Project Report - Securing Information Exchange with Blockchain

Harsh Kedia

December 17, 2018

Abstract

Existing applications for sharing files are central solutions and therefore suffer from the single point of failure risk. Moreover, using central services for securing data means that we have to trust a 3rd party with our data thus exposing it to manipulation risks. Hence, a decentralized application was required to overcome the problems posed by a central application. With the recent developments in Blockchain technology and P2P storage, it's possible to securely store and share data without using any central service.

This report outlines my master's project which is about securing information exchange using Blockchains. I describe here the workings of my application *dShare* built using P2P technologies enabling a secure way of storing and sharing data between two individuals or entities.

For storing files, I use InterPlanetary File System (IPFS), which is a P2P file storage protocol. Before uploading to the IPFS network, files are encrypted using AES-GCM encryption mechanism. Sharing of keys is facilitated using smart contracts built on Ethereum, thus files can be shared by anyone having a Ethereum address. Finally, for immutable timestamping, OriginStamp is used, which submits a file's hash to the Bitcoin blockchain.

I implemented the front-end using React.js, a JavaScript framework. Solidity was used to write smart contracts and deployed on the Ethereum test network (Rinkeby). Next.js was used for server side rendering (SSR) and Firebase was used as a database for storing public Ethereum key of the users. Latest version of the application is deployed on Heroku at https://file-share-dapp.herokuapp.com/.

1 Project Motivation

Today's supply chain spans multiple geographies but the documents involved in the industry such as delivery certificates are still in physical form. This paperwork prevents manipulations but leads to various delays across the whole chain, thus affecting everyone involved.

Above problem can be solved by digitizing all documents, time-stamping them using a trusted Time stamping authority (TSA) and upload them to a cloud service. However, the tools used to accomplish this solution are central services, and, therefore, suffers from data manipulations by a 3rd party.

We, therefore, need a solution that is peer-to-peer (P2P) and decentralized. Blockchain offer a means of decentralized time stamping and enables peer-to-peer storage systems that are resistant to manipulations by any 3rd party.

This project describes the working of an application build using decentralized technologies such as Bitcoin, Ethereum and IPFS such that it's capable of immutable timestamping and secure file sharing in a P2P manner.

2 Related Work

Below is a list of related work in immutable timestamping and decentralized storage explored as part of the project.

2.1 Proof of Existence

Proof of Existence[1] is a web based service that implements the concept of immutable timestamping using the Bitcoin blockchain. It notarizes data in the blockchain by submitting the hash of the data in a Bitcoin transaction. Currently, the service required 0.00025BTC[2] for every certification which makes it expensive to timestamp large volumes of data.

2.2 OriginStamp

OriginStamp[2][3] extends the concept of Proof of Existence by providing a scalable protocol which overcomes the transaction limitations of the Bitcoin blockchain.

When a user submits a file, the hash of the data is recorded. Instead of creating a Bitcoin transaction for each submitted hash, it combines all the hashes submitted over a time period and generates an aggregated hash. After some additional hashing and encoding operations, a Bitcoin address is created, to which the smallest possible transactional amount of Bitcoins is transferred. Performing this transaction embeds the hash and the timestamp permanently to the Bitcoin blockchain.

2.3 Chainpoint

Chainpoint[4] works similarly to OriginStamp. The service runs on the Tierion[5] Network, providing a scalable protocol for anchoring data in the blockchain and generating blockchain receipts. These receipts are called chainpoint proofs which defines a path of operations that cryptographically links the data to one or more blockchains.

2.4 Sia

Sia[3] is a decentralized cloud storage system that allow its users to rent storage among peers by means of storage contracts which are cryptographically secured by saving on a blockchain. This makes the storage contracts tamper-proof and publicaly auditable.

To ensure that storage provider holds a clients data at a given time, they constantly need to submit storage proofs. The network consensus allows automatic

[1] https://poex.io/
[2] https://poex.io/prove
[3] https://originstamp.org/
[4] https://chainpoint.org/
[5] https://tierion.com/

verification of storage proofs and enforcement of storage contracts. The availability of data is ensured using redundancy techniques such as erasure codes.

Sia uses a variant of Bitcoin blockchain for storing the contracts and the user must use Siacoin, an ERC-20 token in order to transact on the Sia network.

