skizzle

Whitepaper

v1.0

Author
Mayur Relekar

Editors
Saurav Kanchan
Abhishek Chaudhary
Aravindh Kumar

Abstract

We share files everyday, for work or for personal use. Some of these files happen to be more important than others. They contain sensitive/valuable information like Intellectual Property, Trade Secrets, Personal Health/Identity Information, Financial documents… We share these files as attachments on Gmail/Outlook or as links to files stored on Google Drive, Dropbox, but…

- Attachments in email are not encrypted and are liable to being breached[1][2]
- When encrypted by the service provider, files are accessible to them and are liable to be mined/hacked[3][4]

To mitigate this and achieve a certain level of security and privacy, we may even password protect or encrypt files ourselves, but…

- Setting, remembering, sharing and just managing passwords/keys is hard[5][6][7]
- Weak passwords and encryption schemes are easily exploited[8]

To overcome this, those of us that can, implement full blown security solutions like DLPs and CASBs or resort to the premium service add-ons from existing service providers, but…

- Doing so, introduces severe process complexities that simply get bypassed so people can get the job done[9]
- High-end security solutions or premium service add-ons are simply too expensive[10]

No matter how big or small, every individual, business and institution has a right to expect a solution that is:

- **Secure** - where your files are immune to hacks.
- **Privacy Preserving** - where your data is not mined or sold.
- **Transparent** - where you can always verify what is happening with your data and who has access to it.
- **Easy to use** - where you don’t need 3-4 apps to simply share a file securely and using it every day is not a chore.
- **Affordable** - where you don’t burn a hole in your pocket to implement and use a solution.

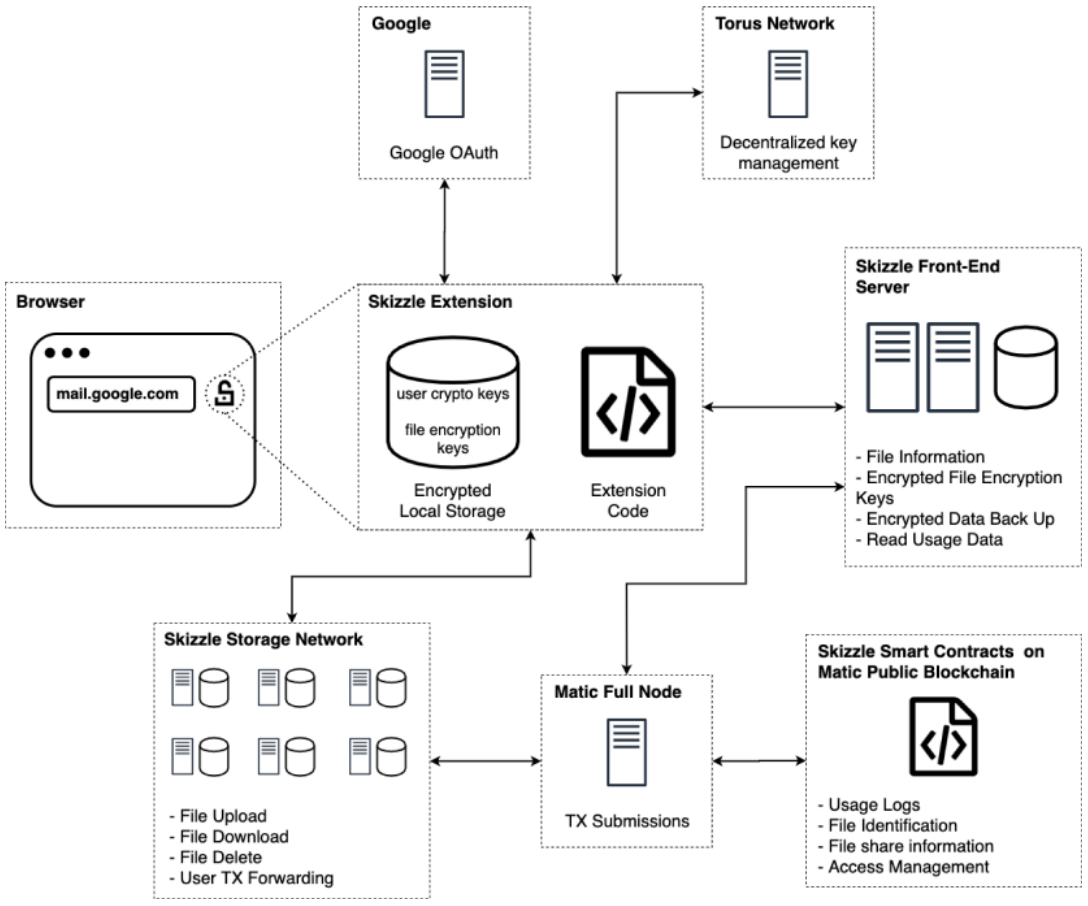
We are building Skizzle to do exactly this and we are on a mission to democratize security and privacy of your data.

This document outlines the tech behind Skizzle.

Motivation

Our motivation lies in the principles of Privacy by Design by Ann Cavoukian.

Architecture



- **Skizzle Extension**
  The Skizzle extension is a VueJS application for the Google Chrome browser. It leverages Chrome's local storage to store private user information and cryptographic material, locally.

- **Authentication Libraries**
  Skizzle uses the DirectAuth sdk from TorusLabs to allow users to securely sign in with their Google accounts(via Google's OAuth) and assign public/private key pairs to them. Every Skizzle user in effect has an Ethereum Wallet assigned to them without ever needing to manage the keys themselves.

- **Skizzle Server**
  For purposes of faster processing and better UX, the Skizzle server is used to:
  - Store information about files that a user has Sent and Received.
  - Store the encrypted file encryption keys to be shared with valid recipients.
  - Process and hold usage logs, fetched from the Skizzle smart contracts on the blockchain, to perform billing and payment related activities.

- **Skizzle Storage Network**
  This is a network of storage nodes which host all the encrypted files uploaded by users, from where valid recipients download files. Each node runs custom Tahoe-LAFS software to create a distributed file system. Its primary tasks include:
  - Storing encrypted files.
  - Perform erasure coding which is a form of file encoding. This essentially breaks up the encrypted file uploaded to the network into multiple pieces and stores each piece on a unique storage node, such that only a subset of the nodes are required to fetch the file, thereby offering efficient redundancy.
  - Handling file download/delete requests.
  - Forwarding signed user transactions to the blockchain. This is done to abstract away transaction fee payments from the user and the complexities and poor UX that come with it.
  - All storage nodes sit behind a firewall on Google Cloud/AWS/DigitalOcean servers.

- Skizzle Smart Contracts
  This is how Skizzle derives its trustlessness. Skizzle smart contracts have 2 primary functions:
  - Deriving usage logs for transparency in eventual user billing.
  - Enable access control to ensure transparency on who is the owner of a file and who has access to it.

