Chained RSA Time-lock Puzzle and its Applications

Abstract. Time-lock puzzles are vital cryptographic protocols with numerous practical applications. They enable a party to lock a message such that anyone else cannot unlock it until a certain time elapses. We propose the idea of composing instances of a puzzle scheme, where given instances' composition at once, a server can find one puzzle's solution after another, on time without the need to deal with all of them simultaneously. We propose a candidate construction: "*chained RSA time-lock puzzle*" (CR-TLP). It allows a client to create and send to the server z puzzles at once, while the server can solve every puzzle sequentially, without having to run parallel computations on them. Cost-wise CR-TLP is efficient, its overall computation complexity of solving z puzzles is equivalent to that of solving only the last puzzle. It is also accompanied by an efficient verification algorithm, publicly verifiable. Moreover, we utilise CR-TLP, as a key-escrow, to build an efficient outsourced proofs of retrievability scheme that supports *real-time detection* and *fair payment* while having lower overheads than the state of the art.

1 Introduction

Time-lock puzzles are elegant cryptographic primitives that allow sending information to the future. They enable a party to lock a message such that, no one else can unlock it until a certain time has passed, although it may have many computing resources running in parallel[1]. They have a wide range of real-world applications, such as e-voting [14], fair contract signing [10], and sealed-bid auctions [41]. Over the last two decades, different variants of time-lock puzzles have been proposed.

Nevertheless, all existing puzzle schemes fall short of efficiently dealing with the "*parallel composition problem*", where at once multiple puzzle instances are given to a server to solve. It is a natural generalisation of the single puzzle setting. Specifically, the existing schemes do not offer to a server any efficient remedy for the generic setting where a client generates and sends multiple puzzles at once to a server who should find one puzzle's solution after another. Application areas include, but not limited to: (a) key-escrow scheme, where a client encodes its random keys into puzzles and wants a server to learn each key at distinct points in time, or (b) sealed-bid auctions, where a client encodes its bids into puzzles and submits its bids to different auctions run by a server who should be able an open client's related bid. If the existing puzzle schemes are utilised directly in the (above) multiple puzzles setting, then the server has to independently deal with each puzzle that ultimately yields a significant computation overhead and demands a high level of parallelisation. To date, there is no solution that mitigates the aforementioned problem.

[1] There exist protocols that use an assistance of a third party to support time-release of a secret. This protocols' category is not our focus in this paper.

In this paper, we present "*chained RSA time-lock puzzle*" (CR-TLP), the first time-lock puzzle that addresses the parallel composition problem. It allows a client to create and send to the server z puzzles at once, while the server can solve every puzzle sequentially, without having to run parallel computations on them. Moreover, CR-TLP is equipped with an efficient verification algorithm that allows anyone to check the correctness of a solution found by the server, i.e. publicly verifiable. It can be built in random oracle or standard models, while retaining its overall efficiency. Solving z puzzles in CR-TLP incurs a computation complexity linear with z, i.e. $O(Tz)$ where T is a single puzzle's time parameter. The same procedure also imposes a communication overhead linear with z, i.e. $O(z)$. Furthermore, we apply CR-TLP to "outsourced proofs of retrievability" research line and make a set of contributions in that area, as well.

The key observation that led us to the design of CR-TLP is that when existing time-lock puzzles are used naively in the multi-puzzle setting, the process of solving puzzles (in parallel) has many overlaps that yield a high computation overhead. By eliminating the overlaps, we can considerably lower the overall cost. To attain that goal, in CR-TLP, a client creates puzzles *outside-in* while they can be solved only *inside-out*. In the sense that the client iteratively creates a puzzle for the solution revealed after the rest, and integrates the information needed for solving it into the puzzle for the solution revealed earlier. For the server to find the puzzles' solution, it begins to find a solution revealed earlier than the rest. The found solution gives it enough information to find the next puzzle's solution, after a certain time. This process proceeds until the last solution is discovered. Also, we use the following novel idea to let CR-TLP have a verification algorithm publicly verifiable. The client, when creating a puzzle for a solution, commits to the solution and publishes the commitment. It combines the solution with the commitment opening and creates a puzzle on the combination. Later on, when the server solves the puzzle (unlike traditional commitment schemes in which the committer is the prover), it plays the role of prover and opens the related commitment to the public.

Outsourced Proofs of Retrievability and CR-TLP. Proofs of retrievability (PoR) schemes provide a strong guarantee to a client that its data stored on a cloud server (potentially malicious) can be fully accessed when required. Recently, researchers developed *outsourced* PoR schemes that enable clients to delegate the verification phase to a third-party auditor, potentially malicious. Nevertheless, the existing outsourced PoRs have a set of shortcomings, e.g. lack of real-time detection, lack of fair payment mechanism, or imposing high costs. In this paper, we present "*smarter outsourced proofs of retrievability*" (SO-PoR) scheme, the first efficient outsourced PoR that addresses the above issues and offers a combination of unique appealing features. It allows a client to delegated PoR verification, meanwhile efficiently detecting the server misbehaviour in (almost) *real-time* without the need to re-execute the verification itself. Supporting real-time detection makes SO-PoR suitable for mission-critical data. It supports a *fair payment*, meaning that the server gets paid only if it provides accepting proofs. Cost-wise, SO-PoR outperforms the state of the art. In particular, SO-PoR verification and store phases impose $\frac{1}{4.5}$ and $\frac{1}{46\times10^5}$ of computation costs imposed by the same phases in the fastest outsourced PoR (when the number of delegated verification is 100 and file size is 1-GB). Also, SO-PoR has significantly lower communication cost than the fastest outsourced PoR. Also, the computation cost of verification and prove in SO-PoR is identical to the costs in the well-known *privately* verifiable PoR scheme, while both have $O(1)$ proof size complexity. In SO-PoR, a smart contract performs verifications using message authenticated codes (MAC). Since MACs are only privately verifiable and smart contracts do not maintain a private state, SO-PoR also utilises CR-TLP, as a key-escrow, to efficiently address the privacy issue.

2 Related Work

2.1 Time-lock Puzzles

The idea to send information into the *future*, i.e. time-lock puzzle/encryption, was first put forth by Timothy C. May. A time-lock puzzle allows a party to encrypt a message such that it cannot be decrypted until a certain amount of time has passed. In general, a time-lock scheme should allow that generating (and verifying) a puzzle to take less time than solving it. The time-lock puzzle scheme that May proposed lies on a trusted agent. Later on, Rivest *et al* [41] propose a protocol that does not require a trusted agent, and is secure against a receiver who may have access to many computation resources that can be run in parallel. It is based on Blum-Blum-Shub pseudorandom number generator that relies on modular repeated squaring, believed to be sequential. The scheme in [41] allows (only) the puzzle creator to verify the correctness of the puzzle solution using a secret key and the original secret message. This scheme has been the core of (almost) all later time-lock puzzles schemes that supports encapsulation of an arbitrary message. Later on, [10, 19] proposed timed commitment schemes that offer more security properties, in the sense that they allow a puzzle generator to prove (in Zero-knowledge) to a puzzle solver that the correct solution (e.g. a signature of a public document) will be recovered after a certain time, before the solver starts solving the puzzle. These schemes are more complex, due to the use of zero-knowledge proofs, and less efficient than [41]. Very recently, [37, 11] propose protocols for homomorphic time-lock puzzles, where an arbitrary function can be run over puzzles before they are solved. The schemes mainly use fully homomorphic encryption and the RSA puzzle, proposed in [11], in the nutshell. The main difference between the two protocols is the security assumption they rely on (i.e. the former uses a non-standard assumption while the latter relies on a standard one). Since both schemes use a generic fully homomorphic encryption, it is not hard to make them publicly verifiable. Both protocols are only of theoretical interest as in practice they impose high computation and communication costs, due to the use of fully homomorphic encryptions.

We also cover two related but different notions, pricing puzzles and verifiable delay functions.

Pricing Puzzles. Also known as *client puzzles*. It was first put forth by Dwork *et al.* [15] who defined it as a function that requires a certain amount of computation resources to solve a puzzle. In general, the pricing puzzles are based on either hash inversion problems or number theoretic. In the former category, a puzzle generator generates a puzzle as: $h = \mathtt{H}(m||r)$, where $\mathtt{H}$ is a hash function, m is a public value and r is a random value of a fixed size. Given $h, \mathtt{H}$ and m, the solver must find r such that the above equation holds. The size of r is picked in such a way that the expected time to find the solution is fixed (however it does not rule out finding the solution on the first

attempt). The above hash-based scheme allows a solver to find a solution faster if it has more computational power resources running in parallel. The application area of such puzzle includes defending against denial-of-service (DoS) attacks, reaching a consensus in cryptocurrencies, etc. A variant of such a puzzle uses iterative hashing; for instance, to generate a set of puzzles in the case where the solver receives a service proportional to the number of puzzles it solves [22], or to generate password puzzle to mitigate DoS attacks [35]. However, the iterative hashing schemes are partially parallelizable, in the sense that each single invocation of the hash function can be run in parallel. Later on, [36] investigates the possibility of constructing (time-lock) puzzles in the random oracle model. Their main result was negative, that rules out time-lock puzzles that require more parallel time to solve than the total work required to generate. Also [36] proposes an iterative hash-based mechanism (very similar to [35]) that allows a puzzle generator to generate a puzzle with n parallel queries to the random oracle, but the solver needs n rounds of serial queries. Nevertheless, this scheme is also partially parallelizable, as each instance of the puzzle can be solved in parallel. Note that the above hash-based puzzle schemes would have very limited applications if they are used directly to encapsulate a message: m' of arbitrary size. The reason is that, in these schemes, the solution size: $|r|$ plays a vital role in (adjusting) the time taken to solve the puzzle. If the solution size becomes bigger, as a result of combining r with m', i.e. $r \odot m'$, then it would take longer to find the solution. This means the puzzles can be used only in the cases where the time required to find a solution is long enough, and is a function of $|r \odot m'|$, which seriously restricts its application. Researchers also propose non-parallelizable pricing puzzles based on number theoretic [48, 32, 26] whose main application is to resist DoS attacks. These schemes have a more efficient verification mechanism than the one proposed in [41]. But, they are only privately verifiable and not designed to encapsulate an arbitrary message.
Verifiable Delay Function (VDF). Allows a prover to provide a publicly verifiable proof stating it has performed a pre-determined number of sequential computations. It has many applications, e.g. in decentralised systems to extract trustworthy public randomness from a blockchain. VDF first formalised by Boneh *et al* in [9] that proposed several VDF constructions based on SNARKs along with either incrementally verifiable computation or injective polynomials, or based on time-lock puzzles, where the SNARKs based approaches require a trusted setup. Later on, [49] improved the pervious VDFs from different perspectives and proposed a scheme based on RSA time-lock encryption, in the random oracle model. To date, this protocol is the most efficient VDF. It also supports batch verification, such that given a single proof a verifier can efficiently check the validity of multiple outputs of the verifiable delay function. As discussed above, (most of) VDF schemes are built upon time-lock puzzles, however the converse is not necessarily the case, as VDFs are not designed to encapsulate an arbitrary private message, and they take a public message as input while time-lock puzzles are designed to conceal a private input message.

2.2 Proof of Storage

Traditional Proof of Storage Proof of storage (PoS) is a cryptographic protocol that allows a client to efficiently verify the integrity or availability of its data stored in a remote server, not necessarily trusted [25]. PoS can be categorised into two broad classes: Proofs of Retrievability (PoR) and Proofs of Data Possession (PDP). The main difference between the two categories is the level of security assurance provided. PoR schemes guarantee that the server maintains knowledge of *all* of the client's outsourced data, while PDP protocols only ensure that the server is storing *most* of the client data. From a technical point of view, the main difference in most prior PDP and PoR constructions is that PoR schemes store a redundant encoding of the client data on the server by employing an error-correcting code, e.g. Reed-Solomon codes, while such encoding is not utilised in PDP schemes. Hence, PoR schemes provide stronger security guarantees compared to PDP at the cost of additional storage space and encoding/decoding, (for more details see [31, 46, 13]). Furthermore, each PoR and PDP scheme can be also grouped into two categories; namely, publicly and privately verifiable. In a publicly verifiable scheme, everyone without knowing a secret can verify proof, whereas a verifier in a privately verifiable scheme requires the knowledge of a secret.

The notion of PoR first was introduced and defined in [24], where the authors designed a protocol that utilises random sentinels, symmetric key encryption, error-correcting code and pseudorandom permutation. In this protocol, a client in the setup phase, applies error-correcting code to every file block and encrypts each encoded block. Then, it computes a set of random sentinels and appends them to the encrypted file. It permutes all values (i.e. sentinels and encrypted file blocks) and sends them to the cloud server. To check if the server has retained the file, the client specifies random positions of some sentinels in the encoded file and asks the server to return those sentinel values. Next, the client checks if it gets the sentinels it asked for. In this scheme, the security holds, as the server cannot feasibly distinguish between sentinels and the actual file blocks and the sentinels have been distributed uniformly among the file blocks. But, as individual sentinel is only one-time verifiable, there is an upper bound on the number of verifications performed by the client, and when reached, the client has to re-encode the file. To overcome the issues related to the bounded number of verifications, the authors have also suggested that sentinels can be replaced with MAC on every file block, or a Merkle tree constructed on the file blocks. This protocol is computationally efficient and its communication cost is linear with the number of challenges sent[2]. The sentinel and MAC-based schemes above are privately verifiable while the one uses Merkle tree supports public verifiability. However, the publicly verifiable Merkle tree based approach has a communication cost logarithmic with the file size and the prover has to send a set of file blocks to the verifier that yields a high communication cost.

Later on, [44] improves the previous sentinel-based scheme and definition, and proposes two PoR protocols (with an unlimited number of verifications), one utilises MAC and the other one relies on BLS signatures. In particular, a client at setup phase, encodes its file blocks using error-correcting code and then for each encoded file block it generates a tag that can be either a MAC or BLS signature of that block. In the verification phase, it specifies a set of random indices corresponding to the file's blocks; and accordingly the server sends a proof to it. These two schemes have a low commu-

[2] It is not hard to make the communication cost of this scheme constant; for instance, by letting the server order the challenged sentinels, concatenate them and send a hash of the concatenated value to the client.

nication cost, as due to the homomorphic property of the tags, the prover can aggregate proofs into a single authenticator value and no file blocks are sent to the prover. Also, the protocol based on MAC supports efficient private verification. Nevertheless, the one that uses BLS signature supports public verifiability at the cost of public key operations and it is far less efficient than the MAC-based one. Within the last decade, researchers have extended PoR protocols from several perspectives, e.g. those that support: efficient update [46], delegation of file pre-processing [2], or delegation of verification [3].

On the other hand, PDP was first introduced in [4]. PDP protocols focus only on verifying the integrity of outsourced data. In this setting, clients can ensure that a certain percentage of data blocks are available. The authors in [4] propose two schemes, for public and private verification both of which utilise RSA-based homomorphic verifiable tags generated for each file block and use the spot-checking techniques (similar to [44]) to check a random subset of a file's blocks. In both schemes, the proof size is constant, while the verification cost is high as it requires many exponentiations over an RSA ring. Later on, an efficient and scalable PDP scheme that supports a limited number of verifications is proposed in [5]. The scheme supports only privately verifiable PDP, and is based on a combination of pre-computation technique and symmetric key primitives. Ever since, researcher proposed different variants of PDP, e.g. dynamic PDP [16], multi-replica PDP[17], keyword-based PDP [43].

Thus, (a) in publicly verifiable PoS it is assumed the verifier is fully trusted with the verification correctness, (b) these schemes either have a very high verification cost when the proof size is constant (when signature-based tags are used) or have a communication cost logarithmic with the entire file size if their verification is efficient (when a Merkle tree is utilised), and (c) the privately verifiable proofs can have a constant proof size and efficient verification algoeithm, but the data owner has to perform the verification.

