Blockchain Enabled Privacy Preserving Data Audit

Prabal Banerjee, Nishant Nikam and Sushmita Ruj, *Senior Member, IEEE*

**Abstract**—Data owners upload large files to cloud storage servers, but malicious servers may potentially tamper data. To check integrity of remote data, Proof of Retrievability (PoR) schemes were introduced. Existing PoR protocols assume that data owners and third-party auditors are honest and audit only the potentially malicious cloud server to check integrity of stored data. In this paper we consider a system where any party may attempt to cheat others and consider collusion cases. We design a protocol that is secure under such adversarial assumptions and evaluate a prototype using Ethereum as the blockchain platform and show that our scheme has comparable performance.

**Index Terms**—PoR, Cloud, Storage, Audit, Blockchain, DLT, Privacy

1 INTRODUCTION

RECENT years have seen enormous amount of data generated and people using multiple devices connected to the Internet. To cater to the need for accessing data across devices, cloud storage providers have come up. Similarly, there has been an increase in Anything-as-a-Service(XaaS) which needs data to be uploaded and stored on remote servers. To ensure integrity of uploaded data, Proof-of-Storage algorithms have been proposed. With publicly verifiable Proof-of-Retrievability schemes, data owners can potentially outsource this auditing task to third-party auditors who send challenges to storage servers. The storage servers compute responses for each challenge and gives a response. By validating the challenge-response pair, the auditor ensures that the stored file is intact and retrievable.

The inherent assumption in the existing PoR schemes is that the data owner is always honest. The server may be malicious and try to erase data in an attempt to reduce it’s cost. The PoR schemes offer guarantees that if the server responds correctly to the challenge set, then the file is recoverable with overwhelming probability. The auditor is trusted with checking the integrity of the stored data without gaining access to the data itself, but previous studies showed that the auditor might gain access by carefully selecting the challenge set it sends to the server, thus acting maliciously. Also, data owner may refuse payment to server and auditor in a bid to cut cost and act maliciously.

Blockchain, a distributed tamper-resistant ledger, has seen several use cases lately. With smart contract support, arbitrary logic can be enforced in a distributed manner, even in the presence of some malicious players. These abilities are used to make blockchain a trusted party to resolve disputes. Several works have used blockchain as a judge to settle disputes among involved parties [1], [2]. Most major blockchain platforms have an inherent currency which is used to make meaningful monetary contracts among participants.

In this paper we aim to propose a blockchain based proof-of-storage model and prove its security even if data owner is corrupted. We study the collusion cases and argue that our system is secure unless the data owner, auditor and cloud server together collude and act maliciously. We implement a prototype using a modified version of Ethereum. We use the inherent currency of the blockchain platform to model and enforce our incentive structure. To make minimum blockchain interactions, we use state channels and perform off-chain transactions. Our experiments show that our improved system has comparable performance.

**Our Contribution**

- Propose a blockchain based data audit model, where the auditor can be a third party.
- Design state channel based audit protocol to minimize blockchain commits.
- Use blockchain based payment to incentivize players in the system.
- Give provable security guarantees even when parties collude with one another.
- Implement a prototype on modified Ethereum and show comparable performance with overhead of around 6 seconds per audit phase.

2 PRELIMINARIES AND BACKGROUND

2.1 Notations

We take $\lambda$ to be the security parameter. An algorithm $A(\lambda)$ is a probabilistic polynomial-time algorithm when its running time is polynomial in $\lambda$ and its output $y$ is a random variable which depends on the internal coin tosses of $A$. A function $f : \mathbb{N} \to \mathbb{R}$ is called negligible in $\lambda$ if for all positive integers $c$ and for all sufficiently large $\lambda$, we have $f(\lambda) < \frac{1}{\lambda^c}$. An element $a$ chosen uniformly at random from set $S$ is denoted as $a \overset{R}{\leftarrow} S$. We use a secure digital signature...
algorithm \((\text{Gen}, \text{Sign}, \text{SigVerify})\), where \(\text{Gen}()\) is the key generation algorithm, \(\text{Sign}()\) is the signing algorithm and \(\text{SigVerify}()\) is the signature verification algorithm. We use a collision-resistant cryptographic hash function \(H\).

### 2.2 Bilinear Pairings

**Definition:** Let \(G_1, G_2\) be two additive cyclic groups of prime order \(p\), and \(G_T\) another cyclic group of order \(p\) written multiplicatively. A pairing is a map \(e : G_1 \times G_2 \rightarrow G_T\), which satisfies the following properties:

- **Bilinearity:** \(\forall a, b \in \mathbb{Z}_p^*, \forall P \in G_1, \forall Q \in G_2 : e(aP, bQ) = e(P, Q)^{ab}\)
- **Non-Degeneracy:** \(e \not= 1\)
- **Computability:** There exists an efficient algorithm to compute \(e\).

A pairing is called symmetric if \(G_1 = G_2\). When we use symmetric bilinear pairings, we refer to a map of the form \(e : G \times G \rightarrow G_T\) with \(G\)'s support being \(\mathbb{Z}_p\).

### 2.3 Proofs of Retrievability

#### 2.3.1 File Processing and Query Generation

**File:** A file \(F\) is broken into \(n\) chunks, where each chunk is one element of \(\mathbb{Z}_p\). Let the file be \(b\) bits long. We refer to each file chunk as \(m_i\), where \(1 \leq i \leq n\) and \(n = \lceil b/\log p \rceil\). We use chunk and block interchangeably to refer to each file chunk.

**Query:** \(Q = \{(i, \nu_i)\}\) be an \(l\)-element set, where \(l\) is a system parameter, \(1 \leq i \leq n\) and \(\nu_i \in \mathbb{Z}_p\). The verifier chooses an \(l\)-element subset \(I\) of \([1, n]\), uniformly at random. For each \(i \in I, \nu_i \leftarrow \mathbb{Z}_p\).

#### 2.3.2 2.4 File Processing and Query Generation

We use the Shacham and Waters Compact Proofs of Retrievability scheme with public verifiability \(\mathcal{F}\), which uses symmetric bilinear pairings. Let a user have a key pair \(K = (sk, pk)\) where \(sk = x \in \mathbb{Z}_p, pk = v = g^x \in G\). Let \(u \in G\) be a generator. For file block \(i\), authentication tag \(\sigma_i = [H(i)u^m]^{sk}\). The prover receives the query \(Q = \{(i, \nu_i)\}\) and sends back \(\sigma \leftarrow \prod_{(i, \nu_i) \in Q} \sigma_i^{\nu_i}\) and \(\mu \leftarrow \prod_{(i, \nu_i) \in Q} \nu_i, m_i\). The verification equation is :

\[
e(\sigma, g) = e(\prod_{(i, \nu_i) \in Q} H(i)^{\nu_i}u^{\nu_i}, v) \tag{1}
\]

The scheme has public verifiability because to generate authentication tags the private key \(sk\) is required. On the other hand, for the proof-of-retrievability protocol, public key \(pk\) is sufficient. In this paper, we refer to this scheme as Shacham-Waters.