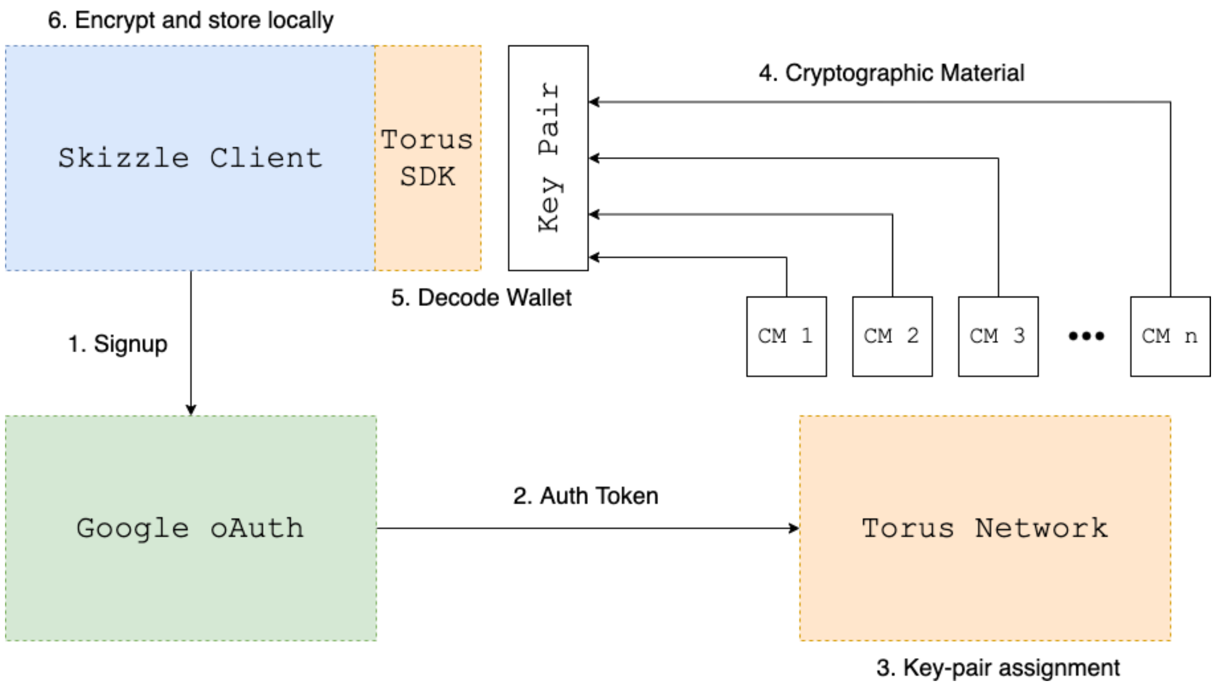
  Because our smart contracts are deployed on the public Matic blockchain, users can publicly and immutably verify their usage/billing and file access related information.

User keys

Parent Keys

Every user is assigned an Ethereum wallet, which is a BIP32 Hierarchically Deterministic(HD) wallet, with a public-private key pair. Even users who have not signed up are assigned a wallet that they claim once they do. This allows a sender to send an email to a non-skizzle user and share encrypted files.

Skizzle uses the directAuth feature from Torus Labs to authenticate and assign a wallet to users. The public-private key pair in this wallet becomes the user's Parent keys.



1. User signs up to Skizzle using Google oAuth.
2. User is authenticated and the auth token is forwarded to Torus Network.
3. Torus assigns a unique public-private key pair, generated from their buffer stream, to the user. This will be used to generate the user's wallet containing the Parent keys for the Gmail account authenticated for Skizzle.
4. The encoded cryptographic material associated with the wallet is fetched from the various nodes on the Torus Network and sent to the Skizzle extension.

5. The Torus SDK in the Skizzle extension, decodes the cryptographic material to give the key pair details.
6. These details are encrypted with a password provided by the user and stored on the browser's local storage.

If a user is yet to sign up to Skizzle, usually the recipient of an email with Skizzle encrypted attachments for the first time:

1. Skizzle's server sends the email id of the yet to be onboarded recipient to Torus which assigns a unique wallet, generated from their buffer stream, to this user.
2. The wallet is reserved for this user who, on receiving the email with Skizzle encrypted attachments, is taken through the flow of installing the Skizzle extension and signing up.
3. User is authenticated and the auth token is forwarded to Torus Network.
4. Torus recognises that this user has been pre-assigned a wallet and sends over the encoded cryptographic material to this user's client.
5. The Torus SDK in the Skizzle extension, decodes the cryptographic material to give the wallet details.
6. These details are PBKDF2 encrypted with a password provided by the user and stored on the browser's local storage.

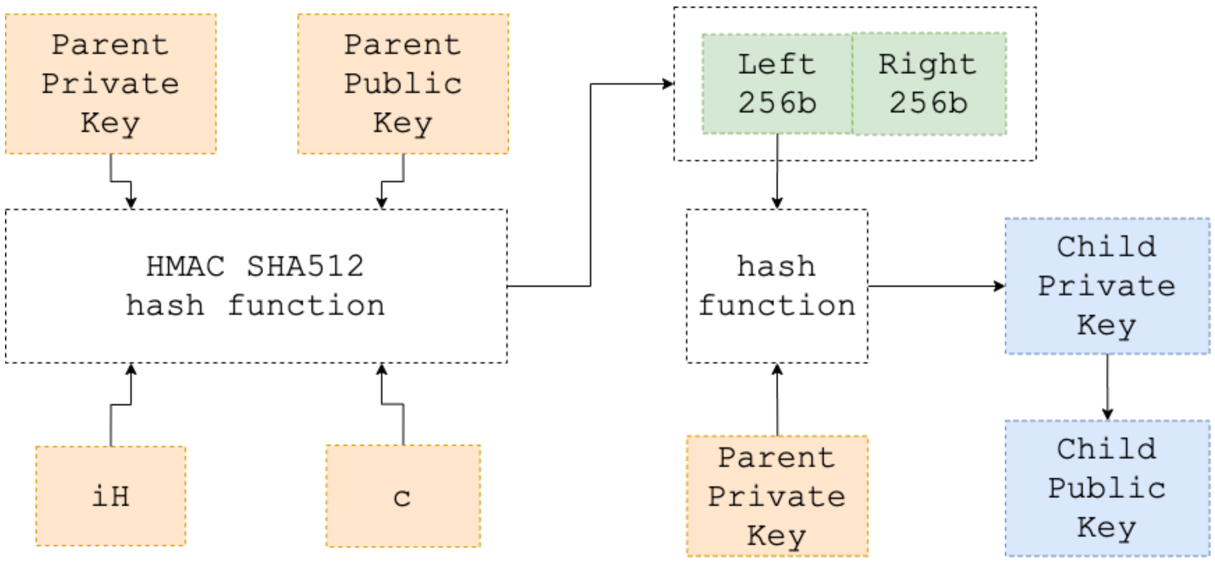
Let us assume 2 users on Skizzle, a Sender "A" and a Recipient "B" with the following wallet details:

Sender Alice - A	**Recipient Bob - B**
Parent Public Key - $\mathbf{P}_{\mathbf{Pub}}^{\mathbf{A}}$	Parent Public Key - $\mathbf{P}_{\mathbf{Pub}}^{\mathbf{B}}$
Parent Private Key - $\mathbf{P}_{\mathbf{Pvt}}^{\mathbf{A}}$	Parent Private Key - $\mathbf{P}_{\mathbf{Pvt}}^{\mathbf{B}}$

Child Keys

Every Skizzle user pre-generates child key pairs on sign up. If the number of pre-generated child keys is lower than an acceptable threshold, more keys are generated, at user login, to meet the threshold.

