Chapter 17 Smarter Data Availability Checks in the Cloud: Proof of Storage via Blockchain

Aydin Abadi
University College London, UK

ABSTRACT

Cloud computing offers clients flexible and cost-effective resources. Nevertheless, past incidents indicate that the cloud may misbehave by exposing or tampering with clients' data. Therefore, it is vital for clients to protect the confidentiality and integrity of their outsourced data. To address these issues, researchers proposed cryptographic protocols called "Proof of Storage" that let a client efficiently verify the integrity or availability of its data stored in a remote cloud server. However, in these schemes, the client either has to be online to perform the verification itself, or has to delegate the verification to a fully trusted auditor. In this chapter, a new scheme is proposed that lets the client distribute its data replicas among multiple cloud servers to achieve high availability, without the need for the client to be online for the verification, and without a trusted auditor's involvement. The new scheme is mainly based on blockchain smart contracts. It illustrates how a combination of cloud computing and blockchain technology can resolve real-world problems.

INTRODUCTION

The importance of cloud computing is swiftly growing. The cloud is receiving increasing attention (Luxner, 2021, March 15; Abadi, 2017), as it enables ubiquitous access to a pool of configurable computing resources that can be scaled up, on demand. It offers elastic and cost-effective storage and computation resources to clients. It has been drawing the attention of individuals and businesses as a vital game-changing technology. There are various benefits for businesses to use the cloud, such as cost flexibility, business scalability, and increased collaboration with external partners (Berman et al., 2012). Nevertheless, the cloud is susceptible to data security breaches such as exposing confidential data, data tampering, and denial of service. Thus, it cannot be fully trusted, and it is crucial for the clients who use the cloud to protect the security of their own data.

DOI: 10.4018/978-1-7998-7712-7.ch017

To address these issues, researchers have proposed Proof of Storage (PoS). It is an interesting cryptographic protocol that allows a client (e.g., a computer system acting on behalf of a party) to efficiently verify the integrity or availability of its data that is stored in a remote cloud server, which is not necessarily trusted (Kamara, 2013). In general, PoS schemes can be classified into two distinct categories; namely, Proofs of Retrievability (PoR) (proposed by Juels and Kaliski, 2007) and Proofs of Data Possession (PDP) (proposed by Ateniese et al., 2007). The former variant offers a stronger security guarantee than the latter, because a PoR scheme guarantees that the entire file is available whereas a PDP scheme guarantees that only a portion of a file remains intact in a remote server. The schemes that offer stronger security guarantees (i.e., PoR) are the main focus of this chapter. Since, in traditional PoR schemes, a client has to either perform the verification itself or delegate it to a fully trusted third-party, researchers proposed *outsourced* PoR schemes that let a client delegate the verifications, without having to fully trust a single entity. An efficient outsourced PoR scheme has recently been put forward by Abadi and Kiayias (2021, March 4). The scheme uses the decentralised nature of the blockchain (and smart contracts) to eliminate the involvement of a single trusted third-party. It allows a client to outsource its data to a single server, and lets the client delegate the verification of its data availability to a smart contract, which can periodically check data availability on the client's behalf.

In this chapter, it will be shown how we can improve upon the state-of-the-art outsourced PoR. In particular, a new variant of the PoR that lets a client store and distribute replicas of its sensitive data among *multiple cloud servers,* is discussed. The new PoR variant does not require the client to be always available to perform data availability checks itself. Instead, a smart contract efficiently performs the checks, on the client's behalf, and pays the servers if they successfully prove to the smart contract that the data is available. In the new scheme, the time intervals between two consecutive verifications can have different sizes which makes this scheme more flexible. To do that, it will be shown how the "chained time-lock puzzle" scheme, that is used in the scheme proposed by Abadi and Kiayias (2021, March 4), can be modified to support different size, time intervals. The modified chained time-lock puzzle scheme will be used in the multi-server outsourced PoR protocol. Thus, there are two primary properties that the multi-server outsourced PoR scheme offers, compared with the state-of-the-art PoR protocol, i.e., supporting (a) multiple cloud servers, and (b) allowing different size time intervals. The proposed scheme is mainly based on symmetric-key primitives that leads to an efficient implementation. The scheme imposes low costs, especially at the verification phase, while preserving all appealing features of the state-of-the-art protocol.

RELATED WORK

PoS is a cryptographic protocol that has been studied for over a decade. It allows a client to efficiently check the integrity or availability of its data stored in a remote cloud server that can potentially be malicious. PoR schemes ensure that the server maintains knowledge of the client's *entire* outsourced data, whereas PDP schemes only ensure the server is storing most of the client's data. Moreover, each PoR and PDP scheme can be grouped into two categories; namely, publicly and privately verifiable. In the former group, everyone without knowing a secret key can verify proofs, while the latter category requires a verifier to have the knowledge of a secret key. The PoR notion was first put forward and defined by Juels and Kaliski (2007), who designed a protocol that uses 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 block of the file it wants to outsource, and then encrypts each encoded block separately. Next, it computes a set of random sentinels and appends them to the encrypted file. It randomly shuffles all values and sends the result 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. After that, the client checks if it gets the sentinels it asked for. In this scheme, the security holds, as the server cannot distinguish between sentinels. But, as an individual sentinel is only one-time verifiable, there is an upper bound on the number of verifications that the client can perform. When the limitation is reached, the client has to download, re-encode the file, and upload the new version of the encoded file.

To overcome the issues related to the bounded number of verifications, Juels and Kaliski (2007) suggested that sentinels can be replaced with message authentication code (MAC) on every file block or a Merkle tree constructed on the file blocks. The sentinel and MAC-based schemes stated above are privately verifiable. However, the scheme that uses a Merkle tree supports public verifiability. The publicly verifiable Merkle tree-based protocol has a communication cost logarithmic with the file size, and the prover has to send a set of file blocks to the verifier. This results in a high communication cost. Shacham and Waters (2008) improved the sentinel-based scheme and the formal definition of the PoR. They proposed two PoR protocols, one using MAC and the other one being based on Boneh-Lynn-Shacham (BLS) signatures. Specifically, the client in the 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 challenges/indices related to the file's blocks. Given the client's message, the server generates and sends a compact proof to it. The protocol that relies on MAC supports efficient private verification. Nevertheless, the one that uses BLS signatures supports public verifiability at the cost of public-key operations, which is less efficient than the one that uses MAC. Since then, various other PoR schemes have been proposed.

Armknecht et al. (2014) and Xu et al. (2016) have proposed *outsourced* PoR protocols that let clients outsource the verification to a third-party auditor, who is not necessarily trusted. The protocol proposed by Armknecht et al. (2014) uses MAC-based tags, zero-knowledge proofs, and error-correcting codes. At a high level, the protocol works as follows. At the setup phase, the client encodes its data using error-correcting codes, generates MAC on every block of the file, and stores the encoded file and tags in a server. Next, the auditor downloads the encoded file and generates another set of MACs on the file blocks. It separately uploads the MACs to the cloud. The auditor also proves to the client in zero-knowledge that it has created each MAC correctly. To do that, it uses a non-interactive zero-knowledge proofs that are sent directly to the client. If the client accepts all proofs, it signs every proof and sends the signatures to the auditor. In the verification phase, the auditor sends two sets of random challenges that are 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 keeps the client's proofs. In the case where the auditor's proof is not accepted, an honest auditor would inform the client, who will come online and check both its proofs and the auditor's proof. The scheme provides two layers of verification to the client: CheckLog and ProveLog. The CheckLog is more efficient, as the auditor sends far 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, however it cannot identify a responsible party at this stage. To pinpoint the malicious party, it needs to proceed to the ProveLog that allows the client to audit the auditor. In this verification process, 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. Even though the protocol offers a set of interesting features, it has several shortcomings, e.g., (1) the auditor can get a free ride when a known highly reputable server generates accepting proofs for both client and auditor - an economically rational auditor can skip performing its part and still get paid by the client, and (2) it offers no guarantee for a real-time detection (or notification) as the client may not be notified in real-time about the data unavailability if the auditor is malicious.

