**Developing a Data Provenance System with Version Control Using Blockchain**

by

Ujan Mukhopadhyay

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 7, 2021

Keywords: Blockchain, Data Provenance, Version Control



Approved by

Cheryl Seals, Chair, Associate Professor, Computer Science and Software Engineering
Ashish Gupta, Co-Chair, Associate Professor, Systems and Technology, Business School
Yang Zhou, Assistant Professor, Computer Science and Software Engineering
Mark Yampolskiy, Associate Professor, Computer Science and Software Engineering

Abstract

The origin and integrity of data is a concept that is of great importance. If the history of the data is not preserved, there is no way to ascertain data integrity and whether it was modified; if yes, the identity of the modifier and the frequency and nature of the modification; or if copies of the data have been made. Even in a closed system, the integrity of the data might be compromised, leaving room for the modifier to repudiate. There are not many systems that provide Data Provenance as a service. Of the few systems that do, none are large scale, robust or commercially available, and are limited in their performance as well. Git provides version control but is not equipped to provide data provenance, and even then, the use cases are limited to codes and documents. In this thesis, we propose a distributed system that combines Git and Blockchain to provide secure data provenance with version control. In the said system, multiple users create, access, and modify files that reside in a separate database. Every change is documented in the Blockchain, and every version of a file is stacked in the database.

Table of Contents

List of Figures

List of Tables

# Chapter 1

# Overview

## 1.1 Introduction

The origin and integrity of data is a concept that is of great importance. If the history of the data isn't preserved, there is no way to know whether it was modified; if yes, the identity of the modifier and the frequency and nature of the modification; or if copies of the data have been made [6]. Even in a closed system, the integrity of the data might be compromised, leaving room for the modifier to repudiate.

## 1.2 Attributes of Data Provenance

In cloud computing or the Web in general, where proof of past data possession [7] is difficult to obtain and current/past data location is hard to know, there is a need to efficiently maintain a reliable audit trail. Certifying chain of custody for files is both difficult and important. Custody information supports forensics and situations where multi-organization data sharing can result in a lack of trust in data quality. Providing secure data provenance information for cloud computing would enable greater trust in cloud computing by enabling better digital forensics. By allowing increased data sharing for law enforcement, without fear of corruption or manipulation, criminal prosecution of online crime will become more likely.

Provenance Metadata is a class of information that explains how data were derived. It is used to:

- Find sources of errors.
- Provide confidence to the system users in the resources and results.

- Provide justification for decisions.

However, while the existing provenance systems provide storage and visualization, it lacks in security. In other words, the existing systems keep track of the whereabouts of the data, but does nothing to ensure that they are not tampered with.

## 1.3 Discussion of Related Technologies

In this section, we introduce and discuss certain technologies, which we will be using to develop our project. We will look at how they work, their strengths and limitations, and the potentials of the system we are proposing.

### 1.3.1 Blockchain

Blockchain is a distributed public ledger used in cryptocurrencies [8] that stores transaction information. Each transaction, once verified, is accumulated in a block [9]. Each block consists of a group of verified transactions [10].

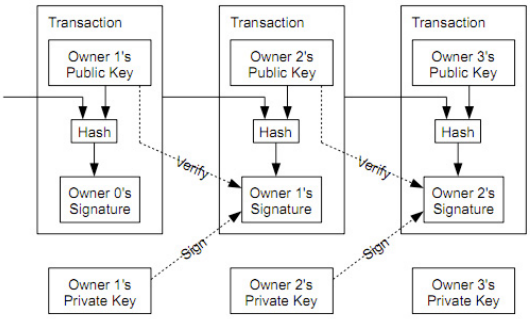



Figure 1.1: A Bitcoin Blockchain (adapted from [1])

In cryptocurrencies, a transaction is created when a *payer* sends currency to a *payee*. Mining validates transactions and adds them to this public ledger [11]. Miners are required to solve a resource-intensive task to perform mining. The resource intensive nature of mining serves two purposes [11].

When a new transaction takes place, the miner checks if the currency belongs to the payer, and if the payer is trying to double spend [12]. The ownership of the currency is recorded in the Blockchain. A malicious user may create multiple fake identities and try to validate

an invalid transaction. Resource intensiveness makes it expensive and hence infeasible for a malicious user to create enough false identities to outnumber benign users and validate an invalid transaction.

Every cryptocurrency offers the miners a reward for a successful mining. For example, for mining a block in Bitcoin, the miner receives 25 bitcoins (as of 5/6/2017). This value reduces by half after every 210,000 mined blocks. Resource-intensiveness limits the number of blocks mined, and hence the number of currencies created per day. This limits inflation.

Blockchain can be a powerful tool to implement secure data provenance. There are four aspects that make blockchain suitable for maintaining provenance.

- **Exhaustiveness**: Blockchain is the distributed ledger that logs each and every change taking place in the system. Thus no action can go unaccounted for.
- **Chronology**: Blockchain is strictly chronological. The order of actions is maintained and reflected via timestamps.
- **Authorship**: The identities of the entities are preserved via Digital Signatures, ensuring non-repudiation.
- **Integrity**: The entries in blockchain are cryptographically linked making it impossible to alter data, hence preserving provenance.

### 1.3.2 Git

Git is an established, active open source project developed in 2005 by Linus Torvalds [13] and is one of the most popular Version Control Systems or VCS [14]. A VCS is a system that chronologically records changes made to a file or group of files [15], such that the changes are not overwritten. So any older versions of the file are available when required [14].

The major difference between Git and any other VCS is the storage of the changes [15]. Other version control systems, for example CVS [16], Subversion [17], or Bazaar [18] implement a file based storage. That means, whenever a change is made, a copy of the unchanged

version is saved, and then the change is committed to the main file [14]. When required, a specific version of a current file, along with the changes committed to achieve it, can be retrieved. [19].

Git handles the layers of edits to a group of files as revisions, as opposed to versions [19]. First of all, Git is distributed, i.e., every individual associated with the project has a separate copy of the information saved, which gets updated when the mainframe is updated. This copy is known as a branch [20]. Users can make changes on files on their branches, and when sure, can commit the changes to the main branch. In Git, an exhaustive history of all the edits made to a branch is stored in form of a graph, enabling the developers to access the changes as revisions [19].

## 1.4 Limitations of Standalone Systems

Both Blockchain and Version Control Systems can preserve data history to some extent. However, both these technologies have some limitations when they are implemented alone.

Blockchain keeps track of every change that is made to the data in the system. It records the time of modification and the author of the change and makes sure that the event of the modification is visible to all the users. Blockchain also prevents these records from being tampered. However, the the data prior to the change is not stored in Blockchain, only the current version remains.

Git and other VCSs were formed to provide this utility. In a VCS, all the versions of the data prior to every change is stored, and can be accessed concurrently. However, VCS allows to rewrite history with permission from the author [21]. Older commits an be undone without affecting the newer commits. Thus history of data in VCS can be tampered. Also, while it is easy to revert to an older version of the data, there are no provisions to efficiently navigate through the history to excavate data provenance [22].

## 1.5 Problem Statement

There aren't many systems which provide Data Provenance as a service. Of the few systems that do, none are large scale, or commercially available, and are limited in their performance as well. Git provides version control, but is not equipped to provide data provenance, and even then, the use cases are limited to codes and documents.

