2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00049

Recurring Contingent Service Payment

Aydin Abadi
University College London, UK
aydin.abadi@ucl.ac.uk

Steven J. Murdoch
University College London, UK
s.murdoch@ucl.ac.uk

Thomas Zacharias
University of Edinburgh, UK
thomas.zacharias@ed.ac.uk

***Abstract*—Fair exchange protocols let two mutually distrustful parties exchange digital data in a way that neither party can cheat. They have various applications such as the exchange of digital items, or the exchange of digital coins and digital services between a buyer/client and seller/server.**

In this work, we formally define and propose a *generic* blockchain-based construction called "*Recurring Contingent Service Payment*" (RC-S-P). It (i) lets a fair exchange of digital coins and verifiable service *reoccur securely* between clients and a server while ensuring that the server is paid if and only if it delivers a valid service, and (ii) ensures the parties' privacy is preserved. RC-S-P supports arbitrary verifiable services, such as "Proofs of Retrievability" (PoR) or verifiable computation and imposes low on-chain overheads. Our formal treatment and construction, for the first time, consider the setting where either client or server is malicious.

We also present a concrete *efficient* instantiation of RC-S-P when the verifiable service is PoR. We implemented the concrete instantiation and analysed its cost. When it deals with a 4-GB outsourced file, a verifier can check a proof in only 90 milliseconds, and a dispute between a prover and verifier is resolved in 0.1 milliseconds.

At CCS 2017, two blockchain-based protocols were proposed to support the fair exchange of digital coins and a certain verifiable service; namely, PoR. In this work, we show that these protocols (i) are susceptible to a *free-riding attack* which enables a client to receive the service without paying the server, and (ii) are not suitable for cases where parties' privacy matters, e.g., when the server's proof status or buyer's file size must remain private from the public. RC-S-P simultaneously mitigates the above attack and preserves the parties' privacy.

1. Introduction

Fair exchange is an interesting problem in which two mutually distrustful parties want to swap digital items such that neither party can cheat the other, in the sense that either each party gets the other's item, or neither party does. It captures various real-world scenarios; for instance, when two parties want to exchange digital items or when a seller wants to sell a digital verifiable service in exchange for digital coins. Solutions to the problem are usually certain cryptographic schemes, called fair exchange protocols, and have been studied for decades. It has been shown that fairness is unachievable without the aid of a trusted third party [14].

With the advent of decentralised cryptocurrencies and blockchain, it seemed fair exchange protocols can be designed without having to rely on a single trusted third party, in the sense that the third party's role can be turned into a computer program, i.e., smart contract, which is maintained and executed by the decentralised blockchain. This ultimately results in a stronger security guarantee, as there would be no need to trust a single entity, anymore. Ever since various fair exchange protocols that rely on blockchain have been proposed. They mainly support the fair exchange of two digital items (e.g., documents, films) or a digital item and coins (except for the ones in [12] that will be discussed shortly).

Our Contributions. In this work, we:

1) define and propose the first generic construction, called "*recurring contingent service payment*" (RC-S-P), that (i) supports the fair exchange of digital verifiable services and coins and (ii) preserves the parties' privacy. RC-S-P makes black-box use of any scheme that offers a verifiable service and remains secure in the recurring setting where the construction is executed many times.
2) propose the first recurring contingent PoR payment (RC-PoR-P). It is a concrete efficient instantiation of the RC-S-P scheme. To achieve efficiency, it avoids generic cryptographic tools and utilises mainly symmetric-key primitives and smart contracts.
3) implement RC-PoR-P and analyse its cost. Our cost analysis illustrates RC-PoR-P is highly efficient. When it deals with a 4-GB outsourced file, in each verification, a verifier can check a proof in only 90 milliseconds, and a dispute between a prover and a verifier can be resolved in 0.1 milliseconds. Also, the contracts' computation is constant in file size. We have made the implementation source code publicly available.
4) identify a free-riding attack in the state-of-the-art fair exchange protocols that were designed to support the fair exchange of a verifiable service and coins, i.e., the two (publicly and privately verifiable) protocols of Campanelli *et al*. [12]. We show that the attack lets an adversary use a service without paying the fee.
5) show that the protocols of [12] are not suitable for cases where parties' privacy matters. We argue that the schemes leak in real-time non-trivial fresh information about the seller and buyer to the public, e.g., deposits' actual amount and proof status.

The identified issues in the protocols of [12] indicate a need for a secure mechanism like RC-S-P. Our RC-S-P can be used to prevent a variant of *Authorised Push Payment* (APP) fraud, called purchase fraud, where a service provider may wish to receive a certain amount of coin

© 2023, Aydin Abadi. Under license to IEEE.

DOI 10.1109/EuroSP57164.2023.00049
Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

without delivering the service[1]. Defining and designing generic RC-S-P is challenging, for three reasons: (i) there exists no generic definition for *verifiable service* (VS) schemes in the literature, (ii) most of the application-specific VS schemes (e.g., proofs of retrievability [48], or verifiable computation [25], verifiable searchable encryption [42]) assume the client is trusted, while in a fair exchange setting either party can be an active adversary, and (iii) the majority of VS schemes do not (need to) consider the privacy of exchanged messages, as they are in the traditional setting where the client and server directly interact with each other; hence, their messages' privacy can be protected from the public by using secure channels.

The primary novelties of this work include: (a) RC-S-P, a fair exchange protocol which remains provably secure in the case where either server or client acts malicious while preserving the parties' privacy, (b) RC-PoR-P, which inherits all appealing features of RC-S-P for the special case of PoR without using zero-knowledge proofs, and (c) identifying and addressing the free-riding attack in the state-of-the-art fair exchange protocols, proposed in [12].

2. Related Work

In this section, we summarise related work. In the paper's full version [4], we present a detailed survey. Maxwell [39] proposes a fair exchange scheme, called "zero-knowledge contingent payment" that supports the fair exchange of digital goods and coins. It is based on Bitcoin's smart contracts, a hash function, and zero-knowledge (zk) proofs. After the advancement of the "succinct non-interactive argument of knowledge" (zk-SNARK) [26] that yields more efficient zk proofs, the scheme was modified to use zk-SNARKs. Later, Campanelli *et al.* [12] identified an issue in the above scheme. The issue lets a malicious buyer receive the item without paying. To address it, the authors propose the "zero-knowledge Contingent Service Payments" (zkCSP) scheme that also supports contingent payment for digital services. It is based on Bitcoin smart contracts, hash functions, and witness indistinguishable proof of knowledge. To improve efficiency, they use zk-SNARKs where the buyer generates a public parameter, i.e., CRS, and the seller performs minimal checks on the CRS. The authors, as the zkCSP's concrete instantiations, propose public and private verifiable schemes where the service is "proofs of retrievability" (PoR) [48]. To date, they are the only ones designed for the fair exchange of digital coins and a digital service. Shortly, we will explain their shortcomings undetected in the literature.

Fuchsbauer [24] identifies a flaw in the zkCSP and shows that the seller's minimal check in the zkCSP does not prevent the buyer from cheating. Later, Nguyen *et al.* [45] show that by relying on a stronger assumption, the zkCSP remains secure. Tramer *et al.* [50] propose a fair exchange scheme that uses trusted hardware and Ethereum smart contracts. Dziembowski *et al.* [22] propose FairSwap, a fair exchange scheme using the Ethereum smart contracts and the notion of proof of misbehaviour [13]. Later, Eckey *et al.* [23] propose OPTISWAP that improves FairSwap's performance. Similar to FairSwap, OPTISWAP uses a smart contract and proof of misbehaviour, but it relies on an interactive dispute resolution protocol. Recently, outsourced fair PoRs letting a client delegate the verifications to a smart contract were proposed in [3], [21]. The scheme in [3] uses message authentication codes (MACs) and time-lock puzzles. The one in [21] uses polynomial commitment and involves a high number of exponentiations. As a result, it imposes higher costs, of proving and verifying, than the former scheme. The schemes in [3], [21] assume the client is honest.

To date, the zkCSP (in [12]) remains the only protocol designed to support the fair exchange of digital coins and a verifiable service; accordingly, it is the closest work to ours. The rest of the above schemes are out of the scope of our work because they only support the exchange of two items or the exchange of an item and digital coins.

1. We refer readers to [51] for further discussion about APP fraud.

3. Preliminaries

We use λ as the security parameter. We write $x \xleftarrow{\$} X$ to denote that x is chosen uniformly at random from set X. We write $\mathsf{negl}(\lambda)$ to denote that a function is negligible in λ, i.e., asymptotically smaller than the inverse of any polynomial. In the formal definitions in this paper, we use the notation $\Pr\left[\frac{\mathsf{Exp}}{\mathsf{Cond}}\right]$, where Exp is an experiment that involves an adversary $\mathcal{A}$, and Cond is the set of the corresponding winning conditions for $\mathcal{A}$. We use $\mathcal{C}$, $\mathcal{S}$, and $\mathcal{R}$ to denote the client, server, and arbiter, respectively. We let pl be $\mathcal{S}$'s public price list, o be the amount paid to $\mathcal{S}$ for each valid proof, and l be the amount (misbehaving) $\mathcal{C}$ or $\mathcal{S}$ pays to $\mathcal{R}$ for resolving a dispute for each verification, o_{max} be the maximum amount paid to $\mathcal{S}$ for a valid proof, l_{max} be the maximum amount to resolve a potential dispute, and z be the total number of verifications and $(o, l, o_{max}, l_{max}) \in pl$.

We provide a notation table in Appendix A. Similar to the *optimistic* fair cryptographic protocols that aim efficiency, e.g., in [7], [8], [18], we assume the existence of a trusted third party arbiter which remains offline most of the time and is only invoked to resolve disputes.

3.1. Smart Contract (SC)

Cryptocurrencies, such as Bitcoin [44] and Ethereum [53], beyond offering a decentralised currency, support computations on transactions. In this setting, often a certain computation logic is encoded in a computer program, called a *"smart contract"*. To date, Ethereum is the most predominant cryptocurrency framework that enables users to define arbitrary smart contracts. In this framework, contract code is stored on the blockchain and executed by all parties maintaining the cryptocurrency. To prevent a denial of service attack, the framework requires a transaction creator to pay a fee, called *"gas"*. In this work, we require *minimal capabilities* of Ethereum smart contracts, i.e., given a set of addresses, a certain amount of deposit that the contracts hold, certain integer variables, and simple equations registered to the contracts (i.e., linear combinations of the variables), it distributes among the account holders, a portion of the deposit specified by the equations' output. We assume the smart contract and underlying blockchain are secure, i.e., the used signature scheme is unforgeable and the blockchain is immutable.

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

3.2. Building Blocks

We outline the main cryptographic primitives that we utilize in our protocols. We provide a detailed description of the said primitives in Appendix B.1.

- *Pseudorandom Fuction* (PRF): we apply a pseudorandom function $\texttt{PRF} : \{0,1\}^{\psi} \times \{0,1\}^{\eta} \rightarrow \{0,1\}^{\iota}$ that on input a random ψ-bit key and η-bit message, it outputs a ι-bit pseudorandom value (cf. Appendix B.1.1).
- *Commitment Scheme*: we deploy a binding and hiding commitment scheme. In the *commit* phase, the sender commits to a message x as $\texttt{Com}(x,r) = \texttt{Com}_x$, that involves a secret value, r. In the *open* phase, the sender sends the opening $\ddot{x} := (x,r)$ to the receiver which verifies its correctness: $\texttt{Ver}(\texttt{Com}_x, \ddot{x}) \stackrel{?}{=} 1$ and accepts if the output is 1 (cf. Appendix B.1.2).
- *Publicly Verifiable Non-interactive Zero-knowledge Proof* (NIZK): is a non-interactive proof where a prover $\mathcal{P}$, given a witness w for some statement x in an NP language L, wants to convince in zero-knowledge a verifier $\mathcal{V}$ of the validity of $x \in L$. A NIZK is publicly verifiable when any party can verify the validity of $x \in L$ by obtaining the proof (cf. Appendix B.1.3).
- *Symmetric-key Encryption Scheme*: it consists of a key generation algorithm SKE.keyGen, an encryption algorithm Enc, and a decryption algorithm Dec. We require that the scheme satisfies IND-CPA security (cf. Appendix B.1.4).
- *Digital Signature Scheme*: it consists of a key generation algorithm Sig.keyGen, a signing algorithm Sig.sign, and a verification algorithm Sig.ver. We require that the digital signature scheme satisfies EUF-CMA security (cf. Appendix B.1.5).
- *Merkle Tree*: A Merkle tree scheme [40], [41] is a data structure often used for efficiently checking the integrity of an outsourced file. The Merkle tree scheme includes three algorithms; namely, MT.genTree, MT.prove, and MT.verify. Briefly, the first algorithm constructs a Merkle tree on file blocks, the second generates a proof of a block's (or set of blocks') membership, and the third one verifies the proof (cf. Appendix B.1.6).

3.3. Proofs of Retrievability (PoR)

A PoR scheme considers the case where an honest client wants to outsource the storage of its file to a potentially malicious server, i.e., an active adversary. It is a challenge-response interactive protocol, where the server proves to the client that its file is intact and retrievable. Informally, a PoR's soundness requires that if a prover convinces the verifier, then the file is stored by the prover. This is formalized via the notion of an extractor algorithm that can extract the file in interaction with the adversary. Appendix B.2 presents the PoR's formal definition.

We briefly describe the privately verifiable PoR in [48] because it was used as a subroutine in zkCSP's concrete instantiation in [12] which we are going to expose its vulnerabilities. In the setup phase, the client splits its file u into fixed size blocks $u = m_1, ..., m_n$. It generates a tag, i.e., Message Authentication Code (MAC), for each block m_i as $\sigma_i = r_i + \alpha \cdot m_i$, where $r_i = \texttt{PRF}(k,i)$, α is a random value and k is the client's secret key. It outsources the storage of all blocks and tags to the server. Later, to check whether the file is still retrievable, it sends c pair of the form (j, v_j) to the server, where j is a block index and v_j is a random value. Let set J contain all indices that the client sends to the server. The server generates and then sends to the client a proof pair (σ, β), where $\sigma = \sum_{\forall j \in J} v_j \cdot \sigma_j$ and $\beta = \sum_{\forall j \in J} v_j \cdot m_j$. The client can verify the proof, by checking if $\sigma = \alpha \cdot \beta + \sum_{\forall j \in J} v_j \cdot r_j$.

Shacham and Waters [48] also propose a publicly verifiable PoR, which requires the client to generate a signature, as a tag, for each block.

4. Putting Forth the RC-S-P Concept

In this section, we briefly describe the Recurring Contingent Service Payment (RC-S-P) concept and explain why it is needed. RC-S-P concerns fair exchange of a digital verifiable service (offered by a server) and digital coins (offered by a client), and supports any verifiable digital service. Broadly speaking, it ensures that a client pays a predefined amount of digital coins to the server *if and only if* the server (proves that it) provided the promised service. It considers the case where either client or server is potentially an active/malicious adversary. RC-S-P ensures the above security guarantees hold even if the payments reoccur, e.g., the server deals with multiple clients. RC-S-P also preserves the two parties' privacy.

4.1. Overview of RC-S-P

RC-S-P involves a client $\mathcal{C}$, a server $\mathcal{S}$, an arbiter $\mathcal{R}$, and a smart contract SC. It comprises eight phases: (1) *key generation*, in which $\mathcal{C}$ generates the system parameters, (2) *client-side initiation*, where $\mathcal{C}$ encodes the service input, generates metadata, computes a proof asserting that the input and metadata are well-formed, and masks the actual amount of coin it wants to pay for the service, (3) *server-side initiation*, where $\mathcal{S}$ checks $\mathcal{C}$'s proof, ensures the input and metadata are well-formed, and masks the actual deposits it puts on SC, (4) *client-side query generation*, where $\mathcal{C}$ generates an encoded query, (5) *server-side proof generation*, where $\mathcal{S}$ generates an encoded proof asserting that the service was delivered correctly, (6) *client-side proof verification*, where $\mathcal{C}$ checks $\mathcal{S}$'s proof, (7) *dispute resolution*, where $\mathcal{R}$ compiles the validity of a complaint made by $\mathcal{S}$/$\mathcal{C}$, and (8) *coin transfer*, where SC distributes parties' deposits depending on their (mis)behaviour.

Intuitively, RC-S-P is said to be secure if it satisfies three main security properties: (i) *security against a malicious server*: an adversary corrupting $\mathcal{S}$ wins only with a negligible probability if it does not provide the promised service but persuades $\mathcal{C}$ to accept it, or makes $\mathcal{C}$ or $\mathcal{R}$ withdraw an incorrect amount of coins they deposited in SC; (ii) *security against a malicious client*: an adversary corrupting $\mathcal{C}$ wins only with a negligible probability if it provides an invalid metadata/query but convinces the $\mathcal{S}$ or $\mathcal{R}$ to accept it, or make $\mathcal{S}$ or $\mathcal{R}$ withdraw an incorrect amount of coins from SC; (iii) privacy: the privacy of (a) the service input (e.g., outsourced file), and (b) the service proof's status is preserved (for a predefined time period).

RC-S-P offers stronger security guarantees compared to the existing solutions in the real world and literature; below, we explain why that is the case.

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

4.2. Service Payment in the Real World

To date, in the real world, each client must pay *in advance* to a server for a digital service that it wishes to use in future, e.g., Dropbox. If a malicious server does not provide the promised service, then the client has to either (i) spend time and effort to follow up the matter (i.e., by arguing the matter with the server or taking legal actions) or (ii) ignore the problem resulting in the loss of the money it paid for the service. Thus, it is important to ensure that an honest client pays only if the server provides the promised service.

4.3. Service Payment in the Literature

The state-of-the-art protocol in [12] that has been designed to support a fair exchange of digital services and coins has an important oversight; namely, it does not protect an honest server from a malicious client. This oversight allows a malicious client to mount an attack which enables it to use the service without paying the server, i.e., a free-riding attack (see Section 5.2 for further details). So, it is important to ensure that an honest server provides the service only if the client pays for it.

4.4. Privacy

The existing fair exchange protocol designed for service payment in [12] uses a public blockchain to hold clients' deposits and to transfer the deposit to the server if the service is delivered. Nevertheless, the use of blockchain in the fair exchange protocol reveals in real-time non-trivial information about the parties involved, e.g., the deposits' actual amount or service type (see Section 5.1 for further discussion). Therefore, it is important to have a fair exchange protocol that could also preserve the parties' privacy.

5. The Privacy Issue and Attack

In this section, we elaborate on the lack of privacy of zkCSP and the attack. We first explain why the zkCSP schemes do not preserve privacy. Later, in the attack description, we show how the lack of privacy can benefit an attacker. We focus only on the zkCSP protocols in [12], as they have been specifically designed for a fair exchange of verifiable services and digital coins, whereas the other protocols studied in Section 2 were designed for a fair exchange of digital items, e.g., a file and coins.

5.1. Lack of Privacy in zkCSP

The zkCSP protocols reveal in real-time non-trivial fresh information about the server and clients to the public. The revealed information includes (i) proof status and (ii) deposit amount. We explain them below.