2.5 Storj

Storj[4] works similarly to Sia. It's built on Kademlia DHT, connecting peers who can transact with each other. A transaction can involve negotiation of storage contract, transfer of data, verifying remote data, download data or payments to other nodes. Each peer is capable of doing transactions independently without any human involvement.

Storj uses the Ethereum blockchain for managing its storage contracts. They are stored as versioned data structure describing the relationship between a client and a storage provider. Users must use Storjcoin, an ERC-20 token to perform transactions on the Storj network.

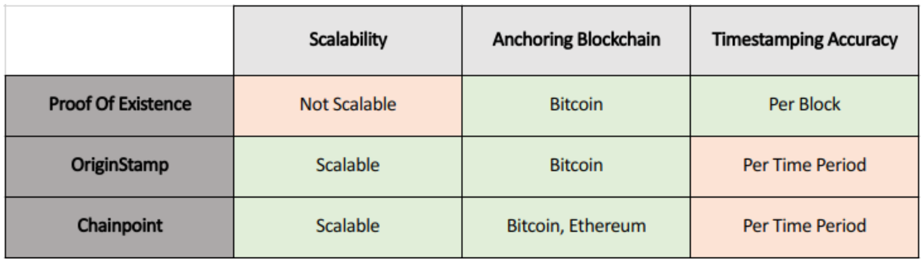
2.6 InterPlanetary File System (IPFS)

Unlike Sia and Storj, IPFS[1] is a P2P file transfer protocol which connects all peers in the network by a shared file system. It achieves this by combining previous peer-to-peer systems such as DHT, BitTorrent, and Git. The data in the IPFS network are modeled as a Merkle DAG thus providing a throughput storage system with content-addressed hyperlinks.

To transact on the IPFS network, a user does not need any tokens.

3 Approach

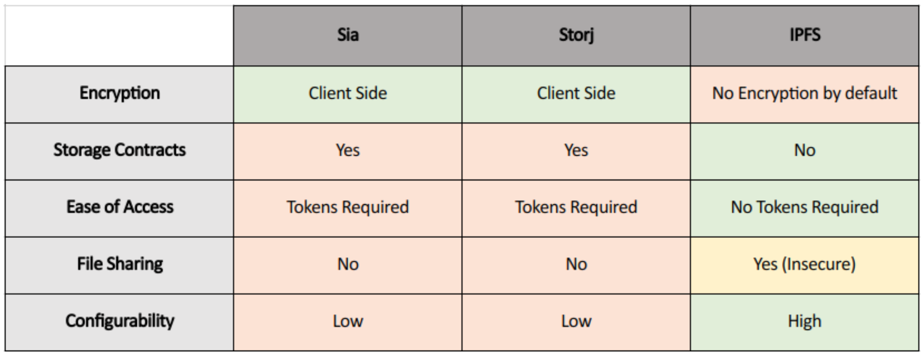
Below is a comparison between different timestamping methods and decentralized storage techniques explored above.



	Scalability	Anchoring Blockchain	Timestamping Accuracy
Proof Of Existence	Not Scalable	Bitcoin	Per Block
OriginStamp	Scalable	Bitcoin	Per Time Period
Chainpoint	Scalable	Bitcoin, Ethereum	Per Time Period

Figure 1: Comparing Decentralized Timestamping

Comparing the decentralized timestamping solutions, Proof of Existence creates a Bitcoin transaction for each hash submitted by the user. Moreover, each certification costs 0.00025 BTC. These limitations make it impractical and expensive for timestamping large volume of data. Both OriginStamp and Chainpoint, instead of creating a transaction for each submitted hash, concatenates the submitted hashes over a period and creates a single transaction with the aggregated hash. Thus they overcome the limitations of Proof of Existence and provide a scalable protocol which can handle large volumes of data.