In this thesis, we propose a distributed system that combines Git and Blockchain to provide secure data provenance with version control. In the said system, multiple users create, access, and modify files which reside in a separate database. Every change is documented in the Blockchain, and every version of a file is stacked in the database.

# Chapter 2

# Literature Review

## 2.1 Cryptocurrencies

A cryptocurrency is defined as a peer-to-peer, cryptographically enabled digital exchange system. Cryptographically is employed to generate and distribute currency units [23]. This process requires distributed verification of transactions without a central authority. Transaction verification confirms the transaction amounts. If the payer owns the currency, then the process verifies that currency units are not spent twice. This verification process is called *mining* [11], and verifiers are called miners [11]. Mining is a competitive process. The first miner to complete the verification is rewarded with new cryptocurrency units. Thus mining introduces new cryptocurrency units or 'coins' in the system. Existing cryptocurrencies use a variety of mining technologies, according to their requirements. Some cryptocurrencies focus on restricting the number of transactions validated per unit time, while others concentrate on achieving fast, lightweight services [24]. Some mining algorithms are deliberately memory intensive; while others are computation expensive [25].

### 2.1.1 History of cryptocurrencies and other distributed monetary systems

The first fully implemented decentralized cryptocurrency was Bitcoin, published by Satoshi Nakamoto in 2008-09 [1]. Before this, articles about peer-to-peer currency systems were published, but none were implemented. Following the success of Bitcoin, several other cryptocurrencies came into existence [26].

David Chaum created an anonymous electronic money system called eCash in 1983 [27]. The major difference between eCash and cryptocurrencies is that eCash was centralized (dependent on banks). The user's money was stored in their local computer by the eCash software in a digital format, cryptographically signed by a bank [27].

PayPal is an Online Money Transfer System established in 1998 [28]. PayPal provides users with an account, which can be linked with bank accounts and credit cards, and users can pay someone or receive payment through the PayPal accounts. PayPal does not have a currency of its own.

A money transfer system called, M-Pesa [29] was established by Vodafone initially in Africa, which later spread to other continents. M-Pesa is a mobile online payment system, where the user can deposit money into an account stored in their cell phones and send PIN secured SMS texts to other users in order to send money [29].

These online monetary systems were all based on fiat currencies [30], where a cryptocurrency is its own currency. It is purely a digital construct with no central bank authority, unlike fiat currencies, whose values are determined by the government. Cryptocurrency valuations are based on a combination of market forces and algorithmic constraints.

### 2.1.2 Working Principle of Cryptocurrencies

Bitcoin was one of the pioneers and the most influential cryptocurrencies. So technical terms coined by Bitcoin are used to describe various cryptocurrency systems. Here are the definition of some terms related to the working of cryptocurrencies:

- **Transaction:** A transaction is the transfer of cryptocurrency units between two individuals.

- **Hash**: A cryptographic hash used in cryptocurrency mining systems is a one-way function [31] that converts data of any size into a output of constant length. For hash algorithms used in mining, calculating a hash should be fast and easy, while reversing the process should be expensive and difficult. Reversal should require a brute force algorithm to recover the original input string that produced the hash. Any minor change in the input

should propagate through the entire output value, so that outputs of similar input values have no predictable similarity. Hashes are chosen because they are one-way functions.

- **Address:** A string of 26-35 alphanumeric characters used to determine where the currencies are going to be sent. As with email for a situation where a given user has multiple addresses, Bitcoins may be transmitted to a person using any one of their addresses. Unlike e-mail addresses, a person may have multiple Bitcoin addresses, and every transaction should use an unique address [32].

  Bitcoin uses two currency formats [32]:

  - The P2PKH format starting with the integer 1,
    (*e.g.,* 1BvBMSEYddIetqTFn9jX4m4GFg7xJaNVN2)
  - The P2SH format starting with the integer 3,
    (*e.g.,* 3J98t1WpEV47CNmQvstopjkiWrnqRhWNLy).

  Each address is associated with a unique public/private key pair is obtained by hashing the public key with the RIPEMD160 [33] hash function.

- **Wallet** A Bitcoin wallet consists of public key-private key pairs and may also refer to client software used to manage those keys and to make transactions on the Bitcoin network [34]. A wallet can be used once or multiple times, providing the user with the flexibility of changing the wallet with each transaction.

- **Block:** A Block [9] is a data structure containing transactions. 465147 Blocks of Bitcoin are in existence as of 5/6/2017.

- **Nonce**: A nonce is a number, usually chosen at random, which is used once for a specific purpose after which it is discarded. Nonce collisions, which happen when two randomly chosen nonces turn out to be the same, are ignored.

- **Blockchain**: A Blockchain is a ledger which makes all cryptocurrency transactions available to public [8]. It consists of a distributed, chronological chain of blocks. It expands

constantly as “completed” blocks with new transactions are posted in it . Blocks are made up of transactions and information from previous blocks. Each block incorporates the previous block’s hash. Multiple computers (nodes) distributed across a network or the Internet are used to record the blockchain.

- **Mining**: Mining fulfills three requirements in a cryptocurrency. It is the required verification step for adding a cryptocurrency transaction and transaction records to the public ledger (the Blockchain). Mining also introduces new cryptocurrency units in the system [11] and limits inflation.

- **Fork**: When two identical blocks containing the same transactions are created a few seconds apart by different miners, a fork is generated [8], because there is a conflict about which block to add to the existing Blockchain. It is resolved by posting the block received first to the Blockchain and discarding the other. Subsequent blocks are added to the latest included block.

### 2.1.3 Overview of Blockchain

A Blockchain is a ledger where transactions are made available to public [8]. Each transaction, once verified, is accumulated in a block [9]. Each block consists of a group of verified transactions. The maximum size of a block is fixed in each cryptocurrency system, which provides a maximum limit to the quantity of transactions included. For instance, the maximum size of a Bitcoin [1] block is 1MB currently. Figure 2.1 shows the structure of a Bitcoin block.

A Bitcoin Block consists of five fields [9], with the following information:

- **Magic number** Magic number is a fixed 4 byte constant.

- **Block size** Block size indicates the ‘size of the data structure’ in bytes. The size allocated for the field is 4 bytes.

- **Block Header** incorporates the previous block’s hash, the time stamp, the block version number, the hash based on a nonce and all the transactions in a Block. The Block header is of size 80 bytes

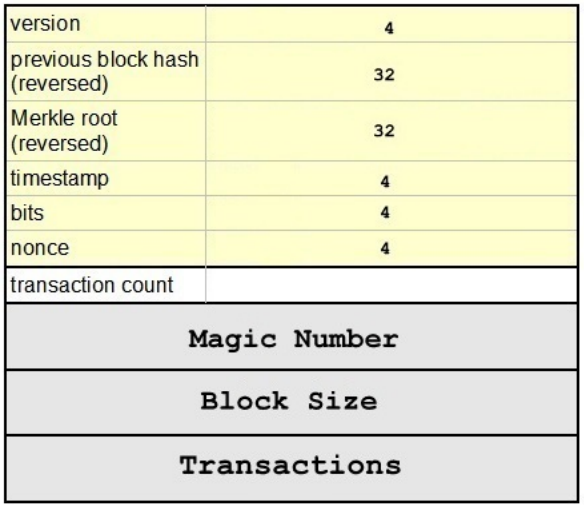



Figure 2.1: A Bitcoin Block (adapted from [2])

- **Transaction counter** states how many transactions are in the block. Its size varies from 1 to 9 bytes.