5.1.1. Proof status. In the traditional setting, the client and server directly interact with each other to verify and prove the integrity of agreed-upon services. In this case, the verification's result is only apparent to them. Nevertheless, in the blockchain era, where a blockchain plays a role in the verification and payment phases (e.g., in the zkCSP schemes) it becomes visible in *real-time* to *everyone* whether the verification (proof) has been accepted, which reflects whether the server has successfully delivered the service. This issue remains even if the service proofs are not stored (in plaintext) in the blockchain, as the *coins transfer* itself reveals the status of proofs.

In certain settings, this leakage might be undesirable and could have *immediate* consequences for both the server and (business) clients, e.g., stock value drop [10], [31], or can benefit attackers (as we will explain in Section 5). For further discussion on proofs status leakage, we refer readers to the paper's full version [4].

5.1.2. Deposit amount. The amount of deposit placed in the smart contract, swiftly reveals non-trivial information about the client to the public. In the case of PoR, an observer learns the approximate size of outsourced data, service type, or in certain cases even the region of clients' outsourced data, by comparing the amount of deposit with the service provider's price list which is often publicly available, e.g., in [5], [20], [29].

5.2. Free-Riding Attack

In this section, we describe an attack scenario in which a malicious client (the attacker) is served by an honest server but it would not pay the service fee, i.e., the free-riding attack. We explain the attack for a concrete instantiation of the zkCSP, when it uses the privately verifiable PoR of Shacham and Waters [48] as a subroutine. This instantiation was presented in [12, Section 5.2].

Briefly, to mount the attack, the client exploits (1) the lack of privacy in zkCSP and (2) the lack of server-side verification mechanism in the PoR that zkCSP uses.

First, we explain how the lack of privacy in zkCSP benefits the attacker. As we discussed in Section 5.1.1, proof status is revealed in real time to everyone including the attacker. Observing proofs' status (when a server deals with multiple clients) over a sufficiently long time (e.g., a few months) allows the attacker to construct comprehensive background knowledge of the server's behaviour, e.g., the server has been acting honestly or not suffering from hardware failures that affect its clients' data.

Next, we explain how the lack of server-side verification mechanism in the PoR that zkCSP uses can benefit the attacker, who acts as follows. At the setup phase in PoR, it generates **ill-formed tags**. Specifically, instead of honestly generating a tag (MAC) σ_i on a file block m_i as $\sigma_i = r_i + \alpha \cdot m_i$ (cf. Subsection 3.3), it generates a tag for some arbitrary block m'_i; i.e., $\sigma'_i = r_i + \alpha \cdot m'_i$, where $m_i \neq m'_i$. It follows the rest of PoR protocol honestly, with one exception; namely, during the verification, when it sends a query containing some pairs (j, v_j), it also includes pair (i, v_i) in the query, where i is the index of a block whose tag is ill-formed.

In this case, the server cannot pass the verification; we show why it is the case. For simplicity, we let the attacker send to the server only two pairs (j, v_j) and (i, v_i). Given the pairs, the honest server computes proof pair (σ, β) as follows $\sigma = v_j \cdot \sigma_j + v_i \cdot \sigma'_i = v_j \cdot (r_j + \alpha \cdot m_j) + v_i \cdot (r_i + \alpha \cdot m'_i)$ and $\beta = v_j \cdot m_j + v_i \cdot m_i$. However, this proof is invalid, as if a verifier follows the PoR verification (described in Section 3.3), it will get the inequality:

$$\begin{aligned}\sigma &= v_j \cdot (r_j + \alpha \cdot m_j) + v_i \cdot (r_i + \alpha \cdot m'_i) \neq \\ &\neq \alpha \cdot (v_j \cdot m_j + v_i \cdot m_i) + v_j \cdot r_j + v_i \cdot r_i = \\ &= \alpha \cdot \beta + v_j \cdot r_j + v_i \cdot r_i.\end{aligned}$$

This means that, in zkCSP, an honest server cannot generate valid proof when i-th block is challenged. In

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

zkCSP, the server cannot detect whether the attacker generated ill-structured tags, as the used PoR does not require a client to prove the tags' correctness. Even after the server fails to generate a valid proof, it cannot tell (or prove to anyone) whether itself (e.g., due to hardware failures) or the malicious client was the source of the problem.

Hence, the malicious client can mount the free-riding attack to avoid paying the server who delivered the services honestly, i.e., kept the file intact during the period between the setup and verification phases. To do that, it (i) ensures the server is honest (by exploiting the lack of privacy in zkCSP), (ii) generates ill-formed tags (by exploiting the lack of server-side verification mechanism in the PoR that zkCSP uses), and then (iii) asks the server to generate proof for the blocks related to ill-formed tags.

In another concrete instantiation of the zkCSK, Campanelli *et al*. used a publicly verifiable PoR as a subroutine. The idea behind the above free-riding attack can be easily applied to that instantiation too.

Main source of the attack. The reason that the attacker can construct ill-structure tags, without being detected, is that the zkCSP scheme uses a subprotocol that offers a weaker security guarantee than required. Specifically, it assumes *either party* can be potentially corrupted by an active adversary, yet it uses a certain verifiable service protocol (i.e., PoR) that is secure against *only* a malicious server and assumes the client is fully honest. This **mismatch of security assumption/requirement** lets a malicious client misbehave without any consequences.

6. Overview of our Solution

Ensuring Security Holds When Either Party is Malicious. We design the RC-S-P in a modular fashion. First, we formally define the notion of Verifiable Service (VS) and then upgrade VS to a "Verifiable Service with IDentifiable abort" (VSID) inspired by the notion of "secure multi-party computation with identifiable abort" [33]. The latter guarantees that not only the service takes into consideration that the client can be malicious too, but also a third-party 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 (similar to the protocol in [3]) and it is forced to provide correct inputs, via NIZK (similar to the scheme in [6]); otherwise, its deposit is sent to the server. Third, we require parties to post their messages to the contract, to avoid any potential repudiation issue, which is a standard technique. Forth, we let the party which resolves disputes get paid by a corrupt party (similar to the protocol in [19]). The combination of the above techniques allows RC-S-P to deal with the free-riding attack as well.

Now we explain how the solution works. Before using the service, the client 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. The server deposits l coins. Then, the client and server engage in the VSID protocol such that (the encryption of) messages exchanged between them are put in the contract. They perform the verifications locally. When a party detects misbehaviour, it can raise a dispute that invokes the arbiter which checks the party's claim, off-chain. The arbiter sends the output of the verification to the contract. If the party's claim is valid, then it can withdraw its coins and the arbiter is paid by the misbehaving party. If the party's claim is invalid, that party has to pay the arbiter and the other party can withdraw its deposit. If 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.

We provide a formal definition of VS in Appendix C. Since VSID as a separate notion which might be of independent interest too, we provide its definition, construction, and proof in Appendix D. We note that in the concrete instantiation of our generic solution in which the VS is PoR, we will use a Merkle tree and *proof of misbehaviour* letting us avoid using NIZK and reduce arbiter-side computation (cf. Section 9).

Preserving Parties' Privacy. To preserve the parties' privacy and prevent real-time information leakage, we use the following ideas. First, to hide proof status, we let the client and server take control of the time of the information release. This lets them keep the information private from the public within a certain period, and release it when it loses its sensitivity.[2] Specifically, they agree on the period in which the information must remain hidden, "private time bubble". During this period, all messages sent to the contract are encrypted and the parties do not raise any dispute. They raise disputes after the private time bubble ends (or bubble bursts).

Nevertheless, the client/server can still find out whether a proof is valid when it is provided by its counterparty, because it can locally verify the proof. Second, to hide the amount of deposit, we let each party "mask" its coins, by increasing the actual coins amount to the maximum amount of coins in the server's price list. So, the masked coins hide the actual coins amount from the public. But, this raises another challenge: *how can the mutually untrustful parties claim back their masking coins (i.e., the difference between the maximum and actual coins amount) after the bubble bursts, while hiding the actual coins amount from the public in the private time bubble?*

Our third idea, which addresses this challenge, is to let the client and server, at the beginning of the protocol, agree on a private statement specifying the deposit details (e.g., parties' actual coins amount for the service, dispute resolution, or masking). Later, when they want to claim their coins, they also provide the statement to the contract which checks the statement validity and if it is accepted, it distributes coins according to the statement (and the contract status). We will show how they can efficiently agree on such a statement, by using a statement agreement protocol (SAP). In Appendix E.1.2 we also show how they can promise their locked share of coins to a third party.

Our generic framework that offers the above features is called "Recurring Contingent Service Payment" (RC-S-P). Also, as a concrete instantiation of RC-S-P, we present the RC-PoR-P protocol, in which the VS is PoR.

Strawman Solutions. To address the issue related to the leakage of deposit amount, one may use privacy-preserving cryptocurrency frameworks, e.g., Zerocash [11]

2. The concept of delayed information release has already been used by researchers, e.g., in smart metering in [32], and in the real world through the declassification approach taken by most democratic countries which declassify sensitive information after the information loses its sensitivity.

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

or Hawk [36]. Although such frameworks solve this problem, they impose additional high cost to their users, as each transaction involves a generic proofs system that are computationally expensive. Also, one might want to let the server pick a fresh address for each verifier to preserve its pseudonymity with the hope that an observer cannot link clients to a server (so proofs status and deposit amount issues can be addressed). But, for this to work, we have to assume that multiple service providers use the same protocol on the blockchain and all of them are pseudonymous which is a strong assumption.

Design Choices. We make certain design choices that are not present in the zkCSP of [12], to keep our RC-S-P efficient. Specifically, since not only the server but also the clients can be malicious, we (a) involve a third-party arbiter, and (b) require parties to store more messages on a blockchain, for dispute resolution to be feasible. To achieve privacy while avoiding costly cryptographic computations, we require parties to deposit extra coins to mask the actual value of their deposit. To allow fair distribution of the parties' deposit (without having to use the arbiter for the distribution) we use smart contracts.

7. RC-S-P Definition

In this section, we introduce a formal definition of RC-S-P. Before presenting the formal definition of RC-S-P, we outline what a *verifiable service* (VS) is. At a high level, a VS scheme is a two-party protocol in which a client chooses a function, F, and provides (an encoding of) F, its input u, and a query $\boldsymbol{q}$ to a server, which is expected to evaluate F on u and $\boldsymbol{q}$ (and some public parameters) and respond with the output. Then, the client verifies that the output is indeed the output of the function computed on the provided input. In verifiable services, either the computation (on the input) or both the computation and storage of the input are delegated to the server. We present a full formal definition of a VS scheme in Appendix C.

Definition 1 (RC-S-P Scheme)**.** A recurring contingent service payment scheme RC-S-P involves four parties; namely, a client, server, arbiter, and smart contract (which represents a bulletin board). The scheme is parameterized by five functions:

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

The scheme consists of eight algorithms defined below.

$\underline{\texttt{RCSP.keyGen}(1^\lambda) \rightarrow \boldsymbol{k}}$: It is run by client $\mathcal{C}$. It takes as input security parameter 1^λ. It outputs $\boldsymbol{k} := (k, k')$ that contains a secret and public verification key pair $k := (sk, pk)$ and a set of secret and public parameters, $k' := (sk', pk')$. It sends pk and pk' to the contract.

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

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

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

$\underline{\texttt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp}) \rightarrow (b_j, m_{\mathcal{S},j}, \boldsymbol{\pi}^*_j)}$: It is run by $\mathcal{S}$. It takes as input u^*, metadata σ, c^*_j, pk, and T_{qp}. It checks the validity of decoded query pair $c_j = D(c^*_j, T_{qp})$. If it is rejected, then it sets $b_j = 0$ and constructs a complaint $m_{\mathcal{S},j}$. Otherwise, it sets $b_j = 1$ and $m_{\mathcal{S},j} = \perp$. It outputs $b_j, m_{\mathcal{S},j}$, and encoded proof $\boldsymbol{\pi}^*_j = E(\pi_j, T_{qp})$, where $\boldsymbol{\pi}_j$ contains $h_j = F(u^*, \boldsymbol{q}_j, pp)$ and a proof δ_j asserting the evaluation is performed correctly ($\boldsymbol{\pi}_j$ may contain dummy values if $b_j = 0$). The smart contract is given $\boldsymbol{\pi}^*_j$.

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

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

$\boldsymbol{y}'_{\mathcal{P}}$ (by setting an element of it to one) if party $\mathcal{P}$ has provided a complain that does not allow it to identify a misbehaved party, in the j-th verification, i.e., when the arbiter is unnecessarily invoked.

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

The above algorithms $\texttt{RCSP.genQuery}$, $\texttt{RCSP.prove}$, $\texttt{RCSP.verify}$, and $\texttt{RCSP.resolve}$ implicitly take $(a, coin^*_{\mathcal{S}}, p_{\mathcal{S}})$ as other inputs and execute only if $a = 1$ and $coin^*_{\mathcal{S}} = p_{\mathcal{S}}$; but, for simplicity we avoided explicitly stating it in the definition. An RC-S-P scheme must meet correctness and security. Correctness requires that by the end of the protocol's execution (that involves honest client and server), the 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 a 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 (that is not involved) receives nothing. Correctness is formally stated below.

Definition 2 (Correctness)**.** An RC-S-P scheme with functions F, M, E, D, Q is *correct* for auxiliary information aux if for any z polynomial in λ, any price list pl, and any service input u, it holds that the following probability is equal to 1:

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

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

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

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

Definition 3 (Security Against Malicious Server)**.** An RC-S-P scheme with functions F, M, E, D, Q is *secure against a malicious server* for auxiliary information aux, if for any z polynomial in λ, any price list pl, every j (where $1 \leq j \leq z$), and any PPT adversary $\mathcal{A}$, it holds that the following probability is $\mathsf{negl}(\lambda)$:

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

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

Informally, security against a malicious client requires that, for each j-th verification, a malicious client with a negligible probability wins if it provides either (a) valid metadata and query but either makes the server receive an incorrect amount of coins (something other than its deposit plus the service payment), or makes the arbiter withdraw incorrect amounts of coin if it unnecessarily invokes the arbiter or (b) invalid metadata or query but convinces the server to accept either of them (i.e., the invalid metadata or query), or (c) invalid query but persuades the arbiter to accept it, or makes them withdraw an incorrect amount of coins (i.e., $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o$ or $coin_{\mathcal{R},j} \neq l$ coins). Below, we formally state the property. Note that in the following definition, an honest server either does not deposit (e.g., when $a = 0$) or if it deposits (i.e., agrees to serve) ultimately receives its deposit *plus the service payment* (with high probability).

Definition 4 (Security Against Malicious Client)**.** An RC-S-P scheme with functions F, M, E, D, Q is *secure against a malicious client* for auxiliary information aux, if for any z polynomial in λ, any price list pl, every j (where $1 \leq j \leq z$), and any PPT adversary $\mathcal{A}$, it holds

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

that the following probability is $\mathsf{negl}(\lambda)$:

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

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

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

Definition 5 (Privacy)**.** An RC-S-P scheme with functions F, M, E, D, Q preserves *privacy* for auxiliary information aux if for any z polynomial in λ and any price list pl, the following hold:

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

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

where $\boldsymbol{c}^* = [c^*_1, ..., c^*_z]$ and $\boldsymbol{\pi}^* = [\boldsymbol{\pi}^*_1, ..., \boldsymbol{\pi}^*_z]$.

2) For any PPT adversaries $\mathcal{A}_2$, $\mathcal{A}_3$, and $\mathcal{A}_4$ the following probability is no more than $Pr_{\max} + \mathsf{negl}(\lambda)$:

$$\Pr\left[\begin{array}{l}\texttt{RCSP.keyGen}(1^\lambda) \rightarrow \boldsymbol{k}\\ \mathcal{A}_2(1^\lambda, pk, F, M, E, D, Q, z, pl) \rightarrow u\\ \texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, M, z, pl, enc) \rightarrow (u^*, e, T, p_{\mathcal{S}}, \boldsymbol{y}, coin^*_{\mathcal{C}})\\ \texttt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y}) \rightarrow (coin^*_{\mathcal{S}}, a)\\ \text{For } j = 1, \ldots, z \text{ do :}\\ \quad \mathcal{A}_2(1^\lambda, \text{aux}, k, T_{qp}) \rightarrow c^*_j\\ \quad \mathcal{A}_3(u^*, \sigma, c^*_j, pk, T_{qp}) \rightarrow (b_j, m_{\mathcal{S},j}, \boldsymbol{\pi}^*_j)\\ \quad \texttt{RCSP.verify}(\boldsymbol{\pi}^*_j, c^*_j, k, T_{qp}) \rightarrow (d_j, \boldsymbol{m}_{\mathcal{C},j})\\ \hline \mathcal{A}_4(F, M, E, D, Q, \boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl, a) \rightarrow (d_j, j)\end{array}\right]$$

where $\boldsymbol{\pi}^*_j$ has been encoded correctly, $\boldsymbol{\pi}^*_j = [h^*_j, \delta^*_j], h_j = D(h^*_j, T_{qp})$, and $Pr_{\max}$ is defined as follows. Let $Exp^{\mathcal{A}_2,\mathcal{A}_3}_{\mathsf{priv}}(1^\lambda)$ be the above experiment. Let $\boldsymbol{q}_j \in D(c^*_j, T_{qp})$, $pp \in T_{qp}$. We define the events $Con^{(1)}_{0,j} : Q(\text{aux}, k, pp) \neq \boldsymbol{q}_j$, $Con^{(2)}_{0,j} : b_j = 0, Con^{(1)}_{1,j} : Q(\text{aux}, k, pp) = \boldsymbol{q}_j$, $Con^{(2)}_{1,j} : b_j = 1$, $\overline{Con}^{(1)}_{0,j} : F(u^*, \boldsymbol{q}_j, pp) \neq h_j$, $\overline{Con}^{(2)}_{0,j} : d_j = 0$, $\overline{Con}^{(1)}_{1,j} : F(u^*, \boldsymbol{q}_j, pp) = h_j$, and $\overline{Con}^{(2)}_{1,j} : d_j = 1$. For $i \in \{0,1\}$ and $j \in [z]$, let

$$Pr_{i,j} := \Pr\left[\frac{Exp^{\mathcal{A}_2,\mathcal{A}_3}_{\mathsf{priv}}(1^\lambda)}{\begin{array}{l}\Big(Con^{(1)}_{i,j} \ \wedge \ Con^{(2)}_{i,j}\Big) \vee\\ \Big(\overline{Con}^{(1)}_{i,j} \ \wedge \ \overline{Con}^{(2)}_{i,j}\Big)\end{array}}\right]. \text{ Then,}$$

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

In the above definition, for each j-th verification, the adversary $\mathcal{A}_2$ or $\mathcal{A}_3$ produces an invalid query or invalid proof, respectively, with probability $Pr_{0,j}$ and a valid query or valid proof, respectively, with probability $Pr_{1,j}$. It is required that privacy is preserved regardless of the queries and proofs status, i.e., whether they are valid/invalid, as long as they are correctly encoded and provided. In the above definitions, the private time bubble is a time period from the point when $\texttt{RCSP.keyGen}(\cdot)$ is executed up to the time when $\texttt{RCSP.resolve}(\cdot)$ is run. In other words, the privacy holds up to the point where $\texttt{RCSP.resolve}(.)$ is run. This is why the latter algorithm is excluded from the experiments in Definition 5.