Outsourced PoR Recently, [3, 51] propose *outsourced* PoR protocols that allow clients to outsource the verification to a third party auditor not necessarily trusted. The scheme in [3] uses MAC-based tags, zero-knowledge proofs and error-correcting codes. At a high level, the protocol works as follows. At the setup, the client encodes its data using error-correcting codes, generates MAC on every file block, and stores the encoded file and tags on the server. Then, the auditor downloads the encoded file, generates another set of MAC's on the file blocks. It uploads the MAC's to the cloud. Also, the auditor proves to the client in zero-knowledge that it has created each MAC correctly. To do that, it uses a non-interactive zero-knowledge proof in the random oracle model. If the client accepts all zero-knowledge proofs, it sings every proof and sends the signatures back to the auditor. In the verification phase, the auditor sends two sets of random challenges extracted from a blockchain. Upon receiving the challenges, the server provides two separate proofs, one for the auditor and the other one for the client. The auditor verifies the proof generated for it and locally stores the clients' proofs. In the case where the auditor's proof is not accepted, an honest auditor would inform the client who will come online and checks both its proofs and the auditors' proof. The scheme provides two layers of verification to the client: *CheckLog* and *ProveLog*. *CheckLog* is more efficient than the other one, as the auditor sends much fewer challenges to the server to generate the client's proof for each verification. In the case where the client's check fails, the client will assume that either the server or auditor has acted maliciously, and to pinpoint the malicious party, it needs to proceed to *ProveLog* where allows the client to audit the auditor. In this verification, the auditor must reveal its secret keys with which the client checks all the proofs provided by the server to the auditor for the entire period that the client was offline. In this protocol, the verification phase is efficient (due to the use of MACs). In particular, the verification's computation cost for the auditor and client is linear with the number of challenges and their communication cost is constant in the file size.

Although the protocol in [3] is appealing, it has several shortcomings: (a) auditor can get a free ride: in the case where a known highly reputable server, e.g. Google/Amazon, always generates accepting proofs for both client and auditor, an economically rational auditor can skip performing its part (e.g. to save computation cost) and get still paid by the client. This deviation from the protocol can never be detected by the client in the protocol. (b) no guarantee for a real-time detection/notification: the client may not be notified in the real-time about the data unavailability if the auditor is malicious; therefore, this scheme is not suitable for the case where a client's involvement for the data extraction is immediately needed once a misbehaviour is detected, e.g., in the case of hardware depreciation. (c) a high cost of auditor onboarding: revoking an auditor and onboarding a new one, imposes a high cost on the client and new auditor, as new auditor need to re-run the setup phase (that includes data downloading, zero-knowledge proofs) and the client needs to verify and sign the zero-knowledge proofs. (d) *ProveLog* costs even an honest auditor: the only way for the client to ensure the auditor has fully followed the protocol, is to run *ProveLog* that requires the auditor to reveal its secret values. After that, an honest auditor has to re-run the computationally expensive setup again. Since the auditor is not trusted and no guarantee that it alerts the client as soon as the server's misbehaviour is detected, its involvement in the protocol seems unnecessary; and the whole scheme can be replaced with a much simpler one: the client, similar to a standard privately verifiable PoR, encodes its data and stores it in the server. Then, for each verification time, the server gets the random challenges from the blockchain and publishes proofs in a bulletin board (to timestamp it). When the client comes online, it checks the proofs and accordingly takes the required action, e.g. pays the server, tries to recover/ the file.

Xu et al. in [51] propose a publicly verifiable PoR to improve the computation cost at setup. The scheme uses BLS signatures-like tags, polynomial arithmetics, and polynomial commitments; unlike previous schemes, each tag has a more complex structure. In this protocol, an auditor is assumed to be fully trusted during the verifications. Later on, however, when it is revoked by the client it may become malicious, i.e. may collude with the server and reveal its secret. This scheme allows a client to update its tags when a new auditor joins. To do that, the client needs to download all the blocks' tags, refresh them locally and uploaded the fresh tags to the server. The verification for auditor and client involves public key operations and is more expensive than the one in [3]. The use of an auditor's revocation remains unclear in the paper, as auditors are assumed to be honest in this protocol. A delegatable PDP is proposed in [45] to ensure only authorised parties can verify the integrity of remote data. However, the delegated party in this scheme is fully trusted with the correctness of verification it performs. The scheme uses authenticator tags similar to BLS signatures based ones.

Hence, the existing outsourced PoR protocols either suffer from several security issues and guarantee no real-time detection or have to fully trust verifiers with the verification correctness.

Distributed PoR using a (Tailored) Blockchain Distributed PoR allows data to be distributed to numerous storage servers to achieve robustness, and address a single point of failure issue. Permacoin [38] is one of those that distribute data among customised blockchain nodes and repurposes mining resources of Bitcoin blockchain miners. In Permacoin, each miner needs to prove that it has a portion of the file and to do so it provides proof of data retrievability verified by other miners. The miner, in this scheme needs to invest on both computation resources and storage resources (to generate proof of retrievability). The protocol uses a Merkle tree built on top of file blocks to support publicly verifiable PoR. The mining procedure in this protocol is based on iterative hashing. In Permacoin, in each verification, the prover has to send both challenged file blocks and the proof path in the tree, that imposes communication cost logarithmic to the size of the *entire original file*. In this scheme, an accepting proof only indicates a portion of the data holding by that miner (prover) is retrievable. Thus, for a data owner to have a guarantee that the entire data is retrievable, it has to either wait for a long period of time (depending on the file size and the number of miners) or hope that there are enough active miners in each epoch. Also, since the miners are both verifiers and provers (storage providers), it is assumed that the storage providers are economically rational (which is weaker than a malicious one).

Filecoin and KopperCoin in [33, 28] respectively, offer similar features, i.e. repurposing Bitcoin and supporting distributed PoR. However, Filecoin, in addition to a Merkle tree, utilises generic zero-knowledge proofs (i.e. zk-SNARKS) that result in a high overall computation cost and requires a trusted party to generate the system parameters. Filecoin uses proof of work as well as PoR. On the other hand, KopperCoin uses BLS signature-based publicly verifiable PoR that has a constant communication cost but has a high computation cost. Unlike Filecoin, KopperCoin does not use a PoW.

In the same line of research, [29] proposes a customised blockchain that supports distributed PoR as well as preserving the privacy of on-chain payments between storage users and providers. The protocol, however is computationally expensive as it uses publicly verifiable tags based on BLS signatures for PoR and ring signatures for the privacy-preserving payments. Likewise, [42] proposes a high-level distributed PoR framework whose aim is efficient utilization of storage resources offered by storage providers. It also uses BLS signatures and a Merkle tree along with a smart contract for payments. Similarly, [53, 52] propose schemes that allow a user to store its encrypted data in the blockchain. The scheme uses a Merkle tree for PoR and smart contract to transfer fees to the blockchain nodes storing the data. In these two schemes, the data owner is the party who sends random challenges to storage nodes, so it has to be online when a verification is needed. Storj [50] also falls in this category where there is a tailored blockchain comprising a set of storage nodes that store a part of data and provide proofs when they are challenged. In Storj, there are trusted nodes, Satellite, who do the verifications on behalf of the clients. The scheme utilises a Merkle tree-based PoR.

So, existing distributed PoR protocols have either a large proof size or high verification cost. Also, they do not guarantee that the *entire* file is retrievable in *real-time*.

Verifying Remote (off-chain) data via a Blockchain To relax the assumption that an auditor is fully trusted with the correctness of verification in the publicly verifiable PoR schemes while storing data off-chain, e.g. in a cloud, researchers proposed numerous protocols, e.g. [40, 23, 55, 18, 7, 47], that delegate the verification procedure to blockchain nodes. The high-level framework proposed in [40] requires only a hash of the entire file is stored in a smart contract, where when later on the user access the file, it computes the hash of the file and compares it with the one stored in the contract. In this scheme, the entire file has to be accessed by the user for each verification which is what exactly PoR schemes avoid doing. The protocol in [23] uses BLS signature-based tags and Merkle tree where the tags are broadcast to all blockchain nodes. However, surprisingly, the protocol assumes the cloud server is fully trusted and stores data safely, which raises the question that *"why is a data verification needed in the first place?"*. In this protocol, the tags are generated by the cloud who sends them to the nodes. The tags are never checked against the outsourced data. So, the only security guarantee [23] offers is the immutability of tags. The high-level scheme proposed in [55] allows a client to pay the storage server in a fair manner. The scheme uses a blockchain for payment and Merkle tree for PoR. In this protocol, the client has to be online and send random challenges to the server for every verification.

Audita [18] utilises an augmented blockchain and RSA signature-based PDP [4] to achieve its goal. In this scheme miners, for each epoch, pick a dealer who sends a challenge to the storage node(s) to get proofs of data possession. Similar to Permacoin [38], Audita substitutes proof of work with PDP, so if a proof is accepted a new block is added to the chain. But, in Audita every miner carries out expensive public key based verifications. The protocol proposed in [7], similar to [3], uses a third party auditor. It mainly utilises, a smart contract and BLS signature-based tags. In this protocol, unlike [3], when the auditor raises a dispute during the verification it calls a smart contract who performs the verification again to detect a misbehaving party, i.e. the server or auditor. The protocol is computationally more expensive than [3] and inherits the same issues, i.e. issues (a-c) stated in Section 2.2. Sia [47] is a mechanism in which a data owner distributes its data among off-chain storage servers who periodically provide a PoR to a smart contract signed between the data owner and the servers. Each server gets paid if its proof is accepted by the contract. In this scheme, a Merkle tree-based PoR is used.

As evident, the schemes designed for blockchain-based verification of data stored in a storage server either require clients to access whole outsourced data for every verification, or impose a high communication/computation cost, or clients have to be online for each verification.

Fair Exchange of Digital Services Campanelli *et al.* [12] propose a scheme that allows different parties to exchange digital services (and goods) over Bitcoin blockchain. This scheme, for instance in PoR context, allows the storage provider to get paid if and only if the data owner receives an accepting proof. It has two variants, publicly and privately verifiable both of which use a smart contract. In addition, in the privately verifiable variant, a MAC-based tags and generic secure multi-party computation are used, e.g. Yao's garbled circuit [54], while in the publicly verifiable one BLS signature tags and zk-SNARK are utilised.

Nevertheless, this scheme assumes that either the client is available and online when PoR is provided (in privately verifiable variant) or the third party, acting on a client's behalf, performs PoR verification honestly (in publicly verifiable one).

3 Preliminaries

3.1 Smart Contract

Cryptocurrencies, such as Bitcoin and Ethereum, in addition to offering a decentralised currency, support computations on transactions. In this setting, often a certain computation logic is encoded in a computer program, called *"smart contract"*. To date, Ethereum is the most predominant cryptocurrency framework that enables users to define arbitrary smart contracts, due to the support of a Turing-complete language. In this framework, contract code is stored on the blockchain and executed by all parties (i.e. miners) maintaining the cryptocurrency, when the program inputs are provided by transactions. The correctness of the program execution is often guaranteed by the security of the underlying blockchain components (e.g. consensus, digital signature). To prevent a denial of service attack, the framework requires a transaction creator to pay a fee, called *"gas"*, depending on the complexity of the contract running on it.

Nonetheless, Ethereum smart contracts suffer from an important issue; namely, the *lack of privacy*, as it requires every contract's data to be public, which is a major impediment to the broad adoption of smart contracts when a certain level of privacy is desired. To address the issue, researchers/users may either (a) utilise existing decentralised frameworks which support privacy-preserving smart contracts, e.g. [30]. But, due to the use of generic and computationally expensive cryptographic tools, they impose a significant cost to their users. Or (b) design efficient tailored cryptographic protocols that preserve (contracts) data privacy, even though non-private smart contracts are used. We take the latter approach in this work.

3.2 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: m as $\texttt{Com}(m, d) = h$, that involves a secret value: d. At the end of the commit phase, the commitment: h is sent to the receiver. In the open phase, the sender sends the opening: $\rho = (m, d)$ to the receiver who verifies its correctness: $\texttt{Ver}(h, \rho) \stackrel{?}{=} 1$ and accepts if the output is 1. A commitment scheme must satisfy two properties: (a) *hiding*: infeasible for an adversary (i.e. the receiver) to learn any information about the committed message: m, until the commitment: h is opened, and (b) *binding*: infeasible for an adversary (i.e. the sender) to open a commitment: h to different values: $\rho' = (m', d')$ than that used in the commit phase, i.e. infeasible to find ρ', *s.t.* $\texttt{Ver}(h, \rho) = \texttt{Ver}(h, \rho') = 1$, where $\rho \neq \rho'$. There exist efficient non-interactive commitment schemes both in (a) the random oracle model using the well-known hash-based scheme such that $\texttt{Com}(m, d)$ involves computing: $\texttt{H}(m||d) = h$ and $\texttt{Ver}(h, \rho)$ requires checking: $\texttt{H}(m||d) \overset{?}{=} h$, where $\texttt{H}$ is a hash function, and (b) the standard model, e.g. Pedersen scheme [39].

3.3 Pseudorandom Function

Informally, a pseudorandom function ($\texttt{PRF}$) is a deterministic function that takes a key and an input; and outputs a value indistinguishable from that of a truly random function with the same input. Pseudorandom functions have many applications in cryptography as they provide an efficient and deterministic way to turn an input into a value that looks random. A $\texttt{PRF}$ is formally defined as follows [27].

Definition 1. *Let $W : \{0,1\}^{\psi} \times \{0,1\}^{\eta} \rightarrow \{0,1\}^{\iota}$ be an efficient keyed function. It is said W is a pseudorandom function if for all probabilistic polynomial-time distinguishers B, there is a negligible function, $\mu(.)$, such that:*

$$\left| Pr[B^{W_k(\cdot)}(1^{\psi}) = 1] - Pr[B^{w(\cdot)}(1^{\psi}) = 1] \right| \leq \mu(\psi),$$

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

In practice, we are interested in a pseudorandom function that can be efficiently built on smart contracts given the tools and primitives that a smart contract framework (e.g. Ethereum) offers. HMAC [8] satisfies the requirements above.

3.4 RSA Time-lock Puzzle

In this section, we explain the RSA time-lock puzzle (R-TLP) proposed in [41]. Its formal definition and proof are presented in Appendix A.

1. **Setup**: $\texttt{R-TLP.Setup}(1^{\lambda}, \varDelta)$
   (a) Compute the product of two large randomly chosen prime numbers: $N = q_1 q_2$, and then compute Euler's totient function of N, as: $\phi(N) = (q_1 - 1)(q_2 - 1)$
   (b) Set $T = S\varDelta$ as the total number of squaring needed to decrypt an encrypted message m, where $\varDelta$ is the period (in seconds) within which the message should remain private and S is the maximum number of squaring modulo N per second that can be performed by a solver.
   (c) Choose a random key: k for a semantically secure symmetric key encryption that has three algorithms: $(\texttt{GenKey}, \texttt{Enc}, \texttt{Dec})$.
   (d) Pick a uniformly random value r from $\mathbb{Z}_N^*$.
   (e) Compute $a = 2^T \bmod \phi(N)$.
   (f) Set $pk = (N, T, r)$ as public key and set $sk = (q_1, q_2, a, k)$ as secret key.
2. **Generate Puzzle**: $\texttt{R-TLP.GenPuZ}(m, pk, sk)$
   (a) Encrypt the message using the symmetric key encryption: $\theta_1 = \texttt{Enc}(k, m)$
   (b) Encrypt the key: k, as: $\theta_2 = k + r^a \bmod N$

   (c) Sets: $\theta = (\theta_1, \theta_2)$ as ciphertext or puzzle. Next, output θ.
3. **Solve Puzzle**: $\texttt{R-TLP.SolvPuz}(pk, \theta)$
   (a) Find b, where $b = r^{2^T} \bmod N$, by using T number of squaring r modulo N.
   (b) Decrypt the key's ciphertext: $k = \theta_2 - b \bmod N$
   (c) Decrypt the message's ciphertext: $m = \texttt{Dec}(k, \theta_1)$. Then, output m.

Informally, a time-lock puzzle's security relies on the hardness of factoring problem, the security of the symmetric key encryption, and sequential squaring assumption.

3.5 Notations

We summarise our notation in Table 1.

Table 1: Notation Table.