	Sia	Storj	IPFS
Encryption	Client Side	Client Side	No Encryption by default
Storage Contracts	Yes	Yes	No
Ease of Access	Tokens Required	Tokens Required	No Tokens Required
File Sharing	No	No	Yes (Insecure)
Configurability	Low	Low	High

Figure 2: Comparing Decentralized Storage

Comparing the decentralized storage systems, both Sia and Storj provide an encrypted data storage; however current implementations do not allow for file sharing. Moreover, both require platform specific crypto tokens to access the network. IPFS, on the other hand, does not encrypt files by default. Files on the IPFS network are accessed by their hashes; thus anyone who knows the files hash can access the file. There are no storage contracts involved for storing files on the network. IPFS network does not require any crypto token for the users to access the network.

Looking at the limitations of the existing solutions, we propose a solution for secure information exchange with blockchains using smart contracts, immutable timestamping and decentralized storage.

For timestamping both OriginStamp and Chainpoint can be used as they can handle large volumes of data and provide a rich set of APIs for integrating decentralized timestamping in any application. For decentralized storage, we want to use IPFS as it is a general-purpose file storage protocol and does not require any crypto tokens for access to its network and for exchanging document encryption keys, we want to use the Ethereum smart contracts.

4 Implementation

This section describes the workings of the application.

4.1 Smart Contract

The smart contract serves as the bridge between the front-end of the application and the Ethereum Blockchain. Data is read from and written to the blockchain with the help of function calls in the contract. Each function call which modifies some data requires a small fees in the form of gas which defines the cost for a function execution in Ether. Reading from the blockchain does not require any fees.

dShare makes use of two contracts, *FileFactory*, which acts as the factory contract for creation of new files and *File*, which represents an individual file.

4.1.1 FileFactory Contract

FileFactory is the contract which is deployed on the Rinkeby test network. It has several *mappings* which stores the list of file contracts uploaded by a user. Below is a list of variables used to store relevant data regarding files.

```
/// @notice Stores the list of files deployed by an address
mapping(address => address[]) uploadedFiles;

/// @notice Stores the list of files shared by an address
mapping(address => address[]) sharedFiles;

/// @notice Stores the list of files shared with an address
mapping(address => address[]) recipientFiles;

/// @notice Stores the list of files archived by all users
address[] archivedFiles;

/// @notice List of Uploaders
mapping(address => bool) uploaders;

/// @notice List of Recipients
mapping(address => bool) recipients;
```

Whenever a user uploads a file, a function call is made to the *FileFactory* contract, which in turn deploys the *File* contract and updates the *mappings* for `uploadedFiles` and `uploaders`.

```
/// @notice Deploys a file contract to the blockchain
/// @dev Deploys a new contract which stores file details
/// @param _d Hash function digest
/// @param _h Hash function code
/// @param _s Size of _digest in bytes
/// @param _f sha3 hash of the uploaded file
function createFile(bytes32 _d, uint8 _h, uint8 _s, bytes32 _f) public {
  address newFile = new File(_d, _h, _s, _f, this, msg.sender);
  uploadedFiles[msg.sender].push(newFile);
  uploaders[msg.sender] = true;
}
```

4.1.2 File Contract

File contract is deployed to the blockchain whenever a file is successfully uploaded to the IPFS network using *dShare*. Upon deployed, the `constructor` function is called. It takes the values passed by the *FileFactory* contract and saves the details to it's *File* contract.

```
/// @notice Initializes the variables with values passed by the factory
/// @dev The constructor calles upon file deployment
/// @param _d Hash function digest
/// @param _h Hash function code
/// @param _s size of _digest in bytes
/// @param _fi sha3 hash of the file
/// @param _fa The address of the factory contract
/// @param _c The address of the uploader
constructor(bytes32 _d,uint8 _h,uint8 _s,bytes32 _fi,address _fa,address _c) public {
  fileIpfsHash = Multihash(_d, _h, _s);
  manager = _c;
  sha3hash = _fi;
  ff = FileFactory(_fa);
}
```

Below is a list of variables, a *File* contract uses to store a file's details.

```
/// @notice The address of the uploader
address public manager;

/// @notice sha3 hash of the file
bytes32 sha3hash;

struct Multihash {
  bytes32 digest;
  uint8 hashFunction;
  uint8 size;
}

/// @notice The address of the factory contract
FileFactory ff;
```

```
/// @notice The IPFS hash of the file
Multihash fileIpfsHash;

/// @notice List of recipients the file is shared with
address[] recipientsList;

/// @notice Stores the encrypted key's IPFS hash for each recipient
mapping(address => Multihash) keyLocation;
```

4.2 File Upload

Figure 3 visualizes the working of the application when a user uploads a file.