- **Transactions** The enumerated set of verified transactions added by the block is present in the Transaction field. This list is not encrypted and can be browsed by anyone. However, the signatures of the users are encrypted.

The first block of Bitcoin, the "genesis block", contains the first transactions of a given cryptocurrency. The hash of the first block is passed forward to the miner, who uses it and generates a nonce to create a hash for the second block. The miner does so to achieve a **Target** of pre-specified **Difficulty.** For example, in Bitcoin, the 'target' is an integer starting with a certain number of zeroes. The 'difficulty' is the minimum number of zeroes the target has to contain. Likewise, the miner uses the hash of the second block and a nonce to create the hash of the third block. Thus, a strict chronological link, or chain, is created starting with the genesis block up to the current block through the inclusion of hashes. There is a single, unique path from the most recent block to that first block as shown in Fig 1. This relationship makes it extremely difficult for an attacker to tamper with the information in a block, because all subsequent blocks would have to be regenerated, which would be detected because the final hash wouldn't match [8].

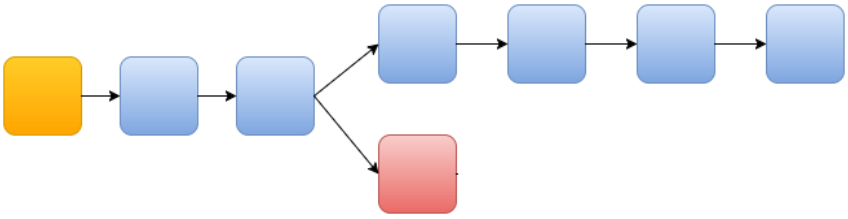


Figure 2.2: Forking in a Blockchain. Yellow indicates genesis block. Blue indicates valid blocks. Red indicates invalid blocks. (adapted from [3]).

In the case of a *fork* in the Blockchain, which can happen when two blocks are created at almost the same time, the block created first according to the timestamp in the block header is accepted in the chain, and subsequent blocks link to the accepted block. Figure 2.2 shows an instance of forking in a Blockchain. Yellow represents the genesis block, Blue represents the longest, hence valid chain of blocks. Red represents the discarded block. The Blockchain algorithms have to be resilient to time variations across their peer ledgers, and thus must allow for forks.

In case of two forks having multiple blocks, the fork with more number of blocks is selected to be included in the chain.

### 2.1.4 Overview of Mining

Cryptocurrency systems rely on a distributed public ledger known as the Blockchain [10]. A transaction is created when a *payer* sends currency to a *payee*. Mining validates transactions and adds them to this public ledger. Miners are required to solve a resource-intensive task. The resource intensive nature of mining serves two purposes.

- When a new transaction takes place, the miner checks if the currency belongs to the payer, and if the payer is trying to double spend [12]. The ownership of the currency is recorded in the Blockchain. A malicious user may create multiple fake identities and try to validate an invalid transaction. Resource intensiveness makes it expensive and hence infeasible for a malicious user to create enough false identities to outnumber benign users and validate an invalid transaction.

- Every cryptocurrency offers the miners a reward for a successful mining. For example, for mining a block in Bitcoin, the miner receives 25 bitcoins (as of 5/6/2017). This value reduces by half after every 210,000 mined blocks. Resource-intensiveness limits the number of blocks mined, and hence the number of currencies created per day. This limits inflation.

Additionally users offer fees to the miners for verifying their transactions. This provides the miners with incentives to expedite certain transactions.

The resource-intensive task can be any of the following:

- *Proof of Work* [35], the easily verifiable result of a resource intensive task, which confirms that the task has been performed. Bitcoin and many other cryptocurrencies use proof of work.

- *Proof of Stake* [36] , requires the miner to show how much of the cryptocurrency the miner owns. Peercoin introduced Proof of Stake.

- *Proof of Retrievability* [37] , requires the miner to store a considerable amount of data and to show the proof that the data stored is intact and can be recovered at will.

- *Proof of Importance* [38], determines the user's participation in the network. Based on the transactions from the user's wallet and the stake the user has in the system [38], the importance of the user is calculated, and the number of blocks allowed to be verified by the user is proportional to the importance [38]. Proof of Importance is used only by the cryptocurrency NEM [39].

- *Proof of Resource* [40], which requires the miner to show evidence of computing power, storage power or network bandwidth to be able to validate a transaction.

Here are the steps involved in mining:

- A miner then checks for the validity of the transactions.

- The miner then performs a resource-intensive task and produces a proof that the work has been done [35]. This task prevents a malicious miner from forming false identities and manipulating, and also limits the number of new currencies created.
- The proof produced is verified to confirm that the task has been performed.
- If the entirety of the transactions are proven valid for a given block, that block is posted in the Blockchain [11].

### 2.1.5 Proofs Required for Mining

#### Proof of Work

The Proof-of-Work algorithm [35] is accomplished by imposing the requirement of delivering specific information that is inconvenient to produce yet easy to verify. This proves that the work has been done. Proof of Work requires trial and error. It is designed to have a low probability of success.

In cryptocurrencies, Proof of Work restricts the number of transactions that can be validated (and consequently the number of blocks added to the ledger) in a given time period. This restriction is necessary because, with each block mined, new currency units (the number of which are finite) are produced. In the case of Bitcoin, every block currently introduces 50 new Bitcoins in the system. The number of new Bitcoins introduced reduces to half after each 210,000 blocks. Consequently, through simple geometric progression, there can be at most 21 million Bitcoins [41].

Usually, cryptocurrencies use one-way functions (typically hashes) for Proof of Work [11]. The miner gets as input the hash of the previous blocks. He or she would have to choose a nonce so that that when the current hash and the nonce are hashed, the result follows a structure defined by the cryptocurrency. Bitcoin requires that the output must have 0s in the $n$ most significant bits [1]. This required output is called the Target [42]. Each Target has a Difficulty [43]. For example, the number most significant bits to have the zeroes in the Difficulty of a Bitcoin Target. More the number of zeroes, more difficult it is for the miner to find the nonce that corresponds. Bitcoin increases the Difficulty for every 2016 blocks.

For example, a miner receives as input the hash

$$H_n$$

of the previous block. The difficulty is set to 6. The miner has to try and generate a nonce X to calculate

$$H_(n+1) = Hash(H_n, X)$$

such that the most significant 6 digits of

$$H_(n+1)$$

are 0s.

Calculating an input from the hash is resource intensive because it needs several trials to generate the correct nonce and the miners need to utilize computational power or memory to achieve the Target, whereas verifying its correctness by calculating the hash is sufficiently fast. Hash functions are designed so that determining the input from the output is extremely time consuming as to be intractable [31]. The miner has to generate nonces and try hashing them with the given input until the requirements are fulfilled [11]. The computational complexity of the reverse hashing function is significantly higher than the hashing function as it is a brute force algorithm. Obtaining the correct nonce is resource-intensive and time consuming as it involves calculating a huge number of hashes, whereas verifying if indeed the nonce, when added to the hash of the previous block, produces a new hash that fulfills the requirements is a matter of one hash computation, and is fast [11].

Proof of Work still remains the most popular mining method for cryptocurrencies. The Proof of Work algorithm of Bitcoin is known as Hashcash. Bitcoin and all of its forks use SHA2 series as their hashes. Litecoin uses Scrypt [44] as the hashing function of the Proof of Work.

