Recurring Contingent Service Payments

Abstract.

1 Definition

At a high-level, a verifiable service scheme is a two-party protocol in which a client chooses a function, F, and provides (an encoding of) F, and its input, u, to a server. The server is expected to evaluate F on u and respond with the output. Then, the client verifies that the output is indeed the output of the function computed on the provided input.

Definition 1 (Verifiable Service Scheme). *A verifiable service scheme VS* $= (\texttt{VS.KeyGen}, \texttt{VS.setup}, \texttt{VS.genChall}, \texttt{VS.prove}, \texttt{VS.verify})$ *consists of five algorithms defined as follows.*

- $\texttt{VS.KeyGen}(1^{\lambda}, F) \rightarrow k : (sk, pk)$*. A probabilistic algorithm run by the client. It takes as input security parameter and a function, F, that will be run on the client's input by the server. It outputs a secret/public verification key: k.*
- $\texttt{VS.setup}(1^{\lambda}, u, k) \rightarrow \sigma$*. A probabilistic algorithm run by the client. It takes as input security parameter, the service input: u, and the key pair: k. It outputs a set of metadata: σ.*
- $\texttt{VS.genChall}(1^{\lambda}, aux, k) \rightarrow c$*. A probabilistic algorithm run by the client. It takes as input security parameter and auxiliary information: aux, and the key pair: k. It outputs a set of (random) challenges c or empty set if the proving/verification algorithms do not require any challenges.*
- $\texttt{VS.prove}(u, \sigma, c, pk) \rightarrow \pi$*. A probabilistic algorithm run by the server. It takes service input: u, metadata: $u\sigma$, random challenges c and public key: pk. It outputs a proof pair, $\pi = (F(u), \delta)$ containing the function evaluation at input u, i.e. $F(u)$ and a proof δ asserting the evaluation is performed correctly.*
- $\texttt{VS.verify}(\pi, c, k) \rightarrow \{0, 1\}$*. A deterministic algorithm run by the client. It takes the proof: π, random challenges c, and key pair k. If the proof is accepted, it outputs 1; otherwise, it outputs 0.*

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

Definition 2 (Correctness). *A verifiable service scheme VS is correct correct for a class of functions $\mathcal{F}$, if for any $F \in \mathcal{F}$, the key generation algorithm produces keys* $\texttt{VS.KeyGen}(1^{\lambda}, F) \rightarrow k : (sk, pk)$ *such that $\forall u \in$* $\texttt{Domain}(F)$*, if* $\texttt{VS.setup}(1^{\lambda}, u, k) \rightarrow \sigma$*, and* $\texttt{VS.genChall}(1^{\lambda}, aux, k) \rightarrow c$ *and* $\texttt{VS.prove}(u, \sigma, c, pk) \rightarrow \pi$*, then* $\texttt{VS.verify}(\pi, c, k) \rightarrow 1$.

Intuitively, a verifiable service is soundness if a malicious server cannot convince the verification algorithm to accept an incorrect output. In other words, if a prover persuades the verifier with a high probability, then the service has been provided by the prover. It is formally stated as follows.

Definition 3 (Soundness). *A verifiable service VS is sound for a class of functions $\mathcal{F}$, if for any $F \in \mathcal{F}$, any input u^*, all probabilistic polynomial time adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function $\mu(.)$, such that:*

$$Pr\left[F(u^*) \neq y \wedge b = 1 \;\middle|\; \begin{array}{l} \texttt{VS.KeyGen}(1^{\lambda}, F) \rightarrow k : (sk, pk) \\ \mathcal{A}_1(1^{\lambda}, pk, F) \rightarrow (u^*, state) \\ \texttt{VS.setup}(1^{\lambda}, u^*, k) \rightarrow \sigma \\ \texttt{VS.genChall}(1^{\lambda}, aux, k) \rightarrow c \\ \mathcal{A}_2(state, c, \sigma) \rightarrow \pi : (y, \delta) \\ \texttt{VS.verify}(\pi, c, k) \rightarrow b \end{array} \right] \leq \mu(\lambda)$$

The above generic definition captures the core requirements of a wide range of verifiable services such as verifiable outsourced storage, i.e. PoR/PDP schemes, verifiable computation, verifiable searchable encryption, verifiable information retrieval, and verifiable delegated (private) set intersection, to name a few. Any additional security properties mandated by certain services, e.g. privacy or extractability, can be easily plugged into the above definition. However, the above definition and the aforementioned verifiable schemes, assume that the server is potentially malicious while the client is trusted.