Setting	Symbol	Description
Generic	z	Number of puzzles or delegated verifications
	h, h_j	Hash values
	d, d_j	Randomness of commitment
	n	Number of file blocks
	m, m_j	Plaintext messages
	$\rho : (m_j, d_j)$	Commitment opening
	$\texttt{H}$	Hash function
SO-PoR	$\texttt{PRF}$	Pseudorandom function
	$\hat{k}, v_j, l_j$	$\texttt{PRF}$'s keys
	ι	Security parameter, $\iota = 128$-bit
	p	Large prime number, $\lvert p\rvert = \iota$
	w	Blockchain block index
	g	Blockchain security parameter: chain quality
	λ'	Blockchain generic security parameter
	$\vec{\boldsymbol{F}}$	Outsourced encoded file
	F_j	A file block
	$\lvert\vec{\boldsymbol{F}}\rvert$	Number of file blocks, $\lvert\vec{\boldsymbol{F}}\rvert = n$
	$\lvert\lvert\vec{\boldsymbol{F}}\rvert\rvert$	File bit-size
	σ_i	Permanent tag
SO-PoR	$\sigma_{b,j}$	Disposable tag
	$\alpha, \alpha_j, r_i, r_{b,j}$	Pseudorandom values
	c	Number of challenges
	$\mathcal{B}_j$	Blockchain's j^{th} block
	(μ_j, ξ_j)	j^{th} PoR
	$\varDelta_1$	Time taken to generate a PoR
	$\varDelta_2$	Time taken a contract gets a message
	e	Coins paid for an accepting PoR
CR-TLP	λ	RSA security parameter
	$\varDelta$	Time win. message remains hidden
	S	Max. squaring done per sec.
	T	$T = S\varDelta$
	s_j	j^{th} solution
	θ_j	j^{th} puzzle, $\theta_j : (\theta_{j,1}, \theta_{j,2})$
	f_j	Time when j^{th} solution is found
	k, k_j	Sym. key encryption keys
	pk, sk	Public and secret keys
	q_1, q_2	Large prime numbers
	N	RSA modulus, $N = q_1 q_2$

4 Chained RSA Time-lock Puzzle (CR-TLP)

4.1 Strawman Solution

In the following, we elaborate on the problems that would arise when an existing time-lock puzzle is used directly to handle multiple puzzles at once. Without loss of generality, to illustrate the problems, we use the well-known R-TLP scheme presented in Section 3.4.

Consider the case where a client wants a server to learn a vector of messages: $\vec{\boldsymbol{m}} = [m_1, ..., m_z]$ at times $[f_1, ..., f_z]$ respectively, where the client is available and online only at an earlier time $f_0 < f_1$. For the sake of simplicity, let $\varDelta = f_1 - f_0$ and $\varDelta = f_{j+1} - f_j$, where $1 \leq j \leq z$. A naive way to address the problem is that the client uses R-TLP to encrypt each message m_j separately, such that it can be decrypted at time f_j

if all ciphertexts and public keys are passed on to the server at time t_0. For the server to decrypt the messages on time, it needs to start decrypting *all of them* as soon as the ciphertexts and public keys are given to it.

Parallel Composition Problem. The above naive approach leads to two vital issues: (a) imposing a high computation burden, as the server has to perform $S\Delta\sum_{j=1}^{z} j$ squaring, to decrypt all messages, and (b) demanding a high level of parallelisation, as each puzzle has to be dealt with separately in parallel to the rest. The issues can be cast as "*parallel composition problem*", where z instances of a puzzle scheme are given at once to a server whose only option, to find solutions on time, is to solve them in parallel.

Moreover, in the above approach, for the client to efficiently compute a_j for each message m_j, where $j > 1$, it has to perform at least one modular multiplication, i.e. $a_j = a_1 a_{j-1} = 2^{jT}$, where $a_1 = 2^T$. So, in this step, in total $z - 1$ modular multiplications are required to compute all a_j values, for z messages.

4.2 An Overview of our Solutions

Our observation is that, in the naive approach, the process of decrypting messages has many overlaps leading to a high computation cost. So, by removing the overlaps, we can considerably lower the overall cost both in *puzzle solving* and *puzzle creating* phases. Our idea is as follows. A client encrypts the messages in $\vec{\boldsymbol{m}}$ *outside-in* but they can be decrypted only *inside-out*. Specifically, it iteratively encrypts the message supposed to be decrypted after the rest, and embeds the information needed for decrypting it into the ciphertext of the message decrypted earlier. In other words, the client integrates the information (i.e. a part of public keys) required to decrypt message m_j into the ciphertext related to message m_{j-1}. In this case, the server after learning message m_{j-1} at time f_{j-1} learns the public key needed to perform the sequential squaring to decrypt the next message: m_j. This means after fully decrypting m_{j-1}, the server starts sequential squaring to decrypt m_j.

Addressing Parallel Composition Problem. The above approach solves the parallel composition problem for two main reasons. First, the total number of squaring required to decrypt all z messages is now much lower, i.e. $S\Delta z$, which is equivalent to the number of squaring needed to solve only the last puzzle, i.e. z^{th} one. Second, it does not demand high parallelisation. Because now the server does not need to deal with all of the puzzles in parallel; instead, it solves them sequentially one after another.

Adding Efficient Publicly Verifiable Algorithm. To let our proposed scheme support efficient public verifiability, we use the following trick. The client uses a commitment scheme to commit to every message: m_i and publishes the corresponding commitment. Then, it uses the time-lock encryption to encrypt the commitment's opening, i.e. a combination of m_i and a random value. But, unlike the traditional commitment, the client does not open the commitment itself. Instead, the server does that after it discovers the puzzle's solution. When it finds a solution, it decodes the solution to find the opening and sends it to the public who can check the solution correctness with the help of the commitment, already published. Therefore, to verify solution correctness, a verifier only needs to run the commitment's verification algorithm that is: (a) publicly verifiable, and (b) efficient. It can be built in the random oracle model, that involves only a hash function invocation, or in the standard model, that involves only 2 modular exponentiation for each puzzle.

Furthermore, the approach allows the server at setup to compute only a single $a = 2^T$ reusable for all z puzzles, which imposes a constant cost, $O(1)$.

4.3 Chained Time-lock Puzzle Definition

In this section, we provide a formal definition of a chained time-lock puzzle. Our starting point is the RSA time-lock puzzle definition, i.e. Definition 12, but we extend it from several perspectives, so it can: (a) handle multiple solutions/messages in setup, (b) produce multiple puzzles for the messages, (c) solve the puzzles given the puzzles and public parameters, and (d) support public verifications. Moreover, in our definition, the completeness property is met even if the solver solves a *subset of puzzles* (in the correct order) and its efficiency property depends on the number of puzzles it tries to solve. In the following, we provide the formal definition of a chained time-lock puzzle.

Definition 2 (Chained Time-lock Puzzle). *A chained time-lock puzzle comprises the following five algorithms, and satisfies completeness and efficiency properties.*

- ***Algorithms**:*
  - $\texttt{Setup}(1^\lambda, \Delta, z) \rightarrow (pk, sk, \vec{\boldsymbol{d}})$*: a probabilistic algorithm that takes as input security:* 1^λ *and time:* Δ *parameters as well as the total number of solutions/puzzles:* z*. Let* $j\Delta$ *be a time period after which* j^{th} *solution is found. It outputs public-private key pair:* (pk, sk) *and a set of fixed size witnesses:* $\vec{\boldsymbol{d}}$*.*
  - $\texttt{GenPuz}(\vec{\boldsymbol{m}}, pk, sk) \rightarrow \Theta$*: a probabilistic algorithm that takes as an input a message vector:* $\vec{\boldsymbol{m}} = [m_1, ..., m_z]$*, witness vector:* $\vec{\boldsymbol{d}}$ *and the public-private key pair:* (pk, sk)*, where* $\mathcal{M}$ *is messages' domain, i.e.* $m_j \in \mathcal{M}$*. It outputs* $\Theta : (\vec{\boldsymbol{\theta}}, \vec{\boldsymbol{h}})$*, where* $\vec{\boldsymbol{\theta}}$ *is a puzzle vector, and* $\vec{\boldsymbol{h}}$ *is a commitment vector. Each* j^{th} *element in vectors* $\vec{\boldsymbol{\theta}}$ *and* $\vec{\boldsymbol{h}}$ *corresponds to a solution* s_j *of the form:* $s_j = m_j || d_j$*.*
  - $\texttt{SolvPuz}(pk, \vec{\boldsymbol{\theta}}) \rightarrow \vec{\boldsymbol{s}}$*: a deterministic algorithm that takes as input the public key:* pk *and puzzle vector:* $\vec{\boldsymbol{\theta}}$*. It outputs a solution vector:* $\vec{\boldsymbol{s}}$*.*
  - $\texttt{Prove}(pk, s_j) \rightarrow \rho_j$*: a deterministic algorithm that takes the public key:* pk *and a solution:* $s_j \in \vec{\boldsymbol{s}}$*. It outputs a proof,* $\rho_j : (m_j, d_j)$*.*
  - $\texttt{Verify}(pk, \rho_j, h_j) \rightarrow \{0, 1\}$*: a deterministic algorithm that takes public key:* pk*, proof:* ρ_j *and commitment:* $h_j \in \vec{\boldsymbol{h}}$*. It outputs either* 0 *if it rejects, or* 1 *if it accepts.*
- ***Completeness**: for any honest prover and verifier, it always holds that:*
  - $\texttt{SolvPuz}(pk, [\theta_1, ..., \theta_j]) = [s_1, ..., s_j]$*, for every* j*,* $1 \leq j \leq z$*.*
  - $\texttt{Verify}(pk, \texttt{Prove}(pk, s_j), h_j) \rightarrow 1$
- ***Efficiency**: the run-time of algorithm* $\texttt{SolvPuz}(pk, [\theta_1, ..., \theta_j]) = [s_1, ...s_j]$ *is bounded by:* $poly(j\Delta, \lambda)$*, where* $poly(.)$ *is a fixed polynomial and* $1 \leq j \leq z$*.*

Informally, a chained time-lock puzzle is secure if it satisfies two properties: a solution's *privacy* and *validity*. The former one requires its j^{th} solution to remain hidden from all adversaries running in parallel within time period: $j\Delta$, while the latter one requires that it is infeasible for a PPT adversary to come up with an invalid solution and passes the verification. The two properties are formally defined in Definitions 3 and 4.

Definition 3 (Chained Time-lock Puzzle's Solution-Privacy). *A chained time-lock puzzle is privacy-preserving if for all λ and Δ, any pair of randomised algorithm $\mathcal{A}$: $(\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ runs in time $O(poly(j\Delta, \lambda))$ and $\mathcal{A}_2$ runs in time $\delta(j\Delta) < j\Delta$ using at most $\pi(\Delta)$ parallel processors, there exists a negligible function $\mu(.)$, such that:*

$$Pr\left[\mathcal{A}_2(pk, \Theta, state) \rightarrow (b_{j'}, j') \;\middle|\; \begin{array}{l} \texttt{Setup}(1^\lambda, \Delta, z) \rightarrow (pk, sk, \vec{\boldsymbol{d}}) \\ \mathcal{A}_1(1^\lambda, pk, z) \rightarrow (\vec{\boldsymbol{m}}, state) \\ \left[b_1, ..., b_z\right], b_j \xleftarrow{\$} \{0,1\} \\ \texttt{GenPuz}((m_{b_1,1}, ..., m_{b_z,z}), pk, sk) \rightarrow \Theta \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

where $j \leq j' \leq z$ and $b_{j'} \in [b_1, ..., b_z]$.

The definition above also ensures the solutions to appear after j^{th} one, remain hidden from the adversary with a high probability, as well. Moreover, it explicitly incorporates the pre-computation and parallel power of an adversary. In particular, similar to [9, 37, 21], it captures the fact that even if $\mathcal{A}_1$ computes on the public parameters for a polynomially long time, the adversary $\mathcal{A}_2$ cannot find j^{th} puzzle's solution in time $\delta(j\Delta) < j\Delta$ utilising $\pi(\Delta)$ parallel processors, with a probability significantly greater than $\frac{1}{2}$. As highlighted in [9], we can set $\delta(\Delta) = (1 - \epsilon)\Delta$ for a small ϵ, where $0 < \epsilon < 1$.

Definition 4 (Chained Time-lock Puzzle's Solution-Validity). *A chained time-lock puzzle preserves a solution validity, if for all λ and Δ, all probabilistic polynomial time adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ that run in time $O(poly(\Delta, \lambda))$ there is negligible function $\mu(.)$, such that:*

$$Pr\left[\begin{array}{l} \mathcal{A}_2(pk, \vec{\boldsymbol{s}}, \Theta, state) \rightarrow (j, \rho_j, \rho') \\ s.t. \\ \rho_j : (m_j, d_j), \rho' : (m', d') \\ m_j \in \vec{\boldsymbol{m}}, d_j \in \vec{\boldsymbol{d}}, m \neq m' \\ \texttt{Verify}(pk, \rho, h_j) = 1 \\ \texttt{Verify}(pk, \rho', h_j) = 1 \end{array} \;\middle|\; \begin{array}{l} \texttt{Setup}(1^\lambda, \Delta, z) \rightarrow (pk, sk, \vec{\boldsymbol{d}}) \\ \mathcal{A}_1(1^\lambda, pk, \Delta, z) \rightarrow (\vec{\boldsymbol{m}}, state) \\ \texttt{GenPuz}(\vec{\boldsymbol{m}}, pk, sk) \rightarrow \Theta \\ \texttt{SolvPuz}(pk, \vec{\boldsymbol{\theta}}) \rightarrow \vec{\boldsymbol{s}} \end{array}\right] \leq \mu(\lambda)$$

where $h_j \in \vec{\boldsymbol{h}} \in \Theta$

In Definition 4, we do not need to bound the adversaries' parallel computation power, as it does not need to solve any puzzles, in fact puzzles' solutions are provided to them. Therefore, they can run in polynomial time $O(poly(\Delta, \lambda))$.

Definition 5 (Chained Time-lock Puzzle Security). *The Chained Time-lock Puzzle scheme is secure if it meets solution-privacy and solution-validity properties.*

4.4 Chained RSA Time-lock Puzzle (CR-TLP) Protocol

Recall, a client wants a server to learn a vector of messages: $\vec{\boldsymbol{m}} = [m_1, ..., m_z]$ at times $[f_1, ..., f_z]$ respectively, where the client is available and online only at an earlier time $f_0 < f_1$. Also, the client wants to ensure that anyone can validate a solution found by the server, i.e. supports public verifiability. For the sake of simplicity, let $\Delta = f_1 - f_0$ and $\Delta = f_{j+1} - f_j$, where $1 \leq j \leq z$ and $T = S\Delta$.

In CR-TLP, the client at setup allocates a random group generator: r_j, a random value: d_j (for a commitment scheme) and a secret key: k_j (of a symmetric key encryption) to each message: m_j. Also, it picks two sufficiently large prime numbers and sets $N = q_1 q_2$. It sets $pk = (N, r_1, T)$ as public key and $sk = (q_1, q_2, \vec{\boldsymbol{r}}, \vec{\boldsymbol{k}}, \vec{\boldsymbol{d}})$ as secret key, where $\vec{\boldsymbol{r}} = [r_2, ..., r_z]$, $\vec{\boldsymbol{k}} = [k_1, ..., k_z]$, and $\vec{\boldsymbol{d}} = [d_1, ..., d_z]$. Note, it keeps secret all generators, except r_1. For the client to generate puzzles for messages in $\vec{\boldsymbol{m}}$, it first generates a single value: $a = 2^T \bmod \phi(N)$. Then, it starts from the last message to be revealed: m_z. It encrypts the message the same way as done in RSA time-lock puzzle with a difference that it first encodes the message as: $m_j || d_j$ and encrypts the encoded message, i.e. encrypts the encoded message under the related secret key: k_z, and blinds the secret key with r_j^a. Then, it moves on to the next message: m_{j-1}, that should be revealed before m_j and creates a puzzle for that. In this case, it first combines (i.e. concatenates) the message with d_{j-1} and r_z (where r_z is the generator used for m_z) and then encrypts the combination under the related key: k_{j-1}. Similarly, it blinds the key: k_{j-1} with r_{j-1}^a. It follows the same procedure to create puzzles for the rest of messages (in descending order). But, after creating a puzzle for m_1, it publishes r_1 as a part of the public key. To solve the puzzles, the server has to start from m_1, i.e. the message that should be revealed before the rest.