Recently, proof of storage-time (PoSt) has been designed by Ateniese et al. (2020). The authors presented two proof of storage protocols; namely, basic PoSt and compact PoSt. Briefly, these protocols offer the guarantee to a client that its outsourced data remains available on a storage server for a predefined time period, T. The idea behind the protocol's design is that a client uploads to a server its data once, then the server generates proofs of storage, such as PoR, on a regular basis, collects them and sends the collection after the time, T, to a verifier who verifies the proof's correctness. The basic PoSt protocol uses Merkle tree-based PoR and a Verifiable Delay Function (VDF). In this protocol, the client generates a set of metadata. It sends the file, metadata, and a challenge to the server. The server uses the challenge to generate a PoR. It feeds the hash of the PoR to VDF to generate an output. It considers the hash of VDF's output as the next challenge from which the next PoR is generated, so a set of challenges can be generated without the client's involvement. This process continues until all "z" PoRs are computed, where "z" is the total number of required proofs or verifications. The server sends all PoRs along with the proofs proving the correctness of VDF's outputs to a verifier. Given the two sets of proofs, the verifier verifies their correctness and concludes that the file is indeed retrievable within the defined time period. The scheme is publicly verifiable; however, the verification's computation cost is very high, because it requires the verifier to validate the correctness of VDF's outputs, that imposes a high cost to the verifier. Although the authors suggest a smart contract can play the validator's role, it seems that this would impose a significant financial cost to the smart contract and users, due to very high costs that stem from the verification and prove algorithms. On the other hand, the compact PoSt lets the server combine PoR proofs which would reduce the communication cost. This protocol's design is very similar to the basic PoSt, with a major difference being that it replaces the Merkle tree approach with simply hashing the entire file and, replaces the VDF with a trapdoor delay function (TDF), which needs a secret to generate and verify the delay function's output. This means the compact PoSt protocol is only privately verifiable. The paper states that the verification in this protocol can be performed by a trusted third-party such as a smart contract. However, this is not the case, because the scheme requires a set of secrets to perform the verification, while the well-known smart contracts (e.g., Ethereum ones) do not maintain a private state. If the challenges are given to the contract, then the server can read the contract and compute all proofs in one go, which ultimately violates the security requirement set out in the paper. Also, the same security issue would arise in the case where the verification is delegated to a third- party auditor who may collude with the server, as the auditor can send all challenges to the server at once. Thus, the only secure option, in the compact PoSt, is that the verifier performs the validation itself.

Hence, the above 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, or are very inefficient. To address these issues, an outsourced PoR has been proposed very recently by Abadi and Kiayias (2021, March 4). It allows a client to efficiently delegate verifications to a smart contract and addresses the previous work issues. The protocol is highly efficient. It is mainly based on cryptographic commitment, chained time-lock puzzle, smart contract, and symmetric-key encryption schemes. Nevertheless, the scheme only supports a single server and the time intervals between consecutive verifications have to have an identical size, which can be considered a limitation. Since the work of Abadi and Kiayias is the focus of this chapter, their work is explained in more detail later.

Moreover, researchers have proposed distributed PoR schemes that use a (tailored) blockchain, and let data be distributed to various storage servers to gain robustness, and address the single point of failure issue. The Permacoin scheme (proposed by Miller et al., 2014) is one of those that distribute data among customised blockchain nodes and repurpose mining resources of Bitcoin blockchain miners. In the 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. In this scheme, the miner needs to invest in both computation and storage resources to generate PoR. The protocol uses a Merkle tree that is built on top of file blocks to support publicly verifiable PoR. The mining procedure in this scheme relies on the iterative hashing. In the Permacoin, the prover has to send both challenged file blocks and the proof path in the tree to the verifiers, which 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 needs to either wait for a sufficiently long period of time, or hope that there are enough active miners in each epoch. The Filecoin and KopperCoin (proposed by Protocol Lab, 2017, and Kopp et al., 2016 respectively) offer similar features. Nevertheless, the Filecoin, in addition to a Merkle tree, uses generic zero-knowledge proofs that yields a high overall computation cost, and requires a trusted party to generate the system parameters. The Filecoin uses proof of work as well as PoR. Nevertheless, the KopperCoin uses a PoR scheme that is based on BLS signatures that results in a constant communication cost (due to the homomorphic property of the signatures), but it imposes a high computation cost.

In the same line of research, Kopp et al. (2017) presented a tailored blockchain that supports distributed PoR and preserves the privacy of on-chain payments between storage users and providers. However, the scheme is computationally expensive because it uses publicly verifiable tags that rely on BLS signatures for PoR, and ring signatures for the privacy- preserving payments. Similarly, Ruj et al. (2018) proposed a distributed PoR scheme that uses BLS signatures and a Merkle tree, as well as a smart contract for payments. Furthermore, Xue at el. (2019) designed schemes that allow a user to store its encrypted data in the blockchain. The scheme uses a Merkle tree for PoR, and a smart contract to transfer fees to the blockchain nodes storing the data. In this scheme, the data owner is the party which sends random challenges to storage nodes, so it has to be online at the time of verification. The Storj scheme (proposed by Wilkinson et al., 2014) 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 the Storj, there exist trusted nodes, called "Satellites", which perform the verifications the clients' behalf. Thus, the existing distributed PoR protocols have either a large proof size, or impose a high verification overhead. Furthermore, they do not guarantee that the entire file is retrievable in real-time.

To weaken the security assumption that an auditor is fully trusted with the correctness of verification in the publicly verifiable PoR schemes, while storing data off-chain, researchers have proposed various protocols, which are briefly explained here. The PoR scheme proposed by Renner et al. (2018) requires only a hash of the entire file to be stored in a smart contract. Later, when the user accesses the file, it computes the hash of the file, and compares it with the one stored in the smart contract. In this scheme, the entire file needs to be accessed by the user for each verification which is its main weakness, as it imposes a high input/output (I/O) cost. The protocol designed by Hao et al. (2018) uses BLS signature-based tags and a Merkle tree, where the tags are broadcast to all blockchain nodes. Nevertheless, the protocol assumes the cloud server is fully trusted and stores data safely. This is a very strong assumption and limits its real-world application. In this protocol, the tags are computed by the cloud server that sends them to the nodes. The tags are never verified against the outsourced data. The only security guarantee that this scheme offers is the tags' immutability. Another high-level scheme presented by Zhang et al. (2018) lets a client 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 must be online and send random challenges to the server for every verification itself. In the same line of research, Francati et al. (2019) designed the Audita scheme, which uses a customised blockchain and RSA signature-based proof of data possession scheme to achieve its goal. In this scheme, miners, for each epoch, pick a dealer who sends a challenge to the storage nodes to get proofs of data possession. The Audita substitutes proof of work with proof of data possession, so if a proof is accepted, then a new block is added to the chain. In the Audita, every miner carries out expensive public key-based verifications. The Sia, proposed by Vorick and Champine (2014), is a mechanism in which a data owner distributes its data among off-chain storage servers, which provide a PoR to a smart contract on a regular basis. Each server receives a fixed amount of coins if its proof is accepted by the smart contract. This scheme uses a Merkle tree-based PoR which often imposes a high communication cost.