We follow the standard hardened derivation method for child keys:



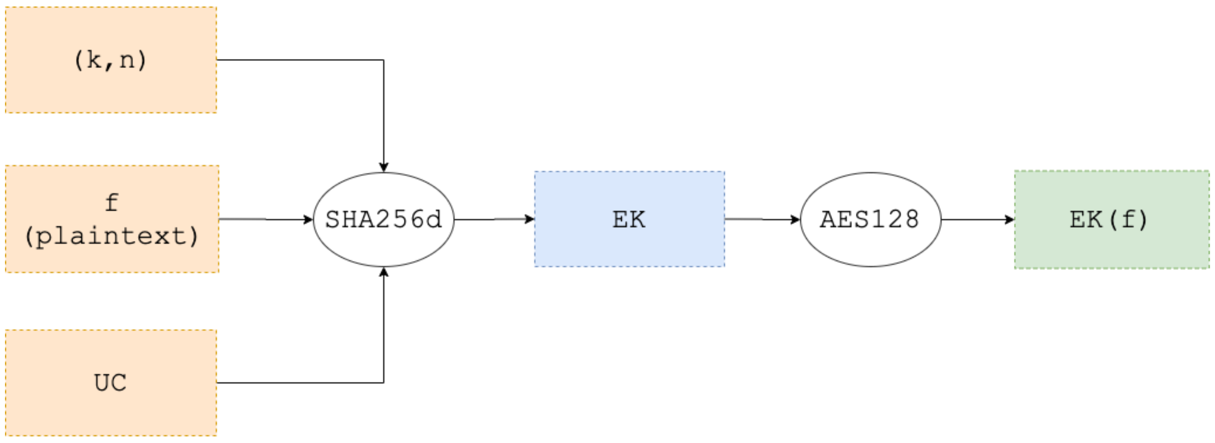
P_{Pvt}	Parent Private Key
P_{Pub}	Parent Public Key
C_{Pvt}	Child Private Key
C_{Pub}	Child Public Key
iH	Hardened key derivation index iH = $[2^{31}, 2^{32}-1]$
c	Parent chain code(Extra 256b of entropy)

1. P_{Pub} and P_{Pvt} are fed into an HMAC SHA512 function, along with **iH** and **c**, to produce a 512b hash.
2. The left 256b of the hash and P_{Pvt} are hashed together to produce the child private key C_{Pvt}.
3. C_{Pvt} is then used to derive the Child Public Key C_{Pub}.
4. This process is repeated multiple times(= some threshold value n) to derive multiple child key pairs, by using different **iH** values:
   [iH1, iH2, ..., iHn]
   [{C_{Pub}1, C_{Pvt}1}, {C_{Pub}2, C_{Pvt}2},...,{C_{Pub}n, C_{Pvt}n}]
5. The child private keys are then deleted and the child public keys and corresponding indexes are stored(the child private keys can be derived from the index and the parent private key, locally and on-demand).

File Encryption

All files are encrypted locally, on the user's browser, before being uploaded. Skizzle performs AES-128 encryption in CTR mode.

f	File
(k,n)	Encoding parameters
UC	Convergence Secret(SHA256d of a unique tag). One per user account.
EK	Encryption Key
E(f)	Encrypted File

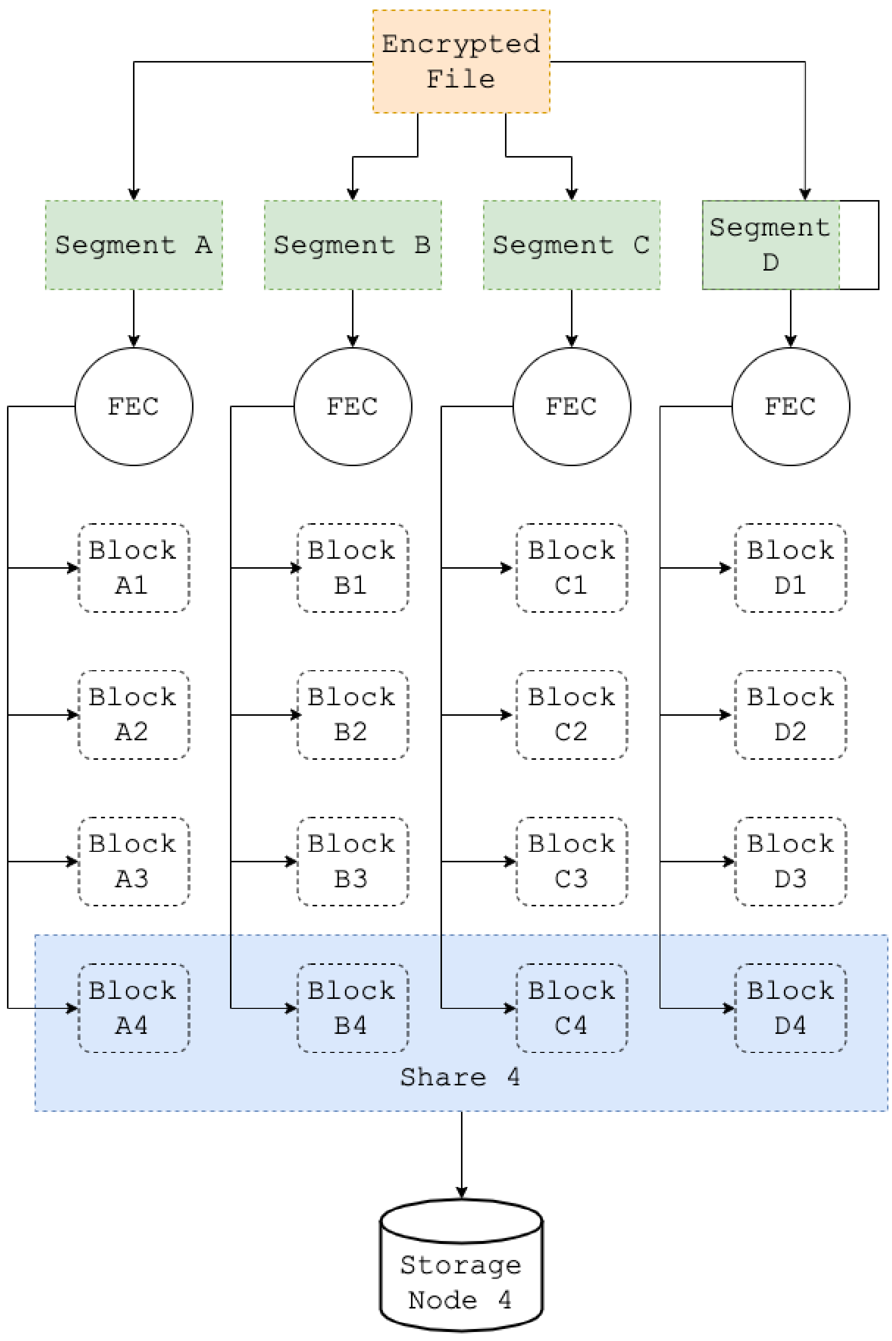


1. User selects a file from their device whose plaintext contents **f** are hashed along with **(k,n)** and **UC** to generate a hash.
2. A portion of this hash is used as the encryption key for the file. This is unique for a combination of file **f** and user i.e. a user U, uploading a file **f**, will always generate an encryption key **EK**.
3. The **EK** and **f** are used to encrypt the file with AES-128 in CTR mode to generate **E(f)**.

