# Recurring Contingent Service Payments

**Abstract.**

## 1 Preliminaries and Notations

### 1.1 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. [11]. 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.

### 1.2 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 [10].

**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:*

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

*where the key, $\hat{k} \xleftarrow{\$} \{0,1\}^{\psi}$, is chosen uniformly at random and $\omega$ is chosen uniformly at random from the set of functions mapping $\eta$-bit strings to $\iota$-bit strings. We let public parameters $\zeta : (\psi, \eta, \iota)$ be the description of* PRF

### 1.3 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: $r \xleftarrow{\$} \{0,1\}^{\lambda}$. At 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 [15], 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, $\mu(\lambda)$.

### 1.4 Non-interactive Publicly Verifiable Zero-knowledge Proofs

### 1.5 Digital Signatures

Some of our protocols utilise digital signatures to commit a party to messages it sends.

### 1.6 Merkle Tree

A Merkle tree scheme introduced by Merkle [12, 13] 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

### 1.7 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 [9, 16]. 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 $1^\lambda$ and outputs a secret verification key $sk$ and a set of public parameters $pk$.

- $\texttt{PoR.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma)$. A probabilistic algorithm, run by the client. It takes as input the security parameter $1^\lambda$, a file $u$, and key $k$. It encodes $u$, denoted by $u^*$ and generates a metadata, $\sigma$. The client adds the size of $u^*$ to $pk$, and sends to the server $u^*$ and $\sigma$.

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

- $\texttt{PoR.prove}(u^*, \sigma, \boldsymbol{q}, pk) \rightarrow \pi$. It is run by a server. It takes as input the encoded file $u^*$, metadata $\sigma$, query $\boldsymbol{q}$, and public parameters $pk$. It outputs a proof, $\pi$, given to the client.

- $\texttt{PoR.verify}(\pi, \boldsymbol{q}, k) \rightarrow d \in \{0, 1\}$. It is run by the client. It takes as input the proof $\pi$, query $\boldsymbol{q}$, and key $k$. 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 $(u^*, \sigma)$ 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 [16], 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 $\texttt{PoR.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma)$ 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 $\texttt{PoR.genQuery}(1^\lambda, k) \rightarrow \boldsymbol{q}$ and give the output to it. $\mathcal{A}$ can locally run $\texttt{PoR.prove}(u^*, \sigma, \boldsymbol{q}, pk) \rightarrow \pi$ to get its outputs as well.

3. $\mathcal{A}$ can request $\mathcal{E}$ the execution of $\texttt{PoR.verify}(\pi, \boldsymbol{q}, k)$ 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 $\sigma$ 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 $\epsilon$-*admissible* if it convincingly answers $\epsilon$ 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 $\sigma$, 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 ($\epsilon$-soundness).** *A PoR scheme is $\epsilon$-sound if there exists an extraction algorithm $\texttt{Ext}()$ such that, for every adversary $\mathcal{A}$ who plays the experiment above, and outputs an $\epsilon$-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. $\texttt{Ext}(k, \sigma, \texttt{P}') \rightarrow u$, except with a negligible probability.*

In contrast to the PoR definition in [9, 16] where $\texttt{PoR.genQuery}(.)$ is implicit, in the above we have explicitly stated it, as it plays an important role in this paper.

### 1.8 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.

## 2 Definition

### 2.1 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 $1^\lambda$ and a function, $F$, that will be run on the client's input by the server. It outputs a secret/public verification key pair $k$.*

- $\texttt{VS.setup}(1^\lambda, u, k, M) \rightarrow (u^*, \sigma)$. *It is run by the client. It takes as input the security parameter $1^\lambda$, 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 encoded input $u^*$ and metadata $\sigma = M(u^*, k)$. Right after that, the server is given $u^*$, $\sigma$ and $pk$.*

- $\texttt{VS.genQuery}(1^\lambda, aux, k, Q) \rightarrow \boldsymbol{q}$. *A probabilistic algorithm run by the client. It takes as input the security parameter $1^\lambda$, auxiliary information aux, the key pair $k$ and query generator deterministic function $Q$, where $Q$ is publicly known. It outputs a query vector $\boldsymbol{q} = Q(aux, k)$. Depending on service types, $\boldsymbol{q}$ may be empty or contain only random strings. The output is given to the server.*

- $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk) \rightarrow \pi$. *It is run by the server. It takes as input the service encoded input $u^*$, metadata $\sigma$, queries $\boldsymbol{q}$ and public key $pk$. It outputs a proof pair, $\pi := (F(u^*, \boldsymbol{q}), \delta)$ containing the function evaluation for service input $u$ and query $\boldsymbol{q}$, i.e. $h = F(u^*, \boldsymbol{q})$, and a proof $\delta$ asserting the evaluation is performed correctly, where generating $\delta$ may involve $\sigma$. The output is given to the client.*

- $\texttt{VS.verify}(\pi, \boldsymbol{q}, k) \rightarrow d \in \{0, 1\}$. *It is run by the client. It takes as input the proof $\pi$, query vector $\boldsymbol{q}$, and key $k$. 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 any $F$, any auxiliary information $aux$, any $Q$, and any $M$, the key generation algorithm produces keys $\texttt{VS.keyGen}(1^\lambda, F) \rightarrow k := (sk, pk)$ s.t. $\forall u \in \texttt{Domain}(F)$, if $\texttt{VS.setup}(1^\lambda, u, k, M) \rightarrow (u^*, \sigma)$, $\texttt{VS.genQuery}(1^\lambda, aux, k, Q) \rightarrow \boldsymbol{q}$ and $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk) \rightarrow \pi$, then $\texttt{VS.verify}(\pi, \boldsymbol{q}, k) \rightarrow 1$*

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 a function $F$, if for any security parameter $\lambda$, any auxiliary information $aux$, any $Q$, any $M$, and any probabilistic polynomial time adversaries $\mathcal{A}$, there exists a negligible function $\mu(.)$, such that:*

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

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 [9, 16] or Provable Data Possession [1, 17], 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 1.7) captures VS definition. In particular, PoR's $\epsilon$-soundness captures VS's soundness. Because in the $\epsilon$-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 $\epsilon$ 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.

### 2.2 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 a verifiable service scheme with that features "verifiable service with identifiable abort" (VSID), inspired by the notion of secure multi-party computation with identifiable abort [8].

**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.*

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

- $\texttt{VSID.setup}(1^\lambda, u, k, M) \to (u^*, e)$. *It is run by the client. It takes as input the security parameter* $1^\lambda$*, 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* $u^*$ *and* $e := (\sigma, w_\sigma)$*, where* $\sigma = M(u^*, k)$ *is a metadata and* $w_\sigma$ *is a proof asserting the metadata is well-structured.*

- $\texttt{VSID.serve}(u^*, e, pk) \to a \in \{0, 1\}$. *It is run by the server. It takes as input the encoded service input* $u^*$*, the pair* $e := (\sigma, w_\sigma)$ *and public key* $pk$*. It outputs* $a = 1$*, if the proof* $w_\sigma$ *is accepted, i.e. if the metadata is well-formed. Otherwise, it outputs* $a = 0$*.*

- $\texttt{VSID.genQuery}(1^\lambda, aux, k, Q) \to c := (\boldsymbol{q}, \boldsymbol{w}_q)$. *A probabilistic algorithm run by the client. It takes as input the security parameter* $1^\lambda$*, auxiliary information aux, the key pair* $k$*, and query generator deterministic function* $Q$*, where* $Q$ *is publicly known. It outputs a pair* $c$ *containing a query vector,* $\boldsymbol{q} = Q(aux, k)$*, and proofs,* $\boldsymbol{w}_q$*, proving the queries are well-structured. Depending on service types,* $c$ *might be empty or contain only random strings.*

- $\texttt{VSID.checkQuery}(c, pk) \to b \in \{0, 1\}$. *It is run by the server. It takes as input a pair* $c := (\boldsymbol{q}, \boldsymbol{w}_q)$ *including queries and their proofs, as well as public key,* $pk$*. It outputs* $b = 1$ *if the proofs* $\boldsymbol{w}_\sigma$ *are accepted, i.e. the queries are well-structured. Otherwise, it outputs* $b = 0$*.*

- $\texttt{VSID.prove}(u^*, \sigma, c, pk) \to \pi$. *It is run by the server. It takes as input the encoded service input* $u^*$*, metadata* $\sigma$*, a pair* $c := (\boldsymbol{q}, \boldsymbol{w}_q)$ *and public key* $pk$*. It outputs a proof pair,* $\pi := (F(u^*, \boldsymbol{q}), \delta)$ *containing the function evaluation, i.e.* $h = F(u^*, \boldsymbol{q})$*, and a proof* $\delta$ *asserting the evaluation is performed correctly, where computing* $\delta$ *may involve* $\sigma$*.*