### 2.6 Privacy Preserving Public Auditing for Secure Cloud Storage

To achieve strong privacy guarantees, we use Privacy Preserving Public Auditing for Secure Cloud Storage scheme by Wang et al. \(\mathcal{F}\). Let \(H_1 : \{0, 1\}^* \rightarrow G_1\) and \(H_2 : G_T \rightarrow \mathbb{Z}_p\) be hash functions and \(g\) be a generator of \(G_2\). Let a user have a key pair \(K = (sk, pk)\) where \(sk = x \in \mathbb{Z}_p\) and \(pk = (v, g, u, e(u, v))\) such that \(v = g^x\) and \(u \leftarrow G_1\). For file block \(i\) with identifier \(id \leftarrow \mathbb{Z}_p\), authentication tag \(\sigma_i = [H_1(W_i)u^m]^{sk}\), where \(W_i = id[i]\). The prover receives the query \(Q = \{(i, \nu_i)\}\). It selects a random element \(r \leftarrow \mathbb{Z}_p\) and calculates \(R = e(u, v)^r\) along with \(\gamma = H_2(R)\). Finally the prover sends back \(R, \sigma = \prod_{(i, \nu_i) \in Q} \sigma_i^{\nu_i}\) and \(\mu \leftarrow r + \gamma \times \prod_{(i, \nu_i) \in Q} \nu_i, m_i\). The verification equation is :

\[
R, e(\sigma^\gamma, g) = e(\prod_{(i, \nu_i) \in Q} H(W_i)^{\nu_i}u^{\nu_i}, v) \tag{2}
\]

The scheme is also publicly verifiable like Shacham-Waters protocol. Additionally, it is privacy preserving. In this paper, we refer to this scheme as PPSCS.

### 2.7 Blockchain

Blockchain is a tamper-resistant, append-only distributed ledger. Apart from acting as a non-repudiable log, blockchain can host distributed applications and perform arbitrary functions in the form of smart contracts. First introduced in Bitcoin \(\mathcal{F}\), it is a hash-linked chain of blocks, each block potentially containing multiple transactions. Participating nodes broadcast ledger updates in form of transactions. While there are various flavours of blockchain systems present based on the type of consensus protocol they use, we use a Proof-of-Work (PoW) based blockchain system. In a blockchain platform following PoW consensus protocol, special nodes, called miners, form blocks containing multiple transactions. The miners compete and solve a hash based challenge and the winner gets to propose the next block along with getting a mining reward.

Ethereum \(\mathcal{F}\) is one of the most popular blockchain platforms supporting Turing-complete languages to write smart contracts. Ether is the cryptocurrency of the Ethereum network.
platform and it is used to incentivize computations on the platform. The contracts are executed inside Ethereum Virtual Machine (EVM) which is uniform across all nodes, so as to have same output across the network. The amount of work done in terms of the number of operations done is calculated in terms of gas. A user submits transactions along with ethers to compensate for the work done by miners, according to the gas price. This acts as a transaction fee for the miners, and at the same time prevents running bad code like infinite loops which might harm the miners. Ethereum has a set of pre-compiled contracts which are codes running inside the host machine and not inside the EVM. Hence, the pre-compiled contracts cost less gas. Ethereum is open-source with an active community and has seen large scale adoption.

While active research is being performed to lower consensus time in public blockchain systems, traditional PoW chains need considerable time before a transaction reaches finality. Hence, it becomes hard to implement multi-commit protocols which are practical. For example, in Ethereum, the average block generation time is between 10-19 seconds. This delay might not be suitable for high frequency applications. On top of that, for each transaction to get mined, the user needs to incur additional cost in terms of transaction fee or gas costs. These problems make protocols like audit unsustainable as an auditor and server need to interact multiple times to exchange challenges and responses. A typical technique used to bypass these problems is performing off-chain transactions by opening state channels between pairs of users. The participants of a state channel exchange signed messages and perform on-chain transactions only when either they are finished with their interaction or some dispute arises. The blockchain either saves the final states of the participants or resolves disputes, whichever applicable. This reduces time and cost for the users.

3 RELATED WORK

On Cloud Storage: The idea of auditing cloud storage servers was first introduced by Atienese et al. [9], who defined Provable Data Possession (PDP) model. Juels and Kaliski [4] first described a PoR scheme for static data using sentinels. A similar construction was provided by Naor et al. [10] using MAC based authenticators. A study on various variants of PoR schemes with private verifiability is done by Dodis et al. [11]. First fully dynamic provable data possession was given by Erway et al. [12]. A secure distributed cloud storage scheme called HAIL (high-availability and integrity layer) is proposed by Bowers et al. [13] which attain POR guarantees. Shacham and Waters provided a PoR construction using BLS signatures, which had both public and private verifiability. In OPOR [14], the authors define a formal framework where the auditing task is outsourced and provide a construction based on Shacham-Waters. Wang et al. argued that the auditor can retrieve information about file and hence proposed a privacy preserving data audit scheme in [6]. For dynamic data, an ORAM based audit protocol was given in [15]. More efficient protocols were given in [16], [17], [18], [19], [20]. Multiple server based PoR schemes were formalized in [21]. All these existing work either assume only the server to be dishonest, or do not consider the collusion cases between the parties.

On Storage with Blockchain: In recent times, various blockchain based cloud servers have come up. IPFS [22] introduced a blockchain based naming and storage system. Several other systems [23], [24], [25] use the concept of cloud storage in a decentralized fashion in a P2P network. In [26], the authors make the storage accountable and show how to integrate with Bitcoin. In [27], the designers use IPFS and a cryptocurrency to make a storage based marketplace. They use Proof of Replication to enforce storage among the peers. SpaceMint [28] introduced a new cryptocurrency that adapts proof of space, also proposed a different blockchain format and transaction types. Moran et al. [29] introduced Proofs of Space-Time (PoSTs) and implemented a practical protocol for these proofs. An in-depth analysis of Proof of Replication mechanisms was done in [30].