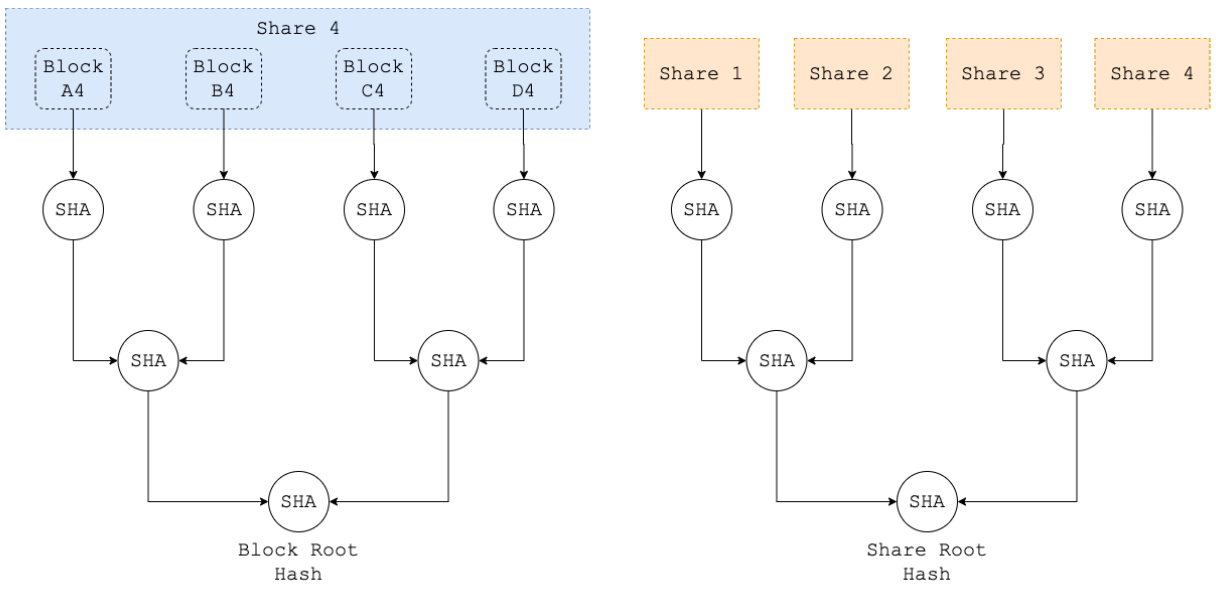
File Upload

Storage on Skizzle is distributed on a network of storage nodes. All files are erasure coded with forward error correction before being stored. It is a form of file encoding which breaks it up into a configured number of pieces such that only a subset of the pieces are required to retrieve the file.
Every file is assigned a DID which is a globally resolvable, unique identifier to the file. The DID resolves to a DID Document which contains information about the owner of the file, who has access to it, ways to interact with the file and some public file meta. Every combination of user and file produces a unique DID i.e. if a user uploads the same file, it will always have the same DID. Both the DID and the DID Document are as per the W3C DID specification.

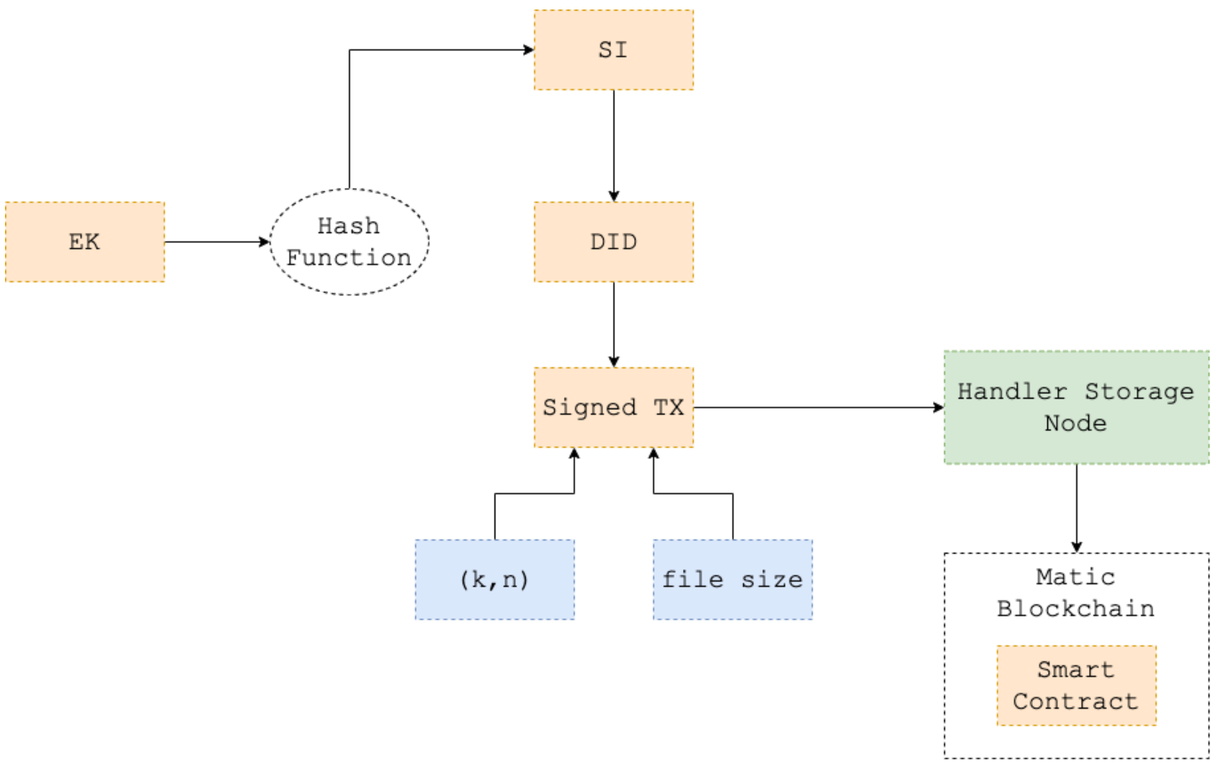
f	File
(k,n)	Encoding parameters
UC	Convergence Secret(SHA256d of a unique tag). One per user account.
EK	Encryption Key
E(f)	Encrypted File



1. Encrypted file E(f) is broken up into a number of segments where each segment is of a certain size.
2. Each segment is erasure coded to get a series of blocks. The number of blocks a segment is broken into is equal to the encoding parameter n.
3. One block from each segment is bundled into what is called a share S.
4. Each share is pushed to a unique storage node on Skizzle's distributed network of storage nodes.
5. In effect, each storage node stores a unique “piece” of an encrypted file uploaded.



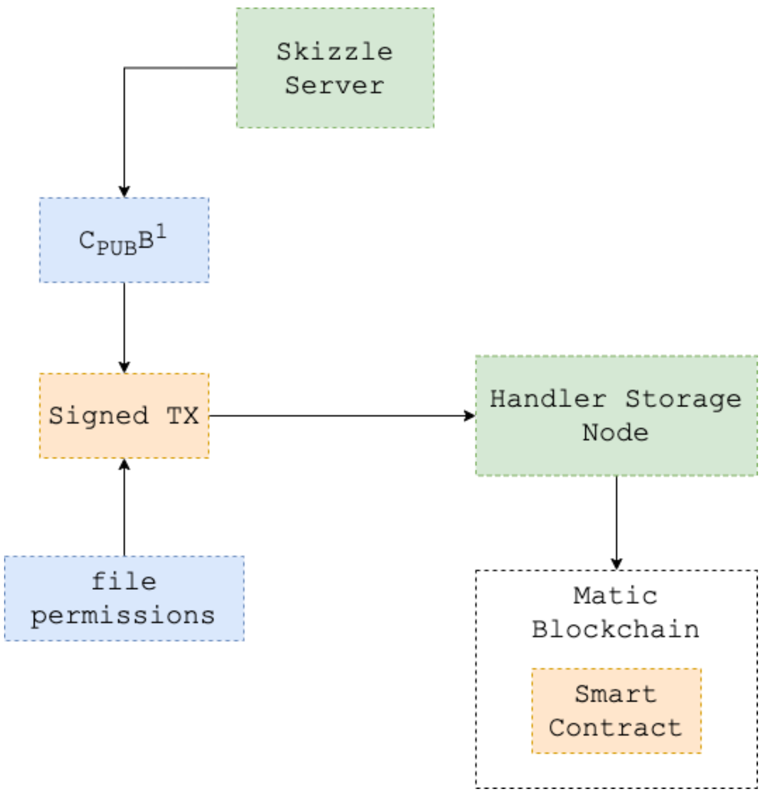
1. In addition to the shares being stored on the storage node, each block within a share is hashed to produce a merkle hash tree, which is also stored on the node.
2. The root of this tree is called the block root hash.
3. Furthermore, the block root hashes of each share are hashed to form a higher order merkle hash tree whose root is called the share root hash.
4. This share root hash, along with the hash of E(f) and few other public file parameters, are put together to form the URI Extension Block UEB.
5. The UEB is then hashed and stored on the blockchain. This is used to efficiently retrieve shares from the storage nodes on download.