### Proof of Stake

Peercoin introduce Proof of Stake as a mining proof strategy. Instead of requiring the miner (known as the *prover* in Peercoin [36]) to perform a certain amount of computation, a Proof of Stake system requires the prover to demonstrate that is has ownership of a specific (minimum) quantity of cryptocurrency [36]. The ownership information is publicly maintained in the ledger. The miners protect their own stake in this approach [36]. With Proof of Stake, the portion a miner can verify is the amount of currency a miner holds [36]. For instance, someone possessing 1% of the cryptocurrency can mine 1% of the "Proof of Stake blocks" [36]. If a miner tries to validate an invalid transaction, a part of their stake is forfeited as a penalty. the amount of penalty is proportional to the amount of stake the miner has.

Proof of Stake is highly energy efficient [45]. It still has to have a block selection policy [45], inclusive of the following:

- "Randomized block selection",
- "Coin-age-based selection",
- "Velocity-based selection", and
- "Voting-based selection".

Proof of Stake, however, is said to be vulnerable to the **Nothing at Stake** problem [45], in which miners have nothing to lose if they vote for a wrong or invalid transaction [45]. Since the penalty is supposed to be proportional to the stake, having no stake might imply no penalty.

Proof of Stake tries to reach a consensus and prevent double spending [36].

Peercoin pioneered Proof of Stake and is still one of the most popular cryptocurrencies to use it. A more user-friendly cryptocurrency is Blackcoin, which requires relatively less amount of currency to claim stake. Nxt also uses Proof of Stake.

### Proof of Retrievability

Proof of Retrievability was introduced by Permacoin [37]. Instead of performing some computationally intensive task, this system requires the miner to store considerable amount of useful

information, and show the proof that it exists and can be accessed at any time. Proof of Retrievability banks on the point that the user would have to own (or afford) extensive amount of storage, and this resource (the storage) is utilized for meaningful information. Thus, while the Proof of Work has no utility beyond restriction of the miners, Proof of Retrievability serves the dual purpose of mining and storage.

Miners, known as Verifiers in Permacoin [37] still have to prove that they have solved a mathematical problem but it is much less computation intensive [37], and is known as a scratch-off puzzle [37]. The puzzle is based on a Floating Preimage Signature [37].

The verifiers must refer to a section of code stored locally on their computer to solve the puzzle [37]. If they successfully solve the problem, then the algorithm can deduce that they are storing that data (at least for a short time) [37]. Thus, all verifiers must be storing a piece of archived data to participate by mining Permacoin [37].

#### Proof of Importance

Proof of Importance [38] was introduced and to date, used only by NEM [39]. An importance score is assigned to all the accounts on the NEM blockchain[46]. This score influences how individual users can mine the blockchain. As a miner's importance score increases, they will have a better chance to be chosen to mine. To be eligible for the importance calculation, users need to have at least 10,000 XEM in their balance [46]. The importance is calculated from a graph-theoretical data structure called Outlink Matrix which describes the weighted net flow of XEM currency in a fixed time period [47].

### 2.1.6 Hash Algorithms

In this section Hash algorithms used by the cryptocurrencies are discussed, namely:

**SHA 256** SHA 2 [48] is a set of Secure Hash Functions that has six algorithms, which produce digests (results) that are of different bit-lengths. SHA 256, creates a digest of 256 bits (that is, when a piece of data is fed into this algorithm, it creates a hash of 256 bits) [31]. SHA 256 satisfies the requirement of unidirectional hashes (that is, any change in the input, however

insignificant, leads to a completely different hash, and determining the input from the hash is practically impossible) [31]. Also, the same input will always produce the same digest [31].

SHA 256 pads input to convert its length to a multiple of 512 bits [48]. Then, it divides the input into blocks of 512 bits each [48]. The message blocks are processed one at a time, starting with a fixed initial value $H^0$ [48] , sequentially computing

$$H^i = H^{i-1} + Ch_{Ma^i}(H^{i-1})[48]$$

where $Ch$ is the SHA-256 compression function and $+$ means word-wise addition modulo $2^{32}$ [48]. $H^N$ becomes the hash [48]. The compression function permutes and compresses the input block and is a combination of bitwise logical operators, such as AND, OR, XOR, Complement, etc. [48].

Doubled SHA 256 [49] is abbreviated as SHA256d. It is simply the SHA 256 hash performed twice serially

Bitcoin uses SHA 256d [49] as its hash function, and the output is specified to have a certain characteristics. For example, the *n* most significant bits of the digest have to be zeroes. The miner has to come up with a nonce that, when appended to the hash of the previous block, yields a digest with the abovementioned property. Other cryptocurrencies, such as Peercoin and Namecoin, that also use SHA256d, may pose different requirements in their outputs [36].

**Scrypt** Scrypt [4] was designed to be a Key Derivation Function (KDF). All Key Derivation Functions are resource-intensive in order to mitigate large-scale custom hardware attacks [44]. Scrypt takes an input and *generates* a large vector of pseudo-random bits. Since these vectors are generated at runtime, the algorithms require large memory. More memory leads to faster computation [44].

Within the algorithm, there are two functions called Smix and Blockmix [4]. Blockmix performs permutation operations on the input blocks using binary logic operands and, in each iteration, the output of the Blockmix is again processed in Smix, which performs bitwise permutations [4]. Figure 2.3 shows the different modules of Scrypt.

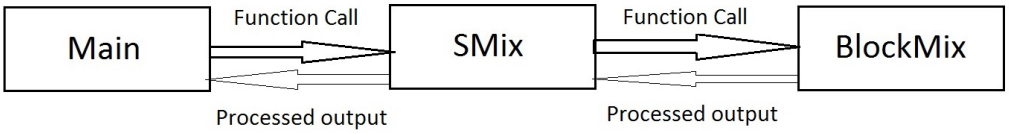



Figure 2.3: Modules of Scrypt (adapted from [4])

Scrypt was modified for the purposes of mining. Since the original Scrypt uses pseudo-random bits, the outputs of the same input would be different. This makes it harder to verify. When Scrypt was used for KDF, there was no need for verification [44].

**EtHash** EtHash [50] is exclusive to Ethereum. It was designed to thwart the dominance of Application Specific Integrated Circuits (ASICs), so that ASIC producers would not have any considerable advantage over people mining with either or both GPUs or CPUs. The verification of correctness of this proof of work is incredibly fast, taking .01 seconds for a light client.

The general EtHash algorithm involves the following steps [50]:

- There exists a seed that can be computed for each block from the data stored in the block headers.
- From the seed, a 16 MB pseudo-random cache can be computed. EtHash uses its own Pseudo-Random Number Generator.
- From the cache, a 1 GB dataset can be generated, such that each item in the dataset depends on only a few items from the cache.
- Mining involves selecting random elements of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that is needed, so it is sufficient to store just the cache.

**Blake** Blake [51] is a cryptographic hash function built from the ChaCha stream cipher [52] with a few modifications.

**X11** X11 [53] is a chained hashing algorithm. It chains 11 different algorithms together [53]. These are as follows: Blake [51], BMW [54], Groestl [55], JH [56], Keccak [57], Skein [58], Luffa [59], CubeHash [60], SHAvite [61], SIMD [62], and Echo [63]. X11 is ASIC-resistant and suitable for both CPU mining and GPU mining [53].