In the following, we provide a formal definition of a service scheme (e.g. PoR) run between the client and server, mutually distrusted.

Definition 4. *A service scheme consists of six algorithms defined as follows.*

- `Service.setup`$(1^{\lambda}, u) \rightarrow (k, \sigma, \theta)$. *A probabilistic algorithm run by the client. It takes as input security parameter and the service input: u. It outputs a secret/public verification key and a set of metadata: σ, and proof: θ asserting σ is a well-constructed metadata on u.*
- `Service.serve`$(u, \sigma, \mu) \rightarrow \{0, 1\}$. *A deterministic algorithm run by the server. It takes the service input u, metadata σ and proof μ. It checks if σ has been well-constructed and the client knows the verification key. It outputs 1, if both proofs are accepted and wants to serve the client; otherwise, it outputs 0.*
- `Service.genChall`$(1^{\lambda}, aux) \rightarrow c$. *A probabilistic algorithm run by the client. It takes as input security parameter and auxiliary information (e.g. size of u). It outputs a set of random challenges c (or empty set if proving/verification algorithms do not require any challenges).*
- `Service.prove`$(u, \sigma, c) \rightarrow \pi$. *A probabilistic algorithm run by the server. It takes service input, u, metadata u, and random challenges c. It outputs a proof, π.*
- `Service.verify`$(\pi, k, c) \rightarrow \{0, 1\}$. *A deterministic algorithm run by the client. It takes the proof: π, verification key k, and random challenges c. If the proof is accepted, it outputs 1; otherwise, it outputs 0.*
- `Service.resolve`$(\pi, k, c, d) \rightarrow \{0, 1\}$. *A deterministic algorithm run by a third-party arbiter. It takes the proof: π, verification key k, random challenges c and $d = (\theta', u', \sigma')$, such that $\theta' \subset \theta, u' \subset u$, and $\sigma' \subset \sigma$. If π and σ' are accepted, then it outputs 1; otherwise, it outputs 0.*

In the above definition, `Service.verify`$()$ algorithm allows a verifier to detect only a misbehaving server; nevertheless, it is not suitable to detect a misbehaving client who may try to falsely accuse the server. Thus, `Service.resolve`$()$ algorithm has also been incorporated in the definition, to allow an arbiter to detect either party's misbehaviour and resolve any dispute between them. Informally, a service scheme has two main properties: correctness and soundness. The correctness requires that for any key, the verification algorithm accepts a proof generated by an honest prover. The soundness requires that if a prover convinces the verifier (i.e. client or arbiter), with a high probability, then the service has been provided by the prover. it seems undeniability and accountability/liability, i.e. a malicious party can be identified and held accountable, are needed too. Thin if it's needed here on in the main payment protocol. The exact formalisation of soundness however totally depends on the kind of service provided.

2 Recurring Contingent Service Payments

2.1 Limitations of zkCSP

Recall, the main purpose of zero-knowledge contingent service payment (zkCSP) protocol [2] is to minimise the role of smart contract as much as possible, so (a) the verification's cost would be much lower, if a Turing-complete smart contract framework (e.g. Ethereum) is used, or (b) it can be implemented on a non-Turing-complete contract framework (e.g. Bitcoin).