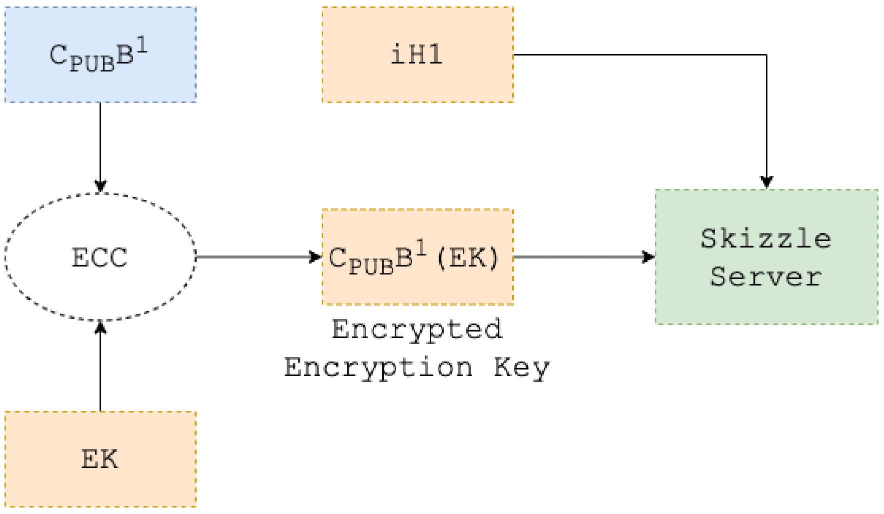


1. EK is hashed(SHA256d) to produce SI which forms the basis of the file's DID. did:newfang:<SI>
2. One of Alice's available child private keys $C_{pvt}1_A$ is used to sign a transaction to create the DID Document DDoc on the blockchain.
3. This signed transaction is forwarded to the blockchain by one of the storage nodes which also fronts the transaction fee(in MATIC token) for executing this transaction on the public blockchain. This meta transaction execution is so that Alice does not have to deal directly with a cryptocurrency.
4. The Skizzle smart contract verifies Alice's signature and creates the DDoc(if Alice is uploading the file for the first time).
5. The DDoc, currently, states that Alice is the owner of the file and also stores some public information about the file(encoding parameters and file size).

File Share

After Alice uploads a file and chooses to share it with Bob(recipient), the DDoc is updated to include one of Bob's child public keys in the access control list. The file encryption key is encrypted with Bob's child key and stored on the server for retrieval on download.



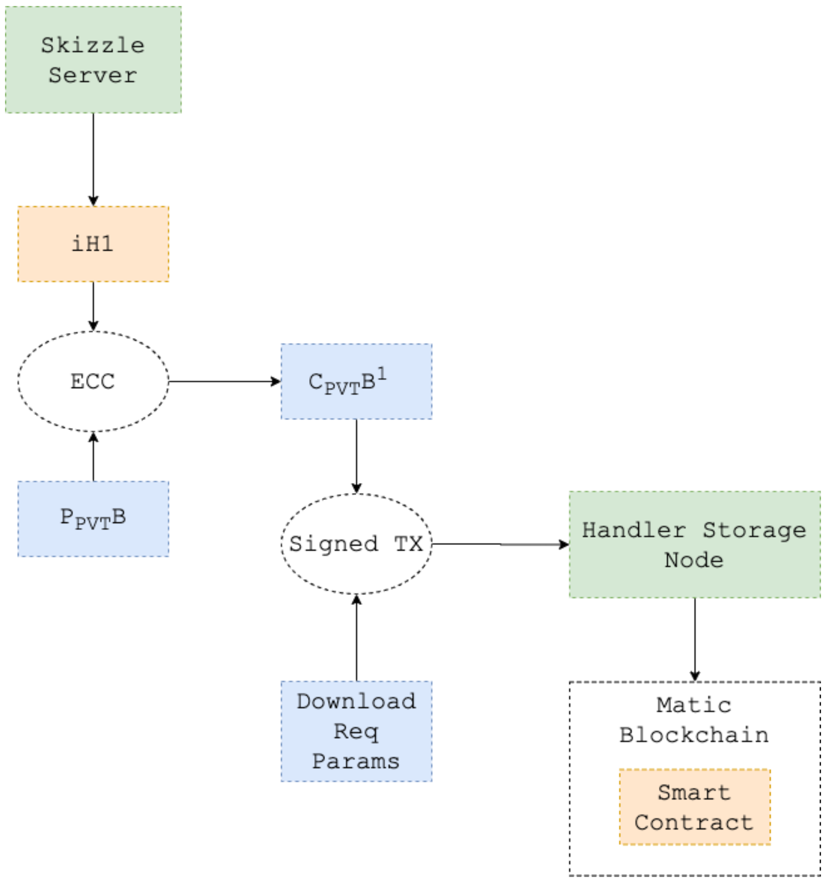


1. Alice fetches one of Bob’s available child public keys($C_{Pub}B_1$) from his list of child public keys on Skizzle’s server and signs a transaction to update the DDoc on the blockchain with $C_{Pub}B_1$.
2. EK is encrypted with $C_{Pub}B_1$ and stored on the Skizzle server.
3. The hardened index iH_1 used to derive $C_{Pub}B_1$ is also stored on the server.
4. If Bob is not yet a user or has run out of child public keys stored on the server(Bob has not accessed his account for a

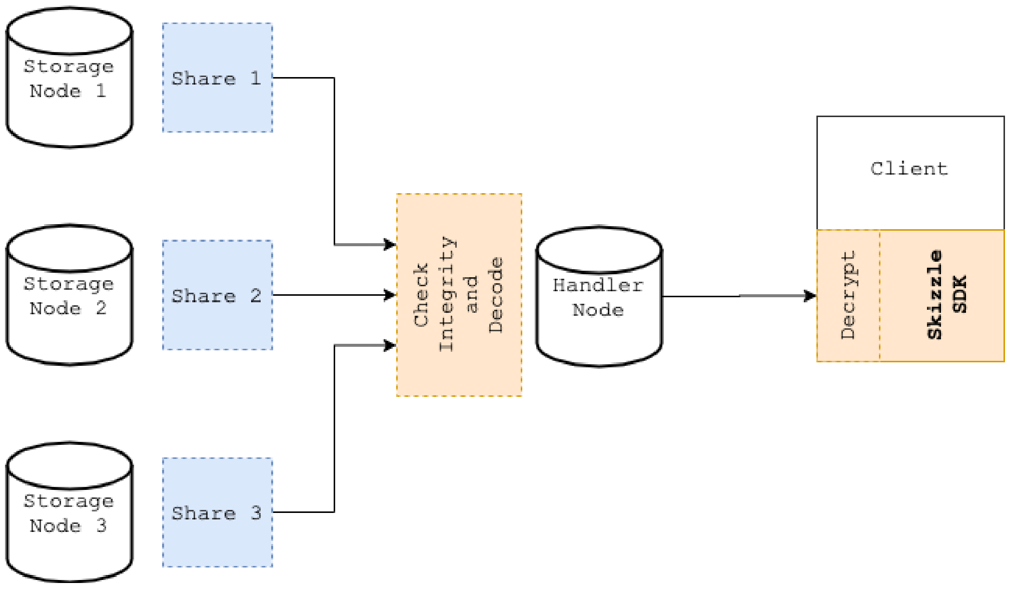
while), his Parent public key $P_{Pub}B$ is fetched and used to encrypt EK.