It is evident that 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 and computation cost, or clients must be online for each verification. Hence, to date the scheme of Abadi and Kiayias (2021, March 4) is the only protocol that efficiently supports delegated PoR. In this chapter, we will show how this scheme's aforementioned limitations can be addressed.

CRYPTOGRAPHIC TOOLS USED IN THE STUDY

In this section, the main cryptographic tools used in this study are outlined.

Time-lock Puzzle Scheme

Time-lock puzzles are 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. The idea to send information into the future, i.e., a time-lock puzzle, was first put forward by May (1993). The scheme that May proposed relies on a trusted agent. Later, Rivest et al. (1996) proposed an RSA-based puzzle scheme that does not require a trusted agent, and is secure against a receiver who may have access to many computation resources that run in parallel. Here, the time-lock puzzle's informal definition is presented, along with the RSA-based time-lock puzzle protocol proposed by these authors.

The focus is on the RSA-based puzzle, as it is simple and has been the core of the majority of time-lock puzzle protocols. Informally, the security of a time-lock puzzle requires that the puzzle's solution remains hidden from all adversaries running in parallel within the time period, E. It also requires that an adversary cannot extract a solution in time $B(E) < E$, using a polynomial number of processors that run in parallel and after a large amount of pre-computation. Below we restate the RSA-based time-lock puzzle (TLP) scheme that Rivest et al. (1996) proposed. This is of relevance to this study because the proposed multi-server outsourced PoR scheme will use it.

1. Setup: TLP.Setup(1^λ, E).
   (a) Pick at random two sufficiently large prime numbers, q_1 and q_2. Then, compute their product, i.e., $N = q_1 . q_2$. Next, compute Euler's totient function of N as follows, $\varphi(N) = (q_1 - 1)(q_2 - 1)$.
   (b) Set T the total number of squaring needed to decrypt an encrypted message m, where $T = S . E$, S is the maximum number of squaring modulo N per second that a solver can perform, and E is the period, in seconds, in which the message must remain private.
   (c) Pick a random key, k, for a semantically secure symmetric-key encryption scheme, where the encryption scheme has three algorithms: (GenKey, Enc, Dec).
   (d) Choose a uniformly random value r.
   (e) Set $a = 2^T \mod \varphi(N)$.
   (f) Set $pk := (N, T, r)$ as public key and $sk := (q_1, q_2, a, k)$ as secret key.

2. Generate Puzzle: TLP.GenPuzzle(m, pk, sk).
   (a) Encrypt the message under key k using the symmetric-key encryption, as follows: $p_1 = \text{Enc}(k, m)$.
   (b) Encrypt the symmetric-key encryption's key k, as follows: $p_2 = k + r^a \mod N$.
   (c) Sets: $p := (p_1, p_2)$ as puzzle and output the puzzle.

3. Solve Puzzle: TLP.Solve(pk, p).
   (a) Find b, where $b = r^{2^T} \mod N$, by using T number of squaring r modulo N.
   (b) Decrypt the key's ciphertext, i.e., $k = p_2 - b \mod N$.

(c) Decrypt the message's ciphertext, i.e., m = Dec(k, p_1). Output the solution, m.

Informally, the security of the RSA time-lock puzzle, presented above, is based on the hardness of the factoring problem, the security of the symmetric-key encryption, and sequential squaring assumption. The following theorem (taken from Abadi and Kiayias, 2021, March 4) formally states it.

Theorem 1. Let N be a strong RSA modulus and E be the period within which the solution stays private. If the sequential squaring holds, factoring N is a hard problem and the symmetric-key encryption is semantically secure, then the RSA-based TLP scheme is a secure time-lock puzzle.

Smart Contracts

In general, cryptocurrencies, such as Bitcoin (Nakamoto, 2009) and Ethereum (Wood, 2014), in addition to offering a decentralised digital currency, allow computations on transactions. In this setting, often a computation logic is encoded in a computer program, called a *"smart contract"*. Although Bitcoin, the first decentralised cryptocurrency, supports smart contracts, the functionality of Bitcoin's smart contracts is very limited, due to the use of the underlying programming language that does not support arbitrary tasks (i.e., it is not a Turing-complete programming language). To address this limitation, Ethereum, as a generic smart contract platform, was designed. Thus far, Ethereum has been the most predominant cryptocurrency framework that lets users define arbitrary smart. This framework allows users to create an account with a unique account number (or address). Such users are often called external account holders, which can send (or deploy) their contracts to the framework's blockchain. In this framework, a contract's code and its related data (or state) is held by every node in the blockchain's network. Ethereum smart contracts are often written in a powerful high-level Turing-complete programming language called ''Solidity'', whose syntax is similar to C++ or JavaScript programming languages, and is usually capable of performing any arbitrary task. Once a smart contract is deployed, a unique address is allocated to it by the network. A deployed contract's functions can be called by external account holders, or even other smart contracts which use the contract's address to interact with, and send their instructions to it. In other words, to communicate with a deployed smart contract, one should know the contract's address and some specifications of the contract (e.g., its logic and functions, as well as the arguments that each function takes). Upon an invocation of a smart contract's function, its code is executed by every miner of the blockchain network. The program execution's correctness is guaranteed by the security of the underlying blockchain components. Interestingly, a deployed smart contract can create other smart contracts if it is programmed to do so. Smart contracts can accept and store digital coins that can be transferred to particular recipients, if certain conditions are satisfied. To communicate with a deployed smart contract, one may use a ready-to-use user interface, such as remix (proposed by Ethereum Foundation, 2018), or design a customised user interface. To prevent a denial of service (DoS) attack, the Ethereum framework requires a transaction creator to pay a fee, called *gas*, depending on the complexity of the contract that processes the transaction. Therefore, an account holder, which wants to execute a smart contract's functions, needs to send a certain amount of coins to the network's nodes to cover their execution cost. If the coin amounts sent to the network exceed the required amount, the unspent balance is returned to the account holder. There exist online platforms or digital wallets, such as Metamask (designed by ConsenSys Software Inc., 2016), that can estimate a required amount of gas in real time. Due to smart contracts' various appealing features (e.g., transparency, autonomous execution, not requiring a single trusted party), their applications have been considered in various areas such as banking (Moyano & Ross, 2017), verifiable computation (Abadi & Kiayias, 2021, March 4), education (Kulkarni, 2021), and supply chain (Moosavi et al., 2021).

However, Ethereum smart contracts suffer from an important drawback; 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., the Hawk scheme proposed by Kosba et al. (2016). However, due to the use of generic and computationally expensive cryptographic tools, the privacy-preserving frameworks 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. In this study, the latter approach is considered.