**CryptoNight** CryptoNight [64] is a memory intensive hash function, resistant to ASIC, GPU and FPGA architectures. CryptoNight involves three steps, generating pseudo-random addresses in a scratchpad [64], read/write operations on the addresses [64], and performing bitwise XOR and shift functions on the scratchpad [64].

SHA256 and Scrypt are the most popularly adopted mining algorithms with extant cryptocurrencies. Only a few cryptocurrencies have developed their own mining algorithms.

### 2.1.7 Cryptocurrencies with most market values

There are many cryptocurrency mining techniques in use. This section lists the top 5 cryptocurrencies with most market capitals and the mining algorithms they use [65].

**Bitcoin** Bitcoin mining uses Proof of Work [1]. The Proof of Work algorithm in use is called Hashcash [66].The hash algorithm used is SHA256 [49].

Bitcoin mining can be done in CPU, but is more efficient in GP-GPUs [67]as they provide more computational power. Application Specific Integrated Circuits (ASIC) [68] have also been developed to mine Bitcoin. The miners are rewarded new Bitcoins as incentives.

Bitcoin mining works as follows [1]:

1. A miner selects transactions he wishes to verify.

2. The miner uses the transactions to build a Merkle Tree [69].

3. Extracts root block hash from the Merkle tree.

4. Adds a nonce, hashes the block header.

5. Keeps incrementing the nonce and hashing until desired result is obtained [11].

6. This result is the Proof of Work [35]. Other users agree/verify that the proof of work matches. Then the transaction is validated and new Bitcoins are introduced to the system.

A Merkle tree [69] is a data structure in which leaves serve as the data elements while their parent nodes contain hashes of the data elements. As the height increases, the lower levels updates the cumulative hashes, and the root always contains the hash of all the elements in the tree. Hash trees provide organized and invulnerable authentication of the elements of big data.

The hash of a transaction, also called the hash pointer, can both be used to look up the transaction and to verify that the transaction retrieved has not been tampered with since it was stored.

To mine coins successfully using SHA-256, the hash rates need to be at the gigahashes per second (GH/s) range or higher [67]. The current average by 2014, time needed to mine a Bitcoin Block with SHA-256 is 10 minutes [11].

**Ethereum** Ethereum was crowdfunded in 2014 [70]. Ethereum also relies on Proof of Work but they do not use any preexisting hash algorithm [70]. Instead, the designers developed their own hashing algorithm called EtHash [70]. EtHash [50] is described further in section 8 below.

The principal objective for constructing a new Proof of Work function instead of using an existing one was to mitigate the problem of mining centralization [71], in which a small group of hardware companies or mining operations can acquire a disproportionately large amount of power to impact or manipulate the network. EtHash is ASIC resistant [50], and has the property of memory hardness (that is, ASICs wouldn't have superiority over other methods and EtHash relies on how fast the memory can move around data) [50].

**Ripple** Ripple [72] does not use mining in its truest sense. Released in 2012, it uses a trust-based system to attain consensus [72]. The aim of consensus is for each server to agree to push the same group of transactions to the current ledger [72]. New ledgers are formed in the

interval of seconds, and the latest completed ledger contains a record of all Ripple accounts and previous transactions [72]. A transaction is any change implemented in the ledger and can be performed by any server. [72]. The servers attempt to arrive at a consensus about a group of transactions to apply to the ledger, creating a new last closed ledger [72].

**Litecoin** Litecoin [73] was the first cryptocurrency to use Scrypt [4] for mining. Scrypt was originally a key derivation function (KDF) [4] developed by Colin Percival and published in 2012. Scrypt's strength lies in the time-memory trade off; that is, a attacker would need more memory to complete the attack faster, and Scrypt's memory requirement makes it expensive, hence slowing down any attack [73]. Scrypt has also been successfully implemented as a proof-of-work verification [4]; Litecoin was the first one to do so [73].

The large memory requirements of Scrypt generates from a large vector of pseudo-random bit strings that are produced as part of the algorithm. After the vector is created, its elements are pseudo-renadomly accessed and combined to generate the derived key [4]. As a Proof of Work, the key would have predefined characteristics and the miner would have to come up with the sequence of bit strings that match the key [73].

Scrypt is much newer, simpler, quicker yet more secure than the SHA-2 series [74]. While SHA is more computation intensive, taking up more time, Scrypt is memory intensive, serving them same purpose in lesser time [74]. Scrypt's hash rates for successful mining usually are in a range from $10^3$ to $10^6$ hashes per second depending on areas of difficulty [4]. Scrypt takes only 2.5 minutes to mine a block with the same difficulty attributes [74].

**Monero** Monero [75] is a digital exchange medium which focuses on privacy. Monero was built on CryptoNote [76]. CryptoNote is an application layer protocol that forms the base of many cryptocurrency Blockchains [76]. However, Monero does not reveal the identities of the users involved in a transaction [75]. Monero uses CryptoNight [64] as its Proof of Work Algorithm.

## 2.2 Version Control Systems

Version Control Systems (VCS) are mostly used by programmers and developers to maintain a track of their code. When a change has to be made to a file, an image or a copy of the file prior to the change is stored, and then the change is implemented.

### 2.2.1 Overview

There are two types of Version Control Systems [22],

- **Centralized Version Control Systems** or CVCS.
- **Distributed Version Control Systems** or DVCS.

A CVCS has only one central repository [22]. This repository contains all the versions of the files along with their metadata, and is typically stored in a server.

A DVCS does not require a central repository. All the users have the entire repository locally, known as the local repository in their own computers [22].

Git is a Distributed VCS.

### 2.2.2 Git

Git is a content-addressable filesystem[77]. It means that at the core of Git is a simple key-value data store. Any kind of content can be inserted into it, and it will give back a key that can be used to retrieve the content again at any time.[5] All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to nodes or file contents. A single tree object contains one or more tree entries, each of which contains a SHA-1 pointer to a blob or subtree with its associated mode, type, and filename [5].

Git has 4 types of objects.

1. Blob : A blob object is used for storing the contents of a single file. Git doesn't store diff of the contents of the files. It stores snapshots(the exact content of the files) at the point a commit is made [5] .