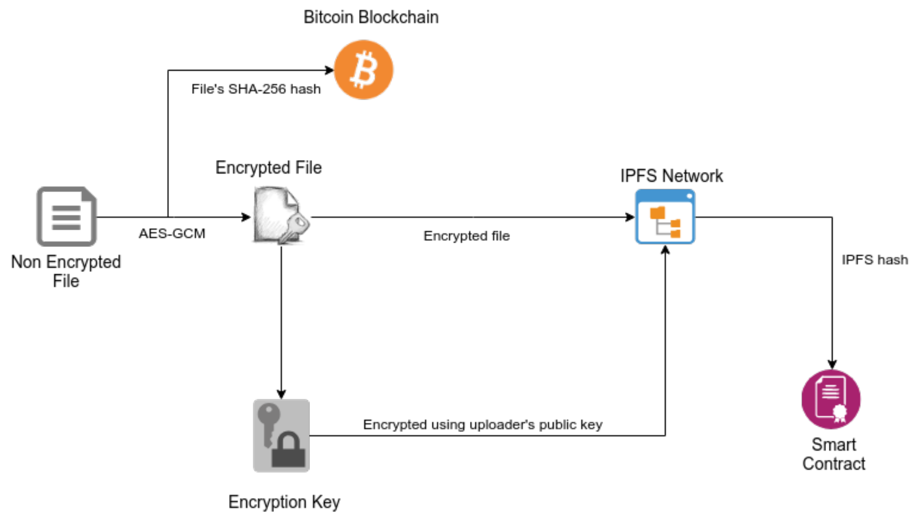


Figure 3: Uploading a File using *dShare*

As soon as a user submits a file to be uploaded, it's sha-256 hash is calculated and a timestamp is created by submitting the hash to the bitcoin blockchain using the OriginStamp API.

```
// get the sha256 hash of file
const sha256hash = await sha256(this.state.buffer);

// create timestamp
try {
  await createTimeStamp(sha256hash, this.state.email);
} catch (error) {
  toast.error("Error in creating timestamp");
  return;
}
```

Next, the file is encrypted using the SubtleCrypto[6] interface with 'AES-GCM' as the encrypting algorithm. The encrypted data is then combined with the random salt to generate a `Uint8Array` buffer ready to be uploaded to the IPFS network.

```
// encrypt the file
const { data, iv, key } = await encrypt(this.state.buffer);
```

[6] https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto

```
const dataArray = new Uint8Array(data);

//combine the data and random value
const data_iv = new Uint8Array([...dataArray, ...iv]);
```

The key used to encrypt the file is converted to JSON and is encrypted using the uploader's Ethereum public key which is retrieved from the database.

```
//encryption key in JSON
const keyData = await window.crypto.subtle.exportKey("jwk", key);

// getting the public key
const snapshot = await db
  .ref("/users/" + this.state.account.toLowerCase())
  .once("value");
const publicKey = snapshot.val() && snapshot.val().public_key;

//encrypt the document key with user's ethereum public key
const encryptedKey = await EthCrypto.encryptWithPublicKey(
  publicKey,
  Buffer.from(JSON.stringify(keyData))
);
```

This encrypted key and the encrypted data is then uploaded to the IPFS network.

```
//Contruct the data to be uploaded to ipfs
const ipfsPayload = [
{
  path: `/tmp/${this.state.fileName}`,
  content: Buffer.from(data_iv)
},
{
  path: `/tmp/${this.state.account}`,
  content: Buffer.from(JSON.stringify(encryptedKey))
}
];

// uploading file to ipfs
await ipfs.files.add(ipfsPayload, (err, res) => {
  if (err) {
    console.error(err);
    return;
  }
  this.setState({ fileIpfsHash: res[2].hash }, () => {
    this.createFile(this.state.fileIpfsHash, sha256hash);
  });
});
```

Once the file is successfully uploaded, `createFile()` in the `FileFactory` contract is called which deploys a new `File` contract with all details regarding the file saved to the blockchain.

```
createFile = async (fileIpfsHash, sha256hash) => {
  const { digest, hashFunction, size } = getBytes32FromMultiash(fileIpfsHash);

  try {
    await factory.methods
    .createFile(digest, hashFunction, size, "0x" + sha256hash)
    .send({
      from: this.state.account
    });
    Router.push("/files/");
  } catch (error) {
    toast.error(error.message);
  }
  this.setState({ loading: false });
};
```

4.3 File Sharing

Sharing a file requires the recipient's Ethereum address and uploader's private key. Figure 4 visualizes the working of the application when a user shares a file with another user.