Definition 6 (RC-S-P Security)**.** An RC-S-P with functions F, M, E, D, Q is *secure* for auxiliary information aux, if it satisfies security against malicious server, security against malicious client, and preserves privacy for aux, w.r.t. Definitions 3, 4, and 5, respectively.

8. Generic RC-S-P Protocol

In this section, we outline the designs of the RC-S-P protocol that realises Definition 6. As we stated in Section 6, this protocol relies on the idea that the server and client can efficiently agree on private statements at the beginning of the protocol. We first present a primitive, called *statement agreement protocol* (SAP), that satisfies the above requirement, and then present the RC-S-P.

8.1. Statement Agreement Protocol (SAP)

An SAP is secure if it meets four security properties:

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

To that end, we use a combination of a smart contract (including digital signatures involved) and a commitment scheme. The idea is as follows. Let x be the statement. The client picks a random value and uses it to commit to x. It sends the commitment to the contract and the commitment opening (i.e., statement and the random value) to the server. The server checks if the opening matches the commitment and if so, it commits to the statement using the same random value and sends its commitment to the contract. Later, for a party to prove to the contract/verifier that it has agreed on the statement with the other party, it

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

only sends the opening of the commitment. The contract/verifier checks if the opening matches both commitments and accepts if it matches. The SAP protocol is provided below. It assumes that each party $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}\}$ already has a blockchain public address $adr_{\mathcal{P}}$.

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

In the paper's full version, we discuss SAP's security and explain why naive solutions are not suitable.

8.2. Overview of RC-S-P Protocol

We have built the RC-S-P protocol using a novel combination of VSID, SAP, the private time bubble notion, symmetric-key encryption schemes, the coin masking and padding techniques. At a high level, it works as follows.

The client and server use SAP to provably agree on two private statements; the first statement includes payment details, while another one specifies a secret key, k, and the pads' length. They also agree on public parameters such as (a) the private time bubble's length, and (b) a smart contract specifying the parameters and the total amount of masked coins each party should deposit. The client deploys the contract. Each party deposits its masked coins in it. To start using the service, they invoke those VSID algorithms which let the server check if the client has generated its metadata correctly (via NIZK) and the server aborts if it decides not to serve.

At the end of each billing cycle, the client generates an encrypted query using k. It pads the encrypted query and sends the result to the contract. In the same cycle, the server retrieves the query and checks its validity (via NIZK). 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. In either case, the server encrypts the proof, pads it, and sends the result to the contract. After the server sends the messages to the contract, the client extracts and locally verifies the proof. If the verification passes, the client knows the server has delivered the service honestly. If the proof is rejected, it waits until the private time bubble passes and dispute resolution time arrives.

During the dispute resolution period, if the client or server rejects a proof, it invokes the arbiter, refers it to the invalid encrypted proofs in the contract, and sends to it the decryption key and the pads' detail. The arbiter checks the key and pads validity. If they are accepted, then the arbiter extracts the related proofs and checks 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. To distribute the coins, the 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. Appendix E presents the RC-S-P protocol in detail.

9. Recurring Contingent PoR Payment

In this section, we present recurring contingent PoR payment (RC-PoR-P) that is a concrete instantiation of the RC-S-P, when the verifiable service is PoR. We use PoR in the concrete instantiation of RC-S-P because PoR it is an active area of research and it will let us compare our solution to the-state-of-the-art, i.e., concrete instantiation of zkCSP of Campanelli *et al*. [12].

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

Avoiding the use of zk proofs. The majority of PoRs assume that only $\mathcal{S}$ is potentially malicious while $\mathcal{C}$ is honest. To ensure a file's availability, they rely on metadata that is either a set of tags (e.g., MACs or signatures) or a root of a Merkle tree, built on the file blocks. In the case where $\mathcal{C}$ can also be malicious, if tags are used then using zk proofs seem an obvious choice, as it allows $\mathcal{C}$ to guarantee to $\mathcal{S}$ that the tags have been constructed correctly (similar to the PoR in [6]). But, this imposes significant computation and communication costs. We observed that using a Merkle tree would benefit our protocol from a couple of perspectives; in short, it removes the need for zk proofs and it supports proof of misbehaviour. Our first observation is that if a Merkle tree is used, then $\mathcal{S}$ can efficiently check the metadata's correctness by reconstructing this tree on the file blocks, without involving zk proofs.

Low arbiter-side cost. In a Merkle tree-based PoR, in each verification, the number of proofs (or paths) are linear with the number of blocks that are probed, say ϕ. In this scheme, the verifier checks all given proofs and rejects them if only one of them is invalid. We observed that if this scheme is used in the RC-PoR-P, then once $\mathcal{C}$ finds an invalid proof, it can send only that single invalid proof as

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

a *proof of misbehaviour* to the arbiter $\mathcal{R}$[3]. This technique significantly reduces $\mathcal{R}$'s computation cost from $\phi \log_2(n)$ to $\log_2(n)$, where n is the number of file blocks.

The RC-PoR-P scheme (cf. Subsection 9.2) deploys the following two building blocks:

1) A PoR scheme, presented in Subsection 9.1, that can be seen as a variant of the standard Merkle tree-based PoR [30], [34], [43]. The security of the construction relies on the security of the underlying Merkle tree and pseudorandom function (cf. Subsection 3.2).
2) A *statement agreement protocol* (SAP), introduced in Subsection 8.1, that lets $\mathcal{S}$ and $\mathcal{C}$ efficiently agree on private statements at the beginning of the RC-PoR-P scheme. The SAP is built upon a binding and hiding commitment scheme, a smart contract, and a secure digital signature scheme used to sign transactions on the blockchain (cf. Subsections 3.2 and 3.1).

9.1. Modified Merkle tree-based PoR

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

1) **Client-side Setup**. $\texttt{PoR.setup}(1^\lambda, u)$
   a) $\mathcal{C}$ uses an error-correcting code, to encode the input file, u. Let u' be the encoded file. Then, it splits u' into m blocks as follows, $u^* = u'_1||1, ..., u'_m||m$.
   b) $\mathcal{C}$ constructs a Merkle tree on u^*'s blocks, i.e., $\texttt{MT.genTree}(u^*)$. Let σ be the root of the tree, and ϕ be the number of blocks that will be probed. It sets public parameters as $pp := (\sigma, \phi, m, \zeta)$, where ζ is a PRF's description, as defined in Subsection 3.2. It sends pp and u^* to $\mathcal{S}$.
2) **Client-side Query Generation**. $\texttt{PoR.genQuery}(1^\lambda, pp)$
   a) $\mathcal{C}$ picks a key $\hat{k}$ for PRF and sends $\hat{k}$ to $\mathcal{S}$.
3) **Server-side Proof Generation**. $\texttt{PoR.prove}(u^*, \hat{k}, pp)$
   a) $\mathcal{S}$ derives ϕ pseudorandom indices as follows. $\forall i, 1 \leq i \leq \phi : q_i = \big(\texttt{PRF}(\hat{k}, i) \bmod m\big) + 1$. Note that $1 \leq q_i \leq m$. Let $\boldsymbol{q} = [q_1, ..., q_\phi]$.
   b) $\mathcal{S}$ generates a proof $\pi_{q_i} = \texttt{MT.prove}(u^*, q_i)$, for each random index q_i. Let the final result be $\boldsymbol{\pi} = [(u^*_{q_i}, \pi_{q_i})]_{q_i \in \boldsymbol{q}}$, where i-th element in $\boldsymbol{\pi}$ corresponds to q_i, and the probed block is $u^*_{q_i}$. It sends $\boldsymbol{\pi}$ to $\mathcal{C}$.
4) **Client-side Proof Verification**. $\texttt{PoR.verify}(\boldsymbol{\pi}, \boldsymbol{q}, pp)$
   a) If $|\boldsymbol{\pi}| = |\boldsymbol{q}| = 1$, then $\mathcal{C}$ sets $\phi = 1$. This step is only for the case where a single proof and query is provided (e.g., in the proof of misbehaviour).
   b) $\mathcal{C}$ checks if $\mathcal{S}$ sent all proofs, by parsing each element of $\boldsymbol{\pi}$ as: $\texttt{parse}(u^*_{q_i}) = u'_{q_i}||q_i$, and checking if its index q_i equals to $\boldsymbol{q}$'s i-th element (note, $\mathcal{C}$ recomputes $\boldsymbol{q}$ given $\hat{k}$). If all checks pass, it takes the next step. Else, it outputs $\boldsymbol{d} = [0, i]$, where i is the index of $\boldsymbol{\pi}$'s element that did not pass the check.
   c) $\mathcal{C}$ checks if every path in $\boldsymbol{\pi}$ is valid, by calling $\texttt{MT.verify}(u^*_{q_i}, \pi_{q_i}, \sigma)$. If all checks pass, it outputs $\boldsymbol{d} = [1, \perp]$; otherwise, it outputs $\boldsymbol{d} = [0, i]$, where i refers to the index of the first element in $\boldsymbol{\pi}$ that does not pass the check.

The above protocol differs from the standard Merkle tree-based PoR from two perspectives; First, in step 4, $\mathcal{C}$ also outputs one of the rejected proofs' indices. Given that index (and vectors of proofs and challenges), this will let a third party *efficiently* verify that $\mathcal{S}$ did not pass the verification. Second, in step 2, instead of sending ϕ challenges, we let $\mathcal{C}$ send only a key/seed of the PRF to $\mathcal{S}$ which can derive a set of challenges from it, such a technique has been used before, e.g., in [15], [17], [27]. This will lead to a decrease in the $\mathcal{C}$'s communication and smart contract's storage costs.

9.2. RC-PoR-P Protocol

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

In the beginning, $\mathcal{C}$ generates a symmetric encryption key $\bar{k}$ and sets the number of dummy values to pad encrypted proofs, pad_π. In its setup step, $\mathcal{C}$ runs $\texttt{PoR.setup}(1^\lambda, u)$ to obtain the encoding u^* and the parameters $pp := (\sigma, \phi, m, \zeta)$. The query/proof secret parameters qp include $(\bar{k}, pad_\pi, pp)$. $\mathcal{C}$ sets the coin secret parameters $cp := (o, o_{max}, l, l_{max}, z)$ (cf. Section 3) that determine $coin^*_{\mathcal{C}}$ and $p_{\mathcal{S}}$, i.e. the total number of masked coins $\mathcal{C}$ and $\mathcal{S}$ must deposit. It initiates two SAP sessions for agreements on qp and cp with $\mathcal{S}$ and deploys a smart contract, SC. It completes setup by providing $\mathcal{S}$ with u^*, the SAP parameters (including qp and cp), and the number of verifications, z, and depositing $coin^*_{\mathcal{C}}$ coins in SC. In server setup, $\mathcal{S}$ checks whether a sufficient amount of coins has been deposited by $\mathcal{C}$ and runs the agreement step of the two SAP sessions initiated by $\mathcal{C}$. If agreement is successful and the public parameters (σ, ϕ, m) verify, it sends $coin^*_{\mathcal{S}} = p_{\mathcal{S}}$ coins to SC.

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

Dispute resolution takes place when $\mathcal{C}$ rejects service proofs or $\mathcal{S}$ rejects the queries. The arbiter $\mathcal{R}$ receives the complaint vectors $\boldsymbol{m}_{\mathcal{C}}$ and $\boldsymbol{m}_{\mathcal{S}}$ from $\mathcal{C}$ and $\mathcal{S}$ along with each party's "views" of the two SAP sessions. Given $\boldsymbol{m}_{\mathcal{S}}$ and the view of $\mathcal{S}$, if $\mathcal{S}$'s view is valid, then $\mathcal{R}$ decides for every complaint in $\boldsymbol{m}_{\mathcal{S}}$ by decrypting the corresponding query and executing $\mathcal{S}$'s steps for that query in the billing

3. This idea is akin to the proof of misbehaviour proposed in [13].

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

cycles phase described above. Given $\boldsymbol{m}_{\mathcal{C}}$ and the view of $\mathcal{C}$, if $\mathcal{C}$'s view is valid, then $\mathcal{R}$ decides for every complaint in $\boldsymbol{m}_{\mathcal{C}}$ by retrieving the rejected proof's details (included in the complaint), decrypting the related query and (i) executing $\mathcal{S}$'s steps for that query, (ii) executing $\mathcal{C}$'s verification for the rejected proof and the related query. The arbiter updates SC's state based upon its decisions. Finally, coin transfer is carried out according to the state of SC, as updated by $\mathcal{R}$.

Before we present the protocol, we discuss how metadata generator function M_{PoR}, the pair of encoding/decoding functions (E_{PoR}, D_{PoR}) and the query generator function Q_{PoR} (involved in the RC-S-P Definition 1) are defined in the PoR context, as they are often implicit in the original definition of PoR. Briefly, M_{PoR} is a function that processes a file and generates metadata. For instance, when PoR uses a Merkle tree, then M_{PoR} refers to $\mathtt{MT.genTree}(w) \rightarrow (tr, \sigma)$, where tr is the tree constructed on in file w and σ is the root of the tree. Encoding by E_{PoR} refers to encrypting with a symmetric key and then adding an appropriate number of pads, while decoding by D_{PoR} refers to removing the pads and then decrypting with the symmetric key. Furthermore, Q_{PoR} can be a PRF that generates a set of pseudorandom strings in a certain range, e.g., file block's indices.

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

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

   Billing-cycles Onset. $\mathcal{C}$ and $\mathcal{S}$ engage in phases 4-6, at the end of every j-th billing cycle, where $1 \leq j \leq z$. Each j-th cycle includes two time points, $\mathtt{G}_{j,1}$ and $\mathtt{G}_{j,2}$, where $\mathtt{G}_{j,2} > \mathtt{G}_{j,1}$, and $\mathtt{G}_{1,1} > \mathtt{T}_2$.
4) **Client-side Query Generation**. $\mathtt{RCPoRP.genQuery}(1^{\lambda}, T_{qp})$
   a) Calls $\mathtt{PoR.genQuery}(1^{\lambda}, pp) \rightarrow \hat{k}_j$, where $pp \in T_{qp}$.
   b) Sends encryption $c^*_j = \mathtt{Enc}(\bar{k}, \hat{k}_j)$ to SC at time $\mathtt{G}_{j,1}$.
5) **Server-side Proof Generation**. $\mathtt{RCPoRP.prove}(u^*, c^*_j, T_{qp})$
   a) Decrypts the query, $\hat{k}_j = \mathtt{Dec}(\bar{k}, c^*_j)$, where $\bar{k} \in T_{qp}$.
   b) Checks the query's correctness by ensuring $\hat{k}_j$ is not empty, and is in the key's universe, i.e., $\hat{k}_j \in \{0,1\}^{\psi}$. If the checks pass, it sets $b_j = 1$; else, it sets $b_j = 0$.
      - if $b_j = 1$, it sets $m_{\mathcal{S},j} = \perp$. It generates proofs vector by calling $\mathtt{PoR.prove}(u^*, \hat{k}_j, pp) \rightarrow \boldsymbol{\pi}_j$. Then, it encrypts the proofs, i.e., for $1 \leq g \leq |\boldsymbol{\pi}_j| : \mathtt{Enc}(\bar{k}, \boldsymbol{\pi}_j[g]) = \boldsymbol{\pi}'_j[g]$. Let $\boldsymbol{\pi}'_j$ contain the encrypted proofs. It pads every encrypted proof in $\boldsymbol{\pi}'_j$ with $pad_{\pi} \in T_{qp}$ random values picked from the encryption's output range, U. Let $\boldsymbol{\pi}^*_j$ be the result. It sends $\boldsymbol{\pi}^*_j$ to SC at time $\mathtt{G}_{j,2}$.
      - if $b_j = 0$, it sets the complaint $m_{\mathcal{S},j} = j$. It constructs a dummy proof $\boldsymbol{\pi}'_j$ with elements randomly picked from U, pads the result as above, and sends the result, $\boldsymbol{\pi}^*_j$, to SC at time $\mathtt{G}_{j,2}$.

      It outputs b_j and $m_{\mathcal{S},j}$.
6) **Client-side Proof Verification**. $\mathtt{RCPoRP.verify}(\boldsymbol{\pi}^*_j, c^*_j, T_{qp})$
   a) Removes the pads from $\boldsymbol{\pi}^*_j$, yielding $\boldsymbol{\pi}'_j$. It decrypts the service proofs $\mathtt{Dec}(\bar{k}, \boldsymbol{\pi}'_j) = \boldsymbol{\pi}_j$ and then verifies the proof by calling $\mathtt{PoR.verify}(\boldsymbol{\pi}_j, \hat{k}_j, pp) \rightarrow \boldsymbol{d}_j$, where $\hat{k}_j = \mathtt{Dec}(\bar{k}, c^*_j)$.
      - if $\boldsymbol{\pi}_j$ passes the verification, i.e., $\boldsymbol{d}_j[0] = 1$, it sets $\boldsymbol{m}_{\mathcal{C},j} = \perp$ and concludes that the service for this verification was delivered.
      - otherwise (i.e., $\boldsymbol{d}_j[0] = 0$), it sets $g = \boldsymbol{d}_j[1]$ and the complaint $\boldsymbol{m}_{\mathcal{C},j} = [j, g]$. Recall, $\boldsymbol{d}_j[1]$ refers to a rejected proof's index in proof vector $\boldsymbol{\pi}_j$.
   b) It outputs $\boldsymbol{d}_j$ and $\boldsymbol{m}_{\mathcal{C},j}$.
7) **Dispute Resolution**. $\mathtt{RCPoRP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, T_{qp})$

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

This phase takes place only in case of dispute, i.e., when $\mathcal{C}$ rejects service proofs or $\mathcal{S}$ rejects the queries.

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

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

Theorem 1. *The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ described in Subsections 9.1 and 9.2 is secure (cf. Definition 6), if the underlying Merkle tree, pseudorandom function, commitment scheme, and digital signature scheme are secure, and the symmetric-key encryption scheme is IND-CPA secure.*

9.3. RC-PoR-P Without Arbiter's Involvement

Due to the efficiency of the arbiter-side algorithm, i.e., $\mathtt{RCSPoR.resolve}(\cdot)$, we can delegate the arbiter's role to the smart contract, SC. In this case, the third-party arbiter's involvement is no longer needed. But, to have the new variant of RC-PoR-P, we need to adjust the original RC-PoR-P's protocol and definition, primarily from two perspectives. First, how a party pays to resolve a dispute would change, which ultimately affects the amount of coins each party receives in the coin transfer phase (see below for more details). Second, there would be no need to keep track of the number of times a party unnecessarily raises a dispute, as it pays the contract when it sends a query, before the contract processes its claim. In the paper's full version, we provide a generic definition for RC-S-P for the case where the arbiter's role can be played by a smart contract. The generic definition also captures the new variant of RC-PoR-P.