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

- $\texttt{VSID.identify}(\pi, c, k, e, u^*) \to I \in \{\mathcal{C}, \mathcal{S}, \perp\}$. *It is run by a third-party arbiter. It takes as input the proof* $\pi$*, query pair* $c := (\boldsymbol{q}, \boldsymbol{w}_q)$*, key pair* $k$*, metadata pair* $e := (\sigma, w_\sigma)$*, and* $u^*$*. If proof* $w_\sigma$ *or* $\boldsymbol{w}_q$ *is rejected, then it outputs* $I = \mathcal{C}$*; otherwise, if proof* $\pi$ *is rejected it outputs* $I = \mathcal{S}$*. Otherwise, if* $w_\sigma, \boldsymbol{w}_q$*, and* $\pi$ *are accepted, it outputs* $I = \perp$*.*

A VSID scheme has four main properties; namely, it is (a) correct, (b) sound, (c) inputs of clients are well-formed, and (d) a corrupt party can be identified by an arbiter. 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* $\texttt{VSID.keyGen}(1^\lambda, F) \to k := (sk, pk)$ *such that* $\forall u \in \texttt{Domain}(F)$ *if* $\texttt{VSID.setup}(1^\lambda, u, k, M) \to (u^*, e)$, $\texttt{VSID.serve}(u^*, e, pk) \to a$, $\texttt{VSID.genQuery}(1^\lambda, aux, k, Q) \to c$, $\texttt{VSID.checkQuery}\ (c, pk) \to b$, $\texttt{VSID.prove}(u^*, \sigma, c, pk) \to \pi$*, and* $\texttt{VSID.verify}(\pi, \boldsymbol{q}, k) \to d$*, then* $\texttt{VSID.identify}(\pi, c, k, e, u^*) \to I = \perp \ \wedge \ a = 1 \ \wedge \ b = 1 \ \wedge \ d = 1$

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 a function $F$, if for any security parameter $\lambda$, any auxiliary information $aux$, any $M, Q$, and any probabilistic polynomial time adversary $\mathcal{A}_1$, there exists a negligible function $\mu(\cdot)$, such that:*

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

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, if for any security parameter $\lambda$, any functions $F, M, Q$, any auxiliary information aux, and any probabilistic polynomial time adversary $\mathcal{A}_2$, there exists a negligible function $\mu(\cdot)$, such that:*

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

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 abort if for any security parameter $\lambda$ any functions $F, M, Q$, and any auxiliary information aux the following hold:*

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

$$\Pr\left[d = 0 \wedge I \neq \mathcal{S} \;\middle|\; \begin{array}{l} \texttt{VSID.keyGen}(1^\lambda, F) \rightarrow k := (sk, pk) \\ \mathcal{A}_1(1^\lambda, pk, F) \rightarrow u \\ \texttt{VSID.setup}(1^\lambda, u, k, M) \rightarrow (u^*, e) \\ \texttt{VSID.genQuery}(1^\lambda, aux, k, Q) \rightarrow c := (\boldsymbol{q}, \boldsymbol{w}_q) \\ \mathcal{A}_1(c, e, u^*) \rightarrow \pi := (h, \delta) \\ \texttt{VSID.verify}(\pi, \boldsymbol{q}, k) \rightarrow d \\ \texttt{VSID.identify}(\pi, c, k, e, u^*) \rightarrow I \end{array}\right] \leq \mu_1(\lambda).$$

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

$$\Pr\left[(a = 0 \vee b = 0) \wedge I \neq \mathcal{C} \;\middle|\; \begin{array}{l} \mathcal{A}_2(1^\lambda, F, M, Q) \rightarrow (u^*, k := (sk, pk), e := (\sigma, w_\sigma)) \\ \texttt{VSID.serve}(u^*, e, pk) \rightarrow a \\ \mathcal{A}_2(aux, k) \rightarrow c := (\boldsymbol{q}, \boldsymbol{w}_q) \\ \texttt{VSID.checkQuery}(c, pk) \rightarrow b \\ \texttt{VSID.prove}(u^*, \sigma, c, pk) \rightarrow \pi \\ \texttt{VSID.identify}(\pi, c, k, e, u^*) \rightarrow I \end{array}\right] \leq \mu_2(\lambda).$$

**Lighter VSID Scheme (VSID$_{\text{light}}$)** In the VSID definition, algorithm 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 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. VSID.identify(), needs to take only $(\pi, c, k, e')$ 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 VSID.identify$(\pi, c, k, e') \rightarrow I$, 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.

### 2.3 Recurring Contingent Service Payment Definition

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

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

- RCSP.cInit$(1^\lambda, u, \boldsymbol{k}, M, z, cp) \rightarrow (u^*, e, t, coin_{\mathcal{C}}^*, \boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}})$. *It is run by the client. It takes as input* $1^\lambda$*, the service input:* $u$*, the keys* $\boldsymbol{k} : [k, k']$*, metadata generator function:* $M$*, total number of verifications:* $z$*, and coin secret parameters* $cp$ *that includes a subset of* $k'$ *and the actual amount of coins for each accepting service proof:* $o$ *and for covering each potential dispute resolution's cost:* $l$*. It encodes* $u$*, that results* $u^*$*. It computes a metadata,* $\sigma = M(u^*, k)$*, and a proof* $w_\sigma$ *asserting the metadata is well-structured. It constructs coin encoding token* $t_{cp}$ *that contains* $cp$*, total coins the server should deposit* $p_{\mathcal{S}}$*, and* $cp$*'s witness,* $g_{cp}$*. It also constructs proof/query encoding token* $t_{pq}$ *that contains secret parameters* $qp \in k'$ *(used to encode the service queries/proofs) and* $qp$*'s witness,* $g_{qp}$*. Given a valid value and its witness anyone can check if they match. It constructs two binary vectors* $\boldsymbol{y}_{\mathcal{C}}$ *and* $\boldsymbol{y}_{\mathcal{S}}$ *where they are set to* 0 *and each string's length is* $z$*. It outputs encoded input* $u^*$*, metadata-proof pair* $e := (\sigma, w_\sigma)$*, the two sets* $t : \{t_{cp}, t_{qp}\}$*, the two binary vectors* $\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}$*, and the encoded coins amount* $coin_{\mathcal{C}}^*$ *(i.e. contains* $o$ *and* $l$ *coins in an encoded form). The client sends* $u^*$*,* $z$*,* $pk$*,* $e$*, and* $t \setminus \{g_{cp}, g_{qp}, p_{\mathcal{S}}\}$ *to the server and sends* $\{g_{cp}, g_{qp}, p_{\mathcal{S}}\}$*,* $\boldsymbol{y}_{\mathcal{C}}$*,* $\boldsymbol{y}_{\mathcal{S}}$ *and* $coin_{\mathcal{C}}^*$ *coins to the smart contract.*

- RCSP.sInit$(u^*, e, pk, z, t, \boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}) \rightarrow (coin_{\mathcal{S}}^*, a)$. *It is run by the server. It takes as input the service encoded input* $u^*$*, metadata-proof pair* $e : (\sigma, w_\sigma)$*, public key* $pk$*, the total number of verifications* $z$*, sets* $\{t_{cp}, t_{qp}\} \in t$ *(where* $\{g_{cp}, g_{qp}, p_{\mathcal{S}}\} \in t$ *are read from the contract), and reads the two binary strings* $\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}$ *from the contract. It verifies the validity of the elements in* $e$ *and* $t$*. Also, it ensures* $\boldsymbol{y}_{\mathcal{C}}$ *and* $\boldsymbol{y}_{\mathcal{S}}$ *have been set to* 0*. If all approved, 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* $a$.

- RCSP.genQuery$(1^\lambda, aux, k, Q, en) \rightarrow c_j^*$. *A probabilistic algorithm run by the client. It takes as input* $1^\lambda$*, auxiliary information aux, the key pair* $k$*, query generator deterministic function* $Q$*, and* $en$ *that contains the encoding/decoding token* $t_{qp}$ *and encoding/decoding functions* $(E, D)$ *used to encode/decode service's proofs and queries. It computes a pair* $c_j$ *containing a query vector* $\boldsymbol{q}_j : Q(aux, k)$*, and proof* $\boldsymbol{w}_{q_j}$ *proving the query is well-structured. It outputs the encoding of the pair,* $c_j^* = E(c_j, t_{qp})$*. The output is sent to the smart contract.*

- RCSP.prove$(u^*, \sigma, c_j^*, pk, en) \rightarrow (b_j, m_{\mathcal{S},j}, \pi_j^*)$. *It is run by the server. It takes service input:* $u^*$*, metadata:* $\sigma$*, encoded query* $c_j^*$*, public key* $pk$*, and* $en := (E, D, t_{qp})$*. 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* $\pi_j$ *contains* $h_j = F(u^*, \boldsymbol{q}_j)$ *and a proof* $\delta_j$ *asserting the evaluation is performed correctly (*$\pi_j$ *may contain dummy values if* $b_j = 0$*). The contract is given* $\pi_j^*$.