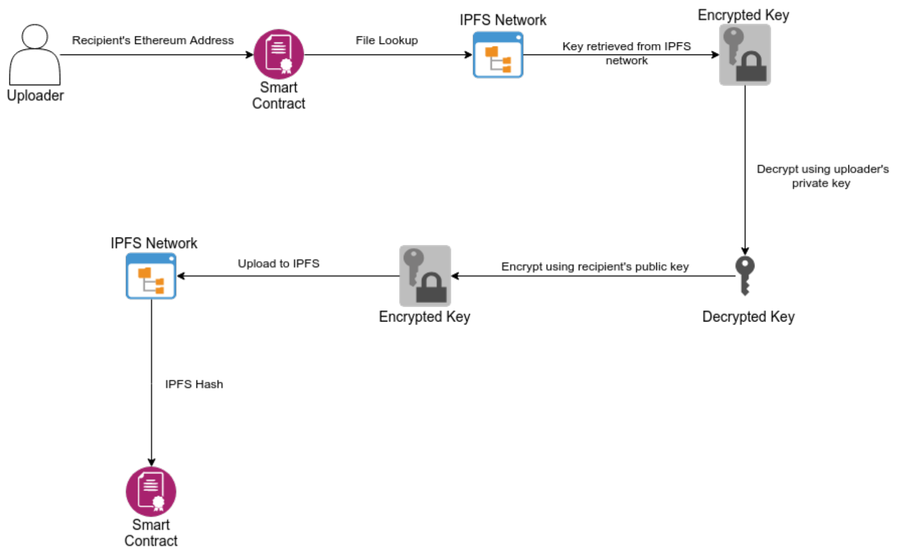


Figure 4: Sharing a File using *dShare*

Firstly, the file's IPFS location is retrieved from the `File` contract. From this location, the encrypted is download and decrypted using uploader's private key.

```
// get File's IPFS hash from Contract
const returnedHash = await fileInstance.methods.getFileDetail().call({
  from: accounts[0]
});

// Retrive the encrypted key
await ipfs.files.cat(
  `${this.state.fileIpfsHash}/${accounts[0]}`,
  (err, file) => {
    this.setState({ fileEncryptedkey: JSON.parse(file.toString("utf8")) });
  }
);

// Decrypt the file key using user's private key
let decryptedKey;
try {
  decryptedKey = await EthCrypto.decryptWithPrivateKey(
    this.state.userPrivateKey,
    this.state.fileEncryptedkey
  );
} catch (error) {
    toast.error("Invalid Private Key");
    this.setState({ loading: false });
    return;
  }
```

Once decrypted, the key is again encrypted using recipient's public key. The new encryted key is again uploaded to the IPFS network.

```
// Encrypt the file key using recipient's public key
const keyForSharing = await EthCrypto.encryptWithPublicKey(
  recipientPublicKey,
  Buffer.from(JSON.stringify(JSON.parse(decryptedKey)))
);

// Contruct the ipfs payload
const ipfsPayload = [
{
  path: `${this.state.recipient}`,
  content: Buffer.from(JSON.stringify(keyForSharing))
}
];

// uploading to ipfs
await ipfs.files.add(ipfsPayload, (err, res) => {
  if (err) {
    return;
  }
  this.setState({ keyIpfsHash: res[0].hash }, () => {
    this.shareFile(this.state.keyIpfsHash);
  });
});
```

Finally, the IPFS location is the key is saved into the `File` contract by calling `shareFile()`.

```
try {
  await fileInstance.methods
    .shareFile(this.state.recipient, digest, hashFunction, size)
    .send({ from: this.state.account });

    Router.push("/files/");
  } catch (error) {
    toast.error(error.message);
  }
```