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

1. *Discrepancies between the Security Guarantees of Service and Fair Exchange Schemes*. zkCSP combines a service scheme secure against only a malicious server (where a client is assumed to be fully trusted) with a fair exchange protocol that takes into the consideration that either server or client might be corrupt. The mismatch leads to serious consequences when the client misbehaves. For instance, in zkCSP the transfer of the deposit requires the client's engagement and approval, after it receives a proof from the server. This allows a malicious client who has been using the service, to avoid sending its approval to the contract or falsely claims the proofs are invalid at the payment time. So, an honest server's resources (e.g. the storage space allocated to the client's data) are wasted. As we will show shortly, in recurring payments (when the server deals with multiple clients) a malicious client can collect convincing information about an honest server that allows it to conclude that it has been served honestly; even though, it does not pay the server and does not check the proof. Thus, it can get a free ride from the server.
2. *Real-time Leakage of Verification Outputs and Deposit Amount*. zkCSP leaks in real-time non-trivial fresh information, about the server and clients, to the public. The leakage includes:
   (a) *proof verification status*: it is visible in the real time to everyone that the proof has been accepted or reject, that reflects whether the server has successfully delivered the agreed-upon service or failed to do so that has serious immediate consequences for both the server and clients, e.g. lost revenue, negative press, stock value drop, or opening doors for attackers to exploit such incident. As an example, observing proof's verification outputs (when a server deals with multiple clients) allows a malicious client to immediately construct a comprehensive background knowledge on the server's current behaviour and status, e.g. the server has been acting honestly. Such auxiliary information can assist the client to more wisely exploit the above deposit issue (that can avoid sending the deposit); for instance, when the sever always acts honestly towards its clients, the client refuses to send the deposit and still has high confidence that the server has delivered the service. As another example, in the case of PoR, a malicious observer can simply find out that the service is suffering hardware/software failure and exploit such vulnerability to mount social engineering attacks on clients or penetrate to the system.
   (b) *deposit amount*: the amount of deposit placed on the contract, swiftly leaks non-trivial information about the client to the public. For instance, in the case of PoR, an observer can learn the size of data outsourced to the server, service type or level of data sensitivity. The situation gets even worse if the client updates its data (e.g. delete or append) or asks the server for additional service (e.g. S3 Glacier or S3 Glacier Deep Archive[1]), as an observer can learn such changes immediately by just observing the amount of deposit put by the client for each payment.

Strawman Solutions for the Two Problems. To address Problem 1, one may slightly adjust the zkCSP protocol such that it would require the client to deposit coins (long) before the server provides the ZK to it, with the hope that the client cannot avoid depositing after the server provides ZK proofs. Nevertheless, this would not work, as the client after accepting the ZK proof, needs to send a confirmation message/transaction to the contract. But a malicious client can avoid doing so and later on, gets its deposit back. Moreover, one might want to let the server pick a fresh address for each verifier/verification, to preserver its pseudonymity with the hope that an observer cannot link clients to a server. However, for this to work, we have to assume multiple service providers use the same protocol on the blockchain and all of them are pseudonymous. But, this is a strong assumption and may not be always feasible. Alternatively, one may let a smart contract to perform the verification on the client's behalf, such that the client deposits its coins in the contract when it starts using the service. Then, the server sends its proof to the contract who performs the verification and pays the server if the proof is accepted. Even though this approach would solve problem 1 above, it imposes a high cost and defeats the purpose of zkCSP design. The reason is that the contract has to always run the verification algorithm that has to be a publicly verifiable one, which usually involves costly public-key primitives.