- $\texttt{RCSP.verify}(\pi_j^*, \boldsymbol{q}_j, k, en) \rightarrow (d_j, m_{\mathcal{C},j})$. *A deterministic algorithm run by the client. It takes encoded proof* $\pi_j^*$*, query* $\boldsymbol{q}_j \in c_j$*, verification key* $k$*, and* $en := (E, D, t_{qp})$*. 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* $m_{\mathcal{C},j} = \perp$

- $\texttt{RCSP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, en) \rightarrow (\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}})$. *It is run by the arbiter. It takes client's complaints* $\boldsymbol{m}_{\mathcal{C}}$*, server's complaints* $\boldsymbol{m}_{\mathcal{S}}$*, total number of verifications* $z$*, all encoded proofs* $\boldsymbol{\pi}^*$*, all encoded queries* $\boldsymbol{c}^*$*, public key* $pk$*, and* $en := (E, D, t_{qp})$*. It verifies the token, decoded queries, and proofs. It outputs two binary strings* $\boldsymbol{y}_{\mathcal{C}}$ *and* $\boldsymbol{y}_{\mathcal{S}}$ *of length* $z$*, it sets each element of* $\boldsymbol{y}_{\mathcal{E}}$ *to one, i.e.* $y_{\mathcal{E},j} = 1$*, if party* $\mathcal{E} \in \{\mathcal{C}, \mathcal{S}\}$ *has misbehaved in* $j$*-th verification (i.e. provided invalid query or service proof).*

- $\texttt{RCSP.pay}(\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, t_{cp}, a, coin_{\mathcal{C}}^*, coin_{\mathcal{S}}^*) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{A}r})$. *It is run by the smart contract and can be invoked by the client or server. It takes two binary strings* $\boldsymbol{y}_{\mathcal{C}}$ *and* $\boldsymbol{y}_{\mathcal{S}}$ *that indicates which party misbehaved in each verification, and coins' token* $t_{cp}$*. 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* $coin_{\mathcal{S}}^* \neq p_{\mathcal{S}} \in t_{cp}$*) it only sends* $coin_{\mathcal{C}}^*$ *to* $\mathcal{C}$*.*

In the above definition algorithms $\texttt{RCSP.genQuery}(), \texttt{RCSP.prove}(), \texttt{RCSP.verify}()$ and $\texttt{RCSP.resolve}()$ implicitly take $a$ as another input and execute only if $a = 1$; however, for the sake of simplicity we avoided explicitly stating it in the definition. Informally, 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 if for any function* $F, Q, M, E, D$*, and auxiliary information aux, the key generation algorithm produces keys* $\texttt{RCSP.keyGen}(1^\lambda, F) \rightarrow \boldsymbol{k}$*, such that* $\forall u \in \texttt{Domain}(F)$ *if* $\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, M, z, cp) \rightarrow (u^*, e, t, coin_{\mathcal{C}}^*, \boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}})$, $\texttt{RCSP.sInit}(u^*, e, pk, z, t, \boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}) \rightarrow (coin_{\mathcal{S}}^*, a)$, $\forall j : \Big(\texttt{RCSP.genQuery}(1^\lambda, aux, k, Q, en) \rightarrow c_j^*$, $\texttt{RCSP.prove}(u^*, \sigma, c_j^*, pk, en) \rightarrow (b_j, m_{\mathcal{S},j}, \pi_j^*)$, $\texttt{RCSP.verify}(\pi_j^*, \boldsymbol{q}_j, k, en) \rightarrow (d_j, m_{\mathcal{C},j})\Big), \texttt{RCSP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, en) \rightarrow (\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}})$, $\texttt{RCSP.pay}(\boldsymbol{y}_{\mathcal{C}}, \boldsymbol{y}_{\mathcal{S}}, t_{cp}, a, coin_{\mathcal{C}}^*, coin_{\mathcal{S}}^*) \rightarrow (\boldsymbol{coin}_{\mathcal{C}}, \boldsymbol{coin}_{\mathcal{S}}, \boldsymbol{coin}_{\mathcal{A}r})$*, then* $(a = 1) \wedge (\bigwedge_{j=1}^{z} b_j = \bigwedge_{j=1}^{z} d_j = 1) \wedge (\boldsymbol{y}_{\mathcal{C}} = \boldsymbol{y}_{\mathcal{S}} = 0) \wedge (\sum_{j=1}^{z} coin_{\mathcal{C},j} = coin_{\mathcal{C}}^* - o \cdot z) \wedge (\sum_{j=1}^{z} coin_{\mathcal{S},j} = coin_{\mathcal{S}}^* + o \cdot z) \wedge (\sum_{j=1}^{z} coin_{\mathcal{A}r,j} = 0)$

A RC-S-P scheme is said to be security 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 the client for each verification gets a valid proof and gets back its deposit minus the service payment or the client gets its deposit back (for $j$-th verification) and the arbiter receives $l$ coins. In other words, 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 makes the client withdraw an incorrect amount of coins (i.e. something other than its deposit minus service payment) 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 amount of coins (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 a function* $F$*, if for any security parameter* $\lambda$*, any auxiliary information aux, every* $j$ *(where* $1 \leq j \leq z$*), any* $Q, M, E, D$

*and any probabilistic polynomial time adversary $\mathcal{A}$, there exists a negligible function $\mu(.)$, such that:*

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

*where $h_j = D(h^*_j, t_{qp})$, $D, t_{qp} \in en$, $\sigma \in e$, and the probability is taken over uniform choice of $k, k' \xleftarrow{\$} \{0,1\}^\lambda$, where $k, k' \in \boldsymbol{k}$, as well as the randomness of $\mathcal{A}$ and the randomness used in witnesses $g_{cp}, g_{qp} \in t$.*

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 makes the server receive incorrect amount of coins (something other than its deposit plus the service payment), or (b) invalid metadata or query but convinces the server to accept either of them, or (c) invalid query but persuades the arbiter to accept it, or makes them to 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 it.

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

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

*where $\boldsymbol{q}_j \in D(c^*_j, t_{qp})$, $D, t_{qp} \in en$, $\sigma \in e$, and the probability is taken over the randomness used in witnesses $g_{cp}, g_{qp} \in t$ as well as the randomness used in proofs $\boldsymbol{w}_{q_j} \in c_j$ and $w_\sigma \in e$ and the randomness of $\mathcal{A}$.*

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 service input (e.g. outsourced file) and service proof's status during the private time bubble. In the following, we formally define them.

**Definition 15 (RC-S-P Privacy).** *A RC-S-P preserves privacy, if for any security parameter $\lambda$, any auxiliary information aux, any $F, Q, M, E$ and $D$, the following hold:*

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

$$Pr\left[\mathcal{A}_1(\boldsymbol{c}^*,coin^*_S,coin^*_C,g_{cp},g_{qp},\boldsymbol{\pi}^*,a)\rightarrow\beta \middle| \begin{array}{l} \texttt{RCSP.keyGen}(1^\lambda,F)\rightarrow \boldsymbol{k} \\ \mathcal{A}_1(1^\lambda,pk,F)\rightarrow (u_0,u_1) \\ \beta\xleftarrow{\$}\{0,1\} \\ \texttt{RCSP.cInit}(1^\lambda,u_\beta,\boldsymbol{k},M,z,cp)\rightarrow (u^*_\beta,e,t,coin^*_{\mathcal{C}},\boldsymbol{y}_{\mathcal{C}},\boldsymbol{y}_{\mathcal{S}}) \\ \texttt{RCSP.sInit}(u^*_\beta,e,pk,z,t,\boldsymbol{y}_{\mathcal{C}},\boldsymbol{y}_{\mathcal{S}})\rightarrow (coin^*_{\mathcal{S}},a) \\ \texttt{RCSP.genQuery}(1^\lambda,aux,k,Q_\beta,j,en)\rightarrow c^*_j \\ \texttt{RCSP.prove}(u^*_\beta,\sigma,c^*_j,pk,en)\rightarrow (b_j,m_{\mathcal{S},j},\pi^*_j) \\ \texttt{RCSP.verify}(\pi^*_j,\boldsymbol{q}_j,k,en)\rightarrow (d_j,m_{\mathcal{C},j}) \end{array}\right]\leq \frac{1}{2}+\mu_1(\lambda)$$