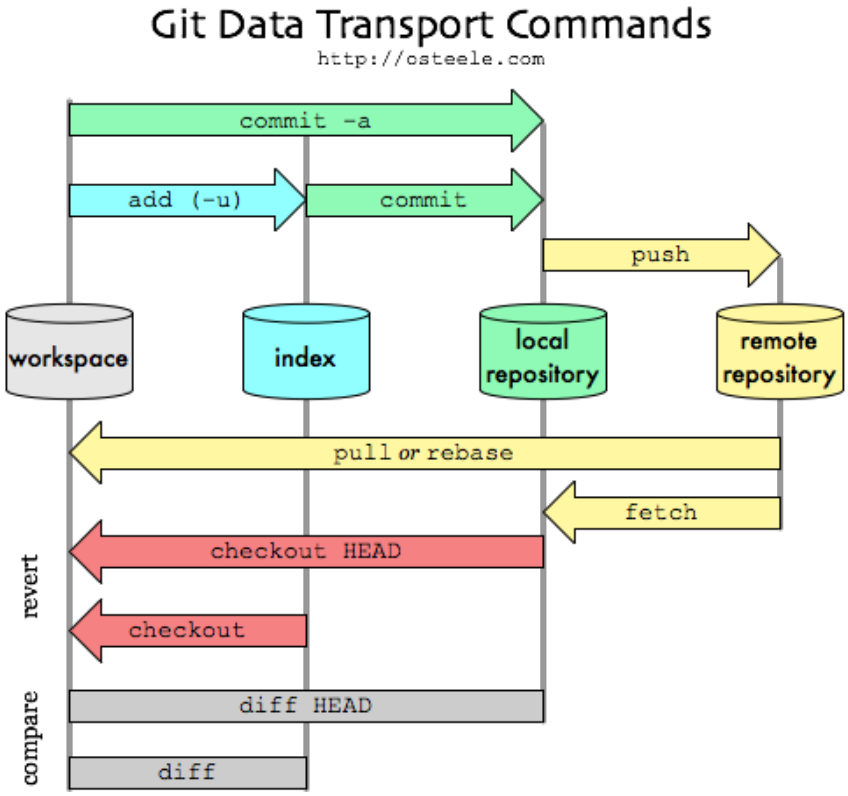



Figure 2.4: Git Data Transport Commands

2. Tree: A tree object contains references to other blobs or subtrees. A tree is a simple list of trees and blobs that the tree contains, along with the names and modes of those trees and blobs. The contents section of a tree object consists of a very simple text file that lists the mode, type, name and sha of each entry [5].

3. Commit: A commit object contains the reference to another tree object and some other information(author, committer etc.) The commit is very simple, much like the tree. It simply points to a tree and keeps an author, committer, message and any parent commits that directly preceded it [5].

4. Tag: A tag or a tag object is just another reference to a commit object and just makes for easier referencing. Tag provides a permanent shorthand name for a particular commit. It contains an object, type, tag, tagger and a message. Normally the type is commit and the object is the SHA-1 of the commit. The tag can also be GPG signed, providing cryptographic integrity to a release or version [5].

Remembering the SHA-1 key for each version of the file isn't practical; plus, the filename isn't stored in the system — just the content. This object type is called a blob. Git can tell the user the object type of any object in Git, given its SHA-1 key [5].

Git constructs a header that starts with the type of the object, in this case a blob. Then, it adds a space followed by the size of the content and finally a null byte. Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content. Git compresses the new content with Zlib [78]. Zlib is a compression algorithm [79].

It is important to note that it is the contents that are stored, not the files. The names and modes of the files are not stored with the blob, just the contents. This means that if there are two files anywhere in the project that are exactly the same, even if they have different names, Git will only store the blob once[5]. This also means that during repository transfers, such as clones or fetches, Git will only transfer the blob once, then expand it out into multiple files upon checkout [5].

Git gets the initial SHA-1 of the starting commit object by looking in the .git/refs directory for the branch, tag or remote specified. Then it traverses the objects by walking the trees one by one, checking out the blobs under the names listed [5].

There are three main actions that Git allows on its files.

- **Git Fetch**: Multiple users could edit the same repository simultaneously. That results in different local versions of the same repository. When a Git Pull action is performed by the user, the contents of the target repository is brought to the local repository for reviewing [5].

- **Git Merge**: When the user decides to commit the fetched contents to the user's repository, it is called a Git Merge [5].

- **Git Pull**: A Git Pull is the combination of a fetch and a merge in that order. When a user pulls another repository, it automatically gets merged with their local repository and doesn't allow the user to review before the merge [5].

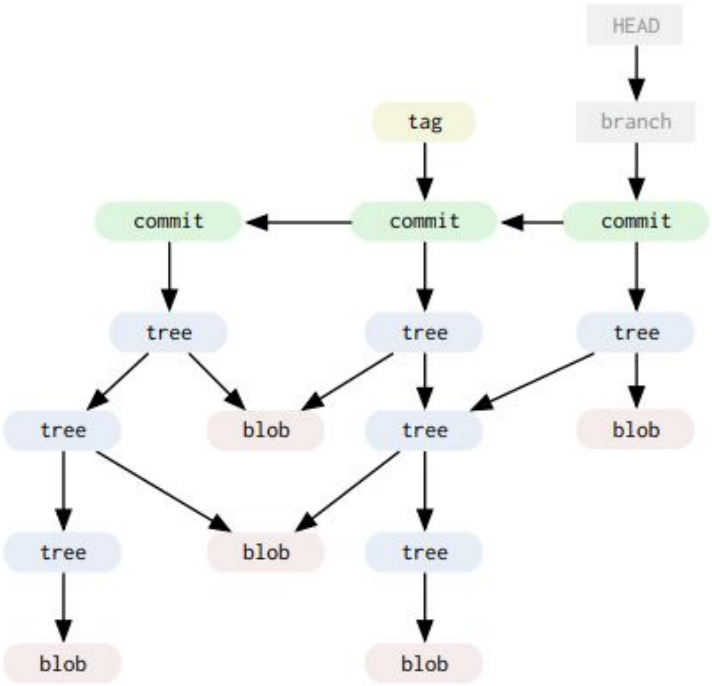



Figure 2.5: Git Workflow [5]

## 2.3 Data Provenance Systems

Provenance of digital scientific data is a vital component in the field of sharing and reuse of scientific data. Provenance provides the information associated with authenticity and evaluates the quality of a particular data set. Provenance collection can be considered as a part of a cyber-infrastructure system, but it is more efficient as a standalone tool [80].

### 2.3.1 Karma

The Karma provenance tool is a standalone tool that can be added to existing cyber-system for the purposes of collection and representation of provenance data [81]. It is developed by Co-I Beth Plale at Indiana University. Karma utilizes a modular architecture that permits support for multiple instrumentation plugins that make it usable in different architectural settings. It provides the downstream provenance of a given data object. The resulting provenance trace has the input data object as the source for all other elements in the provenance trace.

Karma facilitates visualization of provenance data which is more useful with support for manipulating very large structures and for interactivity [82]. This can help a user to navigate their experiment information with a mental map of what is going on in the experiment. The Karma provenance framework records uniform and usable provenance metadata for scientific workflows.

Karma collects two forms of provenance: the first being process oriented which describes the workflow's execution and is used to monitor the workflow progress and mine it for results validation [83]. It also collects data provenance, which provides complementary metadata about the derivation history of data products in the workflow, including the services that create and use it, and the input data transformed to generate it, and forms the basis for quality-oriented data product discovery.

Karma consist of the following components:

1. Server: acts as a notification subscriber of all provenance activities stored in the database.

2. Service bus (RabbitMQ): the Karma server is accessible through either a webservice API or a RabbitMQ enterprise bus [84].

3. Querying interface (NetKarma): A GUI component that queries provenance information from the Karma database [85].

4. Visualization interface: the Karma uses Cytoscape to manipulate large provenance graphs, display views, and for interactivity [86]

5. Database: provenance activities are tightly coupled with the data and are stored in a relational database backend.

### 2.3.2 Komadu

Komadu is another standalone provenance collection tool developed by Indiana University[87]. In Komadu provenance is secured in two ways. It processes log files after the application is executed or it instruments an application directly [88]. Komadu provides a Web Services API and a Messaging API for both provenance collection and querying collected data. Provenance

collection is driven by notifications which represents a particular event related to some activity, entity or agent. Query API can be mainly used to find details about a particular activity, entity or agent and to generate the provenance graph for a particular activity, entity or agent. Unlike Karma, Komadu is not tightly coupled to workflows and scientific provenance collection. Komadu API uses generic terms and operations such that any type of provenance can be captured and queried.