Commitment Scheme

A commitment scheme involves two parties: *sender* and *receiver*. It includes two phases: *commit* and *open*. In the commit phase, the sender commits to a message: m as $\text{Com}(m, d) = h$, 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: $p = (m, d)$ to the receiver who verifies its correctness by checking $\text{Ver}(h, p) = 1$ 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: $p' = (m', d')$ than that used in the commit phase, i.e., infeasible to find p', such that $\text{Ver}(h, p) = \text{Ver}(h, p') = 1$. There exist efficient non-interactive commitment schemes both in (a) the random oracle model using the well-known hash-based scheme that $\text{Com}(m, d)$ involves computing: $\text{H}(m||d) = h$ and $\text{Ver}(h, p)$ requires checking: $\text{H}(m||d) = h$, where H(.) is a collision resistance hash function, and (b) the standard model, e.g., Pedersen scheme proposed by Pedersen (1991). Commitment schemes are of relevance to this study as the proposed multi-server PoR scheme uses it to check a value's correctness.

Pseudorandom Function

Informally, a pseudorandom function (PRF) is a deterministic function that takes as input a key and some argument. It outputs a value indistinguishable from that of a truly random function with the same domain and range. Pseudorandom functions have many applications in cryptography as they provide an efficient and deterministic way to turn an input into a value that looks random. In practice, a pseudorandom function can be obtained from an efficient block cipher; a PRF is formally defined by

Katz and Lindell (2020). A PRF will be used in the proposed PoR scheme to efficiently generate a set of deterministic pseudorandom challenges (or values).

RESEARCH QUESTIONS AND METHODS

Following the above review of relevant literature and discussion of cryptographic tools, the rest of this chapter will address the following research questions (RQs):

RQ1: How can a time-lock puzzle with variable time intervals be built and what are its benefits?

RQ2: Is it possible to construct an efficient outsourced PoR with multiple servers?

RQ3: What is the potential of these developments to improve Proof-of-Storage in the cloud?

The study is conducted by choosing suitable methods and procedures that can tackle the research problem and answer the aforementioned questions. We use the "research onion" of Saunders, Lewis and Thornhill (2009) to identify the study's method elements, outlined briefly as follows. In the *philosophy* layer, we select *pragmatism*, which supports a combination of practice and theories. In the *approach* layer, we select the *deductive* approach that flows from generic to specific. In the strategy layer, we select a combination *of the action* and *archival*, to deal with a specific problem in a specific situation through using existing data and archive documents. In the *choice* layer, we select *multi-methods* that considers both quantitative data and qualitative data. In the *time horizon* layer, our focus is on *longitudinal* that studies events and behaviours using concentrated works over a long period of time. Moreover, in the *technique and procedure* layer, the study analyses the overheads and security of the proposed scheme.

BULDING A TIME-LOCK PUZZLE WITH VARIABLE TIME INTERVALS

As stated previously, the time-lock puzzle, called C-TLP and proposed by Abadi and Kiayias (2021, March 4), requires the time interval's size between finding two consecutive pairs of messages to be equal to the size of the time interval between finding the next pair of two consecutive messages. The issue limits the scheme's flexibility and application. This section outlines how a time-lock puzzle with *variable length time intervals* can be built, so the time intervals can have different sizes. Since the proposed scheme is built on the C-TLP of Abadi and Kiayias (2021, March 4), an overview of this scheme is provided below.

Chained Time-lock Puzzle (C-TLP) Overview

The C-TLP is based on the observation that, in the naive approach when the classical TLP was used in the multiple puzzles setting, the process of decoding messages has many overlaps and also requires the server to solve all puzzles in parallel to the rest. This leads to a high computation overhead and calls for a high level of parallelisation. Therefore, by removing the overlaps, one can considerably lower the overhead, and by making the solving puzzle process sequential the number of required parallel processes (and processors) can be reduced. The core idea in C-TLP is to chain the puzzles. Abadi and Kiayias (2021, March 4) concluded that, although chaining different puzzles seems a relatively obvious approach, designing a secure protocol, which can also make black-box use of a standard time-lock puzzle scheme (i.e., TLP), supports public verifiability, and has low costs, is indeed challenging. In the C-TLP protocol, a client first uses the TLP.Setup (in the TLP scheme) to generate public and private key pair. Note, in this scheme, the private key is a set that includes also a single value, a. Then, it uses them and the algorithm TLP.GenPuzzle to encode the message m that is going to be decoded after the rest, and embeds the information needed for decoding it into the ciphertext of the message m' that will be decoded before message m. In other words, the client integrates the information required to decode message m into the ciphertext related to message m'. It uses the above technique to create a puzzle for every message. In this case, the server uses the TLP.Solve to learn message m' at time t'. By doing so, the server gains sufficient information that allows it to perform the sequential squaring to decode the next message, m, that ultimately will be fully decoded at time t. The server repeats the above process to solve all puzzles.

The authors highlighted that C-TLP solves the above issue for two primary reasons: (a) the total number of squaring required to decrypt all messages is now much lower, and (b) it does not require a high level of parallelisation. The reason is that the server does not need to solve all of the puzzles in parallel; instead, it solves them sequentially, one after another. Furthermore, C-TLP uses the following idea to support a mechanism that allows anyone (i.e., the public) to verify the correctness of solutions found by the server. The client uses a commitment scheme to commit to every message and publishes the commitment. Then, the client uses the TLP to encode the commitment's opening that includes the message and a random value. However, the client does not open the commitment itself; the server does so, after the server solves the related puzzle. In particular, when the server finds a solution, it extracts the opening and message from the solution. Then, it sends them to the verifier (e.g., the public). The verifier can verify the correctness of the solution given the two values and the commitment initially sent by the client. Therefore, in order for a verifier to check the correctness of a solution, it only needs to execute the commitment's verification algorithm, which is efficient and publicly verifiable.

Proposed Solution: Chained Time-lock Puzzle with Non-equal Time Interval Sizes

This section discusses the construction of a chained time-lock puzzle that supports time intervals with different sizes. In this scheme, the time period between the discovery of a pair of consecutive messages can be different from that of another pair of consecutive messages. Briefly, the idea is to slightly modify the algorithm TLP.Setup (presented in "cryptographic tools" section) so it can take different time parameters (instead of only one). It generates a public and private key pair, and outputs different values of ai (as a part of the secret key) for different time intervals.

At a high level, the new scheme works as follows. The client, in the initiation phase, for each time interval, generates a set of parameters (similar to C-TLP). However, in this phase, for every time interval, it also computes additional parameters, say a_i. The additional parameters, a_i, let the messages be encoded into puzzles in a way that the time intervals can have different sizes. In this new scheme, similar to C-TLP, different puzzles are chained. Specifically, when a client generates puzzles, it uses the above parameters to encode the message that will be decoded after the rest of the messages. Nevertheless, it also utilises the above additional parameters, i.e., a_i, when it encodes the related message. It embeds the information that is needed to decode message m into the puzzle of message m' that will be decoded before m. The client repeats the above process until it makes a puzzle for every message. Note that for the client to compute the puzzles, it needs to use the algorithms TLP.Setup and TLP.GenPuzzle that have already been described; moreover, in the above procedure (similar to C-TLP) the client needs to use the commitment scheme to commit to every message. At the end of the puzzle generation phase, the client publishes all puzzles and the commitment results. In the solving phase, when a server wants to solve the puzzles, it starts finding the first message that must be found before the rest; in particular, it finds message m_1, at time t_1. Then, after extracting the first message, it can extract enough information from the message that helps the server to find the next message, m_2 at time t_2. That means, the server, after finding the first message, begins sequentially squaring to extract m_2. The server repeats the above process to solve all puzzles and extract all messages. For the server to prove the correctness of a solution it found, it only needs to send its solution to the verifier, who can check if the solution matches the related commitment (sent initially by the client).