While the use of blockchain as an enforcer and incentive distribution mechanism was tapped in these works, most work did not consider fairness among the services offered by the parties.
Phase 0: Initialization Phase

- **KeyGen**: Initialized by the data owner, this algorithm generates a random public-private key-pair \( K = (sk, pk) \) and public parameters based on the security parameter \( \lambda \).
- **RegisterOwner**: A new data owner uses this function to setup her account. She deposits prerequisite money to initialize her account with, which is used for future payments. The identity information, ownerID, will be used to authorize all further transactions by this data owner. She also submits her public key \( pk_s \) which is used to verify signed messages submitted by the owner.
- **RegisterServer**: An existing data owner uses this function to supply identity information about the publicly known server, serverID, and public key \( pk_s \) of the server with whom she wants to store her data.
- **RegisterAuditor**: This function is called by the operative data owner to specify the publicly known auditor, auditorID, she wants to assign. If the owner wants the selected auditor to only audit a particular server, the owner may additionally supply that information.

Phase 1: Owner - Server

- **FileTransfer**: The owner divides the file \( F \) into \( n \) blocks. Let \( F = m_1, m_2, \ldots, m_n \in \mathbb{Z}_p \). She generates authentication tags, \( \sigma_i \), \( 1 \leq i \leq n \) and calculates hash \( h_i = H(m_i \parallel \sigma_i) \) for \( 1 \leq i \leq n \). She sends \( (m_i, \sigma_i) \), \( \forall i \), to the server. She receives \( ([h'_1, h'_2, \ldots, h'_n], t) \) from server and checks \( \text{SignVerific}([h'_1, h'_2, \ldots, h'_n], t, pk_s) = 1 \) and \( h_i = h'_i(\forall i) \). Then, she sends \( \text{Sign}([h'_1, h'_2, \ldots, h'_n], t, sk_o) \) to the blockchain.

Phase 2: Server - Auditor

- **OpenChannel**: This function collects deposit from server and auditor and opens up a state channel between them to interact off-chain. It freezes the owner money to pay necessary parties once channel is closed.
- **GenQuery**: This function generates an audit query for the auditor based on the randomness derived from the last block of the blockchain. The query is then sent to the server for response.
- **GenResponse**: Given a query, this function generates an audit response. The server sends the response to the auditor.
- **Verify**: Given a response, this function verifies whether the response is correct or not. Based on this outcome of verification, we proceed with the next set of challenges.
- **CloseChannel**: This function receives aggregated challenge-response along with the complaint, if any. It verifies whether the queries were valid and responses pass the audit. In case of complaint, it punishes the guilty party, else, it pays server and auditor as per norms of payment.

Phase 3: Owner - Server

- **FetchFile**: The data owner retrieves the stored file from the server using this function.

4.2 Security Guarantees and Adversarial Model

**Security of Protocol**

- **Authenticity**: The authenticity of storage requires that the cloud server cannot forge a valid proof of storage corresponding to the challenge set \( Q \) without storing the challenged chunks and their respective authentication tags untampered, except with a probability negligible in \( \lambda \).
- **Extractibility**: The extractibility property requires the \( \text{FetchFile()} \) function to be able to recover the original file when interacting with a prover that correctly computes responses for non-negligible fraction of the query space.
- **Privacy**: The privacy of audit requires the auditor not to learn any property of the stored file chunks \( m_i \). The auditor generates queries to receive response. The auditor should not be able to derive \( m_i \), for any \( i \), from the response.
- **Fairness**: We notice that the cloud server and auditor offer services in exchange for payment from the data owner. The fairness property would require the following:
  - If the cloud server stores \( m_i \), \( \forall i \), then it receives adequate incentive. If it fails to keep the files intact, it gets penalized.
  - If the auditor generates queries correctly, verifies responses and submits aggregated response to blockchain, then it receives appropriate incentive. If not, it gets penalized.
  - If the data owner gets services from cloud server and auditor as intended, then it has to pay according to the agreement. If she incurs losses due to a malicious party, she will be paid for the damage.

**Adversarial Model**

**Users**: We would call a user honest, if she follows the protocol. Otherwise, we would call her malicious. A malicious user can deviate arbitrarily.

**Adversary**: An adversary is a polynomial-time algorithm that can make any user malicious at any point of time, subject to some upper bound. Our adversary is dynamic in nature, that is, it can select its target based on current configuration of the system. It can make coordinated attacks, that is, it can control the malicious users and send/receive messages on their behalf. It can, of course, make a malicious user isolated and prescribe arbitrary instructions for her to perform. \[31\]

However, the adversary cannot break cryptographic primitives like hash functions or signatures, except with negligible probability. It cannot interfere with honest users or their exchanges.

**Bounds**: The following are the restrictions:
Among the peers in the blockchain, the adversary can only corrupt up to the bound of the underlying consensus protocol. For our implementation, we have PoW and hence the bound is 49%.

- The three parties apart from the blockchain - owner, server and auditor - cannot be corrupted together. At most two of the three parties can be malicious at any point of time.
- We assume that the adversary will not corrupt without sufficient incentive. We think of the adversary as a rational player in the game.

### 5 Construction of Audit Protocol Using Blockchain

In this section we outline our construction. For a protocol to fit into our framework, the PoR scheme has to be publicly verifiable and needs to produce short aggregated proofs. Although multiple PoR schemes have our required properties, we chose Shacham-Waters in our first protocol named AuB. Shacham-Waters have complete security proofs along with practical overhead in terms of implementation. Also, it uses Homomorphic Linear Authenticators (HLA) which helps us have a very concise proof, which can be submitted to the ledger for verification upon closing state channel. The major drawback of Shacham-Waters is that it lacks privacy. Attacks have been shown that reveals parts of data from audit proofs. Hence, we further define a privacy preserving audit using blockchain named PP AuB using PPSCS which gives privacy guarantees by random masking. We discuss both the designs in this section.

We assume that the server and auditor are known entities in the system, i.e., their public keys, identities and addresses are known throughout the system. Also, we assume the server and auditor has some coins deposited in the system which can be used to penalize them in case of misbehavior. For simplicity, we assume of a single file $F$ uploaded by a single owner $O$ to server $S$. We assume $O$ authorizes auditor $A$ as the third-party auditor for performing audits. $O$ might as well act as $A$ herself and perform the audit protocol. Our security assumptions allow such a case because the protocol is resilient against collusion by owner and auditor. We refer to the smart contract hosted on the blockchain as $L$.

Additionally, we assume that the payout terms are uniform across the system. The terms must contain $auditCount$ (the minimum number of audit challenges for $A$), $C_s$ (money to be paid to honest server) and $C_A$ (money to be paid to honest auditor). All these assumptions can be easily dropped to extend to multi-user generic setting, but we omit those cases for brevity.