Next, we elaborate on how the original RC-PoR-P protocol can be adjusted such that the third-party arbiter's role is totally delegated to the smart contract, SC. Briefly, Phases 1–6 remain unchanged, with an exception. Namely, in step 2d, only two counters $y_{\mathcal{C}}$ and $y_{\mathcal{S}}$ are created, instead of four counters; accordingly, in the same step, vector $\boldsymbol{y}$ is now $\boldsymbol{y} : [y_{\mathcal{C}}, y_{\mathcal{S}}]$, so counters $y'_{\mathcal{C}}$ and $y'_{\mathcal{S}}$ are excluded from the vector. At a high level, the changes applied to phase 7 are as follows: the parties send their complaints to SC now, SC does not maintain $y'_{\mathcal{C}}$ and $y'_{\mathcal{S}}$ anymore, SC takes the related steps (on the arbiter's behalf), and it reads its

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

internal state any time it needs to read data already stored on the contract. Moreover, the main adjustment to phase 8 is that the amount of coins each party receives changes. In the RC-PoR-P and RC-S-P (presented in sections 9.2 and E.1, respectively), the party which raises a dispute does not pay the arbiter when it sends to it the dispute query. Instead, loosely speaking, the arbiter in the coin transfer phase is paid by a misbehaving party. In contrast, when the arbiter's role is played by a smart contract, the party which raises a dispute and sends the dispute query to the contract (due to the nature of Ethereum smart contracts) has to pay the contract *before* the contract processes its query. This means that an honest party which sends a complaint to the contract needs to be compensated (by the corrupt party) for the amount of coins it sent to the contract to resolve the dispute. In the paper's full version, we present the modified RC-PoR-P protocol in more detail.

10. Performance Evaluation of RC-PoR-P

In this section, we provide an analysis of the RC-PoR-P protocol. Table 1 summarises the protocol's concrete cost (we also provide a table for its asymptotic cost in Appendix G). Also, we compare RC-PoR-P with (a) the zero-knowledge contingent (publicly verifiable) PoR payment in [12] and the fair PoR payment scheme in [3] that are more efficient than the state-of-the-art and closest to our work. Table 2 summarises the comparison. The analysis of RC-PoR-P covers both asymptotic and concrete overheads. To conduct the concrete cost study, we have implemented RC-PoR-P. The protocol's off-chain and on-chain parts have been implemented in C++ and Solidity programming languages respectively. To conduct the off-chain experiment, we used a server with dual Intel Xeon Gold 5118, 2.30 GHz CPU and 256 GB RAM. To carry out the on-chain experiment, we used a MacBook Pro laptop with quad-core Intel Core i5, 2 GHz CPU and 16 GB RAM that interacts with the Ethereum private blockchain. We ran the experiment 10 times. In the experiment, we used the SHA-2 hash function and set its output length and the security parameter to 128 bits. We set the size of every block to 128 bits, as in [48]. We used a random file whose size is in the range $[64\text{ MB}, 4\text{ GB}]$. This results in the number of file blocks in the range $[2^{22}, 2^{28}]$. Since in the experiment we used relatively large file sizes, to lower on-chain transaction costs, we allow the parties to use the technique explained in Section E.1, which lets the server and client exchange the (PoR) proofs off-chain in an irrefutable fashion[4]. The prototype implementation utilises the Cryptopp [16] and GMP [47] libraries. The protocol's off-chain and on-chain source code are publicly available in [1] and [2] respectively.

10.1. Computation Cost

In our analysis, the cost of erasure-coding a file is not taken into consideration, as it is identical in all PoR schemes. We first analyse the computation cost of RC-PoR-P. $\mathcal{C}$'s cost is as follows. In phase 2, its cost in step 2a involves $m \cdot \sum_{i=1}^{\log_2(m)} \frac{1}{2^i}$ invocations of a hash function. So its complexity in this step is $O(m)$. Its total cost in steps 2b and 2c involves two invocations of the hash function. Therefore, the client-side total complexity in this phase is $O(m)$. In this phase, its off-chain *run-time* increases about $2\times$ (i.e., from 23.1 to $45.5, ...$, from 732.1 to 1596.6 seconds) when m increases (i.e., from 2^{22} to $2^{23}, ...$, from 2^{27} to 2^{28} blocks). This phase also costs it $123 \cdot 10^{-5}$ ether. In phase 4, $\mathcal{C}$ invokes PRF and symmetric-key encryption ϕ times and once respectively. So, for z verifications its total computation cost is $O(z \cdot \phi)$. Its off-chain run-time in this phase is negligibly small. This phase also costs it $6 \cdot 10^{-5} \cdot z$ ether. In phase 6, $\mathcal{C}$ for each verification decrypts and verifies proofs which mainly involves $\phi \cdot (\log_2(m)+1)$ invocations of the symmetric key encryption and $\phi \cdot \log_2(m)$ invocations of the hash function. So, its total complexity in this phase is $O(z \cdot \phi \cdot \log_2(m))$. Its off-chain run-time in this phase is very low and grows almost $1.1\times$ (i.e., from $0.09 \cdot z$ to $0.11 \cdot z, ...$, from $0.21 \cdot z$ to $0.24 \cdot z$ seconds) when m increases.

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

Next, we analyse $\mathcal{R}$'s cost in phase 7. First, we evaluate $\mathcal{R}$'s cost when it is invoked by an honest $\mathcal{S}$. In this case, it invokes the hash function twice and decrypts $|\boldsymbol{v}_{\mathcal{S}}|$ queries, where $|\boldsymbol{v}_{\mathcal{S}}|$ is the total number of verifications that $\mathcal{S}$ complained about and $|\boldsymbol{v}_{\mathcal{S}}| \leq z$. Now, we evaluate its cost when it is invoked by an honest $\mathcal{C}$. It invokes the hash function twice to check the correctness of the statement, $\ddot{x}_{qp}$, sent by the client. It invokes the hash function $|\boldsymbol{v}_{\mathcal{C}}| \cdot (\log_2(m) + 2)$ times and the symmetric key encryption $|\boldsymbol{v}_{\mathcal{C}}| \cdot (\log_2(m) + 2)$ times, where $|\boldsymbol{v}_{\mathcal{C}}|$ is the total number of verifications that $\mathcal{C}$ complained about. Thus, its cost, in phase 7 is at most $O(z' \cdot \log_2(m))$, where $z' = Max(|\boldsymbol{v}_{\mathcal{C}}|, |\boldsymbol{v}_{\mathcal{S}}|)$ and $z' \leq z$. Note that due to the use of the proof of misbehaviour in the protocol, $\mathcal{R}$'s cost is about $\frac{1}{\phi} = \frac{1}{460}$ of its computation cost in the absence of such technique where it has to check all ϕ proofs for each verification.[6] Its off-chain run-time is very low and increases about $1.3\times$ (i.e., from $2 \cdot 10^{-5} \cdot z'$ to $4 \cdot 10^{-5} \cdot z', ...$, from $9 \cdot 10^{-5} \cdot z'$ to $10^{-4} \cdot z'$ seconds) when m increases. Phase 7 also imposes 10^{-4} ether to $\mathcal{R}$. In phase 8, SC invokes the hash function only twice, so its computation complexity is constant. This phase imposes $6 \cdot 10^{-5}$ ether to the party that calls RCPoRP.pay.

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

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

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

TABLE 1: RC-PoR-P off-chain run-time (in seconds) and on-chain cost, of z verifications; breakdown by phases. In the table, z' is the maximum number of complaints the client and server send to the arbiter, and m is the number of blocks in a file.

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

TABLE 2: Contingent PoRs comparison. In the table, T is a time parameter, and ϕ is the number of challenged blocks.

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

10.2. Communication Cost

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

10.3. Comparison

The fair PoR scheme in [3] assumes that $\mathcal{C}$ is trusted. The initiation phase involves $O(z)$ modular exponentiations and $O(m + z\phi)$ modular multiplications to generate puzzles and MACs respectively. Given the puzzles, $\mathcal{S}$ has to *continuously* solve them sequentially until all z verifications end, which requires $\mathcal{S}$ to perform the exponentiations even between two consecutive verifications. This requires $\mathcal{S}$ to perform $O(Tz)$ exponentiations and z modular multiplications, where T is a time parameter. For z verifications, $\mathcal{S}$ performs $O(z\phi)$ multiplications to generate z proofs. A verifier performs $O(z\phi)$ multiplications to verify all proofs. Now we focus on the scheme in [12]. As we showed in Section 5, this scheme is not secure against a malicious $\mathcal{C}$. In the initiation phase, $\mathcal{C}$ generates a signature for each file block which involves $O(m)$ exponentiations and $O(m)$ hash function invocations. For $\mathcal{S}$ to generate z proofs, it (i) performs $O(z\phi)$ exponentiations to combine the signatures, (ii) invokes the hash function at least $O(1)$ times, and (iii) invokes zk proof system $O(z\phi)$ times. The scheme imposes the same computation complexity on the verifier as it does on the prover. Campanelli *et al.* [12] provide an implementation of zkCSP for publicly and privately verifiable PoRs. We were informed by Campanelli that the total size of the outsourced file used in their experiment is at most 256 bits, which is very small. In contrast, in our experiment, we used a much large file size, i.e., 4-GB.

Since both schemes in [3] and [12] use homomorphic tags, proofs for each verification can be combined resulting in constant proof size, i.e., $O(1)$. These schemes do not address the privacy issue we highlighted in Section 5.1. However, RC-PoR-P is secure against a malicious $\mathcal{C}$ and rectifies the privacy issue. Similar to the other two schemes, its initiation complexity is $O(m)$; but, unlike them, it does not require any modular exponentiations. Instead, it involves only invocations of the hash function which imposes a much lower overhead. Moreover, unlike the other two schemes that have $O(z\phi)$ complexity in the prove and verify phases, RC-PoR-P's complexity, in theory, is slightly higher, i.e., it is $O(z\phi \log_2(m))$. However, the extra factor: $\log_2(m)$ is not very high in practice. For instance, for a 4-GB file (or 2^{28} blocks), it is only 28. RC-PoR-P's prove and verify algorithms, similar to the ones [3], involve only symmetric key operations; whereas, the ones in [12] need asymmetric key operations. Also, RC-PoR-P's the proof size complexity is larger than the other two schemes; but, each message length in RC-PoR-P is much shorter than the one in [12], i.e., 128-bit vs 2048-bit.

Thus, RC-PoR-P is computationally more efficient than [12] and [3] while offering stronger security guarantees (i.e., security against a malicious client and privacy).

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

Acknowledgements

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

References

[1] Aydin Abadi. Off-chain source code of "recurring contingent proofs of retrievability payment" (RC-PoR-P), 2023. https://github.com/AydinAbadi/RC-S-P/blob/main/RC-PoR-P-Source-cod/RC-PoR-P.cpp.

[2] Aydin Abadi. On-chain source code of "recurring contingent proofs of retrievability payment" (RC-PoR-P), 2023. https://github.com/AydinAbadi/RC-S-P/blob/main/RC-PoR-P-Source-cod/RC-PoR-P-Smart-Contract.sol.

[3] Aydin Abadi and Aggelos Kiayias. Multi-instance publicly verifiable time-lock puzzle and its applications. In *, FC*, 2021.

[4] Aydin Abadi, Steven J. Murdoch, and Thomas Zacharias. Recurring contingent service payment. *CoRR*, 2022. https://doi.org/10.48550/arXiv.2208.00283.

[5] Amazon. Amazon s3 pricing, 2021.

[6] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O Karame, Zongren Liu, and Christian A Reuter. Outsourced proofs of retrievability. In *CCS*, 2010.

[7] N. Asokan, Matthias Schunter, and Michael Waidner. Optimistic protocols for fair exchange. In *CCS*, 1997.

[8] N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures (extended abstract). In *EUROCRYPT*, 1998.

[9] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable data possession at untrusted stores. In *CCS*, 2007.

[10] Emily Bary. Zoom stock falls after service outage, 2020.

[11] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *S&P*, 2014.

[12] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *CCS*, 2017.

[13] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In *CCS*, 2011.

[14] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *STOC*, 1986.

[15] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC*, 2005.

[16] Wei Dai. Crypto++ library, 2014. https://cryptopp.com.

[17] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *CRYPTO, 2005, Proceedings*, 2005.

[18] Changyu Dong, Liqun Chen, Jan Camenisch, and Giovanni Russello. Fair private set intersection with a semi-trusted arbiter. In *DBSec*, 2013.

[19] Changyu Dong, Yilei Wang, Amjad Aldweesh, Patrick McCorry, and Aad van Moorsel. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In *CCS*, 2017.

[20] Dropbox. Choose the right dropbox for you, 2021.

[21] Yuefeng Du, Huayi Duan, Anxin Zhou, Cong Wang, Man Ho Au, and Qian Wang. Enabling secure and efficient decentralized storage auditing with blockchain. *TDSC*, 2021.

[22] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. Fairswap: How to fairly exchange digital goods. In *CCS*, 2018.

[23] Lisa Eckey, Sebastian Faust, and Benjamin Schlosser. Optiswap: Fast optimistic fair exchange. In *ASIA CCS*, 2020.

[24] Georg Fuchsbauer. WI is not enough: Zero-knowledge contingent (service) payments revisited. In *CCS*, 2019.

[25] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.

[26] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, 2013.

[27] Satrajit Ghosh and Tobias Nilges. An algebraic approach to maliciously secure private set intersection. In *EUROCRYPT*, 2019.

[28] Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2001.

[29] GoogleOne. Upgrade to a plan that works for you, 2021.

[30] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *CCS*, 2011.

[31] Todd Haselton. Slack service goes down for more than three hours, 2021.

[32] Chunqiang Hu, Xiuzhen Cheng, Zhi Tian, Jiguo Yu, and Weifeng Lv. Achieving privacy preservation and billing via delayed information release. *IEEE/ACM Transactions on Networking*, 2021.

[33] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multiparty computation with identifiable abort. In *CRYPTO*, 2014.

[34] Ari Juels and Burton S. Kaliski Jr. Pors: Proofs of retrievability for large files. In *CCS*, 2007.

[35] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[36] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *S&P*, 2016.

[37] Junzuo Lai, Robert H. Deng, HweeHwa Pang, and Jian Weng. Verifiable computation on outsourced encrypted data. In *ESORICS 2014*, 2014.

[38] Zheli Liu, Tong Li, Ping Li, Chunfu Jia, and Jin Li. Verifiable searchable encryption with aggregate keys for data sharing system. *Future Gener. Comput. Syst.*, 2018.

[39] Gregory Maxwell. Zero knowledge contingent payment, 2011.

[40] Ralph C. Merkle. Protocols for public key cryptosystems. In *S&P*, 1980.

[41] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, 1989.

[42] Y. Miao, Q. Tong, R. Deng, K. R. Choo, X. Liu, and H. Li. Verifiable searchable encryption framework against insider keyword-guessing attack in cloud storage. *IEEE Transactions on Cloud Computing*, 2020.

[43] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *S&P'14*.

[44] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2019.

[45] Ky Nguyen, Miguel Ambrona, and Masayuki Abe. WI is almost enough: Contingent payment all over again. In *CCS*, 2020.

[46] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1991.

[47] GNU Project. The gnu multiple precision arithmetic library, 1991. https://gmplib.org/.

[48] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *ASIACRYPT*, 2008.

[49] Shiuan-Tzuo Shen and Wen-Guey Tzeng. Delegable provable data possession for remote data in the clouds. In *ICICS 2011*.

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

[50] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *EuroS&P*, 2017.

[51] UK Finance. 2021 half year fraud update, 2021. https://www.ukfinance.org.uk/system/files/Half-year-fraud-update-2021-FINAL.pdf.

[52] Xingfeng Wang and Liang Zhao. Verifiable single-server private information retrieval. In *ICICS*, 2018.

[53] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[54] Liang Feng Zhang and Reihaneh Safavi-Naini. Verifiable multi-server private information retrieval. In *ACNS*, 2014.

A. Notations

We summarise our notation in Table 3.

TABLE 3: Notation Table.

Setting	Symbol	Description
Generic	z	Number of verifications
	λ	Security parameter
	PRF	Pseudorandom function
	ζ	PRF's description
	Pr	Probability
	Com	Commit algorithm in commitment
	Ver	Verify algorithm in commitment
	μ	Negligible function
	H	Hash function
	MT	Merkle tree
	sk, pk	Secret and public keys
	PoR	Proof of retrievability
	u	Service input, e.g., file
	u^*	Encoded input
	σ	Metadata
	ω_σ	Proof for metadata's correctness
	e	$e := (\sigma, \omega_\sigma)$
	pp	Public parameter
	$q, \boldsymbol{q}$	Query and query vector
	$\boldsymbol{\omega}_q$	Proof for $\boldsymbol{q}$'s correctness
	c	$c := (\boldsymbol{q}, \boldsymbol{\omega}_q)$
	$\pi, \boldsymbol{\pi}$	Service proof and proof vector
	VS	Verifiable service
	VSID	Verifiable service with identifiable abort
	RCSP	Recurring contingent service payment
	SAP	Statement agreement protocol
	$\mathcal{C}$	Client
	$\mathcal{S}$	Server
	$\mathcal{R}$	Arbiter
	SC	Smart contract
Generic	F	Function run on u^* by $\mathcal{S}$
	M	Metadata generator function
	Q	Query generator function
	aux	Auxiliary information
	m	Number of a file blocks, $m = \lvert u^* \rvert$
	$\lvert\lvert u^* \rvert\rvert$	Bit size of u^*
	δ	Proof of F's evaluation correctness
	j	Verification index, $1 \leq j \leq z$
	adr	Address
	ϕ	Number of challenged blocks
RC-PoR-P or RC-S-P	r_{qp}, r_{cp}	Random values
	$\ddot{x}_{qp}, \ddot{x}_{cp}$	$\ddot{x}_{qp} := (qp, r_{qp}), \ddot{x}_{cp} := (cp, r_{cp})$
	$coin^*_{\mathcal{C}}, coin^*_{\mathcal{S}}$	Encoded coins deposited by $\mathcal{C}$ and $\mathcal{S}$
	enc	Encoding/decoding functions $enc := (E, D)$
	$m_{\mathcal{C}}, m_{\mathcal{S}}$	Complaints of $\mathcal{C}$ and $\mathcal{S}$
	pad_π, pad_q	Number of elements used to pad π and q
	$y_{\mathcal{C}}, y_{\mathcal{S}}$	Number of times $\mathcal{C}$ and $\mathcal{S}$ misbehave towards each other
	$y'_{\mathcal{C}}, y'_{\mathcal{S}}$	Number of times $\mathcal{C}$ and $\mathcal{S}$ unnecessarily invoke $\mathcal{A}r$
	cp	Coin secret parameters
	T_{cp}	Coin encoding token
	qp	Query/proof secret parameters
	T_{qp}	Query/proof encoding token
	T	$T := (T_{cp}, T_{qp})$
	$g_{\mathcal{C}}, g_{\mathcal{S}}$	Commitments computed by $\mathcal{C}$ and $\mathcal{S}$
	pl	Price list: $\{(o, l), ..., (o'', l'')\}$
	o	Coins $\mathcal{S}$ must get for a valid proof, where $o \in pl$
	l	Coins $\mathcal{A}r$ must get for resolving a dispute, where $l \in pl$
	l_{max}	$Max(l, ..., l'')$
	o_{max}	$Max(o, ..., o'')$
	$p_{\mathcal{S}}$	Total coins $\mathcal{S}$ should deposit

B. Preliminaries

B.1. Details on Building Blocks

B.1.1. Pseudorandom Function. Informally, a pseudorandom function (PRF) is a deterministic function that takes as input a key and some argument and outputs a value indistinguishable from that of a truly random function with the same domain and range. Pseudorandom functions have many applications in cryptography as they provide an efficient and deterministic way to turn input into a value that looks random. Below, we restate the formal definition of PRF, taken from [35].