The server performs T squaring, given $r_1 \in pk$, to find encoded m_1 and r_2. Now given r_2, it can perform again T squaring to find m_2 and r_3, and it takes the same steps to finally find the last solution: m_z. To prove the correctness of each solution, the server decodes each encoded m_j to get (m_j, d_j). It publishes the pair. Now everyone, using the pair and the related commitment, can verify the solution validity. Note, the server cannot start from an arbitrary message in $\vec{\boldsymbol{m}}$ or solve the puzzle in an arbitrary order, as the generator related to the arbitrary message is yet unknown to it. So, to find the generator it has to solve the puzzle in (ascending) order. We provide CR-TLP protocol in detail, in Fig. 1.

Remark 1. In Fig. 1, we use the folklore hash-based commitment scheme, in the random oracle model, only to achieve more computation improvement than that can be achieved in the standard model. But CR-TLP can utilise any efficient non-interactive commitment scheme in the *standard model* as well, e.g. Pedersen Commitment.

Remark 2. The efficiency of CR-TLP scheme stems from three crucial factors: (a) removing computation overlaps when solving different puzzles: even though solving j^{th} puzzle, where $j > 1$, requires jT squaring, $(j-1)T$ of the squaring is used to solve previous puzzles that leads to $\frac{z+1}{2}$ times computation cost reduction at the server-side, (b) supporting reusable single public parameter: $a = 2^T$, generated only once that costs $O(1)$, as opposed to R-TLP whose cost is linear: $O(z)$, and (c) supporting efficient

- ***Input***: a vector of message: $\overrightarrow{\boldsymbol{m}} = [m_1, ..., m_z]$ that must be revealed at times $[f_1, ..., f_z]$ respectively. As defined in Section 3.4, $T = S\varDelta$, where $\varDelta = f_{j+1} - f_j$

1. **Setup**: $\texttt{Setup}(1^\lambda, \varDelta, z)$.
   (a) Call: $\texttt{R-TLP.Setup}(1^\lambda, \varDelta) \rightarrow (\hat{pk}, \hat{sk})$. Let $\hat{pk} = (N, T, r_1)$ and $\hat{sk} = (q_1, q_2, a, k_1)$.
   (b) Pick $z-1$ fixed size random generators: $\overrightarrow{\boldsymbol{r}} = [r_2, ..., r_z]$ from $\mathbb{Z}_N^*$.
   (c) Pick $z-1$ random keys: $[k_2, ..., k_z]$ for a symmetric key encryption. Let $\overrightarrow{\boldsymbol{k}} = [k_1, ..., k_z]$, where $k_1 \in \hat{sk}$. Also, pick z fixed size sufficiently large random values: $\overrightarrow{\boldsymbol{d}} = [d_1, ..., d_z]$, e.g. $|d_j| =$ 128-bit or 1024-bit depending on the choice of commitment scheme.
   (d) Set $pk = (\text{aux}, N, T, r_1)$ as public key. Set $sk = (q_1, q_2, a, \overrightarrow{\boldsymbol{k}}, \overrightarrow{\boldsymbol{r}}, \overrightarrow{\boldsymbol{d}})$ as secret key. Note, aux contains a cryptographic hash function's description and the size of the random values. Output pk and sk.
2. **Generate Puzzle**: $\texttt{GenPuz}(\overrightarrow{\boldsymbol{m}}, pk, sk)$
   Encrypt the messages, starting with $j = z$, in descending order as follows. $\forall j, z \geq j \geq 1$:
   (a) Set $pk_j = (N, T, r_j)$ and $sk_j = (q_1, q_2, a, k_j)$. Note, if $j = 1$ then $r_j \in pk$; otherwise (when $j > 1$), $r_j \in \overrightarrow{\boldsymbol{r}}$.
   (b) Generate a puzzle (or ciphertext pair):
      - if $j = z$, then run: $\texttt{R-TLP.GenPuZ}(m_j||d_j, pk_j, sk_j) \rightarrow \theta_j$.
      - otherwise (when $j < z$), run: $\texttt{R-TLP.GenPuZ}(m_j||d_j||r_{j+1}, pk_j, sk_j) \rightarrow \theta_j$.

      Recall that $\theta_j = (\theta_{j,1}, \theta_{j,2})$.
   (c) Commit to each message, e.g. $\texttt{H}(m_j||d_j) = h_j$ and output: h_j.
   (d) Output: $\theta_j = (\theta_{j,1}, \theta_{j,2})$ as puzzle (or ciphertext pair).

   By the end of this phase, vectors of puzzles: $\overrightarrow{\boldsymbol{\theta}} = [\theta_1, ..., \theta_z]$ and commitments: $\overrightarrow{\boldsymbol{h}} = [h_1, ...h_z]$ are generated. All public parameters and puzzles are given to a server/solver at time $t_0 < t_1$, where $\varDelta = f_1 - f_0$.
3. **Solve Puzzle**: $\texttt{SolvPuz}(pk, \overrightarrow{\boldsymbol{\theta}})$
   Decrypt the messages, starting with $j = 1$, in ascending order. $\forall j, 1 \leq j \leq z$:
   (a) If $j = 1$, then set $r_j = r_1$, where $r_1 \in pk$; Otherwise, set $r_j = u$.
   (b) Set $pk_j = (N, T, r_j)$
   (c) Run: $\texttt{R-TLP.SolvPuz}(pk_j, \theta_j) \rightarrow x_j$, where $\theta_j \in \overrightarrow{\boldsymbol{\theta}}$.
   (d) Parse x_j. Note that if $j < z$ then $x_j = m_j||d_j||r_{j+1}$; otherwise (when $j = z$), we have $x_j = m_j||d_j$. Therefore, x_j is parsed as follows:
      - if $j < z$:
         i. Parse $m_j||d_j||r_{j+1}$ into $m_j||d_j$ and $u = r_{j+1}$
         ii. Output $s_j = m_j||d_j$
      - otherwise (when $j = z$), output $s_j = x_j = m_j||d_j$
4. **Prove**: $\texttt{Prove}(pk, s_j)$. Parse s_j into pair: $\rho_j : (m_j, d_j)$ and send the pair to the verifier.
5. **Verify**: $\texttt{Verify}(pk, \rho_j, h_j)$. Verifies the commitment, e.g. $\texttt{H}(m_j, d_j) \overset{?}{=} h_j$. If passed, accept the solution and output 1; otherwise, reject it and output 0.

Fig. 1: Chained RSA Time-lock Puzzle (CR-TLP) Scheme

verification: due to the way each message is encoded (i.e. embedding the opening in a solution), the verification algorithm can use any efficient commitment scheme.

Remark 3. CR-TLP also can efficiently be used in a *multi-server* setting, where there are z servers: $\{S_1, ..., S_z\}$, each S_j needs to solve puzzle θ_j at time f_j and passes on the solution to the next server S_{j+1} to solve the next puzzle by time $f_{j+1} > f_j$. In this setting, due to the scalability property of CR-TLP (and unlike using the existing time-lock puzzles naively), other servers do not need to start solving the puzzle as soon as the client releases puzzles public parameters. Instead, they can wait until the previous solution is issued that saves them significant cost. Furthermore, a server can first verify the correctness of the solution found by the previous server (due to the public verifiability of CR-TLP), if accepted then it starts finding the next solution.

5 CR-TLP Security Proof

In this section, we prove the security of CR-TLP scheme presented in Fig. 1. We first prove that without solving j^{th} puzzle, a solver cannot find the parameter, e.g. a random generator, needed to solve the next puzzle, i.e. $(j+1)^{th}$ one.

Lemma 1 (Next Group Generator Privacy). *In CR-TLP scheme, given puzzle vector: $\vec{c}$ and public key: pk, an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ defined as above, cannot find the next group generator: r_{j+1}, where $j \geq 1$, significantly smaller than $T_j = \delta(j\Delta)$, except with a negligible probability.*

Proof. The proof is straightforward. Since the next generator: r_{j+1}, is: (a) encrypted along with the j^{th} puzzle solution: s_j, and (b) picked uniformly at random from $\mathbb{Z}_N^*$, for the adversary to find r_{j+1} without performing enough squaring, i.e. T_j, it has to (a) either break the symmetric key and extract it from the ciphertext corresponding to s_i that has the probability of success about $2^{-|k|}$ negligible or (b) guess r_{j+1}, that has the probability of success at most $2^{-|N|}$ which is negligible as well. $\square$

In the following, we prove that the privacy of a solution in CR-TLP scheme is preserved according to Definition 3.

Theorem 1 (CR-TLP Solution Privacy). *Let N be a strong RSA modulus and Δ be a time parameter. If the sequential squaring assumption holds, factoring N is a hard problem, $\mathtt{H}(.)$ is a random oracle and the symmetric key encryption is semantically secure, then CR-TLP encoding z solutions is a privacy-preserving chained time-lock puzzle.*

Proof. In the following, we argue for an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ runs in total time $O(poly(j\Delta, \lambda))$, $\mathcal{A}_2$ runs in time $\delta(j\Delta) < j\Delta$ using at most $\pi(\Delta)$ parallel processors, and $j \in [1, z]$, (a) when $z = 1$: to find s_1 earlier than $\delta(\Delta)$, it has to break R-TLP scheme, and (b) when $z > 1$: to find s_j earlier than $T_j = \delta(j\Delta)$, it has to either find at least one of the previous solutions earlier than it is supposed to (that ultimately requires breaking R-TLP scheme again), or find j^{th} generator: r_j, earlier. Also, we argue that the commitments: h_j, are computationally hiding. Specifically, when $z = 1$, the security of CR-TLP is reduced to the security of R-TLP and the scheme is secure as long as R-TLP is, as the two schemes would be identical. On the other hand, when $z > 1$, the adversary has to either find s_j earlier than T_j as soon as the previous solution: s_{j-1} is found that requires breaking R-TLP scheme, or find r_j generator before s_{j-1} is extracted, when $j \in [2, z]$. Nevertheless, R-TLP scheme is secure (under RSA, sequential squaring, and security of symmetric key encryption assumptions) according to Theorem 5, and also the probability of finding the next generator: r_j earlier than T_{j-1} is negligible, according to Lemma 1. Moreover, for an adversary to find a solution earlier, it may also try to find a (partial information of) pre-image of the commitment: h_j before fully (or without) solving the puzzle. But, this is infeasible for a PPT adversary, given output of a random oracle: $\mathtt{H}(.)$. Thus, CR-TLP is a privacy-preserving chained time-lock puzzle scheme. $\square$

Next, we prove that the validity of a solution in CR-TLP scheme is preserved according to Definition 4.

Theorem 2 (CR-TLP Solution Validity). *Let* $\mathtt{H}(.)$ *be a hash function modeled as a random oracle. Then, CR-TLP preserves a solution validity.*

Proof. The proof is straightforward and boils down to the security (i.e. binding property) of the traditional hash-based commitment scheme. In particular, given an opening pair, $\rho : (m_j, d_j)$ and the commitment $h_j = \mathtt{H}(m_j, d_j)$, for an adversary to break the solution validity, it has to come up (m'_j, d'_j), such that $\mathtt{H}(m'_j, d'_j) = h_j$, where $m_j \neq m'_j$, i.e. finds a collision of $\mathtt{H}(.)$. However, this is infeasible for a PPT adversary, as $\mathtt{H}(.)$ is collision resistance, in the random oracle model. $\square$

Theorem 3 (CR-TLP Security). *CR-TLP is a secure chained time-lock puzzle.*

Proof. According to Theorems 1 and 2, the privacy and validity of a solution in CR-TLP are preserved, respectively. Therefore, with regard to Definition 5, CR-TLP is a secure chained time-lock puzzle. $\square$

5.1 Asymptotic Cost Analysis

In this section, we analyse the communication and computation complexity of CR-TLP. We consider a generic setting where the protocol deals with z puzzles. In Table 2, we summarise the analysis results.

Computation Complexity. For a client to generate z puzzles, in total: in step 1a, it performs one exponentiation over $\bmod\ \phi(N)$. In step 2b, it z times calls $\mathtt{R\text{-}TLP.GenPuZ}(.)$. This in total involves z symmetric key-based encryption, z modular exponentiations over $\mathbb{Z}_N$ and z modular additions. Also, in step 2c it performs z invocations of a commitment scheme (to commit), i.e. if a hash-based commitment is used then it would involve z invocations of a hash function, and if Pedersen commitment is used then it would involve $2z$ exponentiations and z multiplications, where all operations, in the latter commitment, are done over a $\bmod\, q$ for a large prime number: q, e.g. $|q| = 1024$-bit. Thus, the overall computation complexity of the client is $O(z)$. For the server to

Table 2: CR-TLP's Detailed Cost Breakdown

(a) Computation Cost

Protocol	Operation	Protocol Function			Complexity
		`GenPuz`	`SolvPuz`	`Verify`	
CR-TLP	Exp.	$z+1$	Tz	$-$	$O(Tz)$
	Add. or Mul.	z	z	$-$	
	Commitment	z	$-$	z	
	Sym. Enc	z	z	$-$	

(b) Communication Cost (in bit)

Protocol	Model	Client	Server	Complexity
CR-TLP	Standard	$3200z$	$1524z$	$O(z)$
	R.O.	$2432z$	$628z$	

solve z puzzles, it z times calls $\texttt{R-TLP.SolvPuz}(.)$. This in total involves Tz modular squaring over $\mathbb{Z}_N$, z modular additions over $\mathbb{Z}_N$, and z symmetric key based decryption. The server's cost of proving, in step 4, is very low, as it involves only parsing z strings. Therefore, the server total computation complexity is $O(Tz)$. The verification cost, in step 5, only involves z invocation of commitment scheme (to verify each opening) and it is *independent* of the RSA security parameters. If the hash-based commitment is used, then it would involve z invocations of a hash function, if Pedersen commitment is utilised, then in total it would involve $2z$ exponentiations and z multiplications performed over $\bmod q$. Thus, the total verification complexity is $O(z)$.

Communication Complexity. In step 2, the client publishes two vectors: $\overrightarrow{\boldsymbol{\theta}}$ and $\overrightarrow{\boldsymbol{h}}$, with $2z$ and z elements respectively. Each element of $\overrightarrow{\boldsymbol{\theta}}$ is a pair $(\theta_{j,1}, \theta_{j,2})$, where $\theta_{j,1}$ is an output of symmetric key encryption, e.g. $|\theta_{j,1}| =$ 128-bit, and $\theta_{j,2}$ is an element of $\mathbb{Z}_N$, e.g. $|\theta_{j,2}| =$ 2048-bit. Also, each element h_j of $\overrightarrow{\boldsymbol{h}}$ is either an output of a hash function, when a hash-based commitment is used, e.g. $|h_j| =$ 256-bit, or an element of $\mathbb{F}_q$ when Pedersen commitment is used, e.g. $|h_j| =$ 1024-bit. Therefore, its total bandwidth is about $2432z$ bits when the former, or $3200z$ bits when the latter commitment scheme is utilised. Also, its complexity is $O(z)$. Note, in CR-TLP, when a server finds a solution, it does not broadcast anything, instead it moves on to the next step: $\texttt{Prove}(.)$. For the server to prove, in step 4, in total, it sends z pairs (m_j, d_j) to the verifier, where m_j is an arbitrary message, e.g. $|m_j| =$ 500-bit, and d_j is either a long enough random value, e.g. $|d_j| =$ 128-bit, when the hash-based commitment is used, or an element of $\mathbb{F}_q$ when Pedersen scheme is used, e.g. $|d_j| =$ 1024-bit. Therefore, its bandwidth is about either $628z$ or $1524z$ bits when the former or latter commitment scheme is used respectively. The solver's total communication complexity is $O(z)$.

6 Smarter Outsourced PoR (SO-PoR) Utilising CR-TLP

As discussed in Section 2.2, the existing outsourced PoR schemes have a set of serious shortcomings, e.g. having high costs, not supporting real-time detection, or fair payment. To address the issues, in this section we present SO-PoR. We first outline our solution, SO-PoR, and then, provide its formal definition. Next, we present the protocol in detail and provide its security proof. After that, we compare its properties and costs with the existing outsourced PoR schemes.