File Download

If Bob passes an access verification check on the blockchain, he can proceed to download the encrypted file E(f). A storage node is assigned to handle this request from Bob which decodes the file from its various shares and sends over E(f) to be further processed.



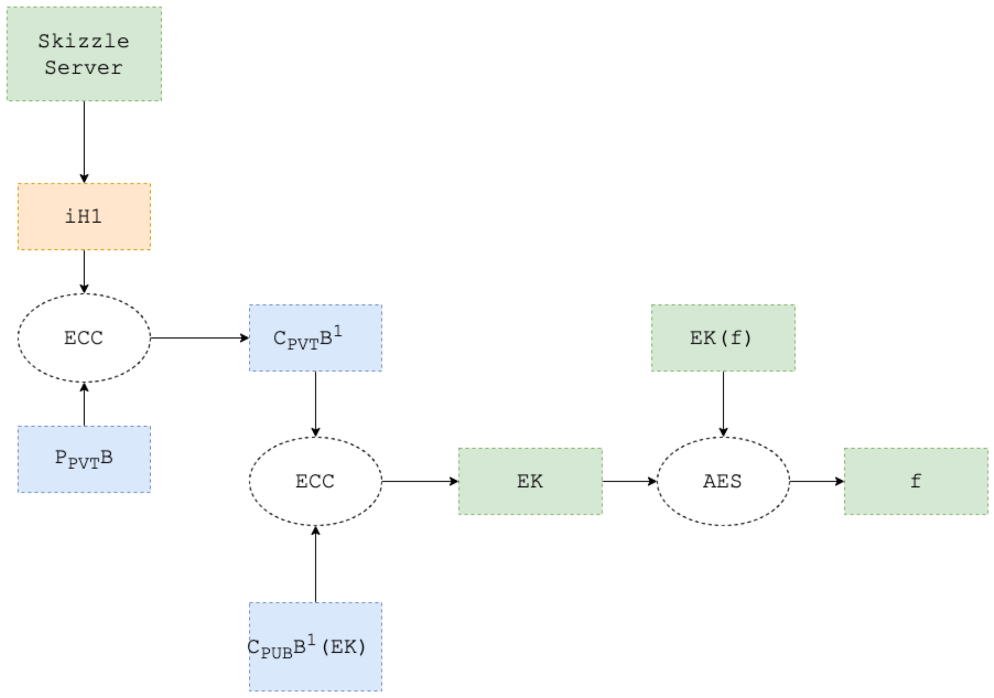
1. Bob fetches iH_1 from the server and uses that along with his Parent private key $P_{Pvt}B$ to derive the child key pair $\{C_{Pub}B_1, C_{Pvt}B_1\}$
2. Bob signs a transaction with $C_{Pvt}B_1$ requesting to download the file.
3. Bob’s signature on this transaction is verified on the smart contract and his address is checked against the access control list in the DDoc.
4. Once verified, the file download initiates.



1. The storage node handling the download request uses the SI in the file’s DID to locate all the shares of E(f).
2. A peer node first transmits the portion of the share hash tree required to validate a block root hash S1.
3. It then transmits S1 itself and the loop repeats till all block root hashes are fetched and validated.
4. The validated blocks are put together to generate the share from a specific peer storage node.
5. Finally, all the shares are put together to give E(f) which is then transmitted to the client.

File Decryption

Bob has E(f) and fetches the encrypted encryption key, decrypts it and then uses EK to decrypt E(f).
Bob’s child keys used to perform this share are deleted from the server so that they may not be used in any other file share step by any other user. This is done to ensure privacy of the public transaction(file share) between Alice and Bob.



1. Bob then fetches the encrypted encryption key $C_{Pub}B_1$(EK) and decrypts it with the associated child private key $C_{Pvt}B_1$.
2. EK is now used to decrypt E(f) to give f.
3. The child key pair $\{C_{Pub}B_1, C_{Pvt}B_1\}$ is then deleted and a new child key pair with a new index is produced to meet the threshold(if necessary).

Access Control

Access control on Skizzle is managed in 2 stages:

Access to the encrypted file

This is checked via an access control list on the blockchain where a user needs to authenticate themselves using a set of child keys and their authenticated identity needs to be on the access control list of the file, maintained in the DID document associated with it.

Potential Issues:

- All actions are signed by users and logs of these actions are maintained on a public blockchain.
- The mapping of a user’s email and their Parent key pair is maintained by Torus and exposed via APIs.
  The combination of these two mean an attacker can derive usage graphs, compromising user’s privacy.

Remedy:
However, Skizzle's use of HD wallets and deriving child key pairs to sign transactions means that every user action potentially utilizes a unique key pair. This effectively nullifies any attempt to draw broad usage graphs, maintaining user privacy.

Access to the encryption key

With the encryption key encrypted with the intended recipient's public key, the decryption of the file with the encryption is only possible via the associated private key. Any user without the private key will not be able to decrypt the encrypted encryption key and as a result not have access to the file.

Potential Issues:

- If any party's Parent keys are compromised, the attacker can generate all previously generated child keys, decrypt all previously encrypted encryption keys and gain access to previously received files.
- If any party's Parent keys are compromised, the attacker can generate all future child keys and gain access to all future files to be received.

Potential Remedy:
Implementing the Double Ratchet algorithm created by the wonderful people at Signal. It fits perfectly with our use case of sharing encrypted messages(encrypted encryption key to a file) asynchronously with the receiving party being potentially offline at the time of sharing.

"The parties derive new keys for every Double Ratchet message so that earlier keys cannot be calculated from later ones. The parties also send Diffie-Hellman public values attached to their messages. The results of Diffie-Hellman calculations are mixed into the derived keys so that later keys cannot be calculated from earlier ones. These properties give some protection to earlier or later encrypted messages in case of a compromise of a party's keys."

This implementation will allow us to develop strong backward secrecy whereby if an attacker manages to compromise an encryption key to a file, all future keys appear random and cannot be derived.
It also affords for strong forward secrecy whereby any compromise of an encryption key to a file does not reveal any of the previously generated keys.