#### 5.1 Audit using Blockchain (AuB)

As outlined in [4], we break our protocol into four phases. We define each of the functionalities involved in those phases in this section.

In our Phase 0 (Initialization Phase), $O$ generates the keys and public parameters using $\text{KeyGen}_{\text{AuB}}()$. It uses $\text{RegisterOwner}()$ to register itself with $L$ and deposit money $(C_s + C_A)$ to the contract. This money is used in future payment to $S$ and $A$. It also deposits contract terms like $auditCount$ into the smart contract for transparency. $\text{RegisterOwner}()$ returns $\text{OwnerID}$ which $O$ uses for all future communication with $L$. $O$ uses $\text{RegisterServer}()$ and $\text{RegisterAuditor}()$ to authorize $S$ and $A$ respectively. If any of the transactions with $L$ fails, the protocol is
Algorithm 1: Honest Owner

Function `KeyGen_AuB(λ)`:
- Select the group $G$ based on $λ$.
- Select the generators $u, g \in G$.
- Select a random element $x \in \mathbb{Z}_p$ Assign $sk = x \in \mathbb{Z}_p$ and $pk = g^x = v \in G$.
- Return $K = (sk, pk), g$ and $u$.

Function `KeyGen_PPAuB(λ)`:
- Select the groups $G_1, G_2$ based on $λ$.
- Select the generator $g \in G_2$.
- Select random elements $x \in \mathbb{Z}_p, u \in G_1$.
- Calculate $g^x = v \in G_2$ Assign $sk = x$ and $pk = (v, g, u, e(u, v))$.
- Return $K = (sk, pk)$.

Function `FileTransfer_AuB(F, u, sk, n, pk, fileid)`:
- Divide the file $F$ into $n$ blocks. Let $F = m_1, m_2, \ldots, m_n \in \mathbb{Z}_p$.
- Generate authentication tags, $σ_i = \left[H(i)u^{e_i}\right]^sk, 1 \leq i \leq n$.
- Calculate hash $h_i = H(m_i || σ_i)$ for $1 \leq i \leq n$ and store it.
- Send $(m_i, σ_i), i, t$ to the Server.
- Receive $([h'_1, h'_2, \ldots, h'_n], t)$ from the Server
- Check $\text{SigVerify}([h'_1, h'_2, \ldots, h'_n], t, pk) = 1$ AND $h_i = h'_i (1 \leq i \leq n)$. If not, terminate.
- Send $\text{Sign}(([h'_1, h'_2, \ldots, h'_n], t, sk))$ to Blockchain by calling function ReceiveSignedDigest().

Function `FileTransfer_PPAuB(F, u, sk, n, pk, fileid)`:
- Divide the file $F$ into $n$ blocks. Let $F = m_1, m_2, \ldots, m_n \in \mathbb{Z}_p$.
- Generate random $\text{fileid} \in \mathbb{Z}_p$.
- Generate $W_i = (\text{fileid}[i]), 1 \leq i \leq n$.
- Generate authentication tags, $σ_i = \left[H(W_i)u^{e_i}\right]^sk, 1 \leq i \leq n$.
- Calculate hash $h_i = H(m_i || σ_i)$ for $1 \leq i \leq n$ and store it.
- Send $(m_i, σ_i), i, t$ to the Server.
- Receive $([h'_1, h'_2, \ldots, h'_n], t)$ from the Server
- Check $\text{SigVerify}([h'_1, h'_2, \ldots, h'_n], t, pk) = 1$ AND $h_i = h'_i (1 \leq i \leq n)$. If not, terminate.
- Send $\text{Sign}(([h'_1, h'_2, \ldots, h'_n], t, sk))$ to Blockchain by calling function ReceiveSignedDigest().

Function `FetchFile(F)`:
- Request to the Server to download the file $F$.
- Receive the acknowledgement from the Server.
- Download the file $F$.

Function `SendCredsToAuditor_AuB(g, u, v, pk)`: Send $g, u, v$, and $pk$ to the Auditor.
- Receive the acknowledgement from the Auditor.

Function `SendCredsToAuditor_PPAuB(pk, fileid, pk)`: Send $pk$, $fileid$, and $pk$ to the Auditor.
- Receive the acknowledgement from the Auditor.
Algorithm 2: Honest Server

Function ReceiveFile($sk_s$):
- `hashArray = []`
- for each $(m_i, \sigma_i)$ received do
  - `hashArray.append(H(m_i||\sigma_i))`
- Send `Sign(hashArray, sk_s)` the Owner.
end

Function SendFile($F$):
- Send $m_i, \forall i \in [1, n]$ to the Owner.
end

Function GenResponse_AuB($Q$):
- $\sigma = \prod_{(i,\nu_i) \in Q} \sigma_i^{\nu_i}$
- $\mu = \sum_{(i,\nu_i) \in Q} \nu_im_i$
- Send $(\sigma, \mu)$ to the Auditor.
end

Function GenResponse_PPAuB($Q$, firstFlag, $r$):
- $\sigma = \prod_{(i,\nu_i) \in Q} \sigma_i^{\nu_i}$
- if `firstFlag == 1` then
  - Generate random element $r \in Z_p$
- end
- $R_r = e(u, v)^r$
- $\gamma = h(R_r)$
- if `firstFlag == 1` then
  - $\mu = r + \gamma \times \sum_{(i,\nu_i) \in Q} \nu_im_i$
- else
  - $\mu = \gamma \times \sum_{(i,\nu_i) \in Q} \nu_im_i$
- end
- Send $(R_r, \sigma, \mu)$ to the Auditor.
end

terminated immediately. This may happen due to multiple reasons like insufficient funds and banned user attempts.

In Phase 1 (Owner-Server Phase), $O$ uses `FileTransfer_AuB()` to break file $F$ into chunks and send file chunks $m_i$ to $S$ along with authentication tags $\sigma_i$. $S$ receives the file chunks using `ReceiveFile()` and stores $(m_i, \sigma_i)$. $S$ sends back signed digest information to $O$ who countersigns it and sends to $L$ by calling `ReceiveSignedDigest()`. In case the digest received is not duly signed, the contract terminates the execution of the protocol. $O$ sends the audit credentials to $A$ using `SendCredsToAuditor_AuB()`, who receives it using `ReceiveCredsFromOwner_AuB()`.