6.1 SO-PoR Overview

Our idea is as follows. We allow a smart contract to act on a client's behalf to do PoR verifications and if approved it pays the server. Generally speaking, smart contracts are believed to execute given instructions correctly if the underlying blockchain is secure. So, the correctness of the PoR verification is guaranteed. However, there are several efficiency and security constraints. In particular, the verification cost must be very low especially when a smart contract is involved, as in general any computation performed by a smart contract costs much higher than when it is done by a local off-chain node. To achieve this goal, SO-PoR builds upon and extends PoR scheme in [44]. It utilises homomorphic MAC-based tags, highly efficient. However, such tags are inherently privately verifiable, which means secret values (hidden from the server) are needed to do the verification. But, smart contracts usually do not maintain a private state. To address the privacy issue, while maintaining server-side efficiency, we utilise CR-TLP, we proposed in Section 4.4, such that the client encodes the secrets required for verifications in puzzles and send the puzzles to the sever. To ensure the server sends a correct solution to the contract, the client stores a short representation of the solutions in the contract such that the representation does not reveal the solutions. For each verification, the server first solves a puzzle and sends PoR to the contract. After a certain time, it solves another puzzle and releases the puzzle solution to the contract who now can verify the correctness of the solution as well as verifying the PoR.

SO-PoR uses a novel combination of disposable tags, pre-computation technique, CR-TLP scheme, smart contract, pseudorandom functions and commitment scheme. At a high-level, SO-PoR works as follows. The client generates a set of authenticator tags on the file. These tags will allow the client to verify its data availability when it is online. Also, for every verification that the client cannot be online, it precomputes a (small) set of *disposable* tags related to the file's blocks that will be challenged for that verification. However, unlike standard PoR schemes in which a subset of file blocks are challenged by picking their indices randomly on the fly just before the verification, in SO-PoR, the challenged blocks' indices are picked *pseudorandomly* by the client in the setup phase. Then, the client for, each verification, encodes the secret key that allows regeneration of the pseudorandom indices and a secret key used for a PoR verification into two puzzles. The client stores the file, tags, and puzzles on the server. It stores commitments of the secret values that will be used for PoR verification in a smart contract. Also, it deposits enough coins in the smart contract, to pay the server if each proof (given by the server) is accepted. At each verification time, the server first solves a puzzle and fully recovers the key for random indices. Using the key, corresponding tags and the file, it generates a PoR and sends it to the contract. After a certain period, for the same verification, it manages to fully find another puzzle solution which is the verification key. It sends the key to the contract who first checks the correctness of the key and then verifies the PoR. If accepted, the contract for that verification pays the cloud server who can now delete all metadata (e.g. tags, encrypted, and decrypted values) for that verification.

6.2 SO-PoR Model

In this section, we provide a formal definition of SO-PoR. As it builds upon the traditional PoR model [44], we provide the latter one in Appendix B. In general, a PoR scheme considers the case where an honest client wants to store its file(s) on a server potentially malicious, i.e active adversary. It is a challenge-response interactive protocol where the server proves to the client that its file is intact and retrievable.

In SO-PoR, unlike the traditional PoR, a client may not be available every time verification is needed. Therefore, it wants to delegate a set of verifications that it cannot carry out itself. In this setting, it (in addition to file retrievability) must have three guarantees: (a) *verification correctness*: every verification is performed honestly, so the client can rely on the verification result without the need to re-do it, (b) *real-time detection*: the client is notified in almost real-time when server's proof is rejected, and (c) *fair payment*: in every verification, the server is paid only if the server's proof is accepted. In SO-PoR, three parties are involved: a client (honest), server potentially malicious and a standard smart contract. SO-PoR also allows a client to perform the verification itself, analogous to the traditional PoR, when it is available.

Definition 6. *A Smart Outsourced PoR (SO-PoR) scheme consists of seven algorithms (*$\mathtt{Setup}, \mathtt{Store}, \mathtt{SolvPuz}, \mathtt{GenChall}, \mathtt{Prove}, \mathtt{Verify}, \mathtt{Pay}$*) defined below:*

- $\mathtt{Setup}(1^{\lambda}, \Delta, z) \rightarrow (\hat{sk}, \hat{pk})$*: a probabilistic algorithm, run by a client. It takes as input a security:* 1^{λ}*, time parameter:* Δ*, and the number of verification delegated:* z*. It outputs a set of secret and public keys.*

- $\mathtt{Store}(\hat{sk}, \hat{pk}, F, z) \rightarrow (\vec{\boldsymbol{F}}, \sigma, \vec{\boldsymbol{\theta}}, u)$*: a probabilistic algorithm, run only once by a client. It takes as input the secret key:* $\hat{sk}$*, public key:* $\hat{pk}$*, a file:* F*, and the number of verifications:* z *that the client wants to delegate. It outputs an encoded file:* $\vec{\boldsymbol{F}}$*, a set of tags:* σ*, a set of* z *puzzles:* $\vec{\boldsymbol{\theta}}$*, and public auxiliary data:* u*. First three outputs are stored on the server and last output:* u*, is stored on a smart contract.*

- $\mathtt{SolvPuz}(\hat{pk}, \vec{\boldsymbol{\theta}}) \rightarrow \vec{\boldsymbol{s}}$*: a deterministic algorithm that takes as input the public key:* $\hat{pk}$ *and puzzle vector:* $\vec{\boldsymbol{\theta}}$*. It for each* j^{th} *verification outputs a pair:* $s_j : (v_j, l_j)$ *of solutions, where* v_j *and* l_j *are outputted at time* t_j *and* t'_j *respectively and* $t'_j > t_j$*. Therefore, the algorithm in total outputs* z *pairs. Value* l_j *is sent to the smart contract right after discovered. This algorithm is run by the server.*

- $\mathtt{GenChall}(j, |\vec{\boldsymbol{F}}|, 1^{\lambda}, s_j, u) \rightarrow \vec{\boldsymbol{c}}$*: a probabilistic algorithm that takes as input a verification index:* j*, the encoded file size:* $|\vec{\boldsymbol{F}}|$*, security parameter:* 1^{λ}*, first component of the related solution pair,* $v_j \in s_j$*, and public parameters:* $pp \in u$ *containing a blockchain and its parameters. It outputs pairs* $c_j : (x_j, y_j)$*, where each pair includes a seudorandom block's index:* x_j *and random coefficient:* y_j*. Also, values* x_j *are derived from* v_j *while* y_j *are derived from* pp*. This algorithm is run by the server for each verification.*

- $\mathtt{Prove}(j, \vec{\boldsymbol{F}}, \sigma, \vec{\boldsymbol{c}}) \rightarrow \pi$*: a probabilistic algorithm that takes the verification index* j*, encoded file:* $\vec{\boldsymbol{F}}$ *, (a subset of) tags:* σ*, and a vector of unpredictable challenges:* $\vec{\boldsymbol{c}}$*, as inputs and outputs a proof of file retrievability. It is run by the server for each verification.*

- $\texttt{Verify}(j, \pi, s_j, u) \rightarrow d : \{0, 1\}$*: a deterministic algorithm that takes the verification index j, proof: π, second component of the related solution pair: $l_j \in s_j$, and public auxiliary data: u. If the proof is accepted, it outputs $d = 1$; otherwise, outputs $d = 0$. The default value of d is 0. This algorithm is run by the smart contract for each verification, and invoked only once for each verification by only the server.*

- $\texttt{Pay}(j, d) \rightarrow d' = \{0, 1\}$*: a deterministic algorithm that takes the verification index j, the verification output: d. If $d = 1$, it transfers e amounts to the server and outputs 1. Otherwise, it does not transfer anything, and outputs 0. The default value of d' is 0. The algorithm is run by the contract, and invoked only by* $\texttt{Verify}(.)$.

A SO-PoR scheme must satisfy two main properties: *correctness* and *soundness*. The correctness requires, for any: file, public-private key pairs, and puzzle solutions, both the verification and pay algorithms, i.e. $\texttt{Verify}(.)$ and $\texttt{Pay}(.)$, output 1 when interacting with the prover, verifier, and client all of which are honest. The soundness however is split into four properties: extractability, verification correctness, real-time detection, and fair payment, formally defined below. Before we define the first property, extractability, we provide the following experiment between an environment: $\mathcal{E}$ and adversary: $\mathcal{A}$ who corrupts $C \subsetneq \{\mathcal{S}, \mathcal{M}_1, ..., \mathcal{M}_\beta\}$, where β is the maximum number of miners which can be corrupted in a secure blockchain. In this game, $\mathcal{A}$ plays the role of corrupt parties and $\mathcal{E}$ simulating an honest party's role.

1. $\mathcal{E}$ executes $\texttt{Setup}(.)$ algorithm and provides public key: $\hat{pk}$, to $\mathcal{A}$.
2. $\mathcal{A}$ can pick an arbitrary file: F', and uses the file to make queries to $\mathcal{E}$ to run: $\texttt{Store}(\hat{sk}, \hat{pk}, F', z) \rightarrow (F'^*, \sigma, \vec{\boldsymbol{\theta}}, u)$ and return the output to $\mathcal{A}$. Furthermore, upon receiving the output of $\texttt{Store}()$, $\mathcal{A}$ can locally run algorithms: $\texttt{SolvPuz}(\hat{pk}, \vec{\boldsymbol{\theta}})$ and $\texttt{GenChall}(j, |F'^*|, 1^\lambda, s_j, u) \rightarrow \vec{\boldsymbol{c}}$ as well as $\texttt{Prove}(j, F^*, \sigma, \vec{\boldsymbol{c}}) \rightarrow \pi$, to get their outputs as well.
3. $\mathcal{A}$ can request $\mathcal{E}$ the execution of $\texttt{Verify}(j, \pi, s_j, u)$ for any F' used to query $\texttt{Store}()$. Accordingly, $\mathcal{E}$ informs $\mathcal{A}$ about the verification output. The adversary can send a polynomial number of queries to $\mathcal{E}$. Finally, $\mathcal{A}$ outputs the description of a prover: $\mathcal{A}'$ for any file it has already chosen above.

It is said a cheating prover: $\mathcal{A}'$ is ϵ-admissible if it convincingly answers ϵ fraction of verification challenges. Informally, a SO-PoR scheme supports extractability, if there is an extractor algorithm: $\texttt{Ext}(\hat{sk}, \hat{pk}, \texttt{P}')$, that takes the secret-public keys and the description of the machine implementing the prover's role: $\mathcal{A}'$ and outputs the file: F'. 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 it.

Definition 7 (ϵ-extractable). *A SO-PoR scheme is ϵ-extractable if for every adversary: $\mathcal{A}$ who corrupts $C \subsetneq \{\mathcal{S}, \mathcal{M}_1, ..., \mathcal{M}_\beta\}$, plays the experiment above, and outputs an ϵ-admissible cheating prover: $\mathcal{A}'$ for a file F', there exists an extraction algorithm that recovers F' from $\mathcal{A}'$, given honest parties public-private keys and $\mathcal{A}'$, i.e.* $\texttt{Ext}(\hat{sk}, \hat{pk}, \mathcal{A}') \rightarrow F'$*, except with a negligible probability.*

In the above game, the environment, acting on honest parties behalf, performs the verification correctly; which is not always the case in SO-PoR. As the verification can be run by miners a subset of which are potentially corrupted. Even in this case, the verification correctness must hold, e.g. if a corrupt server sends an invalid proof then even if $\beta - 1$ miners are corrupt (and colluding with it) the verification function will not output 1 and if the server is honest and submits a valid proof then the verification function does not output 0 even if β miners are corrupt, except with a negligible probability. It is formalised as follows.

Definition 8 (Verification Correctness). *Let β be the maximum number of miners that can be corrupted in a secure blockchain network and λ' be the blockchain security parameter. Also, let $\mathcal{A}$ be the adversary who (plays the above game and) corrupts parties in either $C \subseteq \{\mathcal{S}, \mathcal{M}_1, ..., \mathcal{M}_{\beta-1}\}$ or $C' \subseteq \{\mathcal{M}_1, ..., \mathcal{M}_{\beta}\}$. In SO-PoR, we say the correctness of j^{th} verification is guaranteed if:*

$$\textit{in the former case}: Pr[\texttt{Verify}_{C}(j, \pi, s_j, u) = 1] \leq \mu(\lambda')$$
$$\textit{in the latter case}: Pr[\texttt{Verify}_{C'}(j, \pi, s_j, u) = 0] \leq \mu(\lambda')$$

where $\mu(.)$ is a negligible function.

Also, a client needs to have a guarantee that for each verification it can get a correct result within a (fixed) time period.

Definition 9 (Υ-real-time Detection). *Let $\mathcal{A}$, as defined above, be the adversary who corrupts either C or C'. A client, for each j^{th} delegated verification, will get a correct output of* $\texttt{Verify}(.)$, *by means of reading a blockchain, within time window Υ, after the time when the server is supposed to send its proof to the blockchain network. Formally,*

$$\texttt{Read}(\Upsilon, \texttt{Verify}_{D}(j, \pi, s_j, u)) \rightarrow \{0, 1\}$$

where $D \subsetneq \{C, C'\}$, except with a negligible probability.

Definition 10 (Fair Payment). *SO-PoR supports a fair payment if the client and server fairness are satisfied:*

- ***Client Fairness**: An honest client is guaranteed that it only pays (e coins) iff the server provides an accepting proof, except with a negligible probability.*
- ***Server Fairness**: An honest server is guaranteed that the client gets a correct proof iff the client pays (e coins), except with a negligible probability.*

Formally, let $\mathcal{A}$ be the adversary who corrupts either C or C', as defined above. To satisfy a fair payment:

$$Pr[\texttt{Pay}_{D}(.) = b \cap \texttt{Verify}_{D}(.) = b] \geq 1 - \mu(\lambda'), \tag{1}$$

the following inequality must hold:

$$Pr[\texttt{Pay}_{D}(.) = b' \cap \texttt{Verify}_{D}(.) = b] \leq \mu(\lambda'), \tag{2}$$

where $D \subsetneq \{C, C'\}, b \neq b'$, and $b, b' \subsetneq \{0, 1\}$

The above definition also takes into account the fact that the client at the time of delegated verification is not necessarily available to make the payment itself, so the payment is delegated to a third party, e.g. a smart contract. In this case, the definition ensures that even if the client or/and server are honest, the third party cannot affect the fairness (except with a negligible probability).

Definition 11 (SO-PoR Security). *A SO-PoR scheme is secure if it is ϵ-extractable, and satisfies verification correctness, Υ-real-time detection, and fair payment properties.*

Remark 4. The folklore assumption is that (in a secure blockchain) a smart contract function *always outputs a correct result*. However, this is not the case and it may fail under certain circumstances. For instance, as shown in [34] all rational miners may not verify a certain transaction. As another example, an adversary (although with a negligibly small probability) discards a certain honestly generated blocks, reverses the state of blockchain and contract, or breaks client's signature scheme. Accordingly, in our definitions above, we take such cases into consideration and allow the possibility that a function outputs an incorrect result even though with a negligibly small probability.

Remark 5. SO-PoR model differs from traditional (e.g. [24, 44]) and outsourced PoR (e.g. [3, 51]) models in several aspects. Only SO-PoR model offers all the properties. In particular, traditional PoRs only offer extractability while outsourced ones offer liability as well, that allows a client (by re-running all verifications function) to detect a verifier if it provides an incorrect verification output, so the client cannot rely on the verification result provided. As another difference, SO-PoR model takes into account the case where an adversary can corrupt both the server and some miners at the same time.

Remark 6. SO-PoR should also support the traditional PoR where only client and server interact with each other (e.g. client generates challenges, and verifies proof) when the client is available. To let SO-PoR definition support that too, we can simply define a flag: ξ, in each function, such that when $\xi = 1$, it acts as the traditional PoR; otherwise (when $\xi = 0$), it performs as a delegated one. For the sake of simplicity, we let the flag be implicit in the definitions above, where the default is $\xi = 1$.

6.3 SO-PoR Protocol

In this section, we first present SO-PoR protocol in detail. Then, provide the rationale behind the protocol design.

1. ***Client-side Setup***.