Definition 7. Let $W : \{0,1\}^\psi \times \{0,1\}^\eta \rightarrow \{0,1\}^\iota$ be an efficient keyed function. It is said W is a pseudorandom function if for all probabilistic polynomial-time distinguishers B, there is a negligible function, $\mu(.)$, such that: $\left| \Pr[B^{W_{\hat{k}}(.)}(1^\psi) = 1] - \Pr[B^{\omega(.)}(1^\psi) = 1] \right| \leq \mu(\psi)$, where the key, $\hat{k} \xleftarrow{\$} \{0,1\}^\psi$, is chosen uniformly at random and ω is chosen uniformly at random from the set of functions mapping η-bit strings to ι-bit strings.

B.1.2. Commitment Scheme. A commitment scheme involves two parties, *sender* and *receiver*, and includes two phases: *commit* and *open*. In the commit phase, the sender commits to a message: x as $\texttt{Com}(x, r) = \texttt{Com}_x$, that involves a secret value: $r \xleftarrow{\$} \{0,1\}^\lambda$. In the end of the commit phase, the commitment $\texttt{Com}_x$ is sent to the receiver. In the open phase, the sender sends the opening $\ddot{x} := (x, r)$ to the receiver who verifies its correctness: $\texttt{Ver}(\texttt{Com}_x, \ddot{x}) \stackrel{?}{=} 1$ and accepts if the output is 1. A commitment scheme must satisfy two properties: (a) *hiding*: it is infeasible for an adversary (i.e., the receiver) to learn any information about the committed message x, until the commitment $\texttt{Com}_x$ is opened, and (b) *binding*: it is infeasible for an adversary (i.e., the sender) to open a commitment $\texttt{Com}_x$ to different values $\ddot{x}' := (x', r')$ than that was used in the commit phase, i.e., infeasible to find $\ddot{x}'$, *s.t.* $\texttt{Ver}(\texttt{Com}_x, \ddot{x}) = \texttt{Ver}(\texttt{Com}_x, \ddot{x}') = 1$, where $\ddot{x} \neq \ddot{x}'$. There exist efficient non-interactive commitment schemes both in (a) the standard model, e.g., Pedersen scheme [46], and (b) the random oracle model using the well-known hash-based scheme such that committing is : $\texttt{H}(x||r) = \texttt{Com}_x$ and $\texttt{Ver}(\texttt{Com}_x, \ddot{x})$ requires checking: $\texttt{H}(x||r) \stackrel{?}{=} \texttt{Com}_x$, where $\texttt{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is a collision resistant hash function; i.e., the probability to find x and x' such that $\texttt{H}(x) = \texttt{H}(x')$ is negligible in the security parameter λ.

B.1.3. Publicly Verifiable Non-interactive Zero-knowledge Proof. In a non-interactive zero-knowledge proof (NIZK), a prover $\mathcal{P}$, given a witness w for some statement x in an NP language L, wants to convince a verifier $\mathcal{V}$ of the validity of $x \in L$. The main security property of the scheme is *Zero-knowledge*; meaning, a potentially malicious verifier cannot learn anything beyond the validity of the statement. The procedure is non-interactive, i.e., $\mathcal{P}$ generates a proof π and provides $\mathcal{V}$ with π, who accepts (or rejects) verification. A NIZK is publicly verifiable when any party by obtaining π can verify the validity of $x \in L$. Publicly verifiable NIZKs have been constructed under trust assumptions such as

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

the presence of a common reference string, or setup assumptions such as the existence of a random oracle which is used in this work. For a formal definition of NIZKs we refer the reader to [28].

B.1.4. Symmetric-key Encryption Scheme. A symmetric-key encryption scheme consists of three algorithms $(\texttt{SKE.keyGen}, \texttt{Enc}, \texttt{Dec})$, defined as follows. (1) $\texttt{SKE.keyGen}(1^\lambda) \rightarrow k$ is a probabilistic algorithm that outputs a symmetric key k. (2) $\texttt{Enc}(k, m) \rightarrow c$ takes as input k and a message m in some message space and outputs a ciphertext c. (3) $\texttt{Enc}(k, c) \rightarrow m$ takes as input k and a ciphertext c and outputs a message m.

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

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

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

B.1.5. Digital Signature Scheme. A digital signature is a scheme for verifying the authenticity of digital messages. It involves three algorithms, $(\texttt{Sig.keyGen}, \texttt{Sig.sign}, \texttt{Sig.ver})$, defined as follows. (1) $\texttt{Sig.keyGen}(1^\lambda) \rightarrow (sk, pk)$ is probabilistic algorithm run by a signer that outputs a key pair (sk, pk), consisting of secret key sk, and public key pk. (2) $\texttt{Sig.sign}(sk, pk, u) \rightarrow sig$ is an algorithm run by the signer. It takes as input key pair (sk, pk) and a message u. It outputs a signature sig. (3) $\texttt{Sig.ver}(pk, u, sig) \rightarrow h \in \{0, 1\}$ is an algorithm run by a verifier. It takes as input pk, message u, and signature sig. It checks the signature's validity. If the verification passes, then it outputs 1; otherwise, it outputs 0.

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

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

(2) *Existential unforgeability under chosen message attacks (EUF-CMA)*: a probabilistic polynomial time PPT adversary that obtains pk and has access to a signing oracle for messages of its choice, cannot create a valid pair (u^*, sig^*) for a new message u^*, except with a negligible probability, σ. For a formal definition of digital signatures, we refer readers to [35].

B.1.6. Merkle Tree. In the setting where a Merkle tree is used to remotely check a file, the file is split into blocks and the tree is built on top of the file blocks. Usually, for the sake of simplicity, it is assumed the number of blocks, m, is a power of 2. The height of the tree, constructed on m blocks, is $\log_2(m)$. The Merkle tree scheme includes three algorithms $(\texttt{MT.genTree}, \texttt{MT.prove}, \texttt{MT.verify})$ as follows:

- The algorithm that constructs a Merkle tree, $\texttt{MT.genTree}$, is run by $\mathcal{V}$. It takes file blocks, $u := u_1, ..., u_m$, as input. Then, it groups the blocks in pairs. Next, a collision-resistant hash function, $\texttt{H}(.)$, is used to hash each pair. After that, the hash values are grouped in pairs and each pair is further hashed, and this process is repeated until only a single hash value, called "root", remains. This yields a tree with the leaves corresponding to the blocks of the input file and the root corresponding to the last remaining hash value. $\mathcal{V}$ locally stores the root, and sends the file and tree to $\mathcal{P}$.
- The proving algorithm, $\texttt{MT.prove}$, is run by $\mathcal{P}$. It takes a block index, i, and a tree as inputs. It outputs a vector proof, of $\log_2(m)$ elements. The proof asserts the membership of i-th block in the tree, and consists of all the sibling nodes on a path from the i-th block to the root of the Merkle tree (including i-th block). The proof is given to $\mathcal{V}$.
- The verification algorithm, $\texttt{MT.verify}$, is run by $\mathcal{V}$. It takes as input i-th block, a proof and tree's root. It checks if the i-th block corresponds to the root. If the verification passes, it outputs 1; otherwise, it outputs 0.

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

B.2. Definition of PoR

A PoR scheme considers the case where an honest client wants to outsource the storage of its file to a potentially malicious server, i.e., an active adversary. It is a challenge-response interactive protocol, where the server proves to the client that its file is intact and retrievable. Below, we restate PoR's formal definition initially proposed in [34], [48]. A PoR scheme comprises of five algorithms:

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

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

A PoR scheme has two properties: *correctness* and *soundness*. Correctness requires that the verification algorithm accepts proofs generated by an honest verifier; formally, PoR requires that for any key k, any file $u \in \{0,1\}^*$, and any pair (u^*, σ) output by $\texttt{PoR.setup}(1^\lambda, u, k)$, and any query $\boldsymbol{q}$, the verifier accepts when it interacts with an honest prover. Soundness requires that if a prover convinces the verifier (with high probability) then the file is stored by the prover. This is formalized via the notion of an extractor algorithm, that is able to extract the file in interaction with the adversary using a polynomial number of rounds. Before we define soundness, we restate the experiment, defined in [48], that takes place between an environment $\mathcal{E}$ and adversary $\mathcal{A}$. In this experiment, $\mathcal{A}$ plays the role of a corrupt party and $\mathcal{E}$ simulates an honest party's role.

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

It is said that a cheating prover, $\hat{\mathcal{A}}_\epsilon$, is ϵ-*admissible* if it convincingly answers ϵ fraction of verification challenges (for a certain file). Informally, a PoR scheme supports extractability, if there is an extractor algorithm $\texttt{Ext}(k, \sigma, \hat{\mathcal{A}}_\epsilon)$, that takes as input the key k, metadata σ, and the description of the machine implementing the prover's role $\hat{\mathcal{A}}_\epsilon$ and outputs the file, u. The extractor has the ability to reset the adversary to the beginning of the challenge phase and repeat this step polynomially many times for the purpose of extraction, i.e., the extractor can rewind $\hat{\mathcal{A}}_\epsilon$.

Definition 8 (ϵ-soundness)**.** A PoR scheme is ϵ-sound if there exists an extraction algorithm $\texttt{Ext}(\cdot)$ such that, for every adversary $\mathcal{A}$ who plays experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{PoR}}$ and outputs an ϵ-admissible cheating prover $\hat{\mathcal{A}}_\epsilon$ for a file u, the extraction algorithm recovers u from $\hat{\mathcal{A}}_\epsilon$, given honest party's private key, public parameters, metadata and the description of $\hat{\mathcal{A}}_\epsilon$, except with $\mathsf{negl}(\lambda)$ probability. Formally:

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

In contrast to the PoR definition in [34], [48] where $\texttt{PoR.genQuery}(\cdot)$ is implicit, in the above definition we have explicitly defined $\texttt{PoR.genQuery}(\cdot)$, as it plays an important role in this paper. Also, there are PoR protocols, e.g., in [43], that do not involve $\texttt{PoR.keyGen}(\cdot)$. Instead, a set of public parameters/keys (e.g., file size or a root of Merkle tree) are output by $\texttt{PoR.setup}(\cdot)$. To make the PoR definition generic to capture both cases, we have explicitly included the public parameters pp in the algorithms' definitions too.

C. Verifiable Service (VS) Definition

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

Definition 9 (VS Scheme)**.** A verifiable service scheme $\text{VS} := (\texttt{VS.keyGen}, \texttt{VS.setup}, \texttt{VS.genQuery}, \texttt{VS.prove}, \texttt{VS.verify})$ with function F, metadata generator function M, and query generator function Q consists of five algorithms defined as follows.

- $\texttt{VS.keyGen}(1^\lambda) \rightarrow k := (sk, pk)$. A probabilistic algorithm run by the client. It takes as input the security parameter 1^λ and outputs a secret/public verification key pair k. The server is given pk.
- $\texttt{VS.setup}(1^\lambda, u, k) \rightarrow (u^*, \sigma, pp)$. It is run by the client. It takes as input the security parameter 1^λ, the service input u, and key pair k. If an encoding is needed, then it encodes u, that results in u^*; otherwise, $u^* = u$. It outputs encoded input u^*, metadata $\sigma = M(u^*, k, pp)$, and (possibly input dependent) public parameters pp. Right after that, the server is given u^*, σ, and pp.
- $\texttt{VS.genQuery}(1^\lambda, \text{aux}, k, pp) \rightarrow \boldsymbol{q}$. A probabilistic algorithm run by the client. It takes as input the security parameter 1^λ, auxiliary information aux, the key pair k, and public parameters pp. It outputs a query vector $\boldsymbol{q} = Q(\text{aux}, k, pp)$. Depending on service types, $\boldsymbol{q}$ may be empty or contain only random strings. The output is given to the server.
- $\texttt{VS.prove}(u^*, \sigma, \boldsymbol{q}, pk, pp) \rightarrow \boldsymbol{\pi}$. It is run by the server. It takes as input the service encoded input u^*, metadata σ, queries $\boldsymbol{q}$, public key pk, and public parameters pp. It outputs a proof, $\boldsymbol{\pi} = [F(u^*, \boldsymbol{q}, pp), \delta]$,

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

containing the function evaluation for service input u, public parameters pp, and query $\boldsymbol{q}$, i.e., $h = F(u^*, \boldsymbol{q}, pp)$, and a proof δ asserting the evaluation is performed correctly, where generating δ may involve σ. The output is given to the client.

- $\mathtt{VS.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \rightarrow d \in \{0,1\}$. It is run by the client. It takes as input the proof $\boldsymbol{\pi}$, query vector $\boldsymbol{q}$, key k, and public parameters pp. In the case where $\mathtt{VS.verify}(\cdot)$ is publicly verifiable then $k := (\perp, pk)$, and when it is privately verifiable $k := (sk, pk)$. The algorithm outputs $d = 1$, if the proof is accepted; otherwise, it outputs $d = 0$.

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

Definition 10 (VS Correctness)**.** A verifiable service scheme VS with functions F, M, Q is correct for an auxiliary information aux, if for any service input u it holds that:

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

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

Definition 11 (VS Soundness)**.** A verifiable service VS with functions F, M, Q is sound for an auxiliary information aux, if for any probabilistic polynomial time adversary $\mathcal{A}$, it holds that:

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

The above generic definition captures the core requirements of a wide range of verifiable services such as verifiable outsourced storage, i.e., Proofs of Retrievability [34], [48] or Provable Data Possession [9], [49], verifiable computation [25], [37], verifiable searchable encryption [38], [42], and verifiable information retrieval [52], [54], to name a few. Other additional security properties (e.g., privacy) mandated by certain services can be added to the above definition. Alternatively, the definition can be upgraded to capture the additional requirements. The verifiable service with identifiable abort (VSID) and recurring contingent service payment (RC-S-P) definitions presented in this paper are two examples.

Remark 1. It is not hard to see that the original PoR definition (presented in Section 3.3) is a VS's special case. In particular, PoR's ϵ-soundness captures VS's soundness; in ϵ-soundness, the extractor algorithm interacts (many times) with the cheating prover which must not be able to persuade the extractor to accept an invalid proof with a high probability and should provide accepting proofs for non-negligible ϵ fraction of verification challenges. The former property is exactly what VS soundness states.

D. Verifiable Service with Identifiable Abort (VSID)

A protocol that realises only VS's definition (cf. Appendix C) 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 [42], [49]), it is rather strong and not suitable in the real world, especially when there are monetary incentives (e.g., service payment) that encourage a client to misbehave. Therefore, in the following we enhance the VS notion to allow (a) either party to be malicious and (b) a trusted third party, *arbiter*, to identify a corrupt party. We call an upgraded verifiable service scheme with these features *verifiable service with identifiable abort* (VSID), inspired by the notion of secure multi-party computation with identifiable abort [33].

D.1. VSID Definition

The definition of a VSID scheme is provided below.

Definition 12 (VSID Scheme)**.** A verifiable service with identifiable abort
VSID := ($\mathtt{VSID.keyGen}$, $\mathtt{VSID.setup}$, $\mathtt{VSID.serve}$, $\mathtt{VSID.genQuery}$, $\mathtt{VSID.checkQuery}$, $\mathtt{VSID.prove}$, $\mathtt{VSID.verify}$, $\mathtt{VSID.identify}$)
with function F, metadata generator function M, and query generator function Q involves four entities; namely, client, server, arbiter, and bulletin board. It consists of eight algorithms defined below.

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

$\boldsymbol{q} = Q(\text{aux}, k, pp)$, and proofs, $\boldsymbol{w}_q$, proving the queries are well-structured. Depending on service types, c might be empty or contain only random strings. It sends c to the bulletin board.

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

A VSID scheme has four main properties; namely, it is (a) correct, (b) sound, (c) inputs of clients are well-formed, and (d) a corrupt party can be identified by an arbiter, i.e., detectable abort. In the following, we formally define each of them. Correctness requires that the verification algorithm always accepts a proof generated by an honest prover and both parties are identified as honest. It is formally stated as follows.

Definition 13 (VSID Correctness)**.** A verifiable service with identifiable abort scheme with functions F, M, Q is correct for an auxiliary information aux, if for any service input u it holds that:

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

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

Definition 14 (VSID Soundness)**.** A VSID with functions F, M, Q is sound for an auxiliary information aux, if for any probabilistic polynomial time adversary $\mathcal{A}$, it holds that the following probability is $\mathsf{negl}(\lambda)$:

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

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

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

The above property ensures an honest server can detect a malicious client if the client provides ill-structured inputs. It is further required that a malicious party be identified by an honest third party, arbiter. This ensures that in the case of dispute (or false accusation) a malicious party can be pinpointed. A VSID supports detectable abort if a corrupt party can escape from being identified, by the arbiter, with only negligible probability. Formally:

Definition 16 (VSID Detectable Abort)**.** A VSID with functions F, M, Q supports detectable abort for an auxiliary information aux, if the following hold:

1) For any PPT adversary $\mathcal{A}_1$, the following probability is $\mathsf{negl}(\lambda)$:

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

2) For any PPT adversary $\mathcal{A}_2$, the following probability is $\mathsf{negl}(\lambda)$:

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

D.1.1. Lighter VSID Scheme ($\text{VSID}_{\text{light}}$). In the VSID definition, algorithm $\texttt{VSID.identify}(\cdot)$ allows an arbiter to identify a misbehaving client even in the setup phase. Nevertheless, it is often sufficient to let the arbiter pinpoint a corrupt party *after* the client and server agree to deal with each other, i.e., after the setup when the server runs $\texttt{VSID.serve}(\cdot)$ and outputs 1. A VSID protocol that meets the latter (lighter) requirements, denoted by $\text{VSID}_{\text{light}}$, would impose lower costs especially when u and elements of e are of large size. Because the arbiter is not required to identify a misbehaving client in setup; therefore, it does not need to have access to the entire file u^* and metadata e. This means (a) the server or client does not need to send u^* and e to the arbiter that leads to lower communication cost, and (b) the arbiter skips checking the correctness of metadata in $\texttt{VSID.identify}(\cdot)$, which ultimately saves it computation cost too. In $\text{VSID}_{\text{light}}$, algorithm $\texttt{VSID.identify}(\cdot)$ needs to take only (π, c, k, e', pp) as input, where $e' \subset e$. So, this requires two changes to the VSID definition, (a) the arbiter algorithm would be $\texttt{VSID.identify}(\pi, c, k, e', pp) \rightarrow I$, and (b) in case 2, in Definition 16 we would have $b = 0 \wedge I \neq \mathcal{C}$, so event $a = 0$ is excluded. In this paper, any time we refer to $\text{VSID}_{\text{light}}$, we assume the above minor adjustments are applied to the VSID definition.

D.2. VSID Protocol

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

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

then it proceeds to the next step. Otherwise, it outputs $I = \mathcal{C}$ and halts.

c) If $\boldsymbol{\pi}$ is privately verifiable, then the arbiter first checks if $sk \in k$ (provided by the client along with other opening information) matches $\texttt{Com}_{sk} \in pk$. If they do not match, then the arbiter outputs $I = \mathcal{C}$. Otherwise, it calls $\texttt{VS.verify}(\boldsymbol{\pi}, \boldsymbol{q}, k, pp) \rightarrow d$. If $d = 1$, then it outputs $I = \perp$; otherwise, it outputs $I = \mathcal{S}$.