2. *For any PPT adversary $\mathcal{A}_2$ who plays the above game, there exists a negligible function $\mu_2(\cdot)$ such that*

$$Pr\left[\mathcal{A}_2(\boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, a) \rightarrow (d_j, j)\right] \leq \frac{1}{2} + \mu_2(\lambda)$$

*where $\boldsymbol{c}^* = [c^*_1, ..., c^*_z]$ and $\boldsymbol{\pi}^* = [\pi^*_1, ..., \pi^*_z]$. Also, depending on the choice of $\beta$, algorithm* `RCSP.genQuery()` *outputs an encoded valid query, when the input is $Q_1$, or outputs an encoded invalid query, when the input is $Q_0$. The probability is taken over uniform choice of $k, k' \xleftarrow{\$} \{0,1\}^\lambda$, where $k, k' \in \boldsymbol{k}$, the randomness of $\mathcal{A}_1$ and $\mathcal{A}_2$, the randomness used in witnesses $g_{cp}, g_{qp} \in t$, the randomness used in $E$.*

In the above definition, the experiment is allowed to produce invalid queries. It is required that the privacy holds regardless of the queries status, i.e. whether they are valid or invalid, as long as the are encoded and provided.

**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.*

### 2.4 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: $\sigma = M(u^*, k)$
   (b) Generates non-interactive publicly verifiable zero-knowledge proofs asserting $\sigma$ has been generated correctly, i.e. $\sigma$ is the output of $M$ that is evaluated on $u^*$, $pk$ and $sk$, without revealing $sk$. Let $w_\sigma$ contains the proofs.
   (c) Posts $e := (\sigma, w_\sigma)$ and $u^*$ to the bulletin board.
3. **Server-side Setup**. $\texttt{VSID.serve}(u^*, e, pk)$
   Ensures the metadata $\sigma$ has been constructed correctly, by verifying the proofs in $w_\sigma$ (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**. $\texttt{VSID.genQuery}(1^\lambda, \text{aux}, k, Q)$

(a) Calls $\texttt{VS.genQuery}(1^\lambda, \text{aux}, k, Q) \rightarrow \boldsymbol{q}$, to generate a query vector, $\boldsymbol{q} : Q(\text{aux}, k)$. 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, $pk$ and $sk$, 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**. $\texttt{VSID.checkQuery}(c, pk)$
   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**. $\texttt{VSID.prove}(u^*, \sigma, c, pk)$ This phase starts only if the query was accepted, i.e. $b = 1$
   (a) Calls $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk) \rightarrow \pi$, to generate $\pi := (F(u^*, \boldsymbol{q}), \delta)$. Recall that $\boldsymbol{q} \in c$
   (b) Posts $\pi$ to the board.
7. **Client-side Proof Verification**. $\texttt{VSID.verify}(\pi, \boldsymbol{q}, k)$
   Calls $\texttt{VS.verify}(\pi, \boldsymbol{q}, k) \rightarrow d$, to verify proof $\pi$. It accepts the proof if $d = 1$; otherwise, it rejects it.
8. **Arbiter-side Identification**. $\texttt{VSID.identify}(\pi, c, k, e, u^*)$
   (a) Calls $\texttt{VSID.serve}(u^*, e, pk) \rightarrow a$. If $a = 1$, then it proceeds to the next step. Otherwise, it outputs $I = \mathcal{C}$ and halts.
   (b) Calls $\texttt{VSID.checkQuery}(c, pk) \rightarrow b$. If $b = 1$, then it proceeds to the next step. Otherwise, it outputs $I = \mathcal{C}$ and halts.
   (c) If $\pi$ is privately verifiable, then the arbiter first checks if $sk \in k$ (provided by the client along with other opening information) matches $\texttt{Com}_{sk} \in pk$. If they do not match, then the arbiter outputs $I = \mathcal{C}$. Otherwise, it calls $\texttt{VS.verify}(\pi, \boldsymbol{q}, k) \rightarrow d$. 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. $\texttt{Com}_{sk}$, $\texttt{Com}_{\text{aux}}$, $w_\sigma$ 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, $w_\sigma$ and $\texttt{Com}_{sk} \in pk$, while $M(u^*, k) \neq \sigma$ or to accept query proofs, $\boldsymbol{w}_q$ and $\texttt{Com}_{\text{aux}}$, while $Q(\text{aux}, k) \neq \boldsymbol{q}$, 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 detected 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 $(\pi, c, pk, e')$ 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.

# 3 Recurring Contingent Service Payments

## 3.1 Limitations of zkCSP

Recall, 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.

1. *Discrepancies between the Security Guarantees of Service and Fair Exchange Schemes*. zkCSP combines a service scheme 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.
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:
   (a) *proof verification status*: it is visible in the real time to everyone that the proof has been accepted or reject, that reflects whether the server has successfully delivered the agreed-upon service or failed to do so that has 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.
   (b) *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 to 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 problem 1 above, it imposes high costs and defeats the purpose of zkCSP design. The reason is that the contract has to always 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 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 general, we can further reduce the involvement of the arbiter (even if a dispute is raised) and in a certain case, i.e. PoR setting, its entire role can be efficiently played by a smart contract.

*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.

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

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 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 a statement.

### 3.3 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. **Setup**.
   (a) both parties agree on the SAP smart contract that explicitly states their addresses, $adr_{\mathcal{C}}$ and $adr_{\mathcal{S}}$.
   (b) they sign and deploy the contract to the blockchain.
2. **Agreement**.
   (a) $\mathcal{C}$ picks a random value $r$, and commits to the statement, $\texttt{Com}(x, r) = g_{\mathcal{C}}$
   (b) $\mathcal{C}$ sends $\ddot{x} := (x, r)$ to $\mathcal{S}$. Also, $\mathcal{C}$ using its account sends $g_{\mathcal{C}}$ to the contract.
   (c) $\mathcal{S}$ checks if has been sent from $adr_{\mathcal{C}}$, and $\texttt{Ver}(g_{\mathcal{C}}, \ddot{x}) \stackrel{?}{=} 1$. If the checks pass, then it computes $\texttt{Com}(x, r) = g_{\mathcal{S}}$
   (d) $\mathcal{S}$ using its account sends $g_{\mathcal{S}}$ to the contract.
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**. Given $\ddot{x}$, the contract does the following.
   (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 1, if the checks in the two previous steps (i.e. steps 4a and 4b) were passed. Otherwise, it outputs 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 [6], 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.

Note, 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, 5]. 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).

### 3.4 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, $H$, 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 $\texttt{VSID.keyGen}()$, $\texttt{VSID.setup}()$, and $\texttt{VSID.serve}()$ 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.

At the end of each billing cycle, the client generates an encrypted query, by calling $\texttt{VSID.genQuery}()$ 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 $\texttt{VSID.checkQuery}()$. 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 $\texttt{VSID.prove}()$. 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 $\texttt{VSID.verify}()$. 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 $\texttt{VSID.identify}()$ 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 and the total number of times each of them raised disputes on accepting 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. Moreover, 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 which is known to all parties, (b) it uses that (authorised) address to send transactions to the smart contract, and (c) the contract before recording a transaction, ensures the transaction is originated from an authorised address.

