large files. IACR Cryptology ePrint Archive 2007, 243 (2007)
12. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)
13. 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
14. 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
15. Maxwell, G.: Zero knowledge contingent payment (2011)
16. 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)
17. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings. Lecture Notes in Computer Science, vol. 435, pp. 218–238. Springer (1989)
18. Miller, A., Juels, A., Shi, E., Parno, B., Katz, J.: Permacoin: Repurposing bitcoin work for data preservation. In: S&P'14
19. Nguyen, K., Ambrona, M., Abe, M.: WI is almost enough: Contingent payment all over again. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020
20. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91
21. Shacham, H., Waters, B.: Compact proofs of retrievability. In: ASIACRYPT. pp. 90–107 (2008)
22. Shen, S., Tzeng, W.: Delegable provable data possession for remote data in the clouds. In: ICICS 2011

A 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 who 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 who raises a dispute and sends a dispute query to the contract, (due to the nature of 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 4.3) 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 11:
  - 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, who unnecessarily invokes the contract, pays the contract ahead of time. Therefore, there is no need to keep track of unnecessary contract's invocation.
  - $\texttt{RCSP.resolve}(.)$ is run by a smart contract.
  - $\texttt{RCSP.pay}(.)$ outputs $(\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}})$, so $\boldsymbol{coin}_{\mathcal{A}r}$ is excluded from the output, as a third-party arbiter plays no role anymore.
- In Definition 12: only the above changes are applied to it.
- In Definition 13: 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). *A RC-$\overline{S}$-P is secure against a malicious server, for functions F, Q, M, D, E, and an auxiliary information aux, if for any price list pl, every j (where $1 \leq j \leq z$), and any PPT adversary $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that for any security parameter λ:*

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

  where $\boldsymbol{q}_j \in D(c^*_j, t_{qp})$, $\pi^*_j := (h^*_j, \delta^*_j)$, $h_j = D(h^*_j, T_{qp})$, $D \in enc$, $\sigma \in e$, $m_{\mathcal{C},j} \in m_{\mathcal{C}}, m_{\mathcal{S},j} \in m_{\mathcal{S}}$, $y_{\mathcal{S},j} \in \boldsymbol{y}_{\mathcal{S}} \in \boldsymbol{y}$, and $pp \in T_{qp}$.
- In Definition 14: 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). *A RC-$\overline{S}$-P is secure against a malicious client, for functions F, Q, M, D, E, and an auxiliary information aux, if for every j (where $1 \leq j \leq z$), and any probabilistic polynomial time adversary $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that for any security parameter λ:*

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

  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 15 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. *A RC-$\overline{S}$-P is secure if it satisfies security against malicious server, security against malicious client, and preserves privacy, w.r.t. Definitions 17,18, and 15.*

B 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 7.3, 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 2f, 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 $\boldsymbol{y} : [y_{\mathcal{C}}, y_{\mathcal{S}}, \texttt{Time}, adr_{\text{SC}}]$, 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 $\texttt{K}_1 > \texttt{G}_{z,2} + \texttt{H}$
   (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 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 is 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 calling $\texttt{PoRID.checkQuery}(\hat{k}_i, pp) \rightarrow b_i$. If the query is rejected, i.e. $b_i = 0$, 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 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 $\boldsymbol{m} \in \boldsymbol{m}_{\mathcal{C}}$:
         - Retrieves details of a proof that was rejected in each i-th verification. In particular, it sets $i = \boldsymbol{m}[0]$ and $g = \boldsymbol{m}[1]$. Recall that g refers to the index of a rejected proof in the proof vector which was generated for i-th verification, i.e. $\boldsymbol{\pi}_i$
         - Fetches the related encrypted query $\hat{k}^*_i \in \boldsymbol{q}^*$, and decrypts it, $\hat{k}_i = \texttt{Dec}(\bar{k}, \hat{k}^*_i)$
         - Removes the pads only from g-th padded encrypted proof. Let $\boldsymbol{\pi}'_i[g]$ be the result. Next, it decrypts the encrypted proof, $\texttt{Dec}(\bar{k}, \boldsymbol{\pi}'_i[g])) = \boldsymbol{\pi}_i[g]$
         - Constructs a fresh vector: $\boldsymbol{\pi}''_i$, such that its g-th element equals $\boldsymbol{\pi}_i[g]$ (i.e. $\boldsymbol{\pi}''_i[g] = \boldsymbol{\pi}_i[g]$ and $|\boldsymbol{\pi}''_i| = |\boldsymbol{\pi}_i|$) and the rest of its elements are dummy values.
         - Calls $\texttt{PoRID.identify}(\boldsymbol{\pi}''_i, g, \hat{k}_i, pp) \rightarrow I_i$. If $I_i = \mathcal{S}$, then it increments $y_{\mathcal{S}}$ by 1. Otherwise, it does nothing.