In Phase 2 (Server-Auditor Phase), $A$ opens a channel with $S$ by calling `OpenChannel()` with $L$. $A$ then generates signed queries using `GenQuery()` and sends it to $S$. Upon receiving query set $Q$, $S$ first checks whether nonce and previous states are updated. If not, it immediately closes the channel using `CloseChannel_AuB()`.

In Phase 3 (Owner-Server Phase), $O$ retrieves the file $F$ using `FetchFile()`. $S$ responds using `SendFile()`.

We show that $AuB$ is secure in section 6.

5.2 Privacy Preserving Audit using Blockchain (PPAuB)

We outline only the differences in the protocol flow from $AuB$.

In Phase 0 (Initialization Phase), $O$ generates the keys and public parameters using `KeyGen_PPAuB()`.

In Phase 1 (Owner-Server Phase), $O$ uses `FileTransfer_PPAuB()` to send over the file to $S$. Unlike in $AuB$, the `FileTransfer_PPAuB()`
uses a file identifier and a different tag generation formula. \(O\) sends the audit credentials to \(A\) using 
\texttt{SendCredsToAuditor_PPAuB()} which receives it using \texttt{ReceiveCredsFromOwner_PPAuB()} which additionally deals with the file identifier.

In Phase 2 (Server-Auditor Phase), both \(S\) and \(A\) use \texttt{CloseChannel_PPAuB()} to close the state channel opened using \texttt{OpenChannel()}. Similarly, \(S\) uses \texttt{GenResponse_PPAuB()} for calculating response to queries and \(A\) uses \texttt{VerifyResponse_PPAuB()} to verify received responses. A thing to note here is that after opening the channel, \(S\) uses \texttt{firstFlag=1} once to generate the random number \(r\). On successive calls in the same channel, it uses \texttt{firstFlag=0} and supplies \(r\) to the function \texttt{GenResponse_PPAuB()}, thereby reusing the randomness. \(\mu\) is also calculated differently in both the cases. Also, \(A\) aggregates the responses and then verifies the aggregated response in \texttt{VerifyResponse_PPAuB()}. In case of failure, \(A\) closes channel and raises a complaint. \(S\) should send an audit response over the last query in the query set in case of a complaint, and it should use \texttt{GenResponse_PPAuB()} with \texttt{firstFlag=0} to generate the response and send \((R, r, \sigma + r, m\mu)\) to the smart contract. This is because the verification equation needs the value \(r\) (as before) to pass the check and without \(r\), the smart contract will punish \(S\) even if the response was correct.

Phase 3 (Owner-Server Phase) remains unaltered.

6 Security Analysis

Theorem: If the underlying audit scheme is secure under Authenticity and Extractibility by a dishonest server, the blockchain platform is secure against tampering, the hash function is cryptographically secure, and signature scheme is unforgeable, then no adversary can deny authenticity, extractibility, fairness and privacy, except with negligible probability, unless the adversary controls the data owner, server and auditor together.

Proof: We break this proof into six cases, each representing a combination of malicious actors among the owner, server and auditor. In case all the parties are honest, we need not perform audits as each party executes its role perfectly. We also omit the case where the data owner, storage server and third-party auditor are all malicious. The proof holds for both AuB and PPAuB and we mention the differences if and when applicable.

6.1 Case I: Malicious Server

A malicious server may be given to store a file chunk \(m_i\) but it may claim to have received \(m_i'(\neq m_i)\). This may happen because of error during communication or malicious intent by the server. This attack will fail as both the \texttt{FileTransfer_AuB()} and \texttt{FileTransfer_PPAuB()}
**Algorithm 4: Smart Contracts**

| Function | RegisterOwner\((pk_o, contractTerms)\): |
|-----------|------------------------------------------|
|           | The Owner Deposits required money according to \(contractTerms\). |
|           | Return \(OwnerID\) to the Owner. |
| Function  | RegisterServer\((OwnerID, ServerID)\): |
|           | Store the mapping \((OwnerID, ServerID)\) |
| Function  | RegisterAuditor\((OwnerID, AuditorID)\): |
|           | Store the mapping \((OwnerID, AuditorID)\) |
| Function  | ReceiveSignedDigest\(((m, t), s)\): |
|           | Require \(\text{SigVerify}((m, t), pk_s) = 1\) |
|           | Require \(\text{SigVerify}((m, t), s, pk_o) = 1\) |
| Function  | GetLastBlockHash() |
|           | if \(h_b\) is not set then |
|           | Fetch last block header \(b\). |
|           | Set \(h_b = H(b)\). |
|           | end |
|           | Return \(h_b\). |
| Function  | OpenChannel\((id)\): |
|           | Check if \(id\) matches stored \(AuditorID\). If not, terminate. |
|           | Initialize \(nonce = 0\) and call GetLastBlockHash() |
| Function  | CloseChannel\(_{AuB}\)(\(id, complaintFlag, Q', (σ,μ)\)): |
|           | // \(Q'\) is the query set where each query is signed by \(A\). Let it contain \(k\) queries. |
|           | Unset \(h_b\) |
|           | If \(complaintFlag = 0\), check \(e(σ, g) = e(\prod_{(i,ν_i)∈Q'} H(ν_i)^{μ_i}.u^μ, v)\). If yes, pay \(pk_s\) and \(pk_A\) from deposit according to payment terms. Else, penalize \(A\). Terminate. |
|           | If \(id\) = \(ServerID\), generate \(Q\) from \(h_b\) and compare element wise with \(Q'\). Also check sign in \(Q'\) using \(pk_A\). If sign check or comparison fails, penalize \(A\). Else, penalize \(S\). Terminate. |
|           | If \(id\) = \(AuditorID\), check sign in \(Q'\). If sign verifies, check audit equation and make payment. If audit check fails, penalize \(Server\). |
| Function  | CloseChannel\(_{PPAuB}\)(\(id, complaintFlag, Q', (R,σ,μ)\), fileid): |
|           | Unset \(h_b\) |
|           | Calculate \(W_i = (fileid||i)\) \(∀i ∈ Q'\) |
|           | If \(complaintFlag = 0\), check \(R.e(σ^γ, g) = e(\prod_{(i,ν_i)∈Q'} H(W_i)^{μ_i}.v^μ, v)\). If yes, pay \(pk_s\) and \(pk_A\) from deposit according to payment terms. Else, penalize \(A\). Terminate. |
|           | If \(id\) = \(ServerID\), generate \(Q\) from \(h_b\) and compare element wise with \(Q'\). Also check sign in \(Q'\) using \(pk_A\). If sign check or comparison fails, penalize \(A\). Else, penalize \(S\). Terminate. |
|           | If \(id\) = \(AuditorID\), check sign in \(Q'\). If sign verifies, check audit equation and make payment. If audit check fails, penalize \(Server\). |
function sends a signed hash $h_i$ for each block received. The owner countersigns on this and sends to the blockchain. If the server signs a wrong hash value corresponding to $m_i$, the owner denies to countersign and hence the attempt fails, unless the server finds a collision in the hash function which happens with negligible probability.