1. **Key Generation**.
   (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: $pad_\pi$ and $pad_q$, where $pad_\pi$ and $pad_q$ refer to the number of dummy values that will be used to pad encrypted proofs and encrypted queries respectively [2]. Let $qp := (pad_\pi, pad_q, \bar{k})$. The keys' size is part of the security parameter.
   (b) $\mathcal{C}$ 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. Let $k' : \{o, l, qp\}$ and let $\boldsymbol{k} = [k, k']$.
2. **Initiation**.
   (a) For $\mathcal{C}$ and $\mathcal{S}$ to provably agree on $qp$, $\mathcal{C}$ sends $qp$ to $\mathcal{S}$. Next (if the $\mathcal{S}$ agrees on the parameters) they take the steps in the Setup and Agreement phases in the SAP, at time $T_0$. Let $t_{qp} := (\ddot{x}_{qp}, g_{qp})$ be proof/query encoding token, where $\ddot{x}_{qp}$ is the opening and $g_{qp}$ is the commitment stored on the contract as a result of running SAP.
   (b) Let $cp := (o, o_{max}, l, l_{max}, z)$, 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. For $\mathcal{C}$ and $\mathcal{S}$ to provably agree on $cp$, similar to the previous step, they invoke SAP again, at time $T_1$. Let $t_{cp} := (\ddot{x}_{cp}, g_{cp})$ be coin encoding token, where $\ddot{x}_{cp}$ is the opening and $g_{cp}$ the commitment stored on the contract as a result of running the SAP.
   (c) $\mathcal{C}$ set parameters $coin^*_{\mathcal{C}} = z \cdot (o_{max} + l_{max})$ and $coin^*_{\mathcal{S}} = z \cdot l_{max}$, where $coin^*_{\mathcal{C}}$ and $coin^*_{\mathcal{S}}$ are the total number of masked coins $\mathcal{C}$ and $\mathcal{S}$ should deposit respectively. $\mathcal{C}$ signs and deploys smart contract SC that explicitly specifies parameters $z$, $coin^*_{\mathcal{C}}$ and $coin^*_{\mathcal{S}}$. It deposits $coin^*_{\mathcal{C}}$ coins in the contract.
   (d) $\mathcal{C}$ constructs vector $\boldsymbol{v}_{\mathcal{C}}$, also $\mathcal{S}$ constructs $\boldsymbol{v}_{\mathcal{S}}$, where the vectors are initially empty.
   (e) $\mathcal{C}$ runs $\texttt{VSID.keyGen}(1^\lambda, F) \rightarrow k : (sk, pk)$ and $\texttt{VSID.setup}(1^\lambda, u, k, M) \rightarrow (u^*, e)$. It sends $e$ and $u^*$ to $\mathcal{S}$ and sends the public key's encryption: $\texttt{Enc}(\bar{k}, pk)$ to SC at time $T_2$.
   (f) $\mathcal{S}$ checks the above parameters, and ensures sufficient amount of coins has been deposited. If any check is rejected, then it sets $a = 0$. Otherwise, it decrypts the public key, $\texttt{Dec}(\bar{k}, \texttt{Enc}(\bar{k}, pk)) = pk$. It runs $\texttt{VSID.serve}(u^*, e, pk) \rightarrow a$. Next, it sends $a$ and $coin^*_{\mathcal{S}}$ coins to SC at time $T_3$, where $coin^*_{\mathcal{S}} = \perp$ if $a = 0$
   (g) $\mathcal{C}$ and $\mathcal{S}$ can withdraw their coins at time $T_4$, if the server sends $a = 0$, fewer coins than $coin^*_{\mathcal{S}}$, or nothing to the SC
3. **Billing-cycles**. The parties perform as follows at the end of every $j$-th billing cycle, where $1 \leq j \leq z$. Each $j$-th cycle includes two time points, $G_{j,1}$ and $G_{j,2}$, where $G_{j,2} > G_{j,1}$, and $G_{1,1} > T_4$
   (a) To generate a query-proof pair, $\mathcal{C}$ calls $\texttt{VSID.genQuery}(1^\lambda, \text{aux}, k, Q) \rightarrow c_j$. It encrypts the pair, $\texttt{Enc}(\bar{k}, c_j) = c'_j$. Then, it pads the result with $pad_q$ random values that are picked from the encryption's output range, $U$. It sends the padded encrypted query-proof pair to SC at time $G_{j,1}$
   (b) In this phase, $\mathcal{S}$ generates a service proof. To do so, it constructs two empty vectors, $\boldsymbol{m}_{\mathcal{S}} = \perp$ and $\boldsymbol{v}_{\mathcal{S}} = \perp$. It also removes the pads from the padded encrypted query-proof pair. Let $c'_j$ be the result. Next, it decrypts the result, $\texttt{Dec}(\bar{k}, c'_j) = c_j$. Then, it calls $\texttt{VSID.checkQuery}(c_j, pk) \rightarrow b_j$, to check the correctness of the queries.
      - If $\mathcal{S}$ accepts the query, i.e. $b_j = 1$, then it calls $\texttt{VSID.prove}(u^*, \sigma, c_j, pk) \rightarrow \pi_j$, to generate the service proof. In this case, $\mathcal{S}$ encrypts the proof, $\texttt{Enc}(\bar{k}, \pi_j) = \pi'_j$. Next, it pads the encrypted proof with $pad_\pi$ random values that are picked from $U$. It sends the padded encrypted proof to SC at time $G_{j,2}$
      - Otherwise (if $\mathcal{S}$ rejects the query), it appends $j$ to $\boldsymbol{v}_{\mathcal{S}}$, constructs a dummy proof $\pi'_j \in U$, pads the result as above, and sends the padded dummy proof to SC at time $G_{j,2}$

      When $j = z$ and $\boldsymbol{v}_{\mathcal{S}} \neq \perp$, $\mathcal{S}$ sets $\boldsymbol{m}_{\mathcal{S}} : [\boldsymbol{v}_{\mathcal{S}}, \ddot{x}_{qp}, \text{“dispute”}]$.

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

(c) In this phase, $\mathcal{C}$ verifies the service proof. It first constructs two empty vectors, $\boldsymbol{m}_{\mathcal{C}} = \perp$ and $\boldsymbol{v}_{\mathcal{C}} = \perp$. Then, it removes the pads from the padded encrypted proof. Let $\pi'_j$ be the result. It decrypts the service proof: $\texttt{Dec}(\bar{k}, \pi'_j) = \pi''_j$ and then calls $\texttt{VSID.verify}(\pi''_j, \boldsymbol{q}_j, k) \rightarrow d_j$, to verify the proof. Note that if $\pi'_j = \texttt{Enc}(\bar{k}, \pi_j)$, then $\pi''_j = \pi_j$. If $\pi''_j$ passes the verification (i.e. $d_j = 1$), then $\mathcal{C}$ concludes that the service for this verification has been delivered successfully. Otherwise (when $\pi''_j$ is rejected), $\mathcal{C}$ appends $j$ to $\boldsymbol{v}_{\mathcal{C}}$. If $j = z$ and $\boldsymbol{v}_{\mathcal{C}} \neq \perp$, $\mathcal{C}$ sets $\boldsymbol{m}_{\mathcal{C}} : [\boldsymbol{v}_{\mathcal{C}}, \ddot{x}_{qp}, e', \text{“dispute”}]$, where (as stated in Remark 2) $e'$ contains the opening of $\texttt{Com}_{sk}$ or $\perp$.

4. **Dispute Resolution**. The phase takes place only in case of dispute, e.g. when $\mathcal{C}$ and/or $\mathcal{S}$ reject any proofs in the previous phase.
   (a) $\mathcal{C}$ sends $\boldsymbol{m}_{\mathcal{C}}$ to the arbiter at time $K_1$, where $K_1 > G_{z,2} + H$. Also, $\mathcal{S}$ sends $\boldsymbol{m}_{\mathcal{S}}$ to the arbiter at time $K_1$
   (b) The arbiter sets four counters: $y_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{C}}$, and $y'_{\mathcal{S}}$, that are initially set to 0
   (c) The arbiter, checks the validity of the statement: $\ddot{x}_{qp} \in \boldsymbol{m}_{\mathcal{R}}, \forall \mathcal{R} \in \{\mathcal{C}, \mathcal{S}\}$. To do that, it sends $\ddot{x}_{qp}$ 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 the elements of $\boldsymbol{v}_{\mathcal{R}}$ to $\boldsymbol{v}$, such that $\boldsymbol{v}$ contains only distinct elements which are in the range $[1, z]$. Otherwise (if the party's statement is rejected) it discards the party's request: $\boldsymbol{m}_{\mathcal{R}}$, and increments $y'_{\mathcal{R}}$ by 1
   (d) The arbiter uses $\bar{k} \in \ddot{x}_{qp}$ to decrypt the encrypted public key that was sent to SC, $\texttt{Dec}(\bar{k}, \texttt{Enc}(\bar{k}, pk)) = pk$
   (e) The arbiter for every element $i \in \boldsymbol{v}$:
      i. removes the pads from the related encrypted query-proof pair and encrypted service proof. Let $c'_i$ and $\pi'_i$ 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 $\texttt{VSID.identify}(\pi''_i, c_i, pk, e') \rightarrow I_i$
         - 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 the arbiter is invoked by the client or server respectively.