Theorem 2. *The VSID protocol with functions F, M, Q satisfies the correctness, soundness, inputs well-formedness, and detectable abort properties for auxiliary information aux, (cf. Definitions 13-16), if the underlying VS protocol with functions F, M, Q is correct and sound for aux and the underlying commitment, publicly verifiable non-interactive zero-knowledge, and signature schemes are correct/complete and secure.*

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

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

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

E. Recurring Contingent Service Payment (RC-S-P) Protocol

In this section, we present our RC-S-P protocol.

E.1. Recurring Contingent Service Payment (RC-S-P) Protocol

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

At the end of each billing cycle, the client generates an encrypted query, by calling $\texttt{VSID.genQuery}(\cdot)$ and encrypting its output using the key, k. It pads the encrypted query and sends the result to the contract. The encryption and pads ensure nothing about the client's input (e.g., outsourced file) is revealed to the public within

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

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 $\texttt{VSID.checkQuery}(\cdot)$. If the query is rejected, the server locally stores the index of the billing cycle and then generates a dummy proof. Otherwise, if the server accepts the query, it generates a proof of service by calling $\texttt{VSID.prove}(\cdot)$. In either case, the server encrypts the proof, pads it and sends the result to the contract. Note that sending (padded encrypted) dummy proofs ensures that the public, during the private time bubble, does not learn if the client generates invalid queries. After the server sends the messages to the contract, the client removes the pads, decrypts the proof and locally verifies it, by calling $\texttt{VSID.verify}(\cdot)$. If the verification is passed, then the client knows the server has delivered the service honestly. But, if the proof is rejected, it waits until the private time bubble passes and dispute resolution time arrives. During the dispute resolution period, in the case the client or server rejects any proofs, it invokes the arbiter, refers it to the invalid encrypted proofs in the contract, and sends to it the decryption key and the pads' detail. The arbiter checks the validity of the key and pads, by using SAP. If they are accepted, then the arbiter locally removes the pads from the encrypted proofs, decrypts the related proofs, and runs $\texttt{VSID.identify}(\cdot)$ to check the validity of the party's claim. The arbiter sends to the contract a report of its findings that includes the total number of times the server and client provided invalid proofs. In the next phase, to distribute the coins, either client or server sends: (a) "pay" message, (b) the agreed statement that specifies the payment details, and (c) the statement's proof to the contract which verifies the statement and if approved it distributes the coins according to the statement's detail, and the arbiter's report.

Now we outline why RC-S-P addresses the issues, raised in Section 5. In the setup, if the client provides ill-formed inputs (so later it can accuse the server) then the server can detect and avoid serving it. After the setup, if the client avoids sending any input, then the server still gets paid for the service it provided. Also, in the case of a dispute between the parties, their claim is checked, and the corrupt party is identified. The corrupt party has to pay the arbiter and if that is the client, then it has to pay the server as well. These features not only do guarantee the server's resource is not wasted, but also ensures fairness (i.e., if a potentially malicious server is paid, then it must have provided the service and if a potentially malicious client does not pay, then it will learn nothing). Furthermore, as during the private time bubble (a) no plaintext proof is given to the contract, and (b) no dispute resolution and coin transfer take place on contract, the public cannot figure out the outcome of each verification. This preserves the server's privacy. Also, because the deposited coins are masked and the agreed statement is kept private, nothing about the detail of the service is leaked to the public before the bubble bursts. This preserves the client's privacy. Also, as either party can prove to the contract the validity of the agreed statement, and ask the contract to distribute the coins, the coins will be not be locked forever.

E.1.1. Protocol description. The RC-S-P protocol is parameterized by the functions F, M, Q of the underlying VSID and encoding/decoding functions (E, D) that refer to "encrypt then pad"/"remove pad then decrypt" procedures, respectively. It is assumed that (a) each party $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}, \mathcal{R}\}$ already has a blockchain public address, $adr_{\mathcal{P}}$, which is known to all parties, (b) it uses that (authorised) address to send transactions to the smart contract, (c) the contract before recording a transaction, ensures the transaction is originated from an authorised address, and (d) there is a public price list pl known to everyone. The protocol is presented below.

1) **Key Generation**. $\texttt{RCSP.keyGen}(1^\lambda)$
   a) $\mathcal{C}$ runs $\texttt{VSID.keyGen}(1^\lambda) \rightarrow k := (sk, pk)$. It picks a random secret key $\bar{k}$ for a symmetric-key encryption. Also, it sets two parameters: pad_π and pad_q, where pad_π and pad_q refer to the number of dummy values that will be used to pad encrypted proofs and encrypted queries respectively[7], determined by the security parameter and description of F. Let $sk' := (pad_\pi, pad_q, \bar{k})$. The keys' size is part of the security parameter. Let $\boldsymbol{k} = [k, k']$, where $k' := (sk', pk')$ and $pk' := (adr_{\mathcal{C}}, adr_{\mathcal{S}})$.
2) **Client-side Initiation**. $\texttt{RCSP.cInit}(1^\lambda, u, \boldsymbol{k}, z, pl)$
   a) Calls $\texttt{VSID.setup}(1^\lambda, u, k) \rightarrow (u^*, pp, e)$, to encode service input, and generate metadata. It sets $qp = sk'$ and appends pp to qp.
   b) Calls $\texttt{SAP.init}(1^\lambda, adr_{\mathcal{C}}, adr_{\mathcal{S}}, qp) \rightarrow (r_{qp}, g_{qp}, adr_{\text{SAP}_1})$, to initiate an agreement (with $\mathcal{S}$) on qp. Let $T_{qp} := (\ddot{x}_{qp}, g_{qp})$ be proof/query encoding token, where $\ddot{x}_{qp} := (qp, r_{qp})$ is the opening and g_{qp} is the commitment stored on the contract as a result of running SAP.
   c) Sets coin parameters as follows, o: the amount of coins for each accepting proof, and l: the amount of coins to cover the cost of each potential dispute resolution, given price list pl.
   d) Sets $cp := (o, o_{max}, l, l_{max}, z)$, where o_{max} is the maximum amount of coins for an accepting service proof, l_{max} is the maximum amount of coins to resolve a potential dispute, and z is the number of service proofs/verifications. Then, $\mathcal{C}$ calls $\texttt{SAP.init}(1^\lambda, adr_{\mathcal{C}}, adr_{\mathcal{S}}, cp) \rightarrow (r_{cp}, g_{cp}, adr_{\text{SAP}_2})$, to initiate an agreement (with $\mathcal{S}$) on cp. Let $T_{cp} := (\ddot{x}_{cp}, g_{cp})$ be coin encoding token, where $\ddot{x}_{cp} := (cp, r_{cp})$ is the opening and g_{cp} is the commitment stored on the contract as a result of executing SAP. Let $T := \{T_{qp}, T_{cp}\}$.
   e) Set parameters $coin^*_{\mathcal{C}} = z \cdot (o_{max} + l_{max})$ and $p_{\mathcal{S}} = z \cdot l_{max}$, where $coin^*_{\mathcal{C}}$ and $p_{\mathcal{S}}$ are the total number of masked coins $\mathcal{C}$ and $\mathcal{S}$ should deposit respectively. It also designs a smart contract, SC, that explicitly specifies parameters z, $coin^*_{\mathcal{C}}$, $p_{\mathcal{S}}$, adr_{SAP_1}, adr_{SAP_2}, pk, and pk'. It sets a set of time points/windows, $\texttt{Time} : \{\texttt{T}_0, ..., \texttt{T}_2, \texttt{G}_{1,1}, ..., \texttt{G}_{z,2}, \texttt{J}, \texttt{K}_1, ..., \texttt{K}_3, \texttt{L}\}$, that are explicitly specified in the contract which will accept a certain party's message only in a specified time point/window. The time allocation will become clear in the next phases.

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

f) Sets also four counters $[y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{S}}]$ in SC, where their initial value is 0. It signs and deploys SC to the blockchain. Let adr_{SC} be the address of the deployed SC, and $\boldsymbol{y} : [y_{\mathcal{C}},\ y'_{\mathcal{C}}, y_{\mathcal{S}}, y'_{\mathcal{S}}, adr_{\text{SC}}]$.

g) Deposits $coin^*_{\mathcal{C}}$ coins in the contract. It sends $u^*, z, e, \ddot{x}_{qp}$, and $\ddot{x}_{cp}$ (along with adr_{SC}) to $\mathcal{S}$. Let $\mathtt{T}_0$ be the time that the above process finishes.

3) **Server-side Initiation**.
$\mathtt{RCSP.sInit}(u^*, e, pk, z, T, p_{\mathcal{S}}, \boldsymbol{y})$

a) Checks the parameters in T (e.g., qp and cp) and in SC (e.g., $p_{\mathcal{S}}, \boldsymbol{y}$) and ensures a sufficient amount of coins has been deposited by $\mathcal{C}$.

b) Calls $\mathtt{SAP.agree}(qp, r_{qp}, g_{qp}, adr_{\mathcal{C}}, adr_{\text{SAP}_1}) \rightarrow (g'_{qp}, b_1)$ and $\mathtt{SAP.agree}(cp, r_{cp}, g_{cp}, adr_{\mathcal{C}}, adr_{\text{SAP}_2}) \rightarrow (g'_{cp}, b_2)$, to verify the correctness of tokens in T and to agree on the tokens' parameters, where $qp, r_{qp} \in \ddot{x}_{qp}$, and $cp, r_{cp} \in \ddot{x}_{cp}$. Recall that if both $\mathcal{C}$ and $\mathcal{S}$ are honest, then $g_{qp} = g'_{qp}$ and $g_{cp} = g'_{cp}$.

c) If any above check is rejected, then it sets $a = 0$. Otherwise, it calls $\mathtt{VSID.serve}(u^*, e, pk, pp) \rightarrow a$.

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

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

Billing-cycles Onset. $\mathcal{C}$ and $\mathcal{S}$ engage in the following three phases, i.e., phases 4-6, at the end of every j-th billing cycle, where $1 \leq j \leq z$. Each j-th cycle includes two time points, $\mathtt{G}_{j,1}$ and $\mathtt{G}_{j,2}$, where $\mathtt{G}_{j,2} > \mathtt{G}_{j,1}$, and $\mathtt{G}_{1,1} > \mathtt{T}_2$.

4) **Client-side Query Generation**.
$\mathtt{RCSP.genQuery}(1^\lambda, \text{aux}, k, T_{qp})$

a) Calls $\mathtt{VSID.genQuery}(1^\lambda, \text{aux}, k, pp) \rightarrow c_j := (\boldsymbol{q}_j, \boldsymbol{w}_{q_j})$, to generate a query-proof pair.

b) Encodes c_j, by first encrypting it, $\mathtt{Enc}(\bar{k}, c_j) = c'_j$, where $\bar{k} \in T_{qp}$; and then, padding (each element of) the result with $pad_q \in T_{qp}$ random values that are picked uniformly at random from the encryption's output range, U. Let c^*_j be the result.

c) Sends the padded encrypted query-proof pair, c^*_j, to SC at time $\mathtt{G}_{j,1}$.

5) **Server-side Proof Generation**.
$\mathtt{RCSP.prove}(u^*, \sigma, c^*_j, pk, T_{qp})$

a) Constructs an empty vector, $\boldsymbol{m}_{\mathcal{S}} = \perp$, if $j = 1$.

b) Removes the pads from c^*_j, using parameters of T_{qp}. Let c'_j be the result. Next, it decrypts the result, $\mathtt{Dec}(\bar{k}, c'_j) = c_j$. Then, it runs $\mathtt{VSID.checkQuery}(c_j, pk, pp) \rightarrow b_j$, to check the correctness of the queries.

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

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

6) **Client-side Proof Verification**.
$\mathtt{RCSP.verify}(\boldsymbol{\pi}^*_j, c^*_j, k, T_{qp})$

a) Constructs an empty vector, $\boldsymbol{m}_{\mathcal{C}} = \perp$, if $j = 1$.

b) Removes the pads from $\boldsymbol{\pi}^*_j$, utilising parameters of T_{qp}. Let $\boldsymbol{\pi}'_j$ be the result. It decrypts the service proof: $\mathtt{Dec}(\bar{k}, \boldsymbol{\pi}'_j) = \boldsymbol{\pi}''_j$ and then calls $\mathtt{VSID.verify}(\boldsymbol{\pi}''_j, \boldsymbol{q}_j, k, pp) \rightarrow d_j$, to verify the proof, where $\boldsymbol{q}_j \in c_j$ (c_j is the result of removing pads from c^*_j and then decrypting the result). Note that if $\boldsymbol{\pi}'_j = \mathtt{Enc}(\bar{k}, \boldsymbol{\pi}_j)$, then $\boldsymbol{\pi}''_j = \boldsymbol{\pi}_j$.

- If $\boldsymbol{\pi}''_j$ passes the verification (i.e., $d_j = 1$), then $\mathcal{C}$ concludes that the service for this verification has been delivered successfully.
- Otherwise (when $\boldsymbol{\pi}''_j$ is rejected), $\mathcal{C}$ appends j to $\boldsymbol{m}_{\mathcal{C}}$.

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

7) **Dispute Resolution**.
$\mathtt{RCSP.resolve}(\boldsymbol{m}_{\mathcal{C}}, \boldsymbol{m}_{\mathcal{S}}, z, \boldsymbol{\pi}^*, \boldsymbol{c}^*, pk, T_{qp})$
The phase takes place only in case of dispute, e.g., when $\mathcal{C}$ and/or $\mathcal{S}$ reject any proofs in the previous phases.

a) The arbiter sets counters: $y_{\mathcal{C}}, y'_{\mathcal{C}}, y_{\mathcal{S}}$ and $y'_{\mathcal{S}}$, that are initially set to 0, before time $\mathtt{K}_1$, where $\mathtt{K}_1 > \mathtt{G}_{z,2} + \mathtt{J}$.

b) $\mathcal{C}$ sends $m_{\mathcal{C}}$ and $\ddot{x}_{qp}$ to the arbiter at time $\mathtt{K}_1$. Or, $\mathcal{S}$ sends $m_{\mathcal{S}}$ and $\ddot{x}_{qp}$ to the arbiter at time $\mathtt{K}_1$.

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

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

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

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

iii) calls $\mathtt{VSID.identify}(\boldsymbol{\pi}''_i, c_i, k, e', pp) \rightarrow I_i$

- if $I_i = \mathcal{C}$ and $y'_{\mathcal{C}}$ was not incremented for i-th verification, it increments $y_{\mathcal{C}}$ by 1.
- if $I_i = \mathcal{S}$ and $y'_{\mathcal{S}}$ was not incremented for i-th verification, it increments $y_{\mathcal{S}}$ by 1.
- if $I_i = \perp$, then it increments $y'_{\mathcal{C}}$ or $y'_{\mathcal{S}}$ by 1, if i is in the complaint of $\mathcal{C}$ or $\mathcal{S}$ respectively and $y_{\mathcal{C}}$ or $y_{\mathcal{S}}$ was not incremented in i-th verification.

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

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

8) **Coin Transfer**. $\mathtt{RCSP.pay}(\boldsymbol{y}, T_{cp}, a, p_s, coin^*_c, coin^*_s)$
   a) If SC receives "pay" message at time $\mathtt{T}_2$, where $a = 0$ or $coins^*_s < p_s$, then it sends $coin^*_c$ coins to $\mathcal{C}$ and $coin^*_s$ coins to $\mathcal{S}$. In other words, the parties can withdraw their coins if they do not reach to an agreement in the end of phase 3, i.e., server-side initiation. Otherwise (i.e., they reach to an agreement), they take the following steps.
   b) Either $\mathcal{C}$ or $\mathcal{S}$ sends "pay" message and the statement, $\ddot{x}_{cp} \in T_{cp}$, to SC at time $\mathtt{L} > \mathtt{K}_3$.
   c) SC checks the validity of the statement by sending $\ddot{x}_{cp}$ to the SAP contract which returns either 1 or 0. SC only proceeds to the next step if the output is 1.
   d) SC distributes the coins to the parties as follows:
      - $coin_c = coin^*_c - o \cdot (z - y_s) - l \cdot (y_c + y'_c)$ coins to $\mathcal{C}$.
      - $coin_s = coin^*_s + o \cdot (z - y_s) - l \cdot (y_s + y'_s)$ coins to $\mathcal{S}$.
      - $coin_{\mathcal{R}} = l \cdot (y_s + y_c + y'_s + y'_c)$ coins to the arbiter.