Discussion on the new Chained Time-lock Puzzle

Note that the C-TLP scheme (proposed by Abadi and Kiayias, 2021, March 4) uses the TLP's algorithms (that includes TLP.Setup) in a black-box manner, which is an interesting feature of C-TLP. However, the above proposed chained time-lock puzzle scheme cannot use the TLP scheme in a black-box fashion, because the TLP.Setup has to be modified. This is a necessary trade-off to design a scheme that supports time intervals of different sizes. Furthermore, informally, the security of the above scheme relies on the security of C-TLP, TLP, and commitment schemes. It is not hard to show that if an adversary could break the security of the above scheme, then it would be able to break the security of one of the three schemes. Therefore, the security of the new scheme holds as long as C-TLP, TLP, and commitment schemes are secure. Briefly, the above scheme's overall computation and communication costs of solving and verifying puzzles and their solutions are similar to the costs of those phases in C-TLP.

CONSTRUCTING AN OUTSOURCED PoR WITH MULTIPLE SERVERS

As stated previously, the most recent PoR protocol (proposed by Abadi and Kiayias, 2021, March 4) that efficiently allows a client to delegate the verification process, has two limitations; namely, the time intervals between consecutive verifications have to have identical sizes, and it supports only a single server. In this section, we explain how we can construct a new multi-server outsourced PoR protocol to address these limitations. Since the new scheme is built upon the traditional PoR, introduced by Shacham and Waters (2008), and the outsourced PoR (SO-PoR) designed by Abadi and Kiayias (2021, March 4), an overview of the two schemes is provided, prior to explaining the proposed new scheme.

Traditional Proofs of Retrievability (PoR) Overview

In general, a PoR scheme considers the case where an honest client wants to store its file(s) on a potentially malicious server, i.e., active adversary. It is a challenge-response interactive mechanism where the server proves to the client that its entire file is intact and retrievable. A PoR scheme comprises five algorithms: Setup, Store, GenChal, Prove, and Verify. The Setup is a probabilistic algorithm, run by a client. It takes as input a security parameter and outputs a public-secret key pair. The Store is a probabilistic algorithm, run by a client. It takes as input the secret key and a file. It encodes the file and generates a set of tags. The algorithm outputs the tags and the encoded file. The client stores them on the server. The GenChal is a probabilistic algorithm, run by a client. The algorithm takes as input the encoded file size and security parameter. It outputs a vector of pairs, where each pair includes a file block index and coefficient, both of them are picked uniformly at random. The Prove is a deterministic algorithm, run by the server. It takes the encoded file, tags, and a vector of unpredictable random challenges as inputs and outputs a proof of the file retrievability. The Verify is a deterministic algorithm, run by the client. The algorithm takes the secret key, vector of random challenges, and the proof as inputs. It outputs either 0 if the proof is rejected, or 1 if it is accepted.

Informally, a PoR scheme has two main properties: correctness and soundness. The correctness requires that for any key and any file, the verification algorithm accepts a proof generated by an honest verifier. The soundness requires that if a prover convinces the verifier, then the file is stored by the prover. This is formalized via the notion of *extractability* that involves an extractor algorithm that is able to extract the file in interaction with the adversary, using a polynomial number of rounds. There are two PoR schemes proposed by Shacham and Waters (2008); one uses MAC and the other one relies on BLS signatures. In either scheme, a client first generates a key pair (by running the Setup algorithm). Then, the client in the setup phase encodes its file blocks using error-correcting code, and then the client for each encoded file block generates a tag (by running the Store). The tag can be either a MAC or BLS signature of the block. The encoded file and tags are sent to the server. In the verification phase, the client specifies a set of random indices corresponding to the file's blocks (by running the GenChal algorithm); and accordingly, the server generates and sends a proof to it (by running the Prove algorithm). Next, the client checks if the proof is valid (by calling the Verify algorithm). These two schemes have a low communication cost, because the tags have a certain property (i.e., homomorphic) that allows them to be combined with each other. So, the server can combine all proofs and send only a single proof to the client. In other words, the prover can aggregate proofs into a single authenticator value and no file blocks are sent to the prover. Also, the protocol that relies on MAC supports efficient (private) verification, because it involves only symmetric-key cryptographic tools that impose very low computation cost. However, the one that relies on BLS signatures supports public verifiability at the cost of public-key operations and it is far less efficient than the MAC-based one.

Smart Contract-based Outsourced PoR Overview

Very recently, a smart contract-based outsourced PoR (SO-PoR) was proposed by Abadi and Kiayias (2021, March 4). It adds a set of features to the original PoR scheme (that was initially proposed by Shacham and Waters, 2008). SO-PoR allows the client to delegate its verification process to a (decentralised) smart contract. Also, it offers a combination of real-time detection (i.e., the client is notified in almost real-time when a proof is rejected) and fair payment (i.e., in every verification, the storage server is paid only if a proof is accepted). Specifically, in SO-PoR, unlike the original PoR, a client may not be available every time verification is required. So, the client wants to outsource the verifications that it cannot carry out itself. In this case, the scheme in addition to file retrievability must have three properties: (a) *verification correctness*: each verification process is carried out 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: an honest client, potentially malicious server, and smart contract. The scheme consists of seven algorithms: Setup, Store, SolvPuz, GenChall, Prove, Verify, and Pay.

Here, we briefly describe each of them. The Setup is a probabilistic algorithm, run by a client. It takes as input a security parameter, time parameter, and the number of verifications that will be delegated. It outputs a set of secret and public keys. The Store is a probabilistic algorithm, run only once by the client. It takes as input the secret key, public key, a file, and the number of verifications that the client wants to delegate. It outputs an encoded file, a set of tags, a set of puzzles, and public auxiliary data. The first three outputs are stored on the server and the last output is stored on the smart contract. The SolvPuz is a deterministic algorithm, executed by the server. It takes as input the public key and puzzle vector. It for each verification outputs a pair of solutions. Each solution is sent to the smart contract right after it is discovered. The GenChall is a probabilistic algorithm run by the server. It takes as input a verification index, the encoded file size, security parameter, the related solution pair, and public parameters containing a blockchain and its parameters. It outputs pairs, where each pair includes a pseudorandom block's index and random coefficient. The Prove is a deterministic algorithm, run by the server. It takes the verification index, encoded file, a subset of tags, and a vector of unpredictable challenges, as inputs. It outputs a proof of file retrievability. The Verify is a deterministic algorithm, run by the smart contract. The algorithm takes the verification index, proof, solution pair, and public auxiliary data. If the proof is accepted, then it outputs 1; otherwise, it outputs 0. The SO-PoR protocol is based on homomorphic MAC-based PoR, the chained time-lock puzzle (C- TLP), a smart contract, a pre-computation technique, and a blockchain-based random extraction beacon that was proposed by Abadi et al. (2020).