      Let $K_2$ be the time that the arbiter finishes the above checks.
   (f) The arbiter sends $(y_{\mathcal{C}}, y_{\mathcal{S}})$ and $(y'_{\mathcal{C}}, y'_{\mathcal{S}})$ to SC at time $K_3$
5. **Coin Transfer**.
   (a) Either $\mathcal{C}$ or $\mathcal{S}$ send “pay” message and the statement, $\ddot{x}_{cp}$, to SC at time $L > K_3$
   (b) 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.
   (c) 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 3.* 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 4.* Keeping track of $(y'_{\mathcal{C}}, y'_{\mathcal{S}})$ enables the arbiter to make malicious parties, who *unnecessarily* invoke it for invalid statement in step 4c or accepting proofs in step 4(e)iii, pay for the verifications it performs.

*Remark 5.* 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 6.* 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 7.* In the case where 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 VSID.verify() supports.

**Theorem 2.** *The RC-S-P protocol is secure, w.r.t. Definition 16, if VSID and SAP 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 2.3. We start by proving that RC-S-P satisfies security against a malicious server.

**Lemma 1.** *If SAP is 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 $F(u^*, \boldsymbol{q}_j) = h_j \ \wedge coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o$ that captures the case where the server provides an accepting service proof but makes the client withdraw an incorrect amount of coins, i.e. $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o$. In this case, since the 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 (through dispute resolution phase), or submit to the contract, at the coin transfer phase, an accepting statement $\ddot{x}'_{cp}$ other than what was agreed at the initiation phase, i.e. $\ddot{x}_{cp}$. 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. Moreover, due to the security of SAP, the server cannot persuade the contract to accept any statement 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. Also, due to the correctness of VSID, the arbiter always accepts the honest client's accepting proof.

We now move on to event $F(u^*, \boldsymbol{q}_j) \neq h_j \ \wedge (d_j = 1 \vee y_{\mathcal{S}}[j] = 0 \ \vee (coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} \ \vee coin_{\mathcal{A}r,j} \neq l))$ which captures the case where the server provides an invalid service proof but either persuades the client to accept the proof, or (when the client raises a dispute) persuades the arbiter to accept the proof 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, $\mu(\lambda)$. On the other hand, in the case where the client rejects the proof and raises a dispute, the server may try to convince the arbiter and make it output $y_{\mathcal{S}}[j] = 0$, e.g. by sending a complaint right after the client does. But, for the adversary to win, 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 negligible, $\mu(\lambda)$. Or it makes the arbiter accept an invalid proof, but due to the detectable abort property of VSID, its probability of success is also negligible, $\mu(\lambda)$. In the case where the adversary does not succeed in convincing the client or arbiter, it may still try to make them withdraw an incorrect amount of coins. To this end, at 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 both events above, 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 is 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 $(M(u^*, k) = \sigma \wedge Q(\text{aux}, k) = \boldsymbol{q}_j) \wedge (coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o)$ which captures the case where the client provides accepting metadata and query but makes the server withdraw an incorrect amount of coins, i.e. $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o$. In short, for the event to happen with a high probability, a malicious client has to break the security of SAP. In particular, since the metadata and query's proofs are valid, an honest server accepts them and does not raise any dispute. But, the client may want to make the server withdraw incorrect amounts of coins,

if it manages to either convince the arbiter, in phase 4, that the server has misbehaved, 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 5. 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, 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 that the arbiter is given the honest server's messages (with a high probability). Moreover, due to the security of SAP, the client cannot convince the contract to accept any accepting statement other than what was initially agreed between the client and server, except with probability $\mu(\lambda)$. Furthermore, the correctness of VSID guarantees that the arbiter always accepts the honest server's accepting proof.

We now turn our attention to $(M(u^*, k) \neq \sigma \wedge a = 1) \vee (Q(\text{aux}, k) \neq \boldsymbol{q}_j \wedge b_j = 1)$, that captures the case where the server accepts an ill-formed metadata, or query. However, due to inputs well-formedness of VSID, the probability that either of the events happens is negligible, $\mu(\lambda)$. Next, we move of to $Q(\text{aux}, k) \neq \boldsymbol{q}_j \wedge (y_{\mathcal{C}}[j] = 0 \vee coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o \vee coin_{\mathcal{A}r,j} \neq l)$. It considers the case where the client provides an invalid query, but either convinces the 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. Note, 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 right after the server does so. 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 $y_{\mathcal{C}}[j] = 0$. Nevertheless, due to the detectable abort property of VSID, its probability of success is negligible, $\mu(\lambda)$. If the adversary does not succeed in convincing the server or arbiter, it may still try to make them withdraw an incorrect amount of coins. To this end, at the coin transfer phase, it has to send a different accepting statement than what was initially agreed with the server. However, due to the security of SAP, its success probability is negligible, $\mu(\lambda)$. 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 $\beta$ with probability significantly greater than $\frac{1}{2}$ (where the probability is taken over the choice of $\beta$ and the randomness of $\mathcal{A}$).*

*Proof.* As it is evident, the list and $z$ contains no information about $\beta$. Also, since $z$ is a public value, we could have $coin'^*_{\mathcal{C}} = \frac{coin^*_{\mathcal{C}}}{z} = Max(o_\beta, o_{|\beta-1|}) + Max(l_\beta, l_{|\beta-1|})$. 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 $\beta$. Therefore, given the list, $z$ and $coin'^*_{\mathcal{C}}$ the adversary learns nothing about $\beta$, 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).* 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 result of running SAP) the adversary learns no information about the committed values (e.g. $o, l, pad_\pi, pad_q$, and $\bar{k}$), except with negligible probability, $\mu_1(\lambda)$. Also, given 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 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 greater than $\frac{1}{2} + \mu_1(\lambda)$, under the assumption that $Max(|\boldsymbol{q}_{j,0}|, ..., |\boldsymbol{q}_{j,m}|) = |\boldsymbol{q}_{j,\beta}|$. The 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 probability greater than $\frac{1}{2} + \mu_1(\lambda)$ 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. Again, 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 probability greater than $\frac{1}{2} + \mu_2(\lambda)$, 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 probability greater than $\frac{1}{2} + \mu_2(\lambda)$ 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 $u_\beta$ and $\beta$. Moreover, since each padded encrypted query and proof leak no information and always contains a fixed number of elements, an adversary cannot tell the status of a proof for each $j$-th verification (i.e. whether it is accepted or rejected) with the probability greater than $\frac{1}{2} + \mu_2(\lambda)$, given $\boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*$, and $a$. □

### 3.5 Reducing the Arbiter involvement

Explain how we can use the smart contracts of [4] to delegate the arbiter task to smart contracts. In this case, only if the smart contracts do not reach a consensus the arbiter is involved.

## 4 Recurring Contingent PoR Payment Protocol

In this section, we present recurring contingent PoR payment (RC-PoR-P). Since proofs of retrievability (PoR), whose definition was given in Section 1.7, is a concrete instantiation of the generic verifiable service, RC-PoR-P is a concrete instantiation of the generic recurring contingent service payment (RC-S-P), when the 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) does not involve a third-party arbiter, which ultimately minimises trust assumption and costs. 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 be malicious too, if tags are used then using zero-knowledge proofs seem an obvious choice, as it allows the client to ensure 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 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.

***Efficiently Eliminating Arbiter's Involvement***. To totally eliminate the involvement of a third-party arbiter, one could use the Merkle tree-based PoR in RC-S-P and let a smart contract play the arbiter's entire role. In this case, there will be two naive approaches. One approach is that for each verification, the client verifies the server's proofs[3] and if

[3] 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.

it rejects them, then it sends the proofs (after the private time bubble) to the contract who verifies the proofs again. However, this requires the smart contract to verify *all* proofs which imposes a high cost. Another approach is that for each verification, the server's proofs are always checked only by the contract, without the client's involvement. But, this requires the contract to always perform the verification, even if both parties behaved honestly, which imposes unnecessary hight computation cost too. To achieve our goal, we use a Merkle tree-based PoR and let the client verify the server's proofs locally. However, we use the idea of proof of misbehaviour, put forth in []. In particular, if the client detects invalid proofs for each verification it only sends one invalid proof to the contract who checks the validity of that single proof (as apposed to checking all proofs). Thus, this eliminates the involvement of the arbiter and has a much lower cost.

To present RC-PoR-P protocol, we will use the same approach we 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.