2.2 Overview of Our Solution

Addressing Issue 1. To address issue 1, we use a combination of the following techniques (over simplified). First, we require a client to deposit its coins to the contract right before it starts using the service, e.g. in the case of PoR before it uploaded its data to the server. Second, we upgrade a verifiable service to a " verifiable service with identifiable abort" (VSIA). This guarantees that not only the service takes into the consideration that the client can be malicious as

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

well, but also the public can identify the misbehaving party (i.e. client or server). Third, we allow a trusted third party, arbiter, to resolve dispute between the parties. Now we explain how the solution (using the above techniques) works. The client before using the service deposit a fixed amount of coins in the contract, where the deposit mount includes the service payment e and p coins to cover dispute resolutions' cost. Also, the sever deposits p coins for covering the cost of future disputes. Then, the client and server run VSIA protocol such that (representation of) all messages exchanged between the parties, i.e. the service challenges and proofs, are put in the contract. The parties perform the verifications locally, off-chain. The client has a chance to rise a dispute if it detects any misbehaviour. In this case, it invokes the arbiter who re-runs the verification and checks the client's claim. The arbiter sends the output of the verification to the contract. If the client's claim is valid, then it can withdraw its coins and the arbiter is paid from the server's deposit. However, if the client's claim is invalid, then the server gets back it deposit and along with the coins the client deposited for the service payment. In this case, the arbiter is paid from the client's deposit. If both the client and server behave honestly, then after a certain time, the server gets its deposit back and is paid for the service, and the client gets p coins back. The reason why the above approach solves problem 1, is that the client pay deposit before it starts using the service, and if acts nothing at the payment time, it is assumed that it has accepted the proof. Also, VSIA allows the arbiter to detect the client if it lies about the verification result; but the server still receives the payment. As evident, if the client and server act honestly/rationally, then there will be no need to involve the arbiter. Later, we will show, in general cases, we can further reduce the involvement of the arbiter (even if a dispute is raised) and in a certain case, i.e. PoR setting, its role can be efficiently played by a smart contract.
Addressing Issue 2. We use the following ideas to address issue 2. Instead of trying to hide the information from the public *forever*, we let it become *stale*, to lose its sensitivity, and then it will become publicly accessible. In particular, the client and server agree on the period in which the data should remain hidden, "private time bubble". This requires, the parties to postpone calling the above pay function, or raising any dispute to the time when the private time bubble ends (or the bubble bursts). However, the client can still find out if the proof is valid as soon as the server provides it, because the verification is locally performed. To further hide the amount of deposit, we let each party to mask their coins. But this raises another challenge: *how can the (mutually untrusted) parties claim back their masking coins after the bubble bursts, while hiding the coins amount from the public in the private time bubble?* Note that it would not work if they explicitly encode in the contract the amount of masking, as it would reveal the masking coins amount to the public in the beginning of the protocol. To address the challenge, we let the client and server, mutually untrusted, to agree on a private statement specifying the deposit details (e.g. parties' coins amount for the service, penalty, or masking). Later, when they call pay function to claim their coins, they also provide the statement to the contract which first checks the validity of the statement and if it is accepted, it distributes coins according to the statement (and status of the contract). We will show how they can efficiently agree on a statement, without using any zero knowledge proofs.

2.3 Statement Agreement Protocol (SAP)

In this section, we explain how a client and server, mutually distrusted, can efficiently agree on a private statement, e.g. a string or tuple, that will be used to reclaim clients' masking coins when the private bubble bursts. Informally, there are two security properties that must be met. Firstly, neither party can prove to the verifier that they have agreed on an invalid statement (i.e. statement that both parties have not agreed). Secondly, either no party can successfully prove they have an agreement, or either party can prove it to the verifier. To that end, we use a combination of smart contract and commitment scheme. The idea is as follows. The server picks a random value and commitment to the agreed statement. It sends the commitment to the contract and the commitment opening (i.e. statement and the random value) to the client. The client checks if the opening matches the commitment and if so, it commits to the statement using the same random value and sends its commitment to the contract. Later on, for a party to prove to the contract that it has an agreed on the statement with the other party, it only sends the opening of the commitment. The contract checks if the opening matches both commitments and accepts if it matches.

1. **Setup**. Both parties agree on the SAP smart contract and deploy it.
2. **Agreement**.
   (a) The server picks a random value: r, and commits to the statement: $\mathtt{H}(x||r) = y_S$.
   (b) The server sends r to the client and sends y_S to the contract.
   (c) The client checks: $\mathtt{H}(x||r) \overset{?}{=} y_S$. If the equation holds, it computes $\mathtt{H}(x||r) = y_C$.