At a high-level, SO-PoR works as follows. The client encodes its file using an error-correcting code. Moreover, the client for each verification takes the following steps. It picks two random keys: (v, l) for a pseudorandom function. It uses v to generate c random blocks' indices. The client also uses l to generate a temporary tag (or a MAC) on each challenged block. It uses C-TLP to make two puzzles, one that encodes v, and another that encapsulates l. The client deposits enough coins in the contract to cover z successful verifications. The client sends the encoded file, tags, and puzzles to the server. At the time when a proof is needed, the server discovers the related key, v, by solving a puzzle. This key lets the server determine which file blocks are challenged. The server also uses the blockchain-based random extraction beacon to extract a set of random values from the blockchain. The server, by using the tags, challenged blocks, and beacon's outputs, generates a compact proof of file retrievability. It sends the proof to the smart contract. After that, the server can delete the related temporary tags. For the same verification, after a fixed time, the server extracts the related tag verification key, l. The server sends the key to the smart contract. The contract checks if the solution (i.e., the key) is correct and the proof is valid. If the contract accepts all proofs and solutions, then it pays the server for the current verification; otherwise (if it rejects the messages), then it notifies the client.

Proposed Solution: Multi-server Outsourced PoR

This section explains how an outsourced PoR that can support *multiple servers* can be constructed. The new scheme mainly uses the chained time-lock puzzle scheme with variable length time intervals (i.e., the scheme that was presented above) to address one of the limitations of the PoR scheme that was proposed by Abadi and Kiayias (2021, March 4); namely, identical sized time intervals. This feature ultimately allows the new scheme to be more flexible, as the size of each time interval does not need to be equal to other time intervals' size. This new scheme also uses the highly efficient homomorphic MAC-based PoR scheme (designed by Shacham and Waters, 2008), as it involves only symmetric-key cryptographic primitives, which impose low computation and communication costs to the involved parties. The new scheme, similar to SO-PoR, also leverages a smart contract to verify the MACs (or proofs in PoR) on the client's behalf. The use of a smart contract allows the client to delegate the verification of the proofs to a decentralised blockchain. This ensures that the verification result is correct, as long as the blockchain is secure, without the need for the client to re-check the verification (or in general any computation) performed by the smart contract. Also, it lets the client deposit the total amount of coins that should be paid to each honest server before it starts using the servers. This also guarantees to the servers that they will be paid if they act honestly, as the smart contract (who always acts honestly) will transfer their share, if they provide valid proofs. Note that the use of a combination of smart contract, chained time-lock puzzle, and MACs allows the client to efficiently delegate the verification to a smart contract. In particular, the use of the chained time-lock puzzle allows the client to encode the verification keys for MACs into puzzles, such that each puzzle is solved at a certain time by a certain server which sends the proofs (for PoR) and the puzzle's solution to the contract which can efficiently check the validity of the solution and proofs. Also, the blockchain-based random extraction beacon is used to allow a server to independently extract unpredictable random values from the blockchain where the values' correctness can be later verified by the smart contract.

Figure 1 presents a very high-level schematic outline of the multi-server outsourced PoR in a simplified setting where there are only three cloud servers, and each server involves in only one verification. In particular, (0) the client designs and deploys a smart contract and deposit coins in there, (1) the client stores its data and tags in each cloud server, (2) each server sends a proof to the smart contract, (3) the contract verifies each server's proof, (4) the contract pays those servers which provided valid proofs, and (5) it informs the client about those servers which provided invalid proofs. In the following paragraphs, an overview of the multi-server outsourced PoR scheme is provided. It is assumed that there are *m* servers: $S_1, ..., S_m$, in which the client wants to store a copy of its file. Also, let *z* be the total number of verifications the client wants to delegate. In the setup phase, the client generates a set of public and private parameters. It also designs a smart contract that explicitly states (a) the public keys, (b) the identity of each server, e.g., each server's account number on the Ethereum platform, and (c) the time at which a certain server must provide a proof to the contract. The client sends the above smart contract to the blockchain. It deposits in the contract enough coins to pay each honest server who will successfully provide valid proofs (for the dedicated verifications). After that, the client sends a request and the smart contract's address to each server, which checks the validity of the parameters stated in the contract. If the server agrees on the parameters, it sends to the contract a message that states the server wants to serve the client. In the case where the client does not deposit enough coins or a server disagrees on the contract's parameters, then the servers can terminate the contract which sends the client's and servers' deposit back to them.

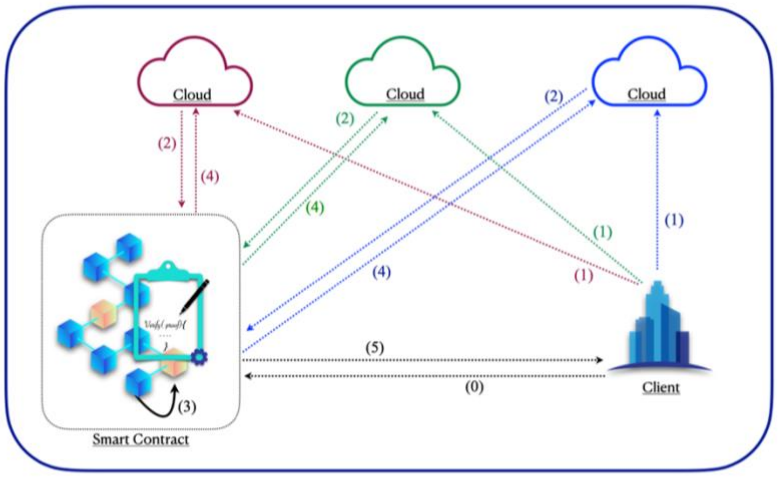

Figure 1. Parties' interaction in the proposed multi-server outsourced PoR.

At the store phase, the client processes its file. This yields a set of encoded file blocks. Next, the client encrypts each file block, by using a symmetric-key encryption scheme. After that, the client, for every verification and every server, takes the following steps. First, it picks two random keys (v, l) for a pseudorandom function. Second, it determines which blocks of the file will be challenged for the current verification; to do so, it uses v to generate a set of (pseudo)random indices of the file blocks. Third, the client uses l to generate a temporary MAC on each file block that will be challenged in the current verification. Recall that it already knows which file blocks will be challenged. Forth, the client uses the time-lock puzzle scheme (that supports variable length time intervals) to encode v and l into two puzzles, such that the former value is discovered at time T while the latter one is discovered at time T', where $T' > T$. Fifth, the client sends the encoded file, MACs, and the puzzles to the related server. Note that by the end of the above process, all servers receive a copy of the encoded file.

In every verification phase, the related server does the following. First, it solves the corresponding puzzle to discover a solution v. Second, the server uses v to determine indices of challenged blocks. Third, the server uses the beacon to retrieve a set of random values from the blockchain. Forth, it uses the MACs (initially given by the client), challenged blocks, and beacon outputs to generate a proof (for PoR). Fifth, it sends the proof to the smart contract. In this phase, the contract does not verify the proofs. Instead, it ensures that a correct server has sent its message on time. In the same verification, after a certain time (i.e., T') the server finds the second puzzle's solution, l, that is a verification key. The server sends the puzzle’s solution and a proof of the solution’s correctness to the smart contract which first verifies the proof’s validity and if it is approved, then it checks the proof (for PoR) that the related server sent to it. If all verifications are passed, then the contract transfers a predefined amount of the client’s deposit to the server. However, if any of the above verifications are rejected, then the contract notifies the client. After all *z* verifications are performed, the client can withdraw its unspent amount of coins from the contract.