E.1.2. Discussion on the RC-S-P protocol. We conclude Subsection E.1 with the following remarks:

- The length of a private time bubble can be agreed between the server and client to be of any size that suits them and can exceed the point where the z-th verifications is completed.
- For the sake of simplicity, in the RC-S-P protocol, we let each $y \in \{y_c, y'_c, y_s, y'_s\}$ be a counter; instead of a binary vector, $\boldsymbol{y} \in \{\boldsymbol{y}_c, \boldsymbol{y}'_c, \boldsymbol{y}_s, \boldsymbol{y}'_s\}$, defined in the RC-S-P definition. However, it is not hard to see that the sum of all elements $\boldsymbol{y}$ of equal y, i.e., $y = \sum_{j=1}^{z} \boldsymbol{y}_j$. The same holds for the amounts of coin each party receives, $coin \in \{coin_c, coin_s, coin_{\mathcal{R}}\}$, in the protocol and the coin vector used in the definition, $\boldsymbol{coin} \in \{\boldsymbol{coin}_c, \boldsymbol{coin}_s, \boldsymbol{coin}_{\mathcal{R}}\}$.
- In the protocol, the pads are added *after* the actual values are encrypted. This is done to save computation cost. Otherwise (if the pads are added prior to the encryption), then the pads would have to be encrypted too, which imposes additional computation cost.
- As stated in Section 7, $\mathtt{RCSP.genQuery}(\cdot), \mathtt{RCSP.prove}(\cdot)$, $\mathtt{RCSP.verify}(\cdot)$ and $\mathtt{RCSP.resolve}(\cdot)$ implicitly take $a, coin^*_s, p_s$ as another inputs and execute only if $a = 1$ and $coin^*_s = p_s$. For the sake of simplicity, we avoided explicitly stating it in the protocol. Also, keeping track of (y'_c, y'_s) enables the arbiter to make malicious parties, that *unnecessarily* invoke it for accepting proofs in step 7(d)iii, pay for the verifications it performs.
- The total coin amounts the client receives is as follows; its initial deposit, i.e., $coin^*_c$, minus the total coin amounts that the server should be paid for those verifications that it has acted honestly towards the client, i.e., $o \cdot (z - y_s)$, minus the total coin amounts the client has to pay to the arbiter when it misbehaved towards the server and the arbiter, i.e., $l \cdot (y_c + y'_c)$. The total coin amounts the server receives is as follows. Its initial deposit, i.e., $coin^*_s$, plus the total coin amounts that it should get paid for those verifications that it acted honestly towards the client, i.e., $o \cdot (z - y_s)$, minus the total coin amounts it has to pay to the arbiter when it misbehaved towards the client and the arbiter, i.e., $l \cdot (y_s + y'_s)$. Moreover the arbiter receives in total $l \cdot (y_s + y_c + y'_s + y'_c)$ coins to cover its cost of resolving disputes, i.e., $l \cdot (y_s + y_c)$, plus the cost imposed to it when it is unnecessarily invoked, i.e., $l \cdot (y'_s + y'_c)$. If all parties behave honestly, then the server receives all its deposit back plus the coin amounts they initially agreed to pay the server if it delivers accepting proofs for all z cycles, i.e., in total it receives $coin^*_s + o \cdot z$ coins. Also, in this case an honest client receives all coins minus the coin amounts paid to the server for delivering accepting proofs for z cycles, i.e., in total it receives $coin^*_c - o \cdot z$ coins. However, the arbiter receives no coins, as it is never invoked.
- The VSID scheme does not (need to) preserve the privacy of the proofs. However, in RC-S-P protocol each proof's privacy must be preserved, for a certain time; otherwise, the proof itself can leak its status, e.g., when it can be publicly verified. This is the reason why in the RC-S-P protocol, *encrypted* proofs are sent to the contract. Moreover, for the sake of simplicity, in the above protocol, we assumed that each arbiter's invocation has a fixed cost regardless of the number of steps it takes. To define a fine-grained costing, one can simply allocate to each step the arbiter takes a certain rate and also separate counter for the client and server.
- In the case where $\mathtt{VSID.verify}(\cdot)$ is privately verifiable and the server invokes the arbiter, the client needs to provide inputs to the arbiter too. Otherwise (when it is publicly verifiable and the server invokes the arbiter), the client's involvement is not required in the dispute resolution phase. In contrast, if the client invokes the arbiter, the server's involvement is not required in that phase, regardless of the type of verifiability $\mathtt{VSID.verify}(\cdot)$ supports. Furthermore, with a minor adjustment to the RC-S-P protocol, we can let the client and server be compensated (by a misbehaving party) for the transaction they send to the contract. To do so, briefly, we can let the parties, in initiation phases, agree on and include in cp parameters, l' and l'', that cover the client's and server's cost of sending a transaction, respectively. The parameters are encoded the same way as l is encoded. In this setting, in the coin transfer phase, the client and server receive $coin^*_c - o \cdot (z - y_s) - l \cdot (y_c + y'_c) + l' \cdot y_s - l'' \cdot y_c$ and $coin^*_s + o \cdot (z - y_s) - l \cdot (y_s + y'_s) - l' \cdot y_s + l'' \cdot y_c$, coins respectively. The amount of coins the arbiter receives remains unchanged.
- The server or client, even during the private time bubble, can spend (or more accurately promise to a third party) the amount of coins kept in the contract and will ultimately be transferred to it. With slight adjustments to the RC-S-P, they can do so in a privacy-preserving manner. We briefly explain how it can be done. For the sake of simplicity, we assume the server will receive $coin_s$ coins after the bubble bursts and

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

wants to promise $\hat{coin}_{\mathcal{S}}$ coins (where $\hat{coin}_{\mathcal{S}} \leq coin_{\mathcal{S}}$) to the third party $\mathcal{D}$ within the bubble. First, the server proves to $\mathcal{D}$ that it will receive $coin_{\mathcal{S}}$ coins after the bubble bursts. To do that, it sends the RC-S-P transcripts (that includes all proofs) to $\mathcal{D}$ which can verify the server's claim, as all proofs are publicly verifiable. Next, if $\mathcal{D}$ is convinced, the server and $\mathcal{D}$ invoke a new instance of the SAP and insert the value $\hat{coin}_{\mathcal{S}}$ into the SAP's private statement. This results in a smart contract, SC_{SAP_3}. Next, if both parties agree on the parameters of SC_{SAP_3}, then the server sends the address of SC_{SAP_3} to the main contract of RC-S-P, i.e., SC. When the bubble bursts, SC transfers the client's share of coins to the client as before. But, SC distributes the server's coins if the server or $\mathcal{D}$ sends to it a valid proof for the above private statement (in addition to the proofs required in the Phase 8 of the original RC-S-P). Upon receiving that proof, SC invokes SC_{SAP_3} to check the validity of the proof. If the proof is accepted, then SC sends $\hat{coin}_{\mathcal{S}}$ to $\mathcal{D}$ and $coin_{\mathcal{S}} - \hat{coin}_{\mathcal{S}}$ to the server. It is evident that this approach leaks no information about the coins amount (including $\hat{coin}_{\mathcal{S}}$) during the bubble to the public, due to the security of the SAP. The above idea can be further extended to support multiple parties. For instance, if the server wants to promise $coin_{\mathcal{S}} - \hat{coin}_{\mathcal{S}}$ coins to $\mathcal{D}'$ (after its promise to $\mathcal{D}$), it needs to send to $\mathcal{D}'$ all the proofs, including the one related to the above private statement.

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

E.2. Security Analysis of RC-S-P Protocol

In this section, we analyse the security of RC-S-P protocol, presented in Section E.1. First, we present the protocol's primary security theorem.

Theorem 3. *The RC-S-P protocol with functions F, M, E, D, Q presented in Section E.1 is secure for auxiliary information aux, (cf. Definition 6), if the underlying VSID protocol with functions F, M, Q satisfies correctness, soundness, inputs well-formedness, and detectable abort for aux_j, the SAP is secure, the signature scheme is secure, and the symmetric-key encryption scheme is IND-CPA secure.*

To prove Theorem 3, we show that RC-S-P meets all security properties defined in Section 7. We start by proving that RC-S-P satisfies security against a malicious server.

Lemma 1. *If the SAP and signature scheme are secure and the VSID protocol satisfies correctness, soundness, and detectable abort for auxiliary information aux, then the RC-S-P protocol presented in Section E.1 is secure against malicious server for aux.(cf. Definition 3).*

Proof. We first consider event

$$\Big(F(u^*, \boldsymbol{q}_j, pp) = h_j\Big) \wedge$$

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

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

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

Nevertheless, the server cannot falsely accuse the client of misbehaviour. This is because, due to the security of SAP (i.e., the underlying commitment's binding property), it cannot convince the arbiter to accept different decryption key or pads other than what was agreed with the client in the initiation phase. Specifically, it cannot persuade the arbiter to accept $\ddot{x}'_{qp}$, where $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$, except with a negligible probability. This ensures that the honest client's message is accessed by the arbiter with a high probability, as the arbiter can extract the client's message using valid pad information and decryption key. On the other hand, if the adversary provides a valid statement, i.e., $\ddot{x}_{qp}$, then due to the correctness of VSID, algorithm $\texttt{VSID.identify}(\cdot)$ outputs $I_j = \perp$. Therefore, due to the security of SAP

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

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

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

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

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

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

$$\Big(F(u^*, \boldsymbol{q}_j, pp) \neq h_j\Big) \wedge$$

$$\Big(d_j = 1 \vee y_{\mathcal{S},j} = 0 \ \vee coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} \ \vee coin_{\mathcal{R},j} \neq l\Big)$$

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

Lemma 2. *If the SAP and signature scheme are secure and the VSID scheme satisfies correctness, inputs well-formedness, and detectable abort for auxiliary information aux, then the RC-S-P protocol presented in Section E.1 is secure against malicious client for aux (cf. Definition 4).*

Proof. First, we consider event

$$\Big(M(u^*, k, pp) = \sigma \ \wedge \ Q(\text{aux}, k, pp) = \boldsymbol{q}_j\Big) \wedge$$

$$\Big((coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o) \ \vee \ (coin_{\mathcal{R},j} \neq l \ \wedge \ y'_{\mathcal{C},j} = 1)\Big)$$

This event captures the case where the client provides accepting metadata and query but makes the server withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o$, or makes the arbiter withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{R},j} \neq l$, if it unnecessarily invokes the arbiter. Since the metadata and query's proofs are valid, an honest server accepts them and does not raise any dispute, so we have $y_{\mathcal{C},j} = 0$. The client could make the server withdraw incorrect amount of coins, if it manages to either convince the arbiter, in phase 7, that the server has misbehaved, i.e., makes the arbiter output $y_{\mathcal{S},j} = 1$, or submit to the contract an accepting statement $\ddot{x}'_{cp}$ other than what was agreed at the initiation phase, i.e., $\ddot{x}_{cp}$, in phase 8, or send a message on the server's behalf to invoke the arbiter unnecessarily. However, it cannot falsely accuse the server of misbehaviour, as due to the security of SAP (i.e., the binding property) it cannot convince the arbiter to accept different decryption key and pads' detail, by providing a different accepting statement $\ddot{x}'_{qp}$ (where $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$), than what was initially agreed with the server, except with negligible probability. This ensures the arbiter is given the honest server's messages, with a high probability. So, with a high probability $y_{\mathcal{S},j} = 0$. On the other hand, if the adversary provides a valid

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

statement, i.e., $\ddot{x}_{qp}$, then due to the correctness of VSID, algorithm $\texttt{VSID.identify}(\cdot)$ outputs $I_j = \perp$. So, due to the security of SAP and correctness of VSID, we would have $y_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$ with a high probability. Moreover, due to the security of SAP, the client cannot convince the contract to accept any statement $\ddot{x}'_{cp}$ other than what was initially agreed between the client and server (i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$), except with negligible probability. Also, it holds that $y'_{\mathcal{S},j} = 0$ because an honest server never invokes the arbiter when the client's messages are well-structured and due to the signature's unforgeability, the client cannot send a signed message on the server's behalf to unnecessarily invoke the arbiter. According to RC-S-P protocol, the total coins the server should receive after z verifications is $coin^*_{\mathcal{S}} + o \cdot (z - y_{\mathcal{S}}) - l \cdot (y_{\mathcal{S}} + y'_{\mathcal{S}})$. Since we focus on the j-th verification, the amount of coins that should be credited to the server for the j-th verification is

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

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

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

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

$$\Big(Q(\text{aux}, k, pp) \neq \boldsymbol{q}_j\Big) \wedge$$

$$\Big(b_j = 1 \vee y_{\mathcal{C},j} = 0 \vee coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o \vee coin_{\mathcal{R},j} \neq l\Big)$$

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

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

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

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

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

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

Lemma 4. *If SAP is secure and the symmetric-key encryption scheme is IND-CPA secure, then the RC-S-P protocol presented in Section E.1 preserves privacy for auxiliary information aux, (cf. Definition 5).*

Proof. We start with case 1, i.e., the privacy of service input. Due to the privacy property of SAP, that stems from the hiding property of the commitment scheme, given the commitments g_{qp} and g_{cp}, (that are stored in the blockchain as a result of running SAP) the adversary learns no information about the committed values (e.g., o, l, pad_π, pad_q, and $\bar{k}$), except with a negligible probability. Also, given price list pl, encoded coins $coin^*_{\mathcal{C}} = z \cdot (o_{max} + l_{max})$ and $coin^*_{\mathcal{S}} = z \cdot l_{max}$, the adversary

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

learns nothing about the actual price that was agreed between the server and client, (o, l), for each verification, due to Lemma 3. Next we analyse the privacy of padded encrypted query vector $\boldsymbol{c}^*$. For the sake of simplicity, we focus on $\boldsymbol{q}_j^* \in c_j^* \in \boldsymbol{c}^*$, that is a padded encrypted query vector for j-th verification. Let $\boldsymbol{q}_{j,0}$ and $\boldsymbol{q}_{j,1}$ be query vectors, for j-th verification, related to the service inputs u_0 and u_1 that are picked by the adversary according to Definition 5 which lets the environment pick $\beta \xleftarrow{\$} \{0,1\}$. Also, let $\{\boldsymbol{q}_{j,0}, ..., \boldsymbol{q}_{j,\bar{m}}\}$ be a list of all queries of different sizes. In the experiment, if $\boldsymbol{q}_{j,\beta}$ is only encrypted (but not padded), then given the ciphertext, due to semantical security of the encryption, an adversary cannot tell if the ciphertext corresponds to $\boldsymbol{q}_{j,0}$ or $\boldsymbol{q}_{j,1}$ (accordingly to u_0 or u_1) with probability significantly greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$, under the assumption that the size of $\boldsymbol{q}_{j,\beta}$ is equal to the size of largest query size [8], i.e., $Max(|\boldsymbol{q}_{j,0}|, ..., |\boldsymbol{q}_{j,\bar{m}}|) = |\boldsymbol{q}_{j,\beta}|$. The above assumption is relaxed with the use of a pad; as each encrypted query is padded to the queries' maximum size, i.e., $Max(|\boldsymbol{q}_{j,0}|, ..., |\boldsymbol{q}_{j,\bar{m}}|)$, the adversary cannot tell with a probability greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$ if the padded encrypted proof corresponds to $\boldsymbol{q}_{j,0}$ or $\boldsymbol{q}_{j,1}$, as the padded encrypted query *always has the same size* and the pad values are picked from the same range as the encryption's ciphertext are defined. The same argument holds for $\boldsymbol{w}_{q_j}^* \in c_j^* \in \boldsymbol{c}^*$. Next we analyse the privacy of padded encrypted proof vector $\boldsymbol{\pi}^*$. The argument is similar to the one presented above, however, we provide it for the sake of completeness. We focus on an element of the vector, $\pi_j^* \in \boldsymbol{\pi}^*$, that is a padded encrypted proof for j-th verification. Let $\pi_{j,0}$ and $\pi_{j,1}$ be proofs, for j-th verification, related to the service inputs u_0 and u_1, where the inputs are picked by the adversary, w.r.t. Definition 5 in which the environment picks $\beta \xleftarrow{\$} \{0,1\}$. Let $\{\pi_{j,0}, ..., \pi_{j,\bar{m}}\}$ be proof list including all proofs of different sizes. If we assume $\pi_{j,\beta}$ is only encrypted, then given the ciphertext, due to semantical security of the encryption, an adversary cannot tell if the ciphertext corresponds to $\pi_{j,0}$ or $\pi_{j,1}$ (accordingly to u_0 or u_1) with a probability significantly greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$, if $Max(|\pi_{j,0}|, ..., |\pi_{j,\bar{m}}|) = |\pi_{j,\beta}|$. However, the assumption is relaxed with the use of a pad. In particular, since each encrypted proof is padded to the proofs' maximum size, the adversary cannot tell with a probability greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$ if the padded encrypted proof corresponds to $\pi_{j,0}$ or $\pi_{j,1}$. Also, since the value of a is independent of u_0 or u_1, and only depends on whether the metadata is well-formed, it leaks nothing about the service input u_β, β, the query-proof pair and service proof. Thus (given $\boldsymbol{c}^*, coin_{\mathcal{S}}^*, coin_{\mathcal{C}}^*, g_{cp}, g_{qp}$, $\boldsymbol{\pi}^*, pl$, and a) the probability that the adversary can tell the value of β is at most $\frac{1}{2} + \mathsf{negl}(\lambda)$, due to IND-CPA security of the symmetric-key encryption scheme.

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

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

F. Proof of Theorem 1

This section contains the security analysis of the RC-PoR-P consrtuction presented in Section 9. First, we prove the security of the PoR scheme in Subsection 9.1 in the following lemma.

Lemma 5. *Let ϵ be non-negligible in the security paramenter λ. Then, the PoR scheme presented in Subsection 9.1 is ϵ-sound w.r.t. Definition 8, if the underlying Merkle tree and pseudorandom function* PRF *are secure.*

Proof (sketch). As stated above, the proposed PoR differs from the standard Merkle tree-based PoR by a couple of perspectives. However, the changes do not affect the security and soundness of the proposed PoR. Its security proof is similar to the existing Merkle tree-based PoR schemes, e.g., [30], [34], [43]. Alternatively, our protocol can be proven based on the security analysis of the PoR schemes that use MACs or BLS signatures, e.g., [48]. In this case, the extractor design (in the Merkle tree-based PoR) would be simpler because it does not need to extract blocks from a linear combination of MACs or signatures, as the blocks are included in proofs, i.e., they are part of the Merkle tree proofs. Intuitively, in either case, the extractor interacts with any adversarial prover that passes a non-negligible ϵ fraction of audits. It initialises an empty array. Then it challenges a subset of file blocks and asks the prover (server) to generate a proof. If the received proof passes the verification, then it adds the related block (in the proof) to the array. It then rewinds the prover and challenges a fresh set of blocks, and repeats the process many times. Since the prover has a good chance of passing the audit, it is easy to show that the extractor can eventually extract a large fraction of the entire file, as it is shown in [48]. In particular, the following hold:

1) Due to the security of the Merkle tree, which relies on the collision resistance of the utlized hash function, whenever the extractor is convinced of a prover's proof of membership, it is ascertained that the retrieved values are the valid and correct file blocks.

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

2) Due to the security of the pseudorandom function, the challenges $q_i = \big(\mathtt{PRF}(\hat{k}, i) \bmod m\big) + 1$, for $i = 1, \ldots, \phi$ are not predictable by $\mathcal{S}$ before the time that the server receives the PRF key $\hat{k}$ by $\mathcal{C}$.

After collecting a sufficient number of blocks, the extractor can use the error-correcting code to decode and recover the entire file blocks, given the retrieved ones. □

By applying Lemma 5, we prove the main theorem of Section 9. Note that, given the value of a counter, e.g., y_S, we can easily parse it as a binary vector, e.g. $y_{S,1}, ..., y_{S,z}$. This is because each counter is maintained and incremented by the smart contract, hence the counter's history/log can be retrieved from the blockchain.

Theorem 1. *The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ described in Subsections 9.1 and 9.2 is secure (cf. Definition 6), if the underlying Merkle tree, pseudorandom function, commitment scheme, digital signature scheme are secure, and the underlying symmetric-key encryption scheme is IND-CPA secure.*

Proof. We show that the RC-PoR-P scheme meets all security properties defined in Section 7 by proving a series of claims. First, we recall that $coin_{\mathcal{P},j}$ denotes the coins that are credited to the party $\mathcal{P} \in \{\mathcal{C}, \mathcal{S}, \mathcal{R}\}$ for the j-th verification and h_j is a value included in the decoded proof π_j that should match $F_{\mathsf{PoR}}(u^*, \hat{k}_j, pp)$. In addition, $y_{\mathcal{C},j} = 1$ (resp. $y_{\mathcal{S},j} = 1$) if $\mathcal{C}$ (resp. $\mathcal{S}$) misbehaved in the j-th billing cycle, and $y'_{\mathcal{C},j} = 1$ (resp. $y'_{\mathcal{S},j} = 1$) if $\mathcal{C}$ (resp. $\mathcal{S}$) has provided a complaint that does not allow $\mathcal{R}$ to identify a misbehaved party in the j-th verification .

Claim 1. The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ is secure against a malicious server (cf. Definition 3), if the SAP and signature scheme are secure, and the PoR scheme satisfies correctness and soundness.