   (a) ***Gen. Public and Private Keys***: Picks a fresh key: $\hat{k}$ and two vectors of keys: $\vec{\boldsymbol{v}}$ and $\vec{\boldsymbol{l}}$, where each vector contains z fresh keys. It picks a large prime number: p whose size is determined by a security parameter, i.e. $|p| = \iota$. Moreover, it runs `Setup(.)` in CR-TLP scheme to generate a key pair: (sk, pk).

   (b) ***Gen. Other Public Parameters***: Sets c to the total number of blocks challenged in each verification. It defines parameters: w and g, where w is an index of a future block: $\mathcal{B}_w$ in a blockchain that will be added to the blockchain (permanent state) at about the time when 1st delegated verification will be performed, and g is also a security parameter referring to the number of blocks (in a row) starting from w. It also sets z: total number of verifications, $||\overrightarrow{\boldsymbol{F}}||$: file bit size, Δ_1: the maximum time taken by the server to generate a proof, Δ_2: time window in which a message is (sent by the server and) received by the contract, and e amount of coins paid to the server for each successful delegated verification. Sets $\hat{pk} : (pk, e, g, w, p, c, z, \Delta_1, \Delta_2)$ that will be sent to the server.

   (c) ***Sign and Deploy Smart Contract***: Signs and deploys a smart contract: $\mathcal{SC}$ to a blockchain. It stores public parameters: $(z, ||\overrightarrow{\boldsymbol{F}}||, \Delta_1, \Delta_2, c, g, p, w)$, on the contract. It deposits ez coins to the contract. Then, it asks the server to sign the contract. The server signs if it agrees on all parameters.

2. ***Client-side Store***.
   (a) ***Encode File***: Splits an error-corrected file, e.g. under Reed-Solomon codes, into n blocks. $\overrightarrow{\boldsymbol{F}} : [F_1, ..., F_n]$, where $F_i \in \mathbb{F}_p$
   (b) ***Gen. Permanent Tags***: Using the key: $\hat{k}$, it computes n pseudorandom values: r_i and single value: α, as follows.

$$\alpha = \texttt{PRF}(\hat{k}, n+1) \bmod p, \forall i, 1 \leq i \leq n : r_i = \texttt{PRF}(\hat{k}, i) \bmod p$$

   It uses the pseudorandom values to compute tags for the file blocks.

$$\forall i, 1 \leq i \leq n : \sigma_i = r_i + \alpha \cdot F_i \bmod p$$

   So, at the end of this step, a set σ of tags are generated, $\sigma : \{\sigma_1, ..., \sigma_n\}$
   (c) ***Gen. Disposable Tags***: For j^{th} verification $(1 \leq j \leq z)$:
      i. chooses the related key: $v_j \in \overrightarrow{\boldsymbol{v}}$ and computes c pseudorandom indices.

$$\forall b, 1 \leq b \leq c : x_{b,j} = \texttt{PRF}(v_j, b) \bmod n$$

      ii. picks the corresponding key: $l_j \in \overrightarrow{\boldsymbol{l}}$ and computes c pseudorandom values: $r_{b,j}$ and single value: α_j, as follows.

$$\alpha_j = \texttt{PRF}(l_j, c+1) \bmod p, \forall b, 1 \leq b \leq c : r_{b,j} = \texttt{PRF}(l_j, b) \bmod p$$

      iii. generates c disposable tags:

$$\forall b, 1 \leq b \leq c : \sigma_{b,j} = r_{b,j} + \alpha_j \cdot F_y \bmod p$$

      where $y = x_{b,j}$. At the end of this step, a set σ_j of c tags are computed, $\sigma_j : \{\sigma_{1,j}, ..., \sigma_{c,j}\}$
   (d) ***Gen. Puzzles***: Sets $\overrightarrow{\boldsymbol{m}} = [v_1, l_1, ..., v_z, l_z]$ and then encrypts the vector's elements, by running: $\texttt{GenPuz}(\overrightarrow{\boldsymbol{m}}, pk, sk)$ in CR-TLP scheme. This yields a puzzle vector: $[(V_1, L_1), ..., (V_z, L_z)]$ and a commitment vector $\overrightarrow{\boldsymbol{h}}$. The encryption is done in such a way that in each j^{th} pair, V_j will be fully decrypted at times t_j and L_j will be decrypted at time t'_j, where $t'_j \geq t_j + \Delta_1 + \Delta_2$.
   (e) ***Outsource File***: Stores $\overrightarrow{\boldsymbol{F}}, n, \hat{pk}, \{\sigma, \sigma_1, ..., \sigma_z, (V_1, L_1), ..., (V_z, L_z)\}$ on the server. Also, it stores $\overrightarrow{\boldsymbol{h}}$ on the smart contract.

3. ***Cloud-Side Proof Generation***. For j^{th} verification $(1 \leq j \leq z)$, the cloud:

(a) ***Solve Puzzle and Regen. Indices***: Receives and parses the output of $\texttt{SolvPuz}(.)$ in CR-TLP, to extract v_j, at time t_j. Next, using v_j, it regenerates c pseudorandom indices.

$$\forall b, 1 \leq b \leq c : x_{b,j} = \texttt{PRF}(v_j, b) \bmod n$$

(b) ***Extract Key***: extracts a seed: s_j, from the blockchain as follows: $s_j = H(\mathcal{B}_\gamma||...||\mathcal{B}_\zeta)$, where $\gamma = w + (j-1) \cdot g$ and $\zeta = w + j \cdot g$

(c) ***Gen. PoR***: computes:

$$\mu_j = \sum_{b=1}^{c} \texttt{PRF}(s_j, b) \cdot F_y \bmod p, \quad \xi_j = \sum_{b=1}^{c} \texttt{PRF}(s_j, b) \cdot \sigma_{b,j} \bmod p$$

where y is a pseudorandom index: $y = x_{b,j}$.

(d) ***Register Proofs***: sends the PoR: (μ_j, ξ_j) to the smart contract within Δ_1

(e) ***Solve Puzzle and Regen. Verification Key***: Receives and parses the output of $\texttt{SolvPuz}(.)$ in CR-TLP to extract l_j, at time t'_j. Also, it runs $\texttt{Prove}(.)$ in CR-TLP, to generate a proof: ρ_j, of l_j's correctness. It sends ρ_j (containing l_j) to the contract, so it can be received by the contract within Δ_2.

4. ***Smart Contract-Side Verification***. For j^{th} verification $(1 \leq j \leq z)$, the contract:

(a) ***Check Arrival Time***: checks the arrival time of the decrypted values sent by the server. In particular, it checks, if (μ_j, ξ_j) was received in the time window: $(t_j, t_j + \Delta_1 + \Delta_2]$ and whether l_j was received in the time window: $(t'_j, t'_j + \Delta_2]$.

(b) ***Verify Puzzle Solution***: runs $\texttt{Verify}(.)$ in CR-TLP to verify ρ_j (i.e. check the correctness of $l_j \in \rho_j$). If approved, then regenerates the seed: $s_j = H(\mathcal{B}_\gamma||...||\mathcal{B}_\zeta)$, where $\gamma = w + (j-1) \cdot g$ and $\zeta = w + j \cdot g$

(c) ***Verify PoR***: regenerates the pseudorandom values and verifies the PoR.

$$\xi_j \stackrel{?}{=} \mu_j \cdot \texttt{PRF}(l_j, c+1) + \sum_{b=1}^{c} (\texttt{PRF}(s_j, b) \cdot \texttt{PRF}(l_j, b)) \bmod p \qquad (3)$$

(d) ***Pay***: if equation 3 holds, pays and asks the server to delete all disposable tags for this verification, i.e. σ_j.

If either of the above checks fails, it aborts and notifies the client.

5. ***Client-server PoR***: When the client is online, it can interact with the server to check its data availability. In particular, it sends c random challenges and random indices to the server who computes POR using only: (a) the messages sent by the client in this step, (b) the file: F, and (c) the tags: $\sigma_i \in \sigma$, generated in step 2b. The proof generation and verification are similar to the MAC-based schemes, e.g. [44].

Remark 7. In each verification, e.g. j^{th}, it is required that the server can: (a) learn the random challenges, (b) compute a proof, and (c) record it in the smart contract, *before* it is able to learn key l_j; otherwise, (i.e. if it learns l_j before sending and registering the proof), it can tamper with the data and pass the verification; because by knowing l_j it can construct valid tags for the data tampered with. That is why, in the protocol, it is required: $t'_j \geq t_j + \Delta_1 + \Delta_2$.

Remark 8. The way disposable tags are generated in SO-PoR differs from those computed in traditional/outsourced PoR schemes, in spite of having similarities structure-wise. Specifically, (unlike existing protocols, e.g. [44, 3]) in SO-PoR, each random value, $r_{b,j}$, utilised to generate disposable tag of a block (for j^{th} verification), is not derived from the block index. Instead, it depends on (a fresh secret key for j^{th} verification and) the total number of blocks challenged in each verification that is a public value. This means, the verifier does not need to know and verify each challenged block's index in the verification phase, which leads to a lower cost.

Strawman Solutions: One might be willing to utilise a combination of publicly verifiable PoR and smart contract, such that the contract performs the verification on the client's behalf. In this setting, the correctness of verification is guaranteed. However, this approach would have a higher computation or communication cost (especially in the verification phase) than our protocol's. Specifically, as stated in Section 2, there are two publicly verifiable PoR schemes, based on either (a) BLS signatures, e.g. [44], or (b) Merkle tree, e.g. [38]. The former approach, in total, requires zc exponentiations in the verification phase, whereas SO-PoR requires no exponentiations in this phase. Also, the BLS signature-based scheme takes a very long time to encode a file, as the required number of exponentiations is linear with the file size. For instance, as measured in [3], it takes about 55 minutes to encode a 64-MB file, that means it would take about 14 hours to encode a 1-GB file. But, in SO-PoR the number of exponentiations, in the store phase, is independent of and much fewer than the file size. On the other hand, the proof size in the Merkle tree-based approach is logarithmic with the file size, i.e. $256zc \log |F|$ bits, that leads a high server-side's communication cost. By contrast, the proof size in SO-PoR is independent of the file size, and is much shorter, i.e. $884z$ bits.

6.4 SO-PoR Security Proof

This section presents the main security theorem of SO-PoR protocol (presented in Section 6.3) followed by its proof.

Theorem 4. *SO-PoR protocol is secure (according to Definition 11) if the tags/MAC's are unforgeable,* $\mathtt{PRF}(.)$ *is a secure pseudorandom function, the blockchain is secure, CR-TLP protocol is secure, and* $H(\mathcal{B}_\gamma||...||\mathcal{B}_\zeta)$ *outputs an unpredictable random value (where* $\zeta - \gamma$ *is a security parameter).*

Sketch. In the following, we prove that SO-PoR protocol satisfies every property defined in Section 6.2.

Verification correctness (according to Definition 8). We first argue that the adversary who corrupts either $C \subseteq \{\mathcal{M}_1, ..., \mathcal{M}_\beta\}$ or $C' \subseteq \{\mathcal{S}, \mathcal{M}_1, ..., \mathcal{M}_{\beta-1}\}$ with a high probability, cannot influence the output of $\mathtt{Verify}(.)$ performed by a smart contract in a blockchain; in other words, the verification correctness holds. In short, the verification correctness boils down to the security of the underlying blockchain. In the case where the adversary corrupts C (when the server provides an accepting proof), for the adversary to make the verification function output 0, it has to: (a) either forge the server's signature, or (b) fork the blockchain so the chain comprising the accepting proof is discarded. In case (a), if it manages to forge the signature, it can generate a transaction that includes a rejecting proof where the transaction is signed on the server's behalf. In this case, it can broadcast the transaction as soon as the transaction containing an accepting proof is broadcast, to make the latter transaction stale. Nevertheless, the probability of such an event is negligible, $\epsilon(\lambda')$, as long as the signature is secure. Moreover, due to the liveness property of blockchain, an honestly generated transaction will eventually appear on honest miners' chain [20]. In case (b), the adversary has to generate long enough (valid) chain that excludes the accepting proof, but this also has a negligible success probability, $\epsilon(\lambda')$, under the assumption that the hash power of the adversary is lower than those of honest miners (i.e. under the honest majority assumption) and due to the liveness property. Now we turn our attention to the case where the adversary corrupts C' (when the server provides a rejecting proof). In this case, for the adversary to make the verification function to output 1, the honest miners must not validate the transaction that contains the proof. Nevertheless, as long as the blockchain is secure and the computational advantage of skipping transaction validation is low, i.e. the validation imposes a low computation cost, the miners check the transaction's validation [34]. Also, as shown in [34] when a transaction validation imposes a high computation cost, two generic techniques can be used to support exact or probabilistic correctness (of a smart contract function output). We conclude the correctness of $\texttt{Verify}(.)$ output is guaranteed with a high probability.

ϵ-extractable (according to Definition 7). In the following, we show that if a proof produced by an adversary: $\mathcal{A}$ who corrupts $C' \subseteq \{\mathcal{S}, \mathcal{M}_1, ..., \mathcal{M}_{\beta-1}\}$ is accepted by $\texttt{Verify}(.)$ with probability at least ϵ, then the file can be extracted by a means of an extraction algorithm. As mentioned before, $\texttt{Verify}(.)$ can use both disposable and permanent tags, in the latter case $\texttt{Verify}(.)$ is run by a smart contract, while in the former one the client runs it. For the sake of simplicity, we first consider the case where $C' = \mathcal{S}$. In this case, the extractability proof is similar to the one in [44], with a few differences, in SO-PoR: (a) the extractor can use both disposable tags and permanent tags (when the former run out), (b) assumes CR-TLP protocol is secure, (c) assumes $\texttt{H}(\mathcal{B}_\gamma||...||\mathcal{B}_\varsigma)$ outputs a random value even if β miners try to influence its output. Note that in [44] only the permanent tags are used, and since the client generates the challenges and performs the verification, it does not use other primitives; hence, it does not require other security assumptions. As proven in Theorems 1 and 2, CR-TLP protocol is secure. Moreover, as analysed and proven in [1, 3], an output of $\texttt{H}(\mathcal{B}_\gamma||...||\mathcal{B}_\varsigma)$ is random value even if some blocks are generated or selectively disseminated by malicious miners. We conclude that the extractor can extract the file when $C' = \mathcal{S}$, and the extractor is interacting a ϵ-admissible prover. Now we move on to the case where $C' \subseteq \{\mathcal{S}, \mathcal{M}_1, ..., \mathcal{M}_{\beta-1}\}$. In this case, the proof (provided by $\mathcal{S}$) is rejecting but the corrupt miners may try to make $\texttt{Verify}(.)$ output 1. Note, if they succeed to do so, then the file can be extracted only by using the permanent tags but not the disposable ones. However, as shown above (i.e. due to the verification correctness), they have a negligible probability of success, when the blockchain is secure.

Υ-real-time Detection (according to Definition 9). In the following, we argue that after the server broadcasts a proof at a certain time, say t, to the network, the client can get a correct output of $\texttt{Verify}(.)$ at most after time period Υ, by a means of reading the blockchain. The proof is split into two parts: (a) correctness of $\texttt{Verify}(.)$ output, and (b) the maximum delay on the client's view of the output. Since we have already shown above that (when C or C' is corrupt) the correctness of $\texttt{Verify}(.)$ output is guaranteed, we focus on the latter property, i.e. the delay. To describe the delay, we need to recall two blockchain notions: *liveness* and *slackness* [20, 6]. Informally, liveness states that an honestly generated transaction will eventually be included more than $\texttt{k}$ blocks deep in an honest party's blockchain [20]. It is parameterised by wait time: $\texttt{u}$ and depth: $\texttt{k}$. We can fix the parameters as follows. We set $\texttt{k}$ as the minimum depth of a block considered as the blockchain's *state* (i.e. a part of the blockchain that remains unchanged with a high probability, e.g. $\texttt{k} \geq 6$) and $\texttt{u}$ the waiting time that the transaction gets $\texttt{k}$ blocks deep. As shown in [6], there is a slackness on honest parties' view of the blockchain. In particular, there is no guarantee that at any given time, all honest miners have the same view of the blockchain, or even the state. But, there is an upper-bound on the slackness, denoted by $\texttt{WindowSize}$, after which all honest parties would have the same view on a certain part of the blockchain state. This means when an honest party (e.g. the server) propagates its transaction (containing the proof) all honest parties will see it on their chain after at most: $\Upsilon = \texttt{WindowSize} + \texttt{u}$ time period. So when the adversary corrupts C or C', but in the latter case the server constructs a valid transaction (regardless of the proof status) the client by reading the blockchain (i.e. probing the miners) can get a correct result after at most time period Υ when the server sends the proof. Also, when parties in C' are corrupt and the transaction (containing the proof) is not valid, as discussed above (for verification correctness) the honest miners would detect the invalid transaction and do not include in the chain, therefore the output of $\texttt{Verify}(.)$ would be the same as its default value: 0; the same holds when the server sends nothing to the network. This concludes the proof related to Υ-real-time detection in SO-PoR protocol.