The malicious server may delete a data block altogether. The authenticity property of Shacham-Waters or PPSCS guarantees that the server will not be able to prove retrievability of the data for non-negligible fraction of query space with overwhelming probability. Hence the malicious server will hence be punished.

It can happen that the server indeed receives and keeps $m_i$ but during recovery phase supplies $m_i'$ for some $i$. The server has no incentive to do this. We consider players who do not deviate from protocol without sufficient incentive.

The privacy property is satisfied as the auditor is honest in this case and hence the auditor does not attempt to extract information about $F$ from server.

The smart contract verifies the aggregated challenge-response in $\text{CloseChannel}_{\text{AuB}}()$ or $\text{CloseChannel}_{\text{PPAuB}}()$. For a malicious server failing verification, it would penalize the server from the deposit submitted during $\text{OpenChannel}()$. The smart contract uses this penalty to pay for damages to the owner. It also pays the auditor because of correctly performing audits. Hence, the smart contract ensures fairness, unless its execution is tampered with, which happens only if majority peers in the blockchain system are dishonest.

### 6.2 Case II: Malicious Auditor

The extractibility and authenticity property is preserved in this case because the server is honest.

A malicious auditor will attempt to compromise the privacy property by trying to extract information about $F$ from response set $R$. In case of AuB, in order to maintain privacy, the auditor should be prevented from sending more than $(l-1)$ audit requests, where $l$ is the query set size. As server is honest, it will adhere to this regulation and send complaint to blockchain if auditor deviates. In case of PPAuB, PPSCS uses random coefficients to mask responses and guarantees prevention of information extraction about $m_i$.

A malicious auditor may send a query $Q$ to server to receive the correct response $R_i$. It may however store $R'_i(\neq R_i)$ as the received response. In the state channel, both parties maintain common states. The response state will not match and hence the channel will be closed by the server by calling $\text{CloseChannel}_{\text{AuB}}()$ or $\text{CloseChannel}_{\text{PPAuB}}()$. The peers can re-audit for the same reason which the server shall pass. The smart contract penalizes the auditor and hence such an attack fails. The auditor may attempt to not log the query-response pair altogether. We note that the query set is public because the source of randomization is public. Hence, it is not possible to suppress queries without getting penalized. Clearly, the fairness is ensured here as the smart contract pays the honest server, penalizes the auditor from the deposit during $\text{OpenChannel}()$ and use that to reimburse the owner for her loss.

### 6.3 Case III: Malicious Owner

A malicious owner may supply $m_i$ but claim to have sent $m_i'(\neq m_i)$. This is safeguarded as signed hash of file is sent from server to owner who should countersign it and send to blockchain. Without this transaction, the smart contract does not proceed with the audit phase. Once countersigned, the owner cannot deny sending $m_i$, unless she finds a collision which happens with negligible probability.

Extractibility for a malicious owner is not applicable as we only ensure that our guarantees hold for honest players. Authenticity and privacy are guaranteed as the server and auditor are both honest.

The data owner is the source of the money to the system. It sends money to honest servers and auditors and hence a malicious owner may try to deny payment. In our protocol, money is sent to the smart contract before the start of the protocol, in $\text{RegisterOwner}()$. Hence, it is not possible for the owner to deny payments, ensuring fairness.

### 6.4 Case IV: Malicious Server and Auditor

If both are malicious, they can only collude on the outcome of challenge-responses. One attack may be the server submits a wrong response. The auditor consciously ignores a failed audit. But when the aggregated responses are submitted to the smart contract, it catches the failed audit and penalizes the auditor.

Auditor can deliberately skip the audit of a file chunk $i$. But, because we derive the randomness from a public source, the smart contract can verify if the queries were correctly generated or not. In case of cheating, the auditor gets penalized. Hence extractibility and authenticity is guaranteed for an honest data owner with all but negligible probability. In case the server and auditor fail to prove to the smart contract that audit was performed correctly, they are penalized and hence the system guarantees fairness for an honest owner.

As the adversary already controls both server and auditor, it has access to the file. Hence, the privacy property is not applicable in this case.

### 6.5 Case V: Malicious Owner and Auditor

Except for the transfer of $u, pk$ from owner to auditor during phase 1, there is no real interaction between the owner and the auditor. The parameters are sent through the smart contract and hence non-repudiation is ensured. The smart contract handles all the payments to the server. In the audit phase, there is no owner involvement and hence this case reduces to that of a malicious auditor. In effect, the owner and auditor can collude but cannot affect the protocol. The extractibility and privacy properties are not applicable as the adversary controls both the owner and auditor. Fairness for an honest server is assured by the smart contract as it already freezes owner deposit before start of protocol.

### 6.6 Case VI: Malicious Owner and Server

A malicious owner and server can only collude to deny payment to the auditor, thus attacking the fairness property. The other properties like extractibility, authenticity and privacy are not applicable in this scenario. In our protocol the smart
contract handles payment and audit verification. Unless the adversary controls majority computation in the blockchain system, it cannot tamper execution and hence fairness for an honest auditor is guaranteed.

7 IMPLEMENTATION AND PERFORMANCE ANALYSIS

In this section, we analyze a realistic cloud setting of blockchain enabled data audit scheme that we have implemented.

7.1 Implementation Setup

We implement and evaluate a prototype using Ethereum\textsuperscript{32} as a blockchain platform. Our entire code is approximately 1500 lines, consisting of Ethereum smart contracts written in Solidity language, Go-Ethereum modifications written in Golang and other experimentation glue code written in Python and Bash\textsuperscript{17}. We needed to perform Bilinear Pairing checks for symmetric pairing inside Ethereum smart contract, to verify audit responses. In this regard, the original Ethereum platform does not support pairing based operations on symmetric groups natively. Post Byzantium, it had introduced pairing operations on a fixed asymmetric group, in order to support Zero-Knowledge proof verification. It was impractical to port some pairing-based cryptography library into Solidity and hence we modified the Ethereum code to include a new pre-compiled contract which supported pairing-based operations. To be specific, the new pre-compiled contract verified the audit equation given in Eq.1

We have used the most popular Ethereum implementation, Go Ethereum also known as geth, which is written in Golang. For the mathematical operations, we needed a library which supports arithmetic in $Z_p$, elliptic curve groups with operations and bilinear pairing computation. Hence, we used the Golang wrapper\textsuperscript{33} of the popular PBC Library\textsuperscript{34}. We included the PBC library inside the geth code and introduced the pre-compiled contract. We used Type A pairings which are very fast, but elements take a lot of space to represent. Because of the modified Ethereum code, we used a private network for our experimentation.