Future Work

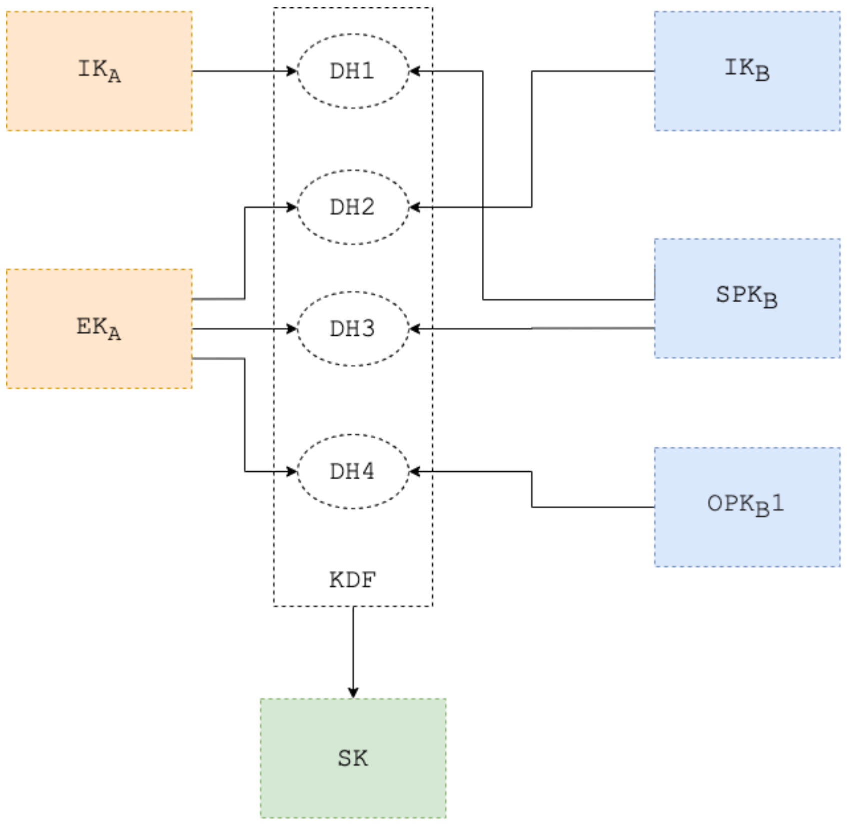
Here we present some key ideas due for implementation in our V2.0 release of Skizzle:

The Double Ratchet

Motivation
To achieve backward and forward secrecy when sharing file encryption keys so a key compromised by an attacker does not allow them to derive previous keys or new keys.

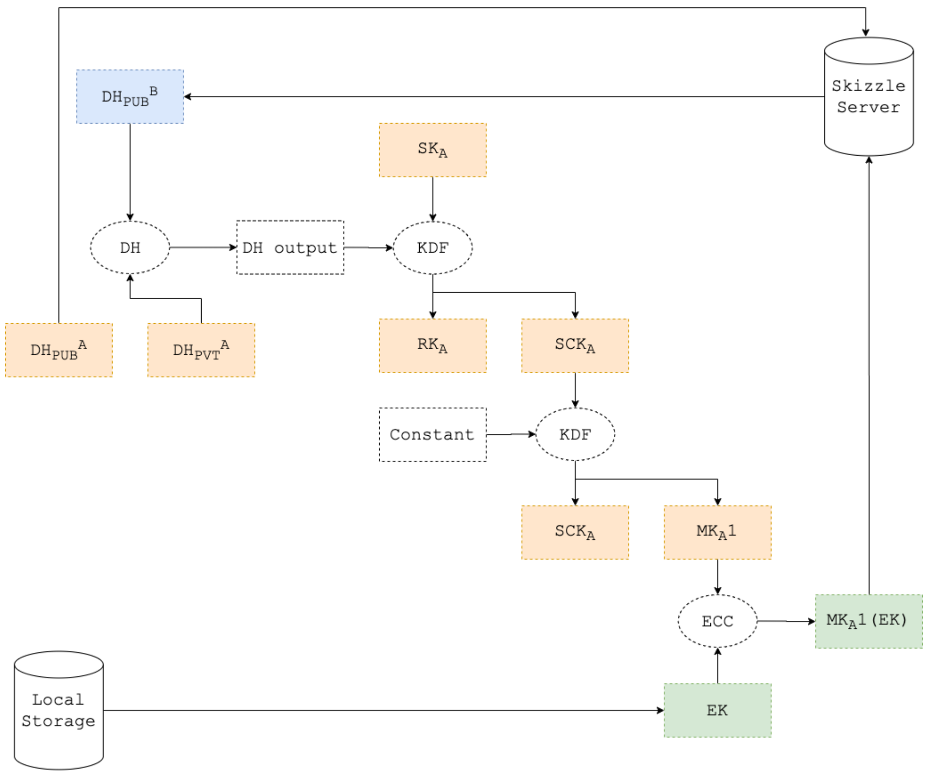
Implementation
The Double Ratchet by Signal.org is a combination of 2 key concepts namely the Symmetric-Key Ratchet and the Diffie-Hellman Ratchet.

1. **Initial Setup**
   Let's say Alice and Bob want to exchange Skizzle encrypted attachments over email. The first step is for them to agree upon a shared secret key(SK) which is achieved by the Extended Triple Diffie-Hellman key exchange protocol(X3DH).



a. Let us assume that Alice is the sender of the email. She first fetches some public information about Bob which is generated on Bob's side when he signs up for the first time and stored on Skizzle's servers.
b. She fetches:
   i. Bob's identity key IK_B
   ii. Bob's signed prekey SPK_B
   iii. Bob's signature used to sign the prekey $Sig(IK_B, Encode(SPK_B))$
   iv. A one-time prekey from a set of prekeys OPK_B1
c. On her side, Alice has her own identity key IK_A and generates an ephemeral key EK_A.
d. She then performs 4 Diffie-Hellman steps:
   i. $DH1 = DH(IK_A, SPK_B)$
   ii. $DH2 = DH(EK_A, IK_B)$
   iii. $DH3 = DH(EK_A, SPK_B)$
   iv. $DH4 = DH(EK_A, OPK_B)$
e. She then passes these 4 values through a Key Derivation Function KDF to arrive at the Secret Key SK.
   $SK = KDF(DH1 \mid\mid DH2 \mid\mid DH3 \mid\mid DH4)$

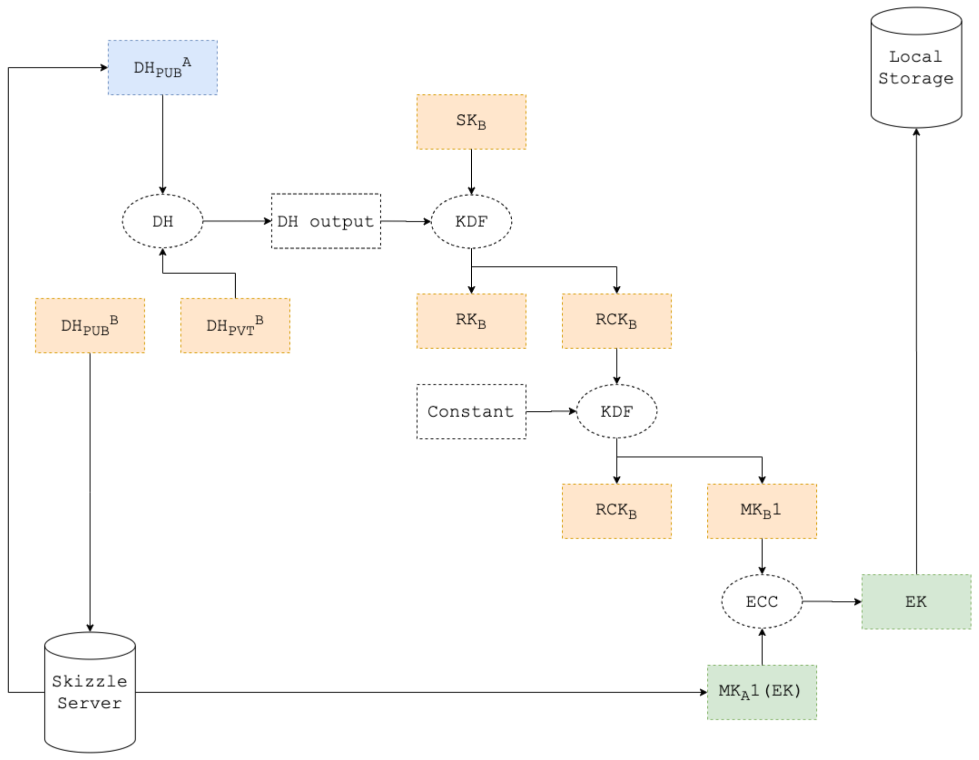
2. Encrypting the file encryption key(Sending an encrypted file)
Alice now performs a Double Ratchet step to encrypt the file encryption key, before sharing it with Bob.