The Komadu architecture consists of these components [80]:

1. Ingest API: Sends notifications while collecting provenance.
2. Query API: Handles queries to the server.
3. Database: A MySQL database that stores notifications and processed components, and generates provenance graphs.
4. Raw Notification Ingester: Receives and transfers raw XML notifications into the database.
5. Asynchronous Raw Notification Processor: Consists of a thread pool that regularly searches for unprocessed notifications remaining in the database.
6. Graph Generator: Creates provenance graphs by adding connected nodes to the initial node ensuring that all connected nodes are included.
7. RabbitMQ Messaging Channel: Receives provenance notifications and sends responses to the incoming queries.
8. Axis2 Web Service Channel: A Web service that runs on Axis2.

### 2.3.3 Others

Some other tools which are used generate and maintain provenance are as follows:

- IPython Notebook: Open-source and based on python, this tool supports data visualization [89].

- Taverna: Combines distributed web services and/or local tools into a complex analysis pipeline [90].
- VisTrails: Supports data exploration and visualization [91].
- Kepler: A Java based tracking tool based on the Ptolemy II systems [92].
- Swift: A scientific workflow management system which is parallel programmed [93].
- Sumatra: A python based provenance tool that tracks provenance of projects using mathematical simulations [94].

## 2.4 Scrybe

Scrybe is a system designed to store provenance securely using blockchains (distributed ledger technology). Substantial modifications have been made to the Bitcoin blockchain architecture and software framework in order to accommodate the needs of this system.

### 2.4.1 Introduction to Scrybe Data Structure

This subsection contains details of the blockchain architecture used in Scrybe and comparisons with the Bitcoin blockchain.

#### Blockchain Architecture

The Scrybe blockchain differs from the Bitcoin blockchain in multiple respects. While the Bitcoin blockchain has to include functionality like mining new currency or reward the miner, the Scrybe blockchain only considers authentication of the user and verification of the data.

Table 2 provides a simple comparison between the two blockchains:

#### Entries

In Bitcoin, a transaction takes place when a payer wallet address sends some currency to a payee wallet address. The transactions are then verified by a miner and listed in a block. The block is then registered into the Blockchain. However, in Scrybe, instead of two, only one

Table 2.1: Comparison between Bitcoin & Scrybe blockchain

| Components | Bitcoin | Scrybe |
|---|---|---|
| Transactions | Proof of Present Ownership<br>Value Oriented | Proof of Past Ownership<br>Data Oriented |
| Block | Difficulty & Nonce | Cryptographically Signed |
| Mining | Resource Intensive | Lightweight, Simple selection Oriented |
| Security | Based on computational difficulty | Based on cryptographic signatures |

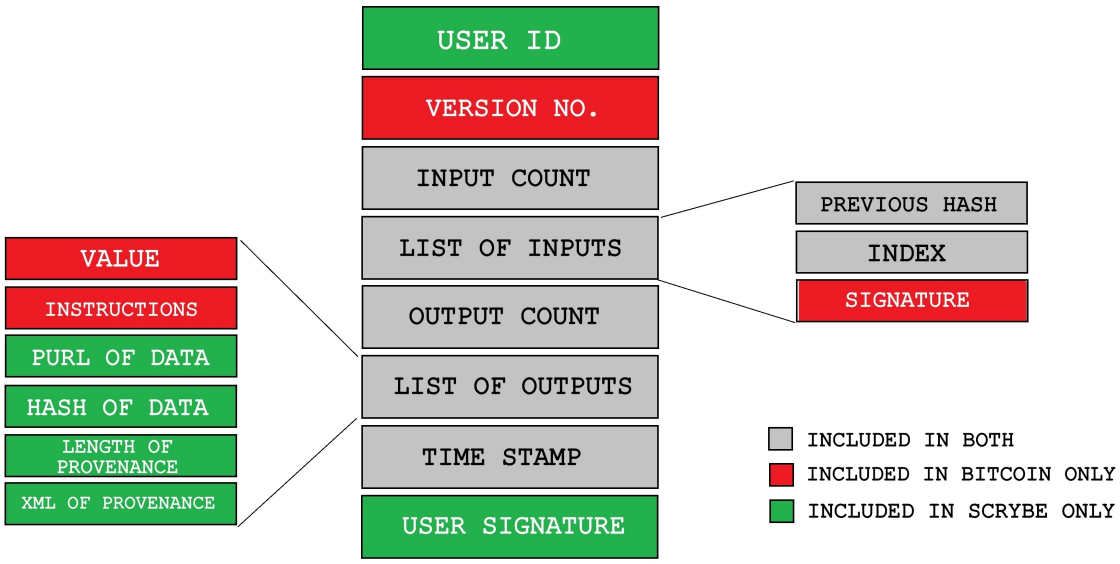



Figure 2.6: Comparison of a Bitcoin Transaction and a Scrybe Entry

actor is involved in an entry. The user commits some change in the system. They can introduce new data in form of a file, or can modify existing data. This entry is recorded in a block by the verifier and the block, in turn, registered to the blockchain. There are a few differences between a Bitcoin Transaction and a Scrybe Entry.

Figure 2.6 explains the differences between a Scrybe entry and a Bitcoin Transaction. The diagram is color coded. Grey indicates fields which are common to both, red indicates fields present only in Bitcoin, whereas green indicates fields present only in Scrybe.

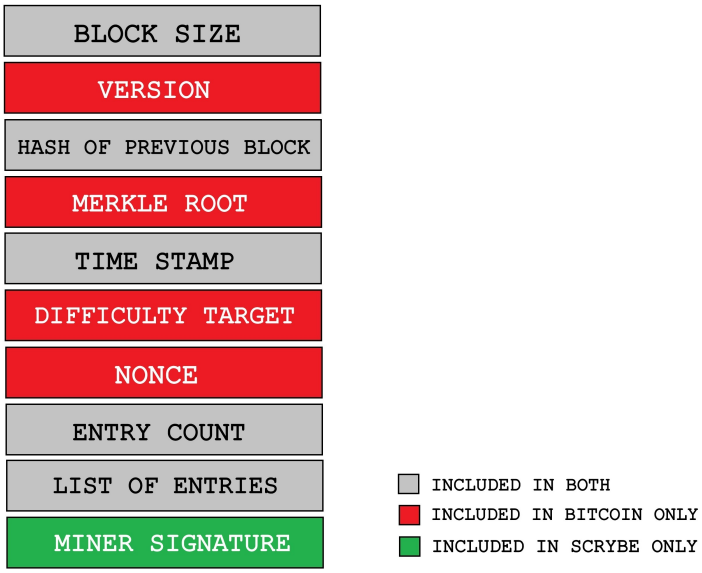



Figure 2.7: Comparison of a Bitcoin and a Scrybe Block

### Blocks

Bitcoin blocks are modified according to requirements to form Scrybe blocks.

Figure 2.7 explains the differences between a Scrybe and a Bitcoin block. The diagram is color coded. Grey indicates fields which are common to both, red indicates fields present only in Bitcoin, whereas green indicates fields present only in Scrybe.

### Lightweight Mining Algorithm

Bitcoin mining requires Proof of Work, and is a computationally intensive process. Scrybe introduces a Lightweight Mining Algorithm which is not resourse intensive. In Bitcoin, the miners compete with each other to mine a block, and the miner to submit the Proof-of-Work first gets to mine the block. Scrybe introduces a fair miner selection process described as follows.