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

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), 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 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:
- $coin^*_{\mathcal{C}} - o(z - y_{\mathcal{S}}) + l(y_{\mathcal{S}} - y_{\mathcal{C}})$ coins to $\mathcal{C}$
- $coin^*_{\mathcal{S}} + o(z - y_{\mathcal{S}}) + l(y_{\mathcal{C}} - y_{\mathcal{S}})$ coins to $\mathcal{S}$

Theorem 6. *The RC-$\overline{\text{PoR}}$-P protocol is secure, w.r.t. Definition 19, if PoRID, SAP, and blockchain are secure and the encryption scheme is semantically secure.*

To prove the above theorem, we show that RC-$\overline{\text{PoR}}$-P meets all security properties defined in Appendix A. 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 Lemma 5) as it does not involve any third-party arbiter.

Lemma 8. *If SAP and blockchain are secure and PoRID scheme supports correctness, soundness, and detectable abort, then RC-$\overline{\text{PoR}}$-P is secure against malicious server, w.r.t. Definition 17.*

Proof (sketch). 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 $coin^*_{\mathcal{C}} - o(z - y_{\mathcal{S}}) + l(y_{\mathcal{S}} - y_{\mathcal{C}})$. Since we focus on j-th verification, the amounts of coin that should be credited to the client for j-th verification is

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

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, due to the correctness of PoRID, values $y_{\mathcal{C}}$ and $y_{\mathcal{S}}$ are not incremented by 1 in 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 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 but 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 PoRID, 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 j-th verification, with a high probability. Furthermore, in general, due to the security of SAP, the adversary cannot block an honest client's messages, "pay" and $\ddot{x}_{cp}$, to the contract in the coin transfer phase. □

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 Lemma 6) as it does not involve any third-party arbiter.

Lemma 9. *If SAP and blockchain are secure and PoRID scheme supports correctness, inputs well-formedness, and detectable abort, then RC-$\overline{\text{PoR}}$-P is secure against malicious client, w.r.t. Definition 18.*

Proof (sketch). 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 $coin^*_{\mathcal{S}} + o(z - y_{\mathcal{S}}) + l(y_{\mathcal{C}} - y_{\mathcal{S}})$. As we focus on j-th verification, the amount of coins that should be credited to the server for j-th verification is

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

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 PoRID, algorithm `PoRID.identify`$(.)$ outputs $I_j = \perp$. Thus, due to the security of SAP and correctness of PoRID, 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 PoRID's inputs well-formedness, the probability the event happens 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 inputs well-formedness of PoRID, the probability that the server outputs $b_j = 1$ is negligible, $\mu(\lambda)$. 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 j-th verification. Also, due to the security of SAP, the adversary cannot block an honest server's messages, "pay" and $\ddot{x}_{cp}$, to the contract in the coin transfer phase. □

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 Lemma 7.

Lemma 10. *If SAP is secure and the encryption scheme is semantically secure, then RC-$\overline{\text{PoR}}$-P preserves privacy, w.r.t. Definition 15.*