(d) The client stores y_C in the contract.

3. **Prove**. For either C or S to prove, it has agreement on s with its counter-party, it sends $\mu = (x, r)$ to the contract.
4. **Verify**. Given μ, the contract does the following.
   (a) checks if $\mathtt{H}(x||r) = y_C = y_S$.
   (b) outputs 1, if the above equation holds; otherwise, it outputs 0.

Strawman solutions: One may simply let each party sign the statement and send it to the other party, so later on each party can send both signatures to the contract who verifies them. However, this would not work, as the party who first receives the other parties signature may refuse to send its own signature, that prevents the other party to prove it has an agreed on the statement with its counter-party. Therefore, one may want to use a protocol for fair exchange of digital signature (or fair contract signing), such as [1, 3]. In this case, after both parties have the other party's signature, they can sign the statement themselves and send the two signatures to the contract; who first checks the validity of the signatures and then distribute the coins as constructed in the statement. However, this approach leads to two main efficiency and practical issues: (a) It imposes very high computation costs, as protocols for fair exchange of signature involve generic zero knowledge proofs and a high number of modular exponentiations. And (b) It is impractical, because protocols for fair exchange of signature protocol support only certain signature schemes (e.g. RSA, Rabin, Schnorr) not supported by the most predominant smart contract framework, Ethereum, as it only supports Elliptic Curve Digital Signature Algorithm (EDCSA).

2.4 Recurring Contingent Service Payment (R-CSP) Protocol

In this section, we provide our main protocol: R-CSP. At a high level the protocol works as follows. In the setup phase, right before the client uses the service, client and server agree on a statement that includes payment details (using SAP). Also they agree on a smart contract that specifies: (a) the total number of proofs/verifications, (b) the total amount of masked coins each party should deposit. They deploy the contract. Each party deposits its masked coins in the contract. If the deposit amount is less than what is stated in the contract, each party has a chance to withdraw its coins and terminate the contract. Next, to start using/providing the service, they run the `Service.setup`() and `Service.serve`() algorithm of the service scheme. If the server decides not to serve (e.g. it detect the client's misbehaviour when it runs the `Service.serve`() algorithm), sends 0 within a fixed time, then the parties can withdraw their deposit and terminate the contract. Otherwise, it sends 1 to the contract. In the proving phase, every time a server generates a proof of service (e.g. PoR) its sends the signed proof to the client who locally verifies the proof. If the verification is passed, then it knows the server has delivered the service honestly. However, if the proof is rejected (or never received) it waits until all proofs are provided and after the agreed time H elapses. Then, the client is given a time window to raise a dispute. During the dispute resolution phase, for each invalid proof, the client sends "dispute" message to the contract and it sends the proof and the verification key to an arbiter. The arbiter (given the proofs), runs `Service.resolve`() algorithm in Service protocol. It sends to the contract "accepted" or "rejected" message if `Service.resolve`() outputs 1 or 0 respectively. In the next phase, to distribute the coins, either client or server sends: (a) "pay" message, (b) the agreed statement, and (c) the statement's proof to the contract, who verifies the statement and if approved it distributes the coins according to the statement and the proofs status, i.e. whether any dispute has raised and if so the arbiter's input.

As stated earlier, if the client avoids sending any input after the setup phase, its coins will be transferred to the server. If the client lies about a proof's validity, its claim will be re-checked and it will be detected. Therefore, the client cannot waste the server resources and coins. Since, during the private time bubble (i.e. during service usage, verification time and the agreed time T) no plaintext proof is given to the contract and 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 deposited coins are masked and the agreed statement is kept private, during the private time bubble nothing about the detail of the service used by the client is leaked. This preserves the client's privacy. Furthermore, 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 and will be distributed among both parties fairly, i.e. as stated in the statement and according to the proofs' status. Also, if the server sends an invalid proof or the client raise tries to frame the server, then the misbehaving party will pay the arbiter cost.