For the Shacham-Waters audit code implementation, we used their extended definition with file sectors. As discussed in algorithm\textsuperscript{1} we split file $F$ into $n$ blocks $m_1, m_2, ..., m_n \in Z_p$. For each block $m_i$, tag $\sigma_i \in G$ is calculated, where $G$ is group whose support is $Z_p$. Calculating tags for the main file $F$ causes a significant overhead if we tag generation as above. So, we used the concept of sectors as introduced by Shacham-Waters. Let $s$ be a parameter and each block consist of $s$ sectors, where $|s| \in Z_p$. As there is only one tag for one block (contains $s$ sectors), tag generation overhead is reduced by $\approx 1/s$ if $n$ is large enough.

7.2 Evaluation

We deployed our implementation on a private Ethereum network consisting of two nodes. We used a single machine with 4-core Intel Xeon E3 – 1200 and 8GB of RAM running Linux (Manjaro 64-Bit XFCE). The storage server, owner and auditor codes were running alongside the Ethereum nodes. The elliptic curve utilized in our experiment is a supersingular curve, with a base field size 512 bits and the embedding degree 2. We use different file sizes, starting from 1KB to 100MB.

Sector size $|s|$, is 19 Bytes in our construction which is dependent upon parameters we used for the construction of elliptic curves. We use 1000 sectors per block in our construction which we noticed is optimized value for current setup.

The main objective of our prototype implementation was to observe the overhead in introducing the distributed computing platform. In particular, we wanted to calculate latency from each of the parties perspective, i.e., how much additional time does the owner spend in uploading and downloading files, the auditor spends in challenge-response and the server spends on file and audit management. To observe this, we perform same experiment with and without the blockchain related calls and look into the latencies in each case.

Firstly, we look into the latency faced by the owner during upload of file. In Fig.\textsuperscript{5}, we note that although for small files the blockchain latency remains considerable, with increasing file sizes, the commit time becomes negligible compared to the file upload time. As practical storage servers store files in order of Gigabytes, the overhead for the owner is negligible. A thing to note is that the same applies for the server as well because the owner latency includes the server signing during file uploads.

In terms of the overhead to the server for proof generation, the protocol does not demand any additional ledger interactions and hence we observed no latency from the servers perspective.

An auditor performs audit over a long period of time. For example, an auditor may send one query to the server every hour. It may have to send the aggregated response to the smart contract only at the end of the day. Hence, in Fig.\textsuperscript{5} although we observe a dominant overhead of blockchain interaction compared to response verification over 10 queries, we note that the time axis in not an honest representation of practice where the audit will be performed over a considerably long period of time as compared to the commit time.

Overall, in Fig.\textsuperscript{5} we see that for the owner and server, our protocol adds minimal overhead. For the auditor, if it is compared against the span of the entire audit process, the additional latency remains negligible, given that the auditor performs the audit over a sufficiently long duration of time.

Table\textsuperscript{1} shows the metrics calculated with 5 different query sizes keeping the file size constant at 1 MB. The gas cost in USD is calculated at average gas price of 3 gwei and an exchange rate of 1 ETH = 153 USD. The empty block size in our private network is 540 bytes. This shows that if a single audit takes 1400 bytes, 1 MB data on the blockchain would accommodate roughly 750 audits. We note here that the block size increase is not linear to the number of queries as only the aggregated response is submitted to the blockchain. Each channel session can communicate a large

1. https://bit.ly/2L6W55n
2. https://bit.ly/2J0z1Ct
number of audits hence in practice, thousands of such audits can be done with an overhead of few kilobytes.

8 DISCUSSION

We wanted to use blockchain as the source of randomness for generating query set. As given in [35], for small amounts of randomness, if the stakes are low enough, the blockchain can be used as a source of randomness. We believe that for our audit purposes, the incentive for parties to collude with miners is low enough. Any other public source of randomness could have been used. External sources of randomness have a separate trust assumption and then we would have needed to consider all the collusion cases with the random source. Random beacons assume honest majority unlike commit based randomness protocols, but both need multiple commits to generate randomness are are hence slow. Our only requirement is that the peers of the blockchain network need to have access to the same source and must access the same random value in order to receive consensus. We referred a single block hash for each contract instance, hence the query set for a channel can be derived at once, after the opening of the channel.

File upload time is very much dependent on number of sectors per block as well as size of sector. Sector size $|s|$, is determined by the parameters and choice of algorithm used for elliptic curve generation.

We have implemented our pairing check as a new pre-compiled contract. Hence, the gas required by the contract has been estimated by us. In a practical situation, either such a symmetric bilinear pairing support comes baked into Ethereum, in which case the community decides upon the gas cost, or, a private network is setup among interested parties where they themselves decide upon the gas requirement. The asymmetric pairing check pre-compiled contract takes $80000 * k + 100000$ as the gas ($k$ is the number of points on the curve). Upon using similar calculation, our audit check transaction took 888387 gas. We have not used this in our performance metric as we think this will depend upon the platform.

The channel closing codes written in our contract is far from ideal. It does not take into account all possible corner cases, but arbitrary complicated code could have been implemented based on the requirements. We have just showed a sample code for the prototype.

9 CONCLUSION

In this paper we introduced a blockchain based privacy preserving audit protocol which is resilient even when any two out of the data owner, storage server and auditor is malicious. We used state channels to minimize blockchain commits thereby improving efficiency. Through smart contracts, we enforced the incentive mechanism in the system. We also build a prototype on modified Ethereum and show that the protocol incurs minimal overhead compared to existing PoR scheme.

In terms of future work, we wish to explore possibilities to enhance efficiency of the protocol by using other elliptic curves. We also aim to adopt an audit protocol without bilinear pairing operations so that it can be readily deployed on blockchain platforms like Ethereum, without modifying the codebase. This would enable us to test on networks beyond a private network, like testnets and main network.

ACKNOWLEDGMENT

This work is partially supported by Cisco University Research Program Fund, CyberGrants ID: #698039 and Silicon Valley Community Foundation. The authors would like to thank Chris Shenefiel and Samir Saklikar for their comments and suggestions.