Proof of Claim 1. First, we consider the event $\Big(F_{\mathsf{PoR}}(u^*, \hat{k}_j, pp) = h_j \ \wedge \Big((coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o) \vee (coin_{\mathcal{R},j} \neq l \ \wedge \ y'_{\mathcal{S},j} = 1)\Big)\Big)$ that captures the case where the server provides an accepting proof, but one of the following happen:

1) an honest client withdraws an incorrect amount of coins, i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} - o$, or
2) the arbiter withdraws an incorrect amount of coins, i.e., $coin_{\mathcal{R},j} \neq l$, if the server unnecessarily invokes the arbiter, i.e., $y'_{\mathcal{S},j} = 1$

Because the proof is valid, an honest client accepts it and does not raise a dispute. However, the server could make the client withdraw incorrect amount of coins, if it manages to (i) convince the arbiter that the client has misbehaved, by making the arbiter output $y_{\mathcal{C},j} = 1$ through the dispute resolution phase, or (ii) submit an accepting statement $\ddot{x}'_{cp}$ to SC which is other than what was agreed in the initiation phase, i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$, so it can change the payments' parameters, or (iii) send a message on the client's behalf to unnecessarily invoke the arbiter. We argue that in any of the above three cases, the server cannot falsely accuse the client of misbehaviour.

First, due to the binding property of the SAP commitment scheme, $\mathcal{S}$ cannot convince the arbiter to accept a different decryption key (that will be used to decrypt queries) other than what was agreed with the client in the SAP initiation phase. In particular, it cannot persuade the arbiter to accept $\ddot{x}'_{qp}$, where $\ddot{x}'_{qp} \neq \ddot{x}_{qp}$, except with $\mathsf{negl}(\lambda)$ probability. This ensures that the honest client's queries are accessed by the arbiter with a high probability. Furthermore, if the adversary provides a valid statement, i.e., $\ddot{x}_{qp}$, then due to the correctness of the PoR and query-checking process (specified in step 5b), no one is identified as a misbehaving party in the dispute resolution phase, i.e., so we would have $I_j = \perp$. Therefore, due to the binding property of SAP and correctness of PoR and query-checking process, the following holds $y_{\mathcal{C},j} = y_{\mathcal{S},j} = 0$.

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

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

We now study the event $\Big(\Big(F_{\mathsf{PoR}}(u^*, \hat{k}_j, pp) \neq h_j\Big) \ \wedge \Big(d_j = 1 \vee y_{\mathcal{S},j} = 0 \ \vee coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z} \ \vee coin_{\mathcal{R},j} \neq l\Big)\Big)$, which captures the case where the server provides an invalid proof and causes at least one of the following to happen:

1) convinces the client/arbiter to accept the proof, or
2) persuades the arbiter to consider it as honest, i.e., keep $y_{\mathcal{S},j} = 0$, or
3) makes the client or arbiter withdraw incorrect amount of coins, i.e., $coin_{\mathcal{C},j} \neq \frac{coin^*_{\mathcal{C}}}{z}$ or $coin_{\mathcal{R},j} \neq l$ respectively.

By the security of the underlying pseudorandom function and the Merkle tree, Lemma 5 holds, i.e., the tree-based PoR building block is sound. This implies that (i)

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

the probability that the adversary can convince an honest client/arbiter to accept invalid proof is $\mathsf{negl}(\lambda)$ and (ii) the file is extractable within a polynomial number of interactions with an ϵ-admissible adversary, where ϵ is some non-negligible function. Therefore, the client outputs $d_j = 0$ with a high probability and raises a dispute.

Furthermore, the server may try to convince the arbiter that the client has misbehaved, and output $y_{\mathcal{C},j} = 1$; if it succeeds, then the arbiter would consider the server as honest. Therefore, it would keep $y_{\mathcal{S},j} = 0$ (according to the protocol's description). In this case, according to RC-PoR-P protocol, if the honest client sends a complaint, its complaint (for the j-th verification) would not be processed by the arbiter, because the arbiter has already detected a misbehaving party, i.e., the client in this case. This allows $y_{\mathcal{S},j}$ to remain 0. Nonetheless, as we argued in the study of the previous event, due to the binding property of the SAP commitment scheme and the EUF-CMA security of the digital signature scheme, the probability that the adversary makes the arbiter recognise the client as misbehaving is $\mathsf{negl}(\lambda)$.

By the above two paragraphs, with high probability the honest client will raise a dispute and the malicious server cannot convince the arbiter that the client has misbehaved. This means that, with high probability, $y_{\mathcal{S},j} = 1$ and $y_{\mathcal{C},j} = 0$, after the arbiter is invoked by the client or server.

In addition, it holds that $y'_{\mathcal{C},j} = 0$ with a high probability, as the honest client will not unnecessarily invoke the arbiter and the server cannot send a message to the arbiter (to unnecessarily invoke it) on behalf of the client (to make the arbiter set $y'_{\mathcal{C},j} = 1$), due to the digital signature's EUF-CMA security. Also, it holds that $y'_{\mathcal{S},j} = 0$, because if the malicious server unnecessarily invokes the arbiter after it is detected, then the arbiter discards its complaint without carrying out any investigation/computation as a malicious party (i.e., the server in this case) has already been identified. Moreover, due to the binding property of the SAP commitment scheme, the probability that the adversary succeeds in changing the payment parameters to make the client or arbiter withdraw an incorrect amount of coins is $\mathsf{negl}(\lambda)$ too. So, according to Equations (1) and (2) the client and arbiter are credited $\frac{coin^*_{\mathcal{C}}}{z}$ and l coins for the j-th verification respectively. Also, due to the liveness of the blockchain and the security and correctness of the smart contract, the second security property of SAP holds, i.e., after the parties agree on the statement, an honest party can almost always prove to the verifier that it has the agreement. Thus, the adversary cannot block an honest client's messages, "pay" and $\ddot{x}_{cp}$, to the contract in the coin transfer phase. ⊣

Claim 2. The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ is secure against a malicious client (cf. Definition 4), if SAP and signature scheme are secure and the Merkle tree scheme is secure.

Proof of Claim 2. We first consider the event $\Big(\Big(M_{\mathsf{PoR}}(u^*, k, pp) = \sigma \ \wedge \ Q_{\mathsf{PoR}}(\mathrm{aux}, k, \ pp) = \hat{k}_j\Big) \ \wedge \ \Big((coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o) \ \vee \ (coin_{\mathcal{R},j} \neq l \ \wedge \ y'_{\mathcal{C},j} = 1)\Big)\Big)$. It captures the case where the client provides accepting metadata (i.e., a Merkle tree and its root) and query but one the following happen:

1) the server withdraws incorrect amount of coins, i.e., $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o$, or
2) the arbiter withdraw incorrect amount of coins, i.e. $coin_{\mathcal{R},j} \neq l$, when the client unnecessarily invokes the arbiter.

Since the metadata and queries are valid and correctly structured, an honest server accepts them and does not raise a dispute, so $y_{\mathcal{C},j} = 0$. However, the client could make the server withdraw an incorrect amount of coins if it manages to (i) persuade the arbiter to recognise the server as misbehaving, i.e., makes the arbiter output $y_{\mathcal{S},j} = 1$, or (ii) submit to the contract an accepting statement $\ddot{x}'_{cp}$ other than what was agreed at the initiation phase, i.e., $\ddot{x}_{cp}$, or (iii) send a message on the server's behalf to unnecessarily invoke the arbiter. Nevertheless, it cannot falsely accuse the server of misbehaviour, as, due to the binding property of SAP commitment scheme, it cannot convince the arbiter to accept different decryption key and pads' details 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 $\mathsf{negl}(\lambda)$ probability. This ensures that the arbiter is given the honest server's messages, with high probability. Therefore, with high probability, (i) and (ii) cannot happen, implying that $y_{\mathcal{S},j} = 0$. In addition, (iii) may happen only with $\mathsf{negl}(\lambda)$ probability, as due to the signature scheme's EUF-CMA security, the client cannot send a message on the server's behalf to unnecessarily invoke the arbiter.

Moreover, it holds that $y'_{\mathcal{S},j} = 0$ because (a) the honest server never invokes the arbiter when the client's queries are well-structured and (b) due to the signature scheme's EUF-CMA security, the client cannot send a message on the server's behalf to unnecessarily invoke the arbiter. Note that, due to the binding property of the SAP commitment scheme, the client cannot change the payment parameters by convincing the contract to accept any statement $\ddot{x}'_{cp}$ other than what was initially agreed between the client and server (i.e., $\ddot{x}'_{cp} \neq \ddot{x}_{cp}$) except with a negligible probability, $\mathsf{negl}(\lambda)$. Recall, according to Equation 3, in RC-PoR-P or RC-S-P protocol, the total coins the server should be credited for j-th verification is $coin_{\mathcal{S},j} = \frac{coin^*_{\mathcal{S}}}{z} + o \cdot (1 - y_{\mathcal{S},j}) - l \cdot (y_{\mathcal{S},j} + y'_{\mathcal{S},j})$. Therefore, given $y_{\mathcal{S},j} = y'_{\mathcal{S},j} = 0$, the server is credited $\frac{coin^*_{\mathcal{S}}}{z} + o$ coins for the j-th verification, with high probability.

Furthermore, as stated above, if the adversary invokes the arbiter, the arbiter with high probability outputs $I_j = \perp$ which yields $y'_{\mathcal{C},j} = 1$. According to Equation 2, the amount of coins the arbiter should be credited for j-th verification is $coin_{\mathcal{R},j} = l \cdot (y_{\mathcal{S},j} + y_{\mathcal{C},j} + y'_{\mathcal{S},j} + y'_{\mathcal{C},j})$. As shown above $y_{\mathcal{C},j} = y_{\mathcal{S},j} = y'_{\mathcal{S},j} = 0$ and $y'_{\mathcal{C},j} = 1$, which implies that with high probability, l coins are credited to the arbiter for the j-th verification if it is unnecessarily invoked by the adversary.

We now turn our attention to $\Big(M_{\mathsf{PoR}}(u^*, k, pp) \neq \sigma \ \wedge \ a = 1\Big)$ which captures the case where the server accepts ill-formed metadata. However, due to the security of the Merkle tree scheme that relies on the collision resistance of the utlized hash function, the probability this event happens is $\mathsf{negl}(\lambda)$; therefore, with a high probability $a = 0$. In this case, the server does not raise any dispute, instead it avoids serving the client.

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

Next, we move on to $\Big((Q_{\mathsf{PoR}}(\text{aux}, k, pp) \neq \hat{k}_j) \wedge (b_j = 1 \vee y_{c,j} = 0 \vee coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o \vee coin_{\mathcal{R},j} \neq l)\Big)$, i.e. the event that considers the case where the client provides an invalid query, and causes at least one of the following to happen:

1) convinces the server or arbiter to accept the query, i.e., $b_j = 1$ or $y_{c,j} = 0$, or
2) makes the server or arbiter withdraw an incorrect amount of coins, i.e., $coin_{\mathcal{S},j} \neq \frac{coin^*_{\mathcal{S}}}{z} + o$ or $coin_{\mathcal{R},j} \neq l$ respectively.

Due to the correctness of the query-checking process, the probability that the server outputs $b_j = 1$ is 0. Note that, when the honest server rejects the query and raises a dispute, the arbiter checks the query and sets $y_{c,j} = 1$. After that, due to RC-PoR-P design, the client cannot make the arbiter set $y_{c,j} = 0$ (unless it manages to modify the blockchain's content later on, but its probability of success is negligible due to the security of blockchain). As already stated, due to the binding property of the SAP commitment scheme and the EUF-CMA security of the digital signature scheme, the client cannot make the arbiter recognise the honest server as a misbehaving party with a probability significantly greater than $\mathsf{negl}(\lambda)$. So, with high probability $y_{\mathcal{S},j} = 0$. Furthermore, since the arbiter has identified a misbehaving party, it holds that $y'_{c,j} = y'_{\mathcal{S},j} = 0$. The adversary may still try to make them withdraw incorrect amount of coins. To this end, in the coin transfer phase, it has to send a different accepting statement than what was initially agreed with the server. But, due to the binding property of the SAP commitment scheme, its success probability is $\mathsf{negl}(\lambda)$. Hence with high probability, according to Equations 3 and 2, for $y_{\mathcal{S},j} = y'_{c,j} = y'_{\mathcal{S},j} = 0$ and $y_{c,j} = 1$, the server and arbiter are credited $\frac{coin^*_{\mathcal{S}}}{z} + o$ and l coins respectively for the j-th verification. Furthermore, due to the liveness of the blockchain and the security and correctness of the smart contract, the second security property of SAP holds, i.e., after the parties agree on the statement, an honest party can almost always prove to the verifier that it has the agreement. Thus, the adversary cannot block an honest server's messages, "pay" and $\ddot{x}_{cp}$, to the smart contract in the coin transfer phase. ⊣

Claim 3. The RC-PoR-P scheme with functions $F_{\mathsf{PoR}}, M_{\mathsf{PoR}}, E_{\mathsf{PoR}}, D_{\mathsf{PoR}}, Q_{\mathsf{PoR}}$ preserves privacy (cf. Definition 5), if SAP is secure and the symmetric-key encryption scheme is IND-CPA secure.

Proof of Claim 3. First, we show that the scheme guarantees the privacy of the service input. According to the associated experiment, let $\mathcal{A}_1$ be an adversary that given pk, F, M, E, D, Q, z, pl, outputs its choice of challenge files, u_0 and u_1. Then, $\mathcal{A}_1$ is provided with the vector of encrypted queries $\boldsymbol{c}^*$, the parties encoded coins $coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}$, the SAP commitments g_{cp}, g_{qp}, $\boldsymbol{\pi}^*$, the price list pl, and the sender's decision bit a on the verification of the parameters and the metadata validity; given the aforementioned input, $\mathcal{A}_1$ attempts to guess the challenge bit β.

Due to the hiding property of the commitment scheme, given commitments g_{qp} and g_{cp} (stored in the blockchain as a result of running SAP), the adversary learns no information about the committed values (e.g. o, l, pad_π and $\bar{k}$), except with $\mathsf{negl}(\lambda)$ probability. Moreover, given price list pl, $coin^*_{\mathcal{C}}$ and $coin^*_{\mathcal{S}}$, the adversary learns nothing about the actual price agreed between the server and client, i.e., (o, l), for each verification, due to Lemma 3. Also, since each proof $\boldsymbol{\pi}^*_j$ is encrypted with the IND-CPA secure encryption scheme and then padded, given $\boldsymbol{\pi}^*_j$ the adversary cannot tell whether $\boldsymbol{\pi}^*_j$ is associated with u_0 or with u_1 with probability significantly greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$. As each c^*_j is an output of IND-CPA secure symmetric-key encryption and its size is fixed, it leaks no information to the adversary. The value of a is also independent of u_0 or u_1, and only depends on whether the parameters and metadata are well-formed, so it leaks nothing about the choice of input file u_β and $\beta \in \{0, 1\}$. Overall, the adversary $\mathcal{A}_1$ cannot distinguish with a probability significantly greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$ which file of its choice has been used as the server input.

Next, we show that the scheme guarantees privacy of the service proof's status. For adversaries $\mathcal{A}_2$, $\mathcal{A}_3$ consider the experiment $Exp^{\mathcal{A}_2,\mathcal{A}_3}_{\mathsf{priv}}(1^\lambda)$ defined in Definition 5. Namely, $\mathcal{A}_2$ is initially given pk, F, M, E, D, Q, z, pl and outputs a file u. After client-side and server-side initiation steps have been honestly completed, $\mathcal{A}_2$ acts as a client that creates potentially invalid queries that are provided to $\mathcal{A}_3$ which, in turn, responds with potentially invalid proofs. For each j-th verification, the adversary $\mathcal{A}_2$ or $\mathcal{A}_3$ produces an invalid query or invalid proof, respectively, with probability $Pr_{0,j}$ and a valid query or valid proof, respectively, with probability $Pr_{1,j}$. Finally, an adversary $\mathcal{A}_4$, given $F, M, E, D, Q, \boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl, a$ is challenged to output a proof's verification bit d_j for some j-th verification. Intuitively, the values $Pr_{0,1}, Pr_{1,1}, ..., Pr_{0,z}, Pr_{1,z}$ express the "background knowledge" that $\mathcal{A}_4$ has on the behavior of $\mathcal{A}_2$ and $\mathcal{A}_3$.

We know that each encoded query $c^*_j \in \boldsymbol{c}^*$ is an IND-CPA encryption of a fresh PRF key that is always provided to the contract. So, c^*_j leaks no information (except with $\mathsf{negl}(\lambda)$ probability), not even the query's status to the adversary. Furthermore, each padded IND-CPA encrypted proof $\boldsymbol{\pi}^*_j$ leaks no information (except with $\mathsf{negl}(\lambda)$ probability) and always contains a fixed number of elements. Besides, due to the hiding property of the SAP commitment scheme, the commitments g_{cp}, g_{qp} leak no information about the committed values (e.g. o, l, pad_π and $\bar{k}$), except with $\mathsf{negl}(\lambda)$ probability. In addition, the client-side and server-side initiation steps are carried out honestly in the experiment $Exp^{\mathcal{A}_2,\mathcal{A}_3}_{\mathsf{priv}}(1^\lambda)$, which implies that a is always 1 and thus, it leaks no information about the proof and query validity. Finally, pl, $coin^*_{\mathcal{C}}$ and $coin^*_{\mathcal{S}}$ are generated independently and contain no information about the query's or proof's status.

By the above, the input $(F, M, E, D, Q, \boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl, a)$ does not provide $\mathcal{A}_4$ with any more than $\mathsf{negl}(\lambda)$ advantage compared to any background knowledge $\mathcal{A}_4$ may have on the behavior of $\mathcal{A}_2$ and $\mathcal{A}_3$. Therefore, for each j-th verification, the probability of $\mathcal{A}_4(F, M, E, D, Q, \boldsymbol{c}^*, coin^*_{\mathcal{S}}, coin^*_{\mathcal{C}}, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl, a)$ guessing d_j is maximized when $\mathcal{A}_4$ exploits its background knowledge, so it is upper bounded by $\max\{Pr_{0,j}, Pr_{1,j}\} + \mathsf{negl}(\lambda)$. Overall, the probability

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.

of $\mathcal{A}_4(F, M, E, D, Q, \boldsymbol{c}^*, coin^*_S, coin^*_c, g_{cp}, g_{qp}, \boldsymbol{\pi}^*, pl, a)$ guessing (d_j, j) for some $j \in \{1, \ldots, z\}$ is upper bounded by

$$\begin{aligned}
&\max\big\{\max\{Pr_{0,1}, Pr_{1,1}\} + \mathsf{negl}(\lambda), \ldots, \\
&\quad \max\{Pr_{0,z}, Pr_{1,z}\} + \mathsf{negl}(\lambda)\big\} \leq \\
\leq &\max\big\{\max\{Pr_{0,1}, Pr_{1,1}\}, \ldots, \max\{Pr_{0,z}, Pr_{1,z}\}\big\} + \\
&\quad + \mathsf{negl}(\lambda) \leq \\
\leq &\max\big\{Pr_{0,1}, Pr_{1,1}, \ldots, Pr_{0,z}, Pr_{1,z}\big\} + \mathsf{negl}(\lambda) = \\
= &Pr_{\max} + \mathsf{negl}(\lambda)\ ,
\end{aligned}$$

so the privacy of the service proof is guaranteed. ⊣

The security of the construction follows from Claims 1, 2, and 3. □

G. A Table Summarizing RC-PoR-P Assymptotic Costs

TABLE 4: RC-PoR-P asymptotic complexity, of z verifications, breakdown by parties.

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

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

Authorized licensed use limited to: Peking University. Downloaded on January 19,2024 at 09:22:23 UTC from IEEE Xplore. Restrictions apply.