Discussion regarding the Multi-server Outsourced PoR

The above proposed scheme offers added features compare to SO-PoR. First, the client can increase its file’s availability by replicating the file among different servers. Second, the time period between finding two puzzle solutions does not need to be equal. Third, nothing about the file (other than its size) is leaked to the servers and the contract, as each file block is encrypted using an encryption scheme whose secret key is known only to the client. The new scheme preserves all appealing features of SO-PoR. In particular, the client can still delegate proofs’ verification to the smart contract, it imposes low costs as it mainly uses symmetric-key cryptographic tools that have low overheads, it allows a client to be notified in (almost) real-time when a proof is rejected, and the servers are paid if, and only if, they provide valid proofs.

CONCLUSION

Cloud computing offers elastic and cost-effective services, such as storage and computation resources, to clients. However, past incidents and recent research illustrated that the cloud cannot be fully trusted, and it may misbehave, e.g., by revealing or tampering with users’ sensitive data. The cloud’s misbehaviour can have serious consequences for individual and business users, e.g., negative press, lost revenue, or stock value drop. So, it is vital for users to ensure the confidentiality, integrity, and availability of the data stored on the cloud. To address the issues, researchers proposed cryptographic schemes called Proof of Storage (PoS). A PoS allows a client to efficiently check the integrity or availability of its data stored in a remote cloud server that potentially can be malicious. But, in PoR schemes, either the client has to be online to perform the verification itself or it has to outsource the verification to a fully trusted single third-party. Recently, an outsourced scheme that addressed the above issues has been proposed. However, it has a set of limitations; namely, the time interval between two consecutive verifications has to have an identical size and supports only a single server. This study has demonstrated how a new scheme can be constructed to addresses the aforementioned issues. In particular, the proposed scheme lets the client replicate its data among multiple servers to achieve high availability, without the need for the client to be online to perform the verification, and without the involvement of a fully trusted third-party auditor, while keeping communication and computation costs low. Also, the scheme is flexible, as the time intervals can have different sizes.

Future research in this field could investigate how to further improve the efficiency of the proposed solution. Such studies may also examine how this solution can be enhanced to support multiple clients, while ensuring that the solution is scalable, and remains secure, even though a subset of the clients collude with each other. The proposals discussed in this chapter and their future enhancements will benefit both the service users (i.e., clients) and the service provider (i.e., cloud computing). In particular, the users will benefit from the increased data availability, security, and increased collaboration with other clients, while the service provider will benefit from high efficiency and scalability that ultimately improve its reputation and revenue.

REFERENCES

Abadi, A. K. (2017). Delegated private set intersection on outsourced private datasets (Doctoral dissertation). Retrieved from the University of Strathclyde. Retrieved March 19, 2021 from http://digitool.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=27943

Abadi, A., Ciampi, M., Kiayias, A., & Zikas, V. (2020). Timed Signatures and Zero-Knowledge Proofs—Timestamping in the Blockchain Era—. In M. Conti, J. Zhou, E. Casalicchio, & A. Spognardi (Eds.), Proceedings of International Conference on Applied Cryptography and Network Security, Rome, Italy (pp. 335-354). Springer. https://doi.org/10.1007/978-3-030-57808-4/_17

Abadi, A., & Kiayias, A. (2021, March 4). Multi-instance publicly verifiable time-lock puzzle and its applications. Financial Cryptography and Data Security conference, Virtual. Retrieved March 10, 2021 from https://fc21.ifca.ai/papers/115.pdf

Armknecht, F., Bohli, J. M., Karame, G. O., Liu, Z., & Reuter, C. A. (2014). Outsourced proofs of retrievability. In G. Joon, M. Yung & N. Li (Eds.), Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA (pp. 831-843). https://doi.org/10.1145/2660267.2660310

Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., & Song, D. (2007). Provable data possession at untrusted stores. In P. Ning, D. Vimercati & P. Syverson (Eds.), Proceedings of the 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA (pp. 598-609). https://doi.org/10.1145/1315245.1315318

Ateniese, G., Chen, L., Etemad, M., & Tang, Q. (2020). Proof of storage-time: Efficiently checking continuous data availability. In Proceedings of the 27th Annual Network and Distributed System Security Symposium, San Diego, California, USA. https://www.ndss-symposium.org/ndss-paper/proof-of-storage-time-efficiently-checking-continuous-data-availability/

Berman, S. J., Kesterson-Townes, L., Marshall, A., & Srivathsa, R. (2012). How cloud computing enables process and business model innovation. *Strategy & Leadership, 40(4),27-35.* https://www.emerald.com/insight/content/doi/10.1108/10878571211242920/full/pdf?title=how-cloud-computing-enables-process-and-business-model-innovation

ConsenSys Software Inc. (2016). Metamask. Retrieved June 08, 2021, from: https://metamask.io

Ethereum Foundation. (2018). Remix-IDE. Retrieved March 06, 2021, from: https://github.com/ethereum/remix-ide/blob/master/docs/index.rst

Luxner, T. (2021, March 15) State of the cloud report from flexera. Forbes. Retrieved July 28, 2021, from: https://www.flexera.com/blog/industry-trends/trend-of-cloud-computing-2020/

Francati, D., Ateniese, G., Faye, A., Milazzo, A. M., Perillo, A. M., Schiatti, L., & Giordano, G. (2019). Audita: A Blockchain-based Auditing Framework for Off-chain Storage. arXiv preprint arXiv:1911.08515. https://arxiv.org/pdf/1911.08515.pdf

Hao, K., Xin, J., Wang, Z., Jiang, Z., & Wang, G. (2018). Decentralized data integrity verification model in untrusted environment. In Y. Cai, Y. Ishikawa & J. Xu (Eds.), Proceedings of Asia-Pacific Web (APWeb) and Web-age information management (WAIM) joint international conference on Web and big data, Macau, China (pp. 410-424). Springer, Cham. https://doi.org/10.1007/978-3-319-96893-3/_31

Lab, P (2017, July 19): Filecoin: A decentralized storage network. Retrieved June 15, 2021, from https://filecoin.io/filecoin.pdf

Juels, A., & Kaliski Jr, B. S. (2007). PORs: Proofs of retrievability for large files. In P. Ning, S. Vimercati & P. Syverson (Eds.), Proceedings of the 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA (pp. 584-597). https://doi.org/10.1145/1315245.1315317

Kamara, S. (2013). Proofs of storage: Theory, constructions and applications. In T. Muntean, D. Poulakis & R. Rolland (Eds.), Proceedings of International Conference on Algebraic Informatics, Porquerolles, France (pp. 7-8). Springer. https://doi.org/10.1007/978-3-642-40663-8/_4

Katz, J., & Lindell, Y. (2020). *Introduction to modern cryptography*. CRC press. Florida, United States. http://www.cs.umd.edu//~jkatz/imc.html

Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. (2016). Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In Proceedings of 2016 IEEE symposium on security and privacy, San Jose, CA, USA. https://doi.org/10.1109/SP.2016.55