a. Alice generates a set of DH keys (DH_{PUB}^{A}, DH_{PVT}^{A})
b. Alice fetches Bob's DH public key DH_{PUB}^{B} and performs a Diffie-Hellman DH step(X25519 DH function) to produce a Diffie-Hellman output.
c. This DH output is the input into a KDF which also takes the SK computed in the earlier X3DH exchange. This is the first step in a chain of steps that forms Alice's Root Chain.
d. The output of the KDF is 2 keys. One being a Root Key RK and the other a Sending Chain Key SCK.
e. The SCK generated is the input, along with a constant(hash of user user email id), into another KDF which also produces 2 keys, namely, the next SCK in Alice's Sending Chain and a Message Key MK_A1.
f. After Alice encrypts the file f with EK(on file upload) and sends the email, MK_A1 is used to encrypt the file encryption key EK and is stored on Skizzle's servers. This completes a Symmetric-Key Ratchet step.
g. Post sending, Alice generates another set of DH keys and repeats b, c and d above, to generate 2 more keys. One being the next RK in her Root Chain and the other a Receiving Chain Key RCK. This is a full DH Ratchet step and she is now ready to receive an email with a Skizzle encrypted attachment from Bob.
h. For every successive send/receive action that Alice performs, the DH Ratchet step is skipped and only new Symmetric-Key Ratchet steps are performed, advancing the Sending/Receiving chains.
i. As chains advance, The older chain keys and even the message key used to encrypt EK are deleted.

3. Decrypting the file encryption key(Receiving an encrypted file)

Decryption is the same set of steps that Alice performs to encrypt EK but flipped.



a. Bob generates a set of DH keys ($DH_{PUB}{}^{B}$, $DH_{PVT}{}^{B}$)
b. Bob fetches Alice’s DH public key $DH_{PUB}{}^{A}$ and performs a Diffie-Hellman DH step(X25519 DH function) to produce a Diffie-Hellman output.
c. This DH output is the input into a KDF which also takes the SK computed in the earlier X3DH exchange. This is the first step in a chain of steps that forms Bob’s Root Chain.
d. The output of the KDF is 2 keys. One being a Root Key RK and the other a Receiving Chain Key RCK. Note that Bob’s RCK is analogous to Alice’s SCK. $RCK_B = SCK_A$.
e. The SCK generated is the input, along with a constant(hash of user user email id), into another KDF which also produces 2 keys, namely, the next RCK in Bob’s Receiving Chain and a Message Key MK_B1 where $MK_B1 = MK_A1$.
f. Bob fetches MK_A1(EK) from Skizzle’s servers and proceeds to decrypt it to produce the file encryption key EK.
g. Bob PBKDF2 encrypts the EK with his account password and stores it locally for future access to the file.
h. Post receiving, Bob generates another set of DH keys and repeats b, c and d above, to generate 2 more keys. One being the next RK in his Root Chain and the other a Sending Chain Key SCK. This is a full DH Ratchet step and he is now ready to send an email with a Skizzle encrypted attachment to Alice.
i. For every successive send/receive action that Bob performs, the DH Ratchet step is skipped and only new

Symmetric-Key Ratchet steps are performed, advancing the Sending/Receiving chains.

j. As chains advance, The older chain keys and even the message key used to encrypt EK are deleted.

Backup
Since encryption keys are only stored locally on user’s devices and with the double ratchet implementation requiring encrypted encryption keys to be deleted from servers after first use, account recovery is not possible.
However, users changing machines or losing control over them(breakdown, loss…) is an eventuality and we need to provide ways for users to recover lost accounts or simply restart from where they left off on new devices.
We intend to solve this by doing regular, automated backup of user’s encrypted encryption keys to files already uploaded to Skizzle, onto locations users control. These could be Google Drive, Dropbox or MS OneDrive folders they control.
For teams and enterprises, we will have the option to backup keys onto self-hosted or cloud services they setup/control.

Asynchronous Downloads

Motivation
To achieve faster downloads.

Implementation
Skizzle encrypted files are erasure coded with each encoded piece stored on a distributed network of storage nodes. On download, these pieces are decoded back to give the encrypted file on one of the storage nodes handling the download request, before being downloaded to the client.
By moving the decode step to the client itself, each encoded piece can be asynchronously fetched from the storage nodes and decoded back, providing significant gains in download times.
Also, performing the cryptographic verification of the encrypted file’s integrity on the client would improve transparency for the user.

Skizzle Blockchain

Motivation
To improve efficiency, performance, security and reduce costs of the entire platform.

Implementation
Taking the example of an Upload action, the action itself is performed on the network of storage nodes with a log of the action written to Skizzle smart contracts on the public Matic blockchain. These are two separate activities that require reconciliation if either step fails, inducing potential inefficiencies.
This current implementation is because the general purpose Matic blockchain, like most public blockchain networks, does not natively support file storage.
Making the execution of the action and the writing/checking of entries in Skizzle's smart contracts, a single atomic operation would remove these inefficiencies. Furthermore, this would also speed up operations and reduce the potential for attack vectors born out of the need for two disparate systems to be in sync.
This can be achieved by running Skizzle's own blockchain, specifically designed to execute Skizzle's use case of atomic action execution and logging/checking on a public blockchain.
This also paves the way for decentralized storage on Skizzle where validators also provide resources to the network in the form of Storage and Bandwidth in exchange for an incentive.

References

1. https://money.cnn.com/2017/10/03/technology/business/yahoo-breach-3-billion-accounts/index.html

2. https://techcrunch.com/2019/04/13/microsoft-support-agent-email-hack/

3. https://www.troyhunt.com/the-773-million-record-collection-1-data-reach/

4. https://www.troyhunt.com/the-773-million-record-collection-1-data-reach/

5. https://security.utexas.edu/iso-policies/cloud-services/risks

6. https://www.venafi.com/sites/default/files/2020-02/AIR_Venafi_White_Paper - Final-V2.pdf

7. https://www.venafi.com/blog/pains-encryption-key-management-why-manual-processes-are-so-hard

8. https://en.wikipedia.org/wiki/Key_management#Challenges

9. https://blog.lastpass.com/2019/05/passwords-still-problem-according-2019-verizon-data-breach-investigations-report/
10. https://i.dell.com/sites/csdocuments/Learn_Docs/en/dell-end-user-security-survey-2017.pdf

11. GSuite premium pricing
12. Microsoft365 premium pricing