### 4.1 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: $\sigma$, by constructing Merkle tree on blocks of $u^*$, i.e. $\texttt{MT.genTree}(u^*)$. Let $\sigma$ be the root of the resulting tree, and $\beta$ be a security parameter. It sets public parameters as $pk := (\sigma, \beta, m, \zeta)$, where $\zeta := (\psi, \eta, \iota)$ is a PRF's description, as it was defined in Section 1
   (c) Sends $pk$ and $u^*$ to $\mathcal{S}$
2. **Client-side Query Generation**. $\texttt{PoR.genQuery}(1^\lambda, pk)$
   (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**. $\texttt{PoR.prove}(u^*, \hat{k}, pk)$
   (a) Derives $\beta$ 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: $\pi_{q_i}$, 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 $\pi_{q_i}$ is path in the tree that proves its corresponding block: $u^*_{q_i}$ is a leaf node of the tree.
   (c) Sends $\boldsymbol{\pi}$ to $\mathcal{C}$
4. **Client-side Proof Verification**. $\texttt{PoR.verify}(\boldsymbol{\pi}, \boldsymbol{q}, pk)$
   (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 $\pi$ 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 4.1, is $\epsilon$-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 $\beta$ 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. [7, 14, 9]. Alternatively, our protocol can be proven based on the security analysis of the PoR schemes that use MACs or BLS signatures, e.g. [16]. 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 $\epsilon$ fraction of audits. It initialises an empty array. Then it challenges a subset of file blocks and asks the prover to generate a proof. If the received proof passes the verification, then it adds the related block (in the proof) to the array. It then rewinds the prover and challenges a fresh set of blocks, and repeats the process many times. Since, the prover has a good chance of passing the audit, it is easy to show that the extractor can eventually extract a large fraction of the entire file. 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 8.* 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.

### 4.2 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 $\texttt{PoR.setup}(1^\lambda, u) \rightarrow (u^*, pk)$, that results in public parameters $pk := (\sigma, \beta, m, \zeta)$ and encoded file: $u^* = u_0^{'}||0, ..., u_m^{'}||m$. Recall, $\zeta := (\psi, \eta, \iota)$ is the PRF's description.
   (b) Posts $pk$ to the bulletin board and sends $u^*$ to $\mathcal{S}$
2. **Server-side Setup**. $\texttt{PoRID.serve}(u^*, pk)$
   Verifies the correctness of public parameters:
   (a) rebuilds the Merkle tree on $u^*$ and checks the resulting root equals $\sigma$
   (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**. $\texttt{PoRID.genQuery}(1^\lambda, pk)$
   (a) Calls $\texttt{PoR.genQuery}(1^\lambda, pk) \rightarrow \hat{k}$, to generate a key, $\hat{k}$
   (b) Posts $\hat{k}$ to the board.
4. **Server-side Query Verification**. $\texttt{PoRID.checkQuery}(\hat{k}, pk)$
   (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**. $\texttt{PoRID.prove}(u^*, \hat{k}, pk)$
   (a) Calls $\texttt{PoR.prove}(u^*, \hat{k}, pk) \rightarrow \boldsymbol{\pi}$, to generate proof vector: $\boldsymbol{\pi}$
   (b) Posts $\boldsymbol{\pi}$ to the board.
6. **Client-side Proof Verification**. $\texttt{PoRID.verify}(\boldsymbol{\pi}, \hat{k}, pk)$
   Calls $\texttt{PoR.verify}(\boldsymbol{\pi}, \hat{k}, pk) \rightarrow \boldsymbol{d}$, to verify the proof. If $\boldsymbol{d}[0] = 1$, it accepts the proof; otherwise, it rejects it.
7. **Arbiter-side Identification**. $\texttt{PoRID.identify}(\boldsymbol{\pi}, g, \hat{k}, pk)$
   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 $\texttt{PoRID.checkQuery}(\hat{k}, pk)$. 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 $\texttt{PoR.verify}(\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{q}}, pk) \rightarrow \boldsymbol{d}'$. If $\boldsymbol{d}'[0] = 0$, then it outputs $I = \mathcal{S}$. Otherwise, it outputs $I = \perp$

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

*Proof (sketch).* The $\epsilon$-soundness of PoRID directly stems from the security of PoR scheme, i.e. $\epsilon$-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 $\epsilon$ fraction of challenges. On the other hand, the inputs well-formedness holds for the following reasons. The metadata generation algorithm, i.e. the Merkle tree algorithm that builds a tree and computes a root, is deterministic and involves only public parameters. Thus, given the tree's leaves (i.e. file blocks), its parameters, and the root, anyone can reconstruct it, check if it yields the same root, and verify the tree's parameters. Also, a query contains a single random key, $\hat{k}$, whose correctness can be checked deterministically, i.e. by checking $\hat{k} \neq \perp$ and $\hat{k} \in \{0,1\}^\psi$. The detectable abort property holds as long as the soundness and inputs well-formedness hold and the 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.

□

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

This section presents recurring contingent PoR payment (RC-PoR-P) protocol. It is built upon PoRID protocol and is in the same security model as RC-S-P is. RC-PoR-P inherits the features of PoRID and RC-PoR-P; however, unlike RC-PoR-P, it does not use any zero-knowledge proofs and there is no third-party arbiter involved. Even though 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, $k$, and a pad's length. 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, $H$) 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 any 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 `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 and sends the encoded file to the server who decrypts them and using the encoded file checks their correctness by calling `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. Otherwise, the server sends 1 to the contract. At the end of each billing cycle, the client generates an encrypted query, by calling `PoRID.genQuery`() and encrypting its output using the key, $k$. It sends the result to the contract. In the same cycle, the server retrieves the query, and decrypts it. Then, it locally checks its correctness, by calling `PoRID.checkQuery`(). If the query is rejected, the server locally stores the index of that billing cycle and generates a dummy PoR proofs. However, if the server accepts the query, it generates PoR proofs by calling `PoRID.prove`(). Then, in either case, the server encrypts the proofs, pads them and sends the result to the contract. After that, the client removes the pads, decrypts the proofs and locally verifies them, by calling `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 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 contract (a) the indices of the billing cycles in which its counterparty provided invalid values, and (b) the statement that contains the decryption key and padding detail. The contract 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 contract checks the party's claim by calling `PoRID.checkQuery`() and `PoRID.identify`() if the server or client calls the contract respectively. The contract also keeps track of the number of times each party provided invalid queries or PoR proofs. 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 the contract distributes the coins according to the statement's detail, and the number of times each party misbehaved.

1. **Key Generation**.
   (a) $\mathcal{C}$ picks a random secret key $\bar{k}$ for a symmetric key encryption. Also, it sets parameter $pad_\pi$ 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.
   (b) $\mathcal{C}$ sets coin parameters as follows, $o$: the amount of coins for each accepting proof, as well as $l$ and $l'$ which are the amount of coins $\mathcal{C}$ and $\mathcal{S}$ respectively need to send to a smart contract to resolve a potential dispute. Let $k' := \{o, l, l', qp\}$.
2. **Initiation**.

(a) For $\mathcal{C}$ and $\mathcal{S}$ to provably agree on $qp$, $\mathcal{C}$ sends $qp$ to $\mathcal{S}$. Next (if the $\mathcal{S}$ agrees on the parameters) they take the steps in the Setup and Agreement phases in the SAP, at time $T_0$. Let $t_{qp} := (\ddot{x}_{qp}, g_{qp})$ be proof encoding token, where $\ddot{x}_{qp}$ is the opening and $g_{qp}$ is the commitment stored on the contract as a result of running SAP.

(b) Let $cp := (o, o_{max}, l, l', l_{max}, z)$, 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. For $\mathcal{C}$ and $\mathcal{S}$ to provably agree on $cp$, similar to the previous step, they invoke SAP, at time $T_1$. Let $t_{cp} := (\ddot{x}_{cp}, g_{cp})$ be coin encoding token, where $\ddot{x}_{cp}$ is the opening and $g_{cp}$ is the commitment stored on the contract as a result of executing the SAP.

(c) $\mathcal{C}$ sets $coin^*_{\mathcal{C}} = z \cdot (o_{max} + l_{max})$ and $coin^*_{\mathcal{S}} = z \cdot l_{max}$, where $coin^*_{\mathcal{C}}$ and $coin^*_{\mathcal{S}}$ are the total number of masked coins that $\mathcal{C}$ and $\mathcal{S}$ should deposit respectively. $\mathcal{C}$ signs and deploys a smart contract, SC, that explicitly specifies parameters $z$, $coin^*_{\mathcal{C}}$ and $coin^*_{\mathcal{S}}$. It deposits $coin^*_{\mathcal{C}}$ coins in the contract.

(d) $\mathcal{C}$ constructs vector $\boldsymbol{w}_{\mathcal{C}}$, also $\mathcal{S}$ constructs $\boldsymbol{v}_{\mathcal{S}}$, where the vectors are initially empty.