4.4 File Download

Downloading a file requires the user's Ethereum private key. Depending on whether the file is a uploaded or shared one, respective function from the `File` contract is called to retrieve the file's details.

```
/** Retrieve the File Name */
await ipfs.files.get(this.state.fileIpfsPath, (err, files) => {
  if (err) {
    throw err;
  }
  this.setState({
  fileName: files[2].path.split("/").pop(),
  fileContent: files[2].content
  });
});

/** Retrive the encrypted key */
await ipfs.files.cat(this.state.keyIpfsPath, (err, file) => {
  if (err) {
    throw err;
  }
  this.setState({ encryptedKey: JSON.parse(file.toString("utf8")) });
});
```

The key is then decrypted using user's private key and is converted to a valid JSON web key (jwk)[7] format.

```
/** Decrypt the key using user's private key */
let decryptedKey;
try {
  decryptedKey = await EthCrypto.decryptWithPrivateKey(
    this.state.userPrivateKey,
    this.state.encryptedKey
  );
} catch (error) {
  toast.error("Invalid Private Key");
  this.setState({ loading: false });
  return;
}

/** Convert key into valid jwk format */
const key = await window.crypto.subtle.importKey(
  "jwk",
  JSON.parse(decryptedKey),
```

[7] https://tools.ietf.org/html/rfc7517

```
  "AES-GCM",
  true,
  ["encrypt", "decrypt"]
);
```

The encrypted file data is then converted to a file buffer and the original file content and the random salt used for encrypting the file is retrieved.

```
// Retrieve the original file Content
const fileBuffer = fileContent.slice(0, fileContent.length - 12);

// Retrive the original random nonce used for encrypting
const iv = fileContent.slice(fileContent.length - 12);
```

Finally, the file is decrypted and saved to the user's local storage.

```
// Decrypt the file
const decryptedFile = await decrypt(fileBuffer, key, iv);

// Contruct the file
const file = new File([decryptedFile], this.state.fileName);
FileSaver.saveAs(file);
```

To stop sharing a file, a function call can be made to the `File` contract with the recipient's address, which deletes the contract reference from the `recipientFiles` array.

```
/// @dev Removes the file's address from sharedFiles and recipientFiles
/// @param _iO The index of file in sharedFiles
/// @param _iR The index of file in recipientFiles
/// @param _r The address of recipient
/// @param _f the Address of uploader
function stopS(uint _iO,uint _iR,address _r,address _f) public isUploader(_f) {
  removeByIndex(_iO, sharedFiles[_f]);
  removeByIndex(_iR, recipientFiles[_r]);
}
```

4.5 File Archiving

Instead of deleting a `File` contract, *dShare* provides a way to achieve files. This is also useful to keep track of archived files and restore them at a later date, if required.

When a file is archived, the `File` contract address is saved in array which is later used for filtering the archived files from the UI.

Restoring a file removes the `File` contract address from the archived files array.

```
/// @dev Adds the archieved file address to archivedFiles array
/// @param _file The address of the deployed file to be archived
/// @param _from the address of the uploader
function archiveFile(address _file, address _from) public isUploader(_from){
  archivedFiles.push(_file);
}

/// @dev Deletes the specified entry from archiveFile array
/// @param _index The index of the file in archiveFile array
/// @param _from The address of the uploader
function restoreFile(uint _index, address _from) public isUploader(_from) {
  removeByIndex(_index, archivedFiles);
}

/// @dev Function to delete element from an array
/// @param _index The index of element to be removed
/// @param _array The array containing the element
function removeByIndex(uint _index, address[] storage _array) internal {
  _array[_index] = _array[_array.length - 1];
  delete _array[_array.length - 1];
  _array.length--;
}
```

5 Results

6 Future Work

References

[1] Benet, J. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561* (2014).

[2] Gipp, B., Meuschke, N., and Gernandt, A. Decentralized trusted timestamping using the crypto currency bitcoin. *arXiv preprint arXiv:1502.04015* (2015).

[3] Vorick, D., and Champine, L. Sia: simple decentralized storage, 2014.

[4] Wilkinson, S., Boshevski, T., Brandoff, J., and Buterin, V. Storj a peer-to-peer cloud storage network.