Fair Payment (according to Definition 10). The proof takes into consideration that the correctness of $\texttt{Verify}(.)$ output is guaranteed (as shown above). It boils down to the correctness of $\texttt{Pay}(.)$ as it is interrelated to $\texttt{Verify}(.)$ output. In particular, for the adversary to make inequality 2 not hold, it has to break $\texttt{Pay}(.)$ correctness, e.g. pays the server despite the verification function outputs 0. Therefore, it would suffice to show the adversary who corrupts C or C' cannot affect $\texttt{Pay}(.)$ output's correctness. To do so, we can apply the same argument used to prove the correctness of $\texttt{Verify}(.)$ above, as the correctness of $\texttt{Pay}(.)$ relies on the security of the blockchain as well. $\square$

6.5 Reducing Smart Contract Storage Cost to Constant

With minor adjustments, we can reduce the smart contract storage cost from $O(z)$ to constant, $O(1)$ and offload the cost to the server. The idea is that the client after computing the commitmnet vector: $\overrightarrow{\boldsymbol{h}} = [h_1, ...h_z]$, in step 2d, it preserves the ordering of the elements (i.e. h_j is associated with j^{th} verification) and constructs a Merkle tree on top of them. It stores the tree and the vector on the server, and stores only the tree's root: R, on the contract. In this case, the server in step 3e after recovering $\rho_j = (l_j, d_j)$, computes: $h_j = \mathtt{H}(l_j || d_j)$, and sends a Merkle tree proof (that h_j corresponds to R) along with ρ_j to the contract. In step 4b, the contract: (a) checks if $h_j = \mathtt{H}(l_j || d_j)$, and (b) verifies the Merkle tree proof. The rest remains unchanged. As a result, the number of values stored in the contract is now $O(1)$. This adjustment comes with an added communication cost: $O(|h_j| \log z)$ for each verification. Nevertheless, the added cost is small and independent of the file size. For instance, when $z = 10^6$ and $|h_j| = 256$, the added communication cost is only about 5.1 kilobit.

6.6 Evaluation

We evaluate SO-PoR by comparing its properties and costs to those protocols that support outsourced PoR (O-PoR), i.e. [3, 51]. In our cost analysis, we consider a generic case where a client outsources z verifications. We summarise the property and cost comparison results in Tables 3 and 4, respectively. Also, in our cost analysis, we compare SO-PoR cost with the cost of the most efficient privately verifiable PoR [44], too.

Table 3: O-PoR Property Comparison

Protocols	Properties			
	Real-time Detection	Fair Payment	Untrusted Auditor	Untrusted Client
SO-PoR	✓	✓	✓	×
[3]	×	×	✓	✓
[51]	×	×	×	×

Table 4: Outsourced PoR Cost Comparison

(a) Computation Cost

Protocols	Operation	Protocol Function			
		`Store`	`SolvPuz`	`Prove`	`Verify`
SO-PoR	Exp.	$z+1$	Tz	$-$	$-$
	Add. or Mul.	$2(n+cz)$	z	$4cz$	$2z(1+c)$
[3]	Exp.	$9n$	$-$	$-$	$-$
	Add. or Mul.	$10n$	$-$	$4z(c+c')$	$z(9c+3)$
[51]	Exp.	$-$	$-$	$z(3+c)$	$6z$
	Add. or Mul.	$4n$	$-$	$2z(3c+4)$	$2cz$
	Pairing	$-$	$-$	$7z$	$-$

(b) Communication Cost (in bit)

Protocols	Client	Server	Verifier	Proof Size
SO-PoR	$128(n+cz+19z)$	$884z$	$-$	$O(1)$
[3]	$128n$	$\lVert\vec{\boldsymbol{F}}\rVert + 256z$	$4672n+256z$	$O(1)$
[51]	$2048n$	$6144z$	$-$	$O(1)$

Properties. We start with a crucial feature that any O-PoR must have: real-time detection. Recall, real-time detection requires a client to receive a correct verification result in (almost) real-time without the need for it to re-execute the verification itself. This is offered only by SO-PoR. By contrast, in [3] the auditor may never notify the client, even if it does, its notification would not be reliable, and the client has to redo the verification to verify the auditor's claim. Similarly, in [51] the client has to fully trust the auditor to get notified on-time. So, [3, 51] are not suitable for the cases where a client must be notified by a potentially malicious auditor as soon as an unauthorised modification on the sensitive data is detected. The fair payment is another vital property in O-PoR, as the cloud server and auditor must be paid fairly, in the *real world* when they serve a client. This feature is explicitly captured by only SO-PoR. The other two protocols do not have any mechanism in place. In [3], one may allow the auditor to pay the server on the client's behalf. But, this is problematic. The server and auditor can collude to save costs, in a way that the server generates accepting proofs for the client but generates no proof for the auditor, and still the auditor pays it. This violates the fair payment and cannot be detected by the client unless it performs all the verification itself. On the other hand, a client in [51] has to fully trust the auditor (with the payment too), otherwise the auditor can collude with the server to violate the fair payment. Another important property is the cost of onboarding a new verifier, as it determines how flexible the client can be, to pick a new auditor when its current one is misbehaving. This cost in [3] is significantly high, as it requires the verifier to download the entire file, generate metadata and prove in zero-knowledge the correctness of metadata to the client. But, that cost in SO-PoR and [51] is very low as the client only sends them a small set of parameters without the need to access the outsourced data. Furthermore, as stated above, a client in [51] has to fully trust the auditor with the correctness of verification but this is not the case in SO-PoR and [3], as they consider a potentially malicious auditor (under different assumptions). Also, the latter protocol is the only outsourced PoR secure against a malicious client.

Computation Complexity. In our analysis, we do not take into account the cost of erasure-coding a file, as it is identical in all schemes. We first analyse the computation cost of SO-PoR. A client in step 2b performs n multiplications and n additions to generate permanent tags. In step 2c, it performs cz multiplications and cz additions to generate disposable tags for z verifications. The client in step 2d invokes $\texttt{GenPuz}(.)$ function, in CR-TLP, that costs $O(z)$. So, the client's total computation cost of preparing and storing a file is $O(cz)$. Now we consider the cost of cloud server that can be categorised into two classes: (a) solving a puzzle: $\texttt{SolvPuz}()$, run only once, and (b) generating PoR run for each verification. In particular, the cloud in step 3a, invokes $\texttt{SolvPuz}()$, in CR-TLP, that costs $O(Tz)$ this includes the cost in step 3e as well. To compute proofs, in step 3c, it performs $2cz$ multiplications and $2cz$ additions. So, the server to generate proof is $O(z)$. Next, we analyse the cost of the smart contract. In step 4b, it invokes $\texttt{Verify}(.)$, in CR-TLP, that in total costs $O(z)$, this involves invoking z hash function's instances. Also, the contract in step 4c performs $z(1+c)$ and $z(1+c)$ modular multiplications and additions respectively. Thus, the total cost is $O(cz)$ involving mainly modular additions and multiplications.

Now we analyse the computation cost of [3]. To prepare file tags, a client performs: n multiplications and n additions. Also, to verify tags generated by the auditor, the client has to engage in a zero-knowledge protocol that requires it to carry out $6n$ exponentiations and $2n$ multiplications. Therefore, the client's computation complexity is $O(n)$. Also, the auditor in total performs $3n$ multiplications, $3n$ additions and $3n$ exponentiations to prepare file's metadata, so its complexity at this phase is $O(n)$. For the cloud to generate z proofs, in total it performs $2z(c+c')$ multiplications and the same number of additions, where c' is the number of challenges sent by the auditor to the cloud (on client's behalf) and $c > c'$, e.g. $c' = (0.1)c$. So, the total complexity of the cloud is $O(z(c + c'))$. Next we consider the verification cost. The auditor performs $z(1 + c)$ multiplications and $z(1 + c)$ additions to verify PoR. It also performs $2cz$ additions in *CheckLog* algorithm that requires the client to perform $z(2c + 1)$ additions and cz multiplications. Nevertheless, as discussed in Section 2, running only *CheckLog* does not allow the client to detect a misbehaving auditor. Thus, it has to run *ProveLog* too, that requires the client to perform cz multiplications and cz additions and requires the auditor to reveal all its secrets to the client. So, the total verification complexity is $O(cz)$. Now we turn our attention to the computation complexity of [51]. To prepare metadata, the client needs to perform $2n$ multiplication and $2n$ additions, so the client's complexity is $O(n)$. For the cloud to generate a proof it needs to perform $3z$ exponentiations, $z(3c + 6)$ multiplications and $z(3c + 2)$ additions. So, its complexity is $O(cz)$. On the other hand, the verifier performs cz exponentiations to compute challenges. To verify the proof, it carries out $6z$ exponentiations, cz multiplications, cz additions, and $7z$ pairings. Therefore, the verifier complexity is $O(cz)$ dominated by expensive exponentiations and pairing operations. Also, we analyse the computation cost of efficient privately verifiable PoR in [44]. A client in the store phase performs n multiplications and n additions to construct the tags. So, its complexity is $O(n)$. A server performs $2cz$ multiplications and $2cz$ additions to generate proofs, so in total $4cz$ or $O(cz)$ modular operations for z verifications it carries out. The client, as verifier this time, performs in total $2z(1 + c)$ or $O(z(1 + c))$ modular operations.

Now we compare the protocols above. The verification in SO-PoR is much faster than the other two protocols; firstly, it requires no exponentiations in this phase, whereas [51] does, and secondly, it requires $\frac{9c+3}{2(1+c)}$ times fewer computation than [3]; Specifically, when[3] $c = 460$, SO-PoR verification requires about 4.5 times fewer computation than the verification in [3] needs. In SO-PoR, the cloud server needs to perform Tz exponentiations to solve puzzles, however this is independent of the file size. The other two protocols do not include the puzzle-solving procedure (and they do not offer all features that SO-PoR does). Furthermore, the proving cost in SO-PoR is similar to that of in [3], and is much better than [51], as the latter one requires both exponentiations and pairing operations while prove algorithm in SO-PoR does not involve any exponentiations. Also, the store phase in SO-PoR has a much lower computation cost than the one in [3]. The reason is that the number of exponentiations required (in this phase) in SO-PoR is independent of file size and is only linear with the number of delegated verifications; however, the number of exponentiations in [3] is linear with the file size. For instance, when $||\overrightarrow{\boldsymbol{F}}|| = $ 1-GB, the total number of blocks is: $n = \frac{1-\text{GB}}{128-\text{bit}} = 625 \times 10^5$. Since the number of exponentiations in [3] is linear with the number of blocks, i.e. $9n$, the total number of exponentiations imposed by store algorithm is: 5625×10^5 which is very high. This is the reason why in the experiment in [3] only a small file size: 64-MB, is used, that can be stored locally without the need to use cloud storage, in the first place. Now, we turn our attention to SO-PoR. Let the verification be done every month for a 10-year period, in this case, $z = 120$. So, the total number of exponentiations required by store in SO-PoR is 121. This means store phase in SO-PoR requires over

[3] As shown in [4], to ensure 99% of file blocks is retrievable, it would suffice to set $c = 460$.

46×10^5 times fewer exponentiations than the one in [3] needs. On the other hand, the store algorithm in [51] does not involve any exponentiations; however, its number of modular additions and multiplication is higher than the ones imposed by SO-PoR store. Also, the verification and prove cost of SO-PoR and privately verifiable PoR [44] are identical.

Communication Complexity. In our analysis, we do not take into account the communication cost of uploading an encoded file, i.e. $||\vec{\boldsymbol{F}}||$, when the client for the first time sends it to the cloud, as it is identical in all schemes. The communication cost of SO-PoR is as follows. The client, in step 2e, sends n permanent tags, zc disposable tags, and the output of $\texttt{GenPuz}(.)$ to the server and contract, where each (permanent/disposable) tag: $\sigma_j \in \mathbb{F}_p$ and $|\sigma_j| = 128$-bit. Note the client also sends a few public parameters: $\hat{pk}$, whose size is short. Therefore, the client's bandwidth is: $128(n + cz + 19z)$ bits, while its communication complexity is $O(n + cz)$. The cloud in step 3d, sends z pairs (μ_j, ξ_j), where $\mu_j, \xi_j \in \mathbb{F}_p$ and $|\mu_j| = |\xi_j| = 128$-bit. Also, in step 3e, it sends to the contract the output of $\texttt{Prove}(.)$, in CR-TLP, whose total size is $628z$ bits. So, the clouds total bandwidth is about $884z$ bits and its complexity is $O(z)$ which is independent of and constant in the file size.

The communication cost of [3] is as follows. The client sends n tags to the server, where the size of each tag is about 128 bits. So its bandwidth is $128n$, and its complexity is $O(n)$. The auditor also sends n tags to the server, where each tag size is also 128 bits. It also sends the tags to the client along with zk proofs that contain $4n$ elements in total, where $2n$ of them are elements of $\mathbb{Z}_N$ and each element size is 2048 bits, and each of the other $2n$ elements is 160 bits long. Also, the auditor in *ProveLog* sends z pairs to the client with the bandwidth of $256z$. So, the auditor's total bandwidth and complexity is $4672n + 256z$ and $O(n + z)$ respectively. Moreover, the cloud sends the entire file, F, to the auditor in the store phase and also sends $2z$ pairs of PoR to the auditor, where each element of the pair is of size 128 bits. Therefore, the cloud's total bandwidth is $||\vec{\boldsymbol{F}}|| + 256z$, while its complexity is $O(||\vec{\boldsymbol{F}}|| + z)$. Now, we analyse the communication cost of [51]. The client bandwidth and complexity are $2048n$ and $O(n)$ respectively, as it sends to the cloud $2n$ tags, where each tag size is 1024 bits. Also, the cloud bandwidth and complexity are $6144z$ and $O(z)$ respectively, as for each verification the cloud sends to the verifier 6 elements each of them is 1024-bit long. Furthermore, in [44] the client bandwidth in the store phase is $128n$, while the server bandwidth is $256z$. In this scheme, the complexity of a proof size is $O(1)$.

To conclude, the verifier-side bandwidth of SO-PoR (and [51]) is much lower than [3]. For instance, when $||\vec{\boldsymbol{F}}|| = 1$-GB and $z = 100$, a verifier in SO-PoR requires 62×10^6 fewer bits than the one in [3] does. The server-side bandwidth of SO-PoR is also lower than the other two protocols. For instance (for the same parameters above) a server in SO-PoR requires 9×10^4 and 7 times fewer bits than those required in [3] and [51] respectively. In general, the overall bandwidth of SO-PoR is much lower than [3], and is about $9\times$ higher than the outsourced PoR that requires a *trusted* verifier, i.e. [51], due to higher client-side bandwidth. Also, a client bandwidth in SO-PoR requires $128(cz + 19z)$ more bits than a client in the privately verifiable PoR [44], while the server's bandwidth in SO-PoR is 3.4 times higher than that in [44]. All schemes above have a constant proof size complexity.

Remark 9. In [3], the additional costs to secure parties against a malicious client stem from only the store phase, where an auditor downloads the entire file, generates zero-knowledge proofs and has the client sign them after verifying the proofs. Therefore, the overheads of proving and verifying phases, in this protocol, would remain unchanged if the protocol considers an honest client.