1. **Setup**.
   (a) C constructs a vector $\vec{v}$, initially empty.
   (b) C and S agree on the number of the service's proofs/verifications: z, the amount of coins for each accepting proof: a, and the amount of coins to cover the cost of each potential dispute' resolution: b. Also, they agree on the amount of masking coins the client and server will use: e and f respectively.
   (c) C and S construct a statement of the form: $x = (a, b, e, f, z)$.
   (d) For C and S to provably agree on x, they take the steps in the Setup and Agreement phases in the SAP, at time T_0. Let μ be the statement's proof.
   (e) S picks a random key k and sends it to C.
   (f) For C and S to provably agree on k, they take the steps in the Setup and Agreement phases in the SAP again, at time T_1. Let μ' be the proof.
   (g) C and S agree on a smart contract: SC, that specifies the total number of the service's proofs: z and total amount of masked coins each party should deposit, i.e. the client and server should deposit $p = z(a + b) + e$ and $q = zb + f$ coins, respectively. They sign and deploy the contract.
   (h) C and S deposit their masked coins in SC at time T_2.
   (i) At time T_3, if the deposit amount either party has put is less than the mount stated in SC, then SC allows each party to withdraw its coins and terminate the contract.
   (j) C runs `Service.setup`$()$ and accordingly the server runs `Service.serve`$()$ algorithm in the service scheme.
   (k) S sends the output of `Service.serve`$()$ to SC at time T_4.
   (l) C and S can withdraw their coins at time T_5, if the server sends 0 or nothing.
2. **Billing-cycles**. At each time G_j, (where $1 \leq j \leq z$ and $G_1 > T_5$), that a proof is generated, the parties perform as follows.
   (a) C calls `Service.genChall`$()$ to generate a set of challenges and sends them to SC.
   (b) S calls `Service.prove`$()$, given the challenges, to generate j-th proof: π_j
   (c) S sends the proof's encryption: $w_j = \texttt{Enc}(k, \pi_j)$ to SC.
   (d) C fetches w_j from SC and decrypts it: $\texttt{Dec}(k, w_j) = \pi_j$
   (e) C runs `Service.verify`$()$ to check the validity of π_j
   (f) C appends j to $\vec{v}$, if π_j is rejected.
3. **Dispute Resolution**. The phase takes place only in case of dispute.
   (a) C at time $K_1 > G_z + H$ sends "dispute" message to the contract.
   (b) C invokes the arbiter and sends $(\vec{v}, k, \mu')$ to it.
   (c) The Arbiter checks the validity of k by sending k and μ' to SAP contract which returns either 1 or 0. The arbiter proceeds to the next step if the output is 1; otherwise, it does nothing.
   (d) The Arbiter decrypts those proofs whose index are in $\vec{v}$. In particular, $\forall i \in \vec{v} : \texttt{Dec}(k, w_i) = \pi_i$
   (e) The Arbiter for every π_i, where $i \in \vec{v}$, invokes `Service.resolve`$()$ to resolve the dispute. Let u and y be total number of times `Service.resolve`$()$ returns 1 and 0 respectively.
   (f) The Arbiter sends u and y to SC at time K_2.
4. **Coin Transfer**.
   (a) Either C or S send "pay" message, the statement: x and the statement proof: μ to SC at time $L > K_2$.
   (b) SC checks the validity of the statement by sending x and μ to SAP contract which returns either 1 or 0. SC only proceeds to the next step if the output is 1.
   (c) SC distributes the coins to the parties as follows:
      - $p - yb - a(z - u)$ coins to C.
      - $q - ub + a(z - u)$ coins to S.
      - $b(y + u)$ coins to the arbiter.

Remark 1. The server schemed defined in Section does not (need to) support the privacy of the proofs. However, in the main protocol above 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 verifiable). This is the reason the encrypted proofs are sent to the contract.

References

1. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000
2. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS'17
3. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02