(e) $\mathcal{C}$ runs $\texttt{PoRID.setup}(1^\lambda, u) \rightarrow (u^*, pk)$. It sends encoded file $u^*$ to $\mathcal{S}$, and sends the public key's encryption: $\texttt{Enc}(\bar{k}, pk)$ to SC at time $T_2$

(f) $\mathcal{S}$ checks the above parameters, and ensures sufficient amount of coins has been deposited. If any check is rejected, then it sets $a = 0$. Otherwise, it decrypts the public key, $\texttt{Dec}(\bar{k}, \texttt{Enc}(\bar{k}, pk)) = pk$. It runs $\texttt{PoRID.serve}(u^*, pk) \rightarrow a$. Next, it sends $a$ and $coin^*_{\mathcal{S}}$ coins to SC at time $T_3$, where $coin^*_{\mathcal{S}} = \perp$ if $a = 0$

(g) $\mathcal{C}$ and $\mathcal{S}$ can withdraw their coins at time $T_4$, if the server sends $a = 0$, fewer coins than $coin^*_{\mathcal{S}}$, or nothing to the SC

3. **Billing-cycles**. The parties do the following, at the end of every $j$-th billing cycle, where $1 \leq j \leq z$. Each $j$-th cycle includes two time points, $G_{j,1}$ and $G_{j,2}$, where $G_{j,2} > G_{j,1}$, and $G_{1,1} > T_4$

(a) $\mathcal{C}$ calls $\texttt{PoRID.genQuery}(1^\lambda, pk) \rightarrow \hat{k}_j$, to generate a query. It sends $\hat{k}_j$ to SC at time $G_{j,1}$

(b) $\mathcal{S}$ calls $\texttt{PoRID.checkQuery}(\hat{k}_j, pk) \rightarrow b_j$ to check the query's correctness.

- If it accepts the query, then it calls $\texttt{PoRID.prove}(u^*, \hat{k}_j, pk) \rightarrow \boldsymbol{\pi}_j$, to generate a PoR proof. In this case, $\mathcal{S}$ encrypts every proof in the proof vector, i.e. $\forall g, 1 \leq g \leq |\boldsymbol{\pi}_j| : \texttt{Enc}(\bar{k}, \boldsymbol{\pi}_j[g]) = \boldsymbol{\pi}'_j[g]$. Let vector $\boldsymbol{\pi}'_j$ contain the encryption of all proofs. It pads every encrypted proof in $\boldsymbol{\pi}'_j$ with $pad_\pi$ random values that are picked from the encryption's output range $U$, (by appending the random values to the encrypted proofs vector). It sends the padded encrypted proofs to SC at time $G_{j,2}$
- Otherwise (if $\mathcal{S}$ rejects the query), it appends $j$ to $\boldsymbol{v}_{\mathcal{S}}$, constructs a dummy proof $\boldsymbol{\pi}'_j$ whose elements are randomly picked from $U$, pads the result as above, and sends the padded dummy proof to SC at time $G_{j,2}$

When $j = z$ and $\boldsymbol{v}_{\mathcal{S}} \neq \perp$, $\mathcal{S}$ sets $\boldsymbol{m}_{\mathcal{S}} : [\boldsymbol{v}_{\mathcal{S}}, \ddot{x}_{qp}, \text{“dispute”}]$.

(c) In this phase, $\mathcal{C}$ verifies the service proof. It first constructs two empty vectors, $\boldsymbol{m}_{\mathcal{C}} = \perp$ and $\boldsymbol{w}_{\mathcal{C}} = \perp$. Next, it removes the pads from the padded encrypted proofs and then decrypts the encrypted proofs: $\texttt{Dec}(\bar{k}, \boldsymbol{\pi}'_j) = \boldsymbol{\pi}_j$. Then, it calls $\texttt{PoRID.verify}(\boldsymbol{\pi}_j, \hat{k}_j, pk) \rightarrow \boldsymbol{d}_j$, to verify them. 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. $\boldsymbol{d}_j[0] = 0$), then it sets $g = \boldsymbol{d}_j[1]$ and appends vector $[j, g]$ to $\boldsymbol{w}_{\mathcal{C}}$. Recall, $\boldsymbol{d}_j[1]$ refers to a rejected proof's index in proof vector $\boldsymbol{\pi}_j$. If $j = z$ and $\boldsymbol{w}_{\mathcal{C}} \neq \perp$, $\mathcal{C}$ sets $\boldsymbol{m}_{\mathcal{C}} : [\boldsymbol{w}_{\mathcal{C}}, \ddot{x}_{qp}, \text{“dispute”}]$.

4. **Dispute Resolution**. 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 $\boldsymbol{m}_{\mathcal{S}}$ to SC, at time $K_1$, where $K_1 > G_{z,2} + H$

(b) SC upon receiving $\boldsymbol{m}_{\mathcal{S}}$ does the following.

i. Sets two counters: $y_{\mathcal{C}}$ and $y_{\mathcal{S}}$ that are initially set to 0. Also, it constructs an empty vector $\boldsymbol{v}$

ii. Checks the validity of statement $\ddot{x}_{qp} \in \boldsymbol{m}_{\mathcal{S}}$, by sending it to SAP contract which returns 1 or 0. If the output is 0, then SC discards the server's complaint, $\boldsymbol{m}_{\mathcal{S}}$, and does not take steps 4(b)iii-4(b)v. Otherwise, it proceeds to the next step.

iii. Uses secret key $\bar{k} \in \ddot{x}_{qp}$ to decrypt the encrypted public key, $\texttt{Dec}(\bar{k}, \texttt{Enc}(\bar{k}, pk)) = pk$

iv. Removes from $\boldsymbol{v}_{\mathcal{S}}$ any element that is duplicated or is not in the range $[1, z]$

v. For any element $i \in \boldsymbol{v}_{\mathcal{S}}$:

- Fetches the related query, $\hat{k}_i$, from SC
- Checks if the query is well-formed, by calling $\texttt{PoRID.checkQuery}(\hat{k}_i, pk) \rightarrow b_i$. 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 $K_2$ be the time SC finishes the above checks.

(c) $\mathcal{C}$ sends $\boldsymbol{m}_{\mathcal{C}}$ to SC, at time $K_3$

(d) SC upon receiving $\boldsymbol{m}_{\mathcal{C}}$, does the following.

i. Checks the validity of statement $\ddot{x}_{qp} \in \boldsymbol{m}_{\mathcal{C}}$. To do that, it sends $\ddot{x}_{qp}$ to SAP contract which returns either 1 or 0. If the output is 0, then SC discards the client's complaint, $\boldsymbol{m}_{\mathcal{C}}$, and does not take steps 4(d)ii-4(d)iv. Otherwise, it proceeds to the next step.

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

iii. Uses secret key $\bar{k} \in \ddot{x}_{qp}$ to decrypt the encrypted public key, $\texttt{Dec}(\bar{k}, \texttt{Enc}(\bar{k}, pk)) = pk$

iv. For every vector $\boldsymbol{w} \in \boldsymbol{w}_{\mathcal{C}}$:

- Retrieves details of a proof that was rejected in each $i$-th verification. In particular, it sets $i = \boldsymbol{w}[0]$ and $g = \boldsymbol{w}[1]$. 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 query, $\hat{k}_i$, from SC.
- 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 $\texttt{PoRID.identify}(\boldsymbol{\pi}''_i, g, \hat{k}_i, pk) \rightarrow I_i$. If $I_i = \mathcal{S}$, then it increments $y_{\mathcal{S}}$ by 1. Otherwise, it does nothing.

Let $K_4$ be the time that SC finishes all the above checks.

5. **Coin Transfer**.

(a) Either $\mathcal{C}$ or $\mathcal{S}$ sends "pay" message and the statement: $\ddot{x}_{cp}$ to SC at time $L > K_4$

(b) 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

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

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

*Remark 9.* The reason in step 4(d)iv 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 10.* 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.

## 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. Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A.: Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM Conference on Computer and Communications Security, CCS 2017 (2017)
5. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02

6. 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)
7. 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)
8. 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)
9. Juels, A., Jr., B.S.K.: Pors: Proofs of retrievability for large files. IACR Cryptology ePrint Archive 2007, 243 (2007)
10. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)
11. 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
12. 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)
13. 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)
14. Miller, A., Juels, A., Shi, E., Parno, B., Katz, J.: Permacoin: Repurposing bitcoin work for data preservation. In: S&P'14
15. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91
16. Shacham, H., Waters, B.: Compact proofs of retrievability. In: ASIACRYPT. pp. 90–107 (2008)
17. Shen, S., Tzeng, W.: Delegable provable data possession for remote data in the clouds. In: ICICS 2011