7 Conclusion

Time-lock puzzles are important cryptographic protocols with various applications. However, existing puzzle schemes are not suitable to deal with multiple puzzles at once. In this work, we put forth the concept of composing multiple puzzles; where given puzzles composition at once, a server can find one puzzle's solution after another. This process does not require the server to deal with all of them in parallel which reliefs the server from having numerous parallel processors and allows it to save considerable computation overhead. We proposed a candidate construction: chained RSA time-lock puzzle (CR-TLP) that possesses the aforementioned features. Furthermore, CR-TLP is equipped with an efficient verification algorithm publicly executable. In this work, we also showed how to use CR-TLP to construct an efficient outsourced proofs of retrievability scheme that supports *real-time detection* and *fair payment* while keeping its costs lower than the state of the art.

In the future, we would like to investigate how multiple puzzles originated from *distinct clients* can be composed while offering the same features, to the server, as CR-TLP does. Exploring other applications of CR-TLP would be another interesting future research direction.

References

1. Abadi, A., Ciampi, M., Kiayias, A., Zikas, V.: Timed signatures and zero-knowledge proofs -timestamping in the blockchain era-. IACR Cryptology ePrint Archive 2019, 644 (2019)
2. Armknecht, F., Barman, L., Bohli, J., Karame, G.O.: Mirror: Enabling proofs of data replication and retrievability in the cloud. In: 25th USENIX Security 16
3. Armknecht, F., Bohli, J.M., Karame, G.O., Liu, Z., Reuter, C.A.: Outsourced proofs of retrievability. In: CCS'14
4. Ateniese, G., Burns, R.C., Curtmola, R., Herring, J., Kissner, L., Peterson, Z.N.J., Song, D.X.: Provable data possession at untrusted stores. In: CCS'07
5. Ateniese, G., Pietro, R.D., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: SECURECOMM'08
6. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: Katz, J., Shacham, H. (eds.) CRYPTO'17
7. Banerjee, P., Nikam, N., Ruj, S.: Blockchain enabled privacy preserving data audit. CoRR abs/1904.12362 (2019)
8. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Advances in Cryptology - CRYPTO '96

9. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO'18
10. Boneh, D., Naor, M.: Timed commitments. In: Bellare, M. (ed.) CRYPTO 2000
11. Brakerski, Z., Döttling, N., Garg, S., Malavolta, G.: Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In: TCC'19,
12. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS'17
13. Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious ram. J. Cryptol. (2017)
14. Chen, H., Deviani, R.: A secure e-voting system based on RSA time-lock puzzle mechanism. In: BWCCA'12 ,
15. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Brickell, E.F. (ed.) CRYPTO'92
16. Erway, C.C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: CCS'09
17. Etemad, M., Küpçü, A.: Transparent, distributed, and replicated dynamic provable data possession. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS'13
18. Francati, D., Ateniese, G., Faye, A., Milazzo, A.M., Perillo, A.M., Schiatti, L., Giordano, G.: Audita: A blockchain-based auditing framework for off-chain storage. CoRR'19
19. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Blaze, M. (ed.) FC'02
20. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol with chains of variable difficulty. In: Katz, J., Shacham, H. (eds.) CRYPTO'17
21. Garay, J.A., Kiayias, A., Panagiotakos, G.: Iterated search problems and blockchain security under falsifiable assumptions. IACR Cryptology ePrint Archive (2019)
22. Groza, B., Petrica, D.: On chained cryptographic puzzles. In: SACI'06,
23. Hao, K., Xin, J., Wang, Z., Jiang, Z., Wang, G.: Decentralized data integrity verification model in untrusted environment. In: APWeb-WAIM'18
24. Juels, A., Jr., B.S.K.: Pors: proofs of retrievability for large files. In: CCS'07
25. Kamara, S.: Proofs of storage: Theory, constructions and applications. In: CAI'13
26. Karame, G., Capkun, S.: Low-cost client puzzles based on modular exponentiation. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS '10
27. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)
28. Kopp, H., Bösch, C., Kargl, F.: Koppercoin - A distributed file storage with financial incentives. In: ISPEC'16
29. Kopp, H., Mödinger, D., Hauck, F.J., Kargl, F., Bösch, C.: Design of a privacy-preserving decentralized file storage with financial incentives. In: EuroS&P Workshops'17
30. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: S&P'16
31. Kupcu, A.: Efficient cryptography for the next generation secure cloud. Brown University (2010)
32. Kuppusamy, L., Rangasamy, J., Stebila, D., Boyd, C., Nieto, J.M.G.: Practical client puzzles in the standard model. In: Youm, H.Y., Won, Y. (eds.) ASIACCS '12
33. Labs, P.: Filecoin: A decentralized storage network (2017), https://filecoin.io/filecoin.pdf
34. Luu, L., Teutsch, J., Kulkarni, R., Saxena, P.: Demystifying incentives in the consensus computer. In: Ray, I., Li, N., Kruegel, C. (eds.) CCS'15
35. Ma, M.: Mitigating denial of service attacks with password puzzles. In: ITCC'05
36. Mahmoody, M., Moran, T., Vadhan, S.P.: Time-lock puzzles in the random oracle model. In: CRYPTO'11

37. Malavolta, G., Thyagarajan, S.A.K.: Homomorphic time-lock puzzles and applications. In: CRYPTO'19
38. Miller, A., Juels, A., Shi, E., Parno, B., Katz, J.: Permacoin: Repurposing bitcoin work for data preservation. In: S&P'14
39. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91
40. Renner, T., Müller, J., Kao, O.: Endolith: A blockchain-based framework to enhance data retention in cloud storages. In: PDP'18
41. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Tech. rep. (1996)
42. Ruj, S., Rahman, M.S., Basu, A., Kiyomoto, S.: Blockstore: A secure decentralized storage framework on blockchain. In: AINA'18
43. Sengupta, B., Ruj, S.: Keyword-based delegable proofs of storage. In: ISPEC'18
44. Shacham, H., Waters, B.: Compact proofs of retrievability. In: ASIACRYPT. pp. 90–107 (2008)
45. Shen, S., Tzeng, W.: Delegable provable data possession for remote data in the clouds. In: ICICS 2011
46. Shi, E., Stefanov, E., Papamanthou, C.: Practical dynamic proofs of retrievability. In: CCS'13
47. Vorick, D., Champine, L.: Sia: Simple decentralized storage. Nebulous Inc (2014)
48. Waters, B., Juels, A., Halderman, J.A., Felten, E.W.: New client puzzle outsourcing techniques for dos resistance. In: CCS'04
49. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT'19
50. Wilkinson, S., Boshevski, T., Brandoff, J., Buterin, V.: Storj: a peer-to-peer cloud storage network (2014)
51. Xu, J., Yang, A., Zhou, J., Wong, D.S.: Lightweight delegatable proofs of storage. In: ESORICS'16
52. Xue, J., Xu, C., Bai, L.: Dstore: A distributed system for outsourced data storage and retrieval. Future Generation Comp. Syst. 2019
53. Xue, J., Xu, C., Zhang, Y., Bai, L.: Dstore: A distributed cloud storage system based on smart contracts and blockchain. In: ICA3PP'18
54. Yao, A.C.: Protocols for secure computations (extended abstract). In: FOCS'1982
55. Zhang, Y., Deng, R.H., Liu, X., Zheng, D.: Blockchain based efficient and robust fair payment for outsourcing services in cloud computing. Inf. Sci. (2018)

A Traditional RSA Time-lock Puzzle

Below, we provide the definition and proof of the RSA puzzle scheme proposed in [41].

Definition 12 (Time-lock Puzzle). *A time-lock puzzle comprises the following efficient three algorithms, such that the puzzle satisfies completeness and efficiency properties.*

- ***Algorithms***:
  - $\texttt{R-TLP.Setup}(1^\lambda, \Delta) \rightarrow (pk, sk)$: *a probabilistic algorithm that takes an input a security:* 1^λ *and time:* Δ *parameters. It outputs a public-private key pairs:* (pk, sk).
  - $\texttt{R-TLP.GenPuz}(s, pk, sk) \rightarrow \Theta$: *a probabilistic algorithm that takes an input a solution:* s *and the public-private key pairs:* (pk, sk). *It outputs a puzzle:* Θ.

- $\texttt{R-TLP.SolvPuz}(pk, \Theta) \rightarrow s$: *a deterministic algorithm that takes as input a the public key: pk and puzzle: Θ. It outputs a solution: s.*

– ***Completeness***: *it always holds* $\texttt{R-TLP.SolvPuz}(pk, \texttt{R-TLP.GenPuz}(s, pk, sk)) = s$

– ***Efficiency***: *the run-time of algorithm* $\texttt{R-TLP.SolvPuz}(pk, \Theta)$ *is bounded by* $poly(\Delta, \lambda)$, *where* $poly(.)$ *is a fixed polynomial.*

Informally, a time-lock puzzle's security requires that the puzzle solution remain hidden from all adversaries running in parallel within the time period, Δ. It is essential that no adversary can find a solution (in particular distinguishes two puzzles computed on solutions provided by the adversary) in time $\delta(\Delta) < \Delta$, utilising up to many processors running in parallel and after a potentially large amount of pre-computation. In other words, it is critical to bound the adversary's allowed parallelism. Therefore, such factors are explicitly incorporated into the puzzle's definitions [9, 37, 21].

Definition 13 (Time-lock Puzzle Security). *A time-lock puzzle is secure if for all λ and Δ, all probabilistic polynomial time adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ where $\mathcal{A}_1$ runs in total time $O(poly(\Delta, \lambda))$ and $\mathcal{A}_2$ runs in time $\delta(\Delta) < \Delta$ using at most $\pi(\Delta)$ parallel processors, there exists a negligible function $\mu(.)$, such that:*

$$Pr\left[\mathcal{A}_2(pk, \Theta, state) \rightarrow b \;\middle|\; \begin{array}{l} \texttt{R-TLP.Setup}(1^\lambda, \Delta) \rightarrow (pk, sk) \\ \mathcal{A}_1(1^\lambda, pk, \Delta) \rightarrow (s_0, s_1, state) \\ b \overset{\$}{\leftarrow} \{0, 1\} \\ \texttt{GenPuz}(s_b, pk, sk) \rightarrow \Theta \end{array}\right] \leq \frac{1}{2} + \mu(\lambda)$$

In general, time-lock puzzles by definition are sequential functions. In their construction, it is required a function inherently sequential, e.g. modular squaring, is called iteratively for a certain number of times. The notions of sequential function and iterated sequential functions, in the presence of an adversary possessing a polynomial number of processors, have been generalised and defined in [9]. Below, for the sake of completeness, we provide those definitions.

Definition 14 $(\Delta, \delta(\Delta))$-Sequential function). *For a function: $\delta(\Delta)$, time parameter: Δ and security parameter: $\lambda = O(\log(|X|))$, $f : X \rightarrow Y$ is a $(\Delta, \delta(\Delta))$-sequential function if the following conditions hold:*

- *There exists an algorithm that for all $x \in X$evaluates f in parallel time Δ using $poly(\log(\Delta), \lambda)$ processors.*
- *For all adversaries $\mathcal{A}$ that run in parallel time strictly less than $\delta(\Delta)$ with $poly(\Delta, \lambda)$ processors:*
$$Pr\left[y_A = f(x) \middle| y_A \overset{\$}{\leftarrow} \mathcal{A}(\lambda, x), x \overset{\$}{\leftarrow} X\right] \leq negl(\lambda)$$
where $\delta(\Delta) = (1 - \epsilon)\Delta$ and $\epsilon < 1$.

Definition 15 (Iterated Sequential function). *Let $g : X \rightarrow X$ be a $(\Delta, \delta(\Delta))$-sequential function. A function $f : \mathbb{N} \times X \rightarrow X$ defined as $f(k, x) = g^{(k)}(x) = \underbrace{g \circ g \circ ... \circ g}_{k}$ is an iterated sequential function, with round function g, if for all $k = 2^{o(\lambda)}$ the function $h : X \rightarrow X$ defined by $h(x) = f(k, x)$ is $(k\Delta, \delta(\Delta))$-sequential.*

The main property of an iterated sequential function is that iteration of the round function g, is the fastest way to evaluate the function. Iterated squaring in a finite group of unknown order, is widely believed to be a candidate for an iterated sequential function. Its definition is as follows.

Assumption 1 (Iterated Squaring)**.** *Let N be a strong RSA modulus, r be a generator of $\mathbb{Z}_N$, Δ be a time parameter, and $T = poly(\Delta, \lambda)$. For any $\mathcal{A}$, defined above, there is a negligible function $\mu(.)$ such that:*

$$Pr\left[\mathcal{A}(N, r, y) \rightarrow b \left| \begin{array}{l} r \stackrel{\$}{\leftarrow} \mathbb{Z}_N, b \stackrel{\$}{\leftarrow} \{0,1\} \\ \textit{if } b = 0,\ y \stackrel{\$}{\leftarrow} \mathbb{Z}_N \\ \textit{otherwise } y = r^{2^T} \end{array} \right.\right] \leq \frac{1}{2} + \mu(\lambda)$$

Theorem 5 (R-TLP Security). *Let N be a strong RSA modulus and Δ be the period within which the solution/secret remains hidden. If the sequential squaring assumption holds, factoring N is a hard problem and the symmetric key encryption is semantically secure, then R-TLP (presented in Section 3.4) is a secure time-lock puzzle.*

sketch. Let a solver be $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ runs in total time $O(poly(\Delta, \lambda))$ and $\mathcal{A}_2$ runs in time $\delta(\Delta) < \Delta$ using at most $\pi(\Delta)$ parallel processors. For the solver to find the secret significantly earlier than $\delta(\Delta)$, given the public parameters, it needs to either: (a) compute $\phi(N)$, so it can generate the blinding factor: b as fast as it is done in the encryption phase, or (b) break the symmetric key encryption, or (c) extract k from θ_2 by finding the blinding factor without performing a sufficient number of squaring (and without knowing $\phi(N)$). However, finding $\phi(N)$ is as hard as factoring N, also as long as the encryption is secure and k is sufficiently large it cannot break the symmetric key encryption. Moreover, the blinding factor is a uniformly random element of the ring (due to Assumption 1), so it prevents the solver from finding the blinding factor without carrying out enough squaring within time significantly less than $\delta(\Delta)$. Thus, any adversary $\mathcal{A}$ cannot find the secret without carrying out a sufficient number of squaring that would take it $\delta(\Delta)$. □

B Traditional PoR Model

A PoR scheme comprises five algorithms:

- $\texttt{Setup}(1^\lambda) \rightarrow sk$: a probabilistic algorithm, run by a client, that takes an input a security: 1^λ and outputs a secret key.

- $\texttt{Store}(sk, F) \rightarrow (F^*, \sigma)$: a probabilistic algorithm, run by a client, that takes an input the secret key: sk and a file: F. It encodes F, denoted by F^* as well as generating a set of tags: σ, where F^* and σ is stored on the server.

- $\texttt{GenChal}(|F^*|, 1^\lambda) \rightarrow \vec{c}$: a probabilistic algorithm, run by a client, that takes an input the encoded file size: $|F^*|$ and security: 1^λ. It outputs a set of pairs: $c_j : (x_j, y_j)$, where each pair includes a file block index: x_j and coefficient: y_j, both of them are picked uniformly at random.

- $\texttt{Prove}(F^*, \sigma, \vec{\boldsymbol{c}}) \rightarrow \pi$: takes the encoded file: F^*, (a subset of) tags: σ, and a vector of unpredictable random challenges: $\vec{\boldsymbol{c}}$ as inputs and outputs a proof of the file retrievability. It is run by a server.

- $\texttt{Verify}(sk, \vec{\boldsymbol{c}}, \pi) \rightarrow \{0, 1\}$: takes the secret key: sk, vector of random challenges: $\vec{\boldsymbol{c}}$, and the proof π as inputs. It outputs either 0 if it rejects, or 1 if it accepts the proof. It is run by a client.

Informally, a PoR scheme has two main properties: correctness and soundness. The correctness requires that for any key, all files, the verification algorithm accepts a proof generated by an honest verifier. The soundness requires that if a prover convinces the verifier (with a high probability) then the file is actually 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. In contrast to the definition in [44] where $\texttt{GenChal}(.)$ is implicit, in the above we have explicitly defined it, as its modified version plays an integral role in SO-PoR definition (and protocol).