# VERIT – Idea Document

# Abstract

SSI Verit aims to create a system to let social media users create a source of truth against their social media posts. It is to serve as a source of message integrity and message authentication. To start with the application would focus on one platform – the twitter model. And the entire system builds on storing immutable information about tweets on a blockchain platform.

# Registration – Setup

### Flow – Chrome Extension

- User logs in to the Verit ecosystem, with a private key
- The chrome extension starts monitoring activity on the twitter page
- The chrome extension (optionally) gets hold of the auth token to communicate with the twitter server on behalf of the user.

### Adding identity to Twitter

- A portal to add keys
- Upon submitting, twitter issues a transaction to the IdentityTable, signing the user's key
- This ensures the correct link between twitter handle and public key

# Tweet – Flow

USER - end

- User types in the tweet
- Clicks on “Post” button
- The chrome extension catches the action and the tweet text, stores it
- The chrome extension watches for submit action on the tweet and fetches the tweet ID of the last tweet.
- Chrome extension matches the text in the tweet with the previously stored tweet and creates the transaction.
- The chrome extension gives out a popup asking the user to put the signature on the last tweet or cancel it.
- The chrome extension unlocks the secret key, signs the transaction and the message hash and sends the transaction to the network.
- The transaction is added to the local storage of past transactions.

**Service Provider (Twitter) – End - Prescriptive**

- Twitter’s backend continuously scans for events on the blockchain network.
- On adding a signature for a particular tweet, it updates its database with the URL to verify the tweet.
- Twitter does a check from its end against the tweet text as well as the user identity and matches it with the identity table on the blockchain network as well as its own database.

## Components

The application will have the following components stitched together:

1. VERIT Chrome Extension – The user facing application end that has the functionalities of creating and signing blockchain transaction, creating the message integrity hashes and issuing transaction to teh the network. Additionally, it will implement features to store and manage Verit accounts for the user.
2. Smart Contracts on blockchain – Set of smart contracts deployed on a scalable blockchain solution. These smart contracts would serve as the storage for the message hashes and signatures and would also offer functionalities to create and verify them. Additionally, the smart contracts would also have an integration layer for identities. To start with this can be another set of smart contracts deployed on the same blockchain platform.
3. VERIT website frontend – Website to verify tweets and transactions on the Verit network. Additionally, it would contain resources to help users to understand and sign up for the platform.
4. VERIT website backend – Set of APIs to cater to the VERIT frontend and chrome extension.

## VERIT Chrome Extension

The VERIT chrome extension is aimed at providing users easy access to the message signing functionalities and to interact with the overall Verit ecosystem. Chrome is the most popular browser out there and it makes sense to go with chrome for the first release of the application. Further, a browser extension is more user-centric than a website, with its own access to local storage – an ideal technology for the purpose of Verit. The list of functionalities follow.

### Create and manage identity

Identity is considered a user-centric property in the Verit ecosystem, so a model similar to the Self-Sovereign identity will be adopted for Verit. The core of this identity lies in a public key pair generated by the user, with the secret key – password protected – is stored in the local wallet. The extension would then use this key to create signatures on the messages as well as the blockchain transactions.

Before this identity could be used for a real transaction, for message authentication, this must be registered into the Verit ecosystem. So, the chrome extension should be able to issue a transaction to the network, registering a newly created key. The public part of this key would then be signed by Twitter creating a link between the user's handle and the public key. Ideally, twitter would expose a functionality to "add" a public key to a user's profile.

### Catch a tweet and let user sign it

The extension should have the ability to detect when the user is about to submit a tweet and throw a popup asking if the user wants to put a signature on the tweet and issue it to the blockchain network. This can be achieved in multiple ways such as monitoring the action of the click of the "tweet" button – a reverse engineered solution that could potentially work on other portals too. Another way would be to get hold of an auth token from twitter log in and using it to fetch the latest tweet and ask the user if she wants to sign it.

### Store user profile details including keys

Storing parts of the data inside the local storage of the chrome extension would allow easy access to manage and monitor the user profile. Firstly, the key pair created by the extension should be securely stored in the local storage in the form of a wallet, also allowing users to create multiple keys for use in same or different portals/accounts. A section for "past transactions" should also be included providing a clear view of recent transactions. For a friendlier experience of the user, certain parts of the local storage would have to be linked with the user's account on the website. So, the wallet would have certain (loosely coupled) components integrated to the Verit backend and frontend.

It is also important that certain data in the extension be password protected by default, since the details of the transaction might be private. The private key must be stored under a second layer of encryption and should be used only while creating the signature and erased from the memory right afterwards.