Kopp, H., Bösch, C., & Kargl, F. (2016). Koppercoin–a distributed file storage with financial incentives. In F. Bao, F., L. Chen, R. Deng & G. Wang (Eds.), Proceedings of International Conference on Information Security Practice and Experience, Zhangjiajie, China (pp. 79-93). Springer, Cham. https://doi.org/10.1007/978-3-319-49151-6/_6

Kopp, H., Mödinger, D., Hauck, F., Kargl, F., & Bösch, C. (2017). Design of a privacy-preserving decentralized file storage with financial incentives. In Proceedings of IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) Paris, France (pp. 14-22). IEEE. https://doi.org/10.1109/EuroSPW.2017.45

Kulkarni, D. (2021). Leveraging Blockchain technology in the Education Sector. Turkish Journal of Computer and Mathematics Education (TURCOMAT), 12(10), 4578-4583. https://www.turcomat.org/index.php/turkbilmat/article/view/5202

May, T.C. (1993). Timed-release crypto. Technical report. Retrieved March 06, 2021, from: http://cypherpunks.venona.com/date/1993/02/msg00129.html

Miller, A., Juels, A., Shi, E., Parno, B., & Katz, J. (2014). Permacoin: Repurposing bitcoin work for data preservation. In Proceedings of 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA (pp. 475-490). IEEE. https://doi.org/10.1109/SP.2014.37

Moosavi, J., Naeni, L. M., Fathollahi-Fard, A. M., & Fiore, U. (2021). Blockchain in supply chain management: a review, bibliometric, and network analysis. Environmental Science and Pollution Research, 1-15. Springer. https://link.springer.com/article/10.1007/s11356-021-13094-3

Moyano, J. P., & Ross, O. (2017). KYC optimization using distributed ledger technology. Business & Information Systems Engineering, 59(6), 411-423. https://link.springer.com/content/pdf/10.1007/s12599-017-0504-2.pdf

Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system. Retrieved March 06, 2021, from: https://bitcoin.org/bitcoin.pdf

Pedersen, T. P. (1991). Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum (Eds.), Proceedings of Annual international cryptology conference, California, USA (pp. 129-140). Springer. https://link.springer.com/chapter/10.1007/3-540-46766-1_9

Renner, T., Müller, J., & Kao, O. (2018). Endolith: A blockchain-based framework to enhance data retention in cloud storages. In I. Merelli, P. Lio & I. Kotenko (Eds.). Proceedings of 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Cambridge, United Kingdom (pp. 627-634). IEEE. https://doi.org/10.1109/PDP2018.2018.00105

Rivest, R. L., Shamir, A., & Wagner, D. A. (1996). Time-lock puzzles and timed-release crypto. Massachusetts Institute of Technology. https://dl.acm.org/doi/book/10.5555/888615

Ruj, S., Rahman, M. S., Basu, A., & Kiyomoto, S. (2018). Blockstore: A secure decentralized storage framework on blockchain. In L. Barolli, M. Takizawa, T. Enokido, M. Ogiela, L Ogiela, & N. Javaid

(Eds.), Proceedings of IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA) Krakow, Poland (pp. 1096-1103). IEEE. https://doi.org/10.1109/AINA.2018.00157

Saunders, M., Lewis, P., & Thornhill, A. (2009). Research methods for business students. Pearson education.https://books.google.co.uk/books/about/Research_Methods_for_Business_Students.html?id=u-txtfaCFiEC&redir_esc=y

Shacham, H., & Waters, B. (2008). Compact proofs of retrievability. In J. Pieprzyk (Eds.), Proceedings of International conference on the theory and application of cryptology and information security, Melbourne, Australia (pp. 90-107). Springer. https://doi.org/10.1007/978-3-540-89255-7/_7

Vorick, D., & Champine, L. (2014). Sia: Simple decentralized storage. Retrieved March 06, 2021, from: https://sia.tech/sia.pdf

Wilkinson, S., Boshevski, T., Brandoff, J., & Buterin, V. (2014). Storj a peer-to-peer cloud storage. Retrieved June 15, 2021, from: network. https://www.storj.io/storj2014.pdf

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, *151*(2014), 1-32. https://ethereum.github.io/yellowpaper/paper.pdf

Xu, J., Yang, A., Zhou, J., & Wong, D. S. (2016). Lightweight delegatable proofs of storage. In I. Askoxylakis, S. Ioannidis, S. Katsikas, & C. Meadows (Eds.), Proceedings of European Symposium on Research in Computer Security, Heraklion, Greece (pp. 324-343). Springer. https://doi.org/10.1007/978-3-319-45744-4/_16

Xue, J., Xu, C., & Bai, L. (2019). DStore: A distributed system for outsourced data storage and retrieval. Future Generation Computer Systems, 99, 106-114. https://www.sciencedirect.com/science/article/abs/pii/S0167739X18324610

Zhang, Y., Deng, R. H., Liu, X., & Zheng, D. (2018). Blockchain based efficient and robust fair payment for outsourcing services in cloud computing. Information Sciences, 462, 262-277. https://doi.org/10.1016/j.ins.2018.06.018

KEY TERMS AND DEFINITIONS

Boneh-Lynn-Shacham (BLS) Signatures: Is a cryptographic signature scheme that let a user verify that a signer is authentic. One of the main features of this scheme is ‘’signature aggregation’’, that allows multiple signatures that are computed under multiple public keys (for different messages) to be combined into a single signature.

Commitment Scheme: A commitment scheme is a cryptographic tool which lets a party to publicly commit to a message while keeping the message hidden to others, with the ability to reveal the committed message later without being able to change the message.

Merkle Tree: In cryptography, a Merkle tree or (hash tree) is a data structure that is used for data verification. It is a tree-like data structure built on top of a file such that a leaf node of the tree is the file block and each non-leaf node is a hash of two child nodes. It has been used in various schemes such as digital signatures and Bitcoin.

Message Authentication Code (MAC). MAC is metadata (or tag) of a message that allows a verifier to authenticate the message or detect the message modification. The algorithm that generates a MAC is usually a symmetric-key cryptographic scheme that is highly efficient. It is often used in cases where the party who generates and verifies a MAC is the same (i.e., supports only private verification).

Proof of Retrievability (PoR): A PoR scheme is a cryptographic protocol that allows a client to efficiently verify the availability of its entire data stored in a potentially malicious remote cloud server. In other words, it the scheme lets the client check if its entire outsourced data is retrievable from a cloud server.

Proof of Data Possession (PDP): A PDP scheme is a cryptographic protocol that lets a client that stored data at a server, which is potentially malicious, and later on verify that the server possesses the original data without requiring to retrieve the entire file. In general, PDP schemes only ensure the integrity of outsourced data (but not the retrievability of it).

Smart Contract: A smart contract is a decentralised computer program that is used to digitally facilitate, verify, or enforce a contract's clause without relying on a single trusted third-party arbiter. The contract code is usually stored in a blockchain and is executed by all nodes of the blockchain network.

Time-lock puzzle: Time-lock puzzles are cryptographic protocols that allow sending messages "to the future". In these schemes, a sender can generate a puzzle with a solution that remains hidden until a moderately large amount of time has elapsed.

Verifiable Delay Function (VDF): A VDF lets a prover generate a publicly verifiable proof, thereby proving that it has carried out a pre-determined number of sequential computations. To generate such a proof, it needs to perform the required sequential computations. The VDF has various applications, e.g., in decentralised systems to extract trustworthy public randomness from a blockchain.