REFERENCES

[1] I. Bentov and R. Kumarasen, “How to Use Bitcoin to Design Fair Protocols.” Springer, Berlin, Heidelberg, 2014, pp. 421–439. [Online]. Available: http://link.springer.com/10.1007/978-3-662-44381-1{
[2] S. Dziembowski, L. Ebeck, and S. Faust, “FairSwap,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS ’18. New York, New York, USA: ACM Press, 2018, pp. 962–984. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3243734.3243857
[3] S. D. Galbraith, K. G. Paterson, and N. P. Smart, “Pairings for cryptographers,” Discrete Applied Mathematics, vol. 156, no. 16, pp. 3113–3121, sep 2008. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167506008000449
[4] A. Juels and B. S. Kaliski Jr, “PORe: Proofs of retrievability for large files,” in Proceedings of the Computer and communications security, 14th ACM conference on, 2007.
[5] H. Shacham and B. Waters, “Compact Proofs of Retrievability.” Springer, Berlin, Heidelberg, 2008, pp. 90–107. [Online]. Available: http://link.springer.com/10.1007/978-3-540-89255-7{
[6] C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou, “Privacy-Preserving Public Auditing for Secure Cloud Storage,” IEEE Transactions on Computers, vol. 62, no. 2, pp. 362–375, feb 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6109247/
[7] N. Satoshi and S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic cash system,” Bitcoin, 2008.
[8] G. Wood, “Ethereum: A Secure Decentralised Generalised Transaction Ledger EIP-150 Revision,” in Ethereum Project Yellow Paper, 2014.
[9] G. Ateniese, S. Kamara, and J. Katz, “Proofs of Storage from Homomorphic Identification Protocols,” in Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology. Springer-Verlag, 2008, pp. 319–333. [Online]. Available: http://link.springer.com/10.1007/978-3-540-89255-7{
[10] M. Naor and G. N. Rothblum, “The complexity of online memory checking,” Journal of the ACM, vol. 56, no. 1, pp. 1–46, jan 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1462153.1462155
[11] Y. Dodis, S. Vadhan, and D. Wichs, “Proofs of Retrivaerbility via Hardness Amplification,” in Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography. Springer-Verlag, 2009, pp. 109–127. [Online]. Available: http://link.springer.com/10.1007/978-3-642-04357-5{
[12] C. Erway, A. Kürçü, C. Papamanthou, and R. Tamassia, “Dynamic provable data possession,” in Proceedings of the 16th ACM conference on Computer and communications security - CCS ’09. New York, New York, USA: ACM Press, 2009, p. 213. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1653662.1653688
[13] K. D. Bowers, A. Juels, and A. Oprea, “HAIP,” in Proceedings of the 16th ACM conference on Computer and communications security - CCS ’09. New York, New York, USA: ACM Press, 2009, p. 187. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1653662.1653688
[14] F. Armknecht, J.-M. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter, “Outsourced proofs of retrievability,” in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014, pp. 831–843.
| No. of Queries | Gas Cost | Gas Cost in USD | Block Size Overhead | Auditor Total Time | Auditor BC Wait Time |
|---------------|----------|----------------|---------------------|-------------------|---------------------|
| 1             | 110087   | 0.0512         | 1394                | 72.808527ms       | 6.220183447s       |
| 3             | 115220   | 0.05358        | 1458                | 65.074692ms       | 6.21555321s        |
| 5             | 120486   | 0.05603        | 1554                | 69.638309ms       | 5.818474706s       |
| 7             | 125619   | 0.05842        | 1618                | 67.183777ms       | 6.219303512s       |
| 10            | 133451   | 0.06206        | 1746                | 69.449573ms       | 6.220183447s       |

TABLE 1: Table to show different measures while auditing with varying query size for upload file of 1MB.

[15] D. Cash, A. Küpçü, and D. Wichs, “Dynamic Proofs of Retrievability Via Oblivious RAM,” Journal of Cryptology, vol. 30, no. 1, pp. 22–57, Jan 2017. [Online]. Available: http://link.springer.com/10.1007/s00145-015-9216-2

[16] J. Dautrich, E. Shi, and E. Stefanov, “Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns,” in USENIX Security Symposium. USENIX, 2014, pp. 749–764. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/dautrich

[17] Y. Wang, Q. Wu, B. Qin, S. Tang, and W. Susilo, “Online/offline provable data possession,” IEEE Transactions on Information Forensics and Security, vol. 12, pp. 1182–1194, 2017.

[18] B. Sengupta and S. Ruj, “Efficient proofs of retrievability with public verifiability for dynamic cloud storage,” IEEE Transactions on Cloud Computing, vol. PP, 10 2017.

[19] M. B. Paterson, D. R. Stinson, and J. Upadhyay, “Multi-prover proof of retrievability,” J. Mathematical Cryptology, vol. 12, pp. 203–220, 2016.

[20] J. Benet, “IPFS - Content Addressed, Versioned, [P2P] File System,” CoRR, 2014.

[21] S. Wilkinson, T. Boshevski, J. Brandoff, J. Prestwich, G. Hall, and Communications Security, ser. ASIA CCS ’16. New York, NY, USA: ACM, 2016, pp. 107–118. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897915

[22] Y. Wang, Q. Wu, B. Qin, S. Tang, and W. Susilo, “Online/offline provable data possession,” IEEE Transactions on Information Forensics and Security, vol. 12, pp. 1182–1194, 2017.

[23] B. Sengupta and S. Ruj, “Efficient proofs of retrievability with public verifiability for dynamic cloud storage,” IEEE Transactions on Cloud Computing, vol. PP, 10 2017.

[24] M. B. Paterson, D. R. Stinson, and J. Upadhyay, “Multi-prover proof of retrievability,” J. Mathematical Cryptology, vol. 12, pp. 203–220, 2016.

[25] J. Benet, “IPFS - Content Addressed, Versioned, [P2P] File System,” CoRR, 2014.
Prabal Banerjee  Prabal Banerjee joined Indian Statistical Institute as a Ph.D. student in 2016. He received his B.Sc. and M.Sc. in Computer Science from St. Xavier’s College, Kolkata and Chennai Mathematical Institute, Chennai respectively. He is currently working on integration of data with blockchain.

Nishant Nikam  Nishant Nikam joined Indian Statistical Institute as a Ph.D. student in 2018. He received his B.Tech. and M.Tech. in Computer Science from National Institute of Technology, Warangal and Indian Statistical Institute, Kolkata respectively. He is currently working on cloud storage security.

Sushmita Ruj  Sushmita Ruj is currently an Assistant Professor at Indian Statistical Institute, Kolkata, India. Her research interests are in blockchains, IoT security, cloud security and privacy.