# Smart Contracts on Blockchain

**VeritTraceRecords**

This would be the main smart contract that contains record of all signatures on tweet. All of this data would be public for now. The plaintext of the messages are not stored on the blockchain, whence the idea is that anyone who has the details of a tweet can easily verify if the tweet is coming from the claimed origin. Whereas anyone without the details won't be able to use this system as a source for tweet data. Most tweets are public knowledge, but there might be tweets that are to be hidden from public and so, this requirement comes as an aid to give some amount of privacy to the tweet data.

A certain message M is a property of the platform (or any other party). For the purpose of storing and verifying a reference to this message, a SHA-256 hash is created for it, namely $m = hash(M)$. Hereinafter, $m$ is treated as the "message" that needs to be verified.

The message $m$ is then sent to the smart contract for validation on the further hash and signature. Since the plaintext of the message can carry an arbitrary text with a certain associated meaning, it is important to give the message a certain application specific identifier before signing it. This would limit its usage to our platform. So the input to the signature algorithm is the hash created as follows:

$$\text{inputMessage} = hash(\text{'VeritTraceRecords\n'} + \text{platformIdentifier} + m + hash(\text{data index/tweet ID}) + \text{timeStamp})$$

The user will create a plain ECDSA signature on the inputMessage(with some associated randomness already present in most libraries).

The contract would contain methods to check the structure of the hashes and signatures. But no methods would take plaintexts as data.

DATA

The data in this smart contract is structured in the form of **Records**. The structure of a single record is given as follows.

**Records** is a map from string -> Record: Contains all the records of all tweets, along with cryptographic material for later verification. Hash of data index (tweet ID) used as the index.

Struct *Record* has the following components:

*Signature* – Signature on the inputMessage, in hex

*InputMessage – String in hex*

*TimeStamp* – Timestamp obtained from the client's machine.

*PlatformIdentifier* – Platform Identifier code

*PlatformSignature* - [Optional] Signature provided by the platform (Twitter) on the plaintext hash

*PlatformMetadata* – Stringified JSON of platform related data

METHODS

- AddRecord (Record) – Records a new transaction containing message hash and signature of the (verified) user. The method recreates the inputMessage and verifies the signature against the public key stored in the IdentityTable.
- VerifyRecord (*index*, *hashOfOrigMessage*, *platformIdentifier*) – Can be used to verify a certain record, given the components.

It might be noted here that using the *index*, the record stored on the blockchain can be fetched and verified offline, without requiring the verit infrastructure.

## VeritIdentityTable

Prior to creating any transaction for message integrity and message authentication, it is important to link an identity to the twitter handle. This also makes it possible to avoid malicious users from claiming a given identity with the wrong key. Here, we propose to create a table that implements an identity hierarchy, inspired by, but a very simple implementation of the PKI infrastructure. The idea here is that a user of the platform should be able to generate a key pair on her own end, get that signed by the provider and use the key independent of the provider in later transactions.

We aim to provide interoperability with other identity services that share the same PKI. For this reason, the individual accounts are created as a reflection to the public key that it serves. A user with a public key from an already registered identity ecosystem can re-use the key in Verit. If the user has credentials or signed claims outside of the Verit ecosystem that the user wants to import, the smart contract allows that.

DATA

**Users** contain the list of users who sign up for the platform. It's a map, indexed by the ethereum address of the public key that the user intends to use for the platform.

Struct *User* has the following properties:

type - StringType of identity – user, platform

address - Ethereum address corresponding to Public key

attestations - [Attestation] Ethereum address corresponding to Public key

Struct *Attestation:*

*platformIdentifier* – String, Platform name string in short, e.g. Twitter

*platformUrlString* – String, e.g. twitter.com

*signature* – hex String, Sign on hash( "Verit Platform Attestation\n" + PlatformHandle + "\n" + address )

*platformHandle* – Plain string of the platform handle

## METHODS

**registerAddress** (type, address, attestations[ ], signature) - method to register a new address to the Verit ecosystem. The signature here is a signature on *hash(*'Verit Platform Registration\n' + address*)*.

**addAttestation** (address, attestation) - method to add a new atttestation to an existing address.

## VERIT website frontend

Functionalities:

- Ability to verify a given tweet
- Information about Verit ecosystem – How it works section
- Ability to view profile, including past activity.
- Signing up for Verit – step by step flow including chrome extension setup
- Top verified tweets section – this could also serve to attract new users.

## VERIT website backend

Functionalities:

- API to verify a given tweet (unauthenticated)
- API to verify a private tweet (authenticated) – this does not store the information passed on to it
- User profile APIs
- Login and Signup APIs
- Top/Featured tweets section APIs