1. Each Miner generates a Random Number

2. Miners broadcast hashes of their respective random numbers

3. Once all the hashes are broadcasted, the miners broadcast the original random numbers.

4. The random numbers would be sorted in ascending order and the miners ranked according to them. i.e. the miner with the lowest random number would be rank 1.

5. The miners calculate a modulo of the sum of the hashes with respect to the number of miners present.

6. The miner with the rank = (sum % N) gets to verify the entry, create the block and broadcast it.

Once a miner is selected, they check the correctness of different fields in an entry, and if all the checks turn out to be valid, the entry is included in a block, and the block is subsequently registered in the blockchain.

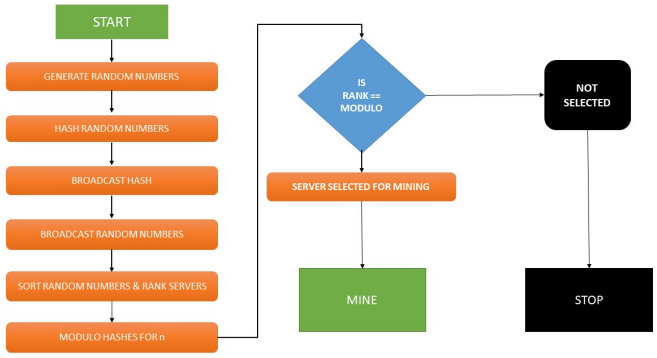



Figure 2.8: The Process of selecting a Miner in Scrybe

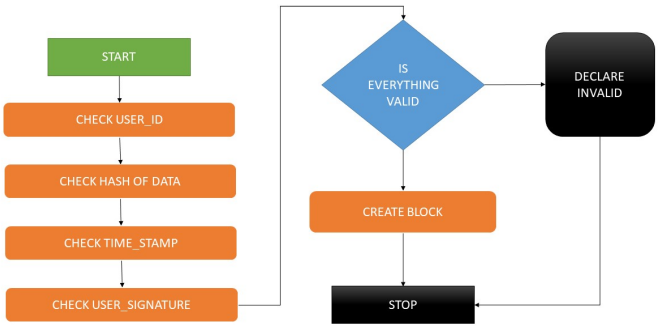



Figure 2.9: The Algorithm of Mining in Scrybe

# Chapter 3

# Methodology

In this project, I propose a Data Provenance system with selective and general version control options. The purpose is achieved by introducing a blockchain linked to the data storage. Each and every change made to storage system would be stored in the blockchain. Changes include introducing a new file, deletion of an existing file, or modification to an existing file.

The Flakes architecture consists of different components.

- The **User End**, where the user of the system makes a change to one or multiple files. These changes include *Adding* a file, *Editing* a file, *Deleting* a file, or *Reverting* to a previous version.
- The **Blockchain**, where every change is stored in a data structure called an *Entry*. Each Entry also contains version numbers for the file/s for which the entry was created as well as for the system and the differential of the previous and the new versions for each file.
- The **Provenance**, where the concerned user looks up the provenance information stored in the blockchain.
- The **Version Control**, where the concerned user uses the information in the blockchain to retrace to an older version of a file or the system.

## 3.1 Flakes Blockchain Architecture

Whenever a change is made to a file, the server accumulates certain information, which are:

- Entry ID, i.e. a serial number that counts the number of Entries.
- File ID, i.e. the PURL (Personalized URL, an unique identifier) of the file.
- Authorship, i.e. the Digital Signature of the person who made the changes.
- Edits, i.e. what changes were made, for example, the ASCII difference between the previous character and the changed character at each index of a text file. We will call that differential (Di).
- Version No., i.e. a serial number starting from 1 incremented by 1 each time the file is edited.
- Timestamp: When were those changes made(Ti).
- Hash: He = SHA256(PURL,Signature,Di,Version No.,Ti).

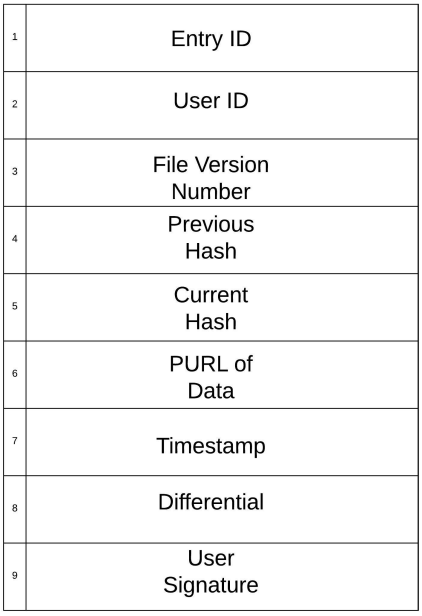
### 3.1.1 Flakes Entry

The Flakes Entry contains various fields along with the link to the actual document.

The fields included in Flakes are:

- **Entry ID**: Contains the serial number of the entry in the system.
- **User ID**: Contains the ID of the user who created the entry.
- **File Version Number**: Since Flakes offers version control, after every change is committed to a file, a new version number is attributed to the file.
- **Previous Hash**: Contains Hash of the document on which the change is made, before the change is made.
- **Current Hash**: Contains Hash of the document on which the change is made, after the change is made.

Figure 3.1: Diagram of a Flakes Entry




- **PURL of Data**: Contains a link to the location of the actual document. Doesn't change with the modifications.
- **Timestamp**: Contains the timestamp of the precise time when the entry was created, i.e. when the change was saved.
- **Differential**: Every version of a file is an update of the previous version. Each change is stored in the entries, so that when a file has to be reverted back to a previous version, all the interim differentials are deleted.
- **User Signature**: Contains the digital signature of the user.

Figure 3.1 shows a Flakes Entry.

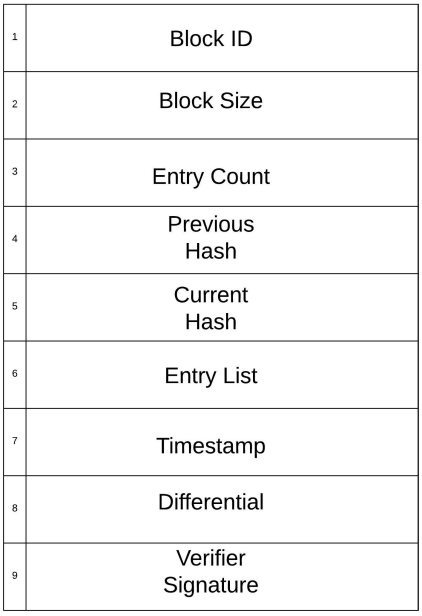
### 3.1.2 Flakes Block

The Flakes Block contains a series of header fields used to identify the actors and secure provenance information.

The fields included in a Flakes block are as follows:

- **Block ID**: Contains the serial number of the block.

Figure 3.2: A Flakes Block




- **Block Size**: Contains the size of the block in KBs.
- **Entry Count**: Contains the number of entries included in the block.
- **Previous Hash**: Contains the hash of the previous block.
- **Current Hash**: Contains the hash of the current block.
- **Entry List**: Contains the list of the entries included in this block.
- **Timestamp**: Contains the timestamp when the block was created.
- **Differential**: Contains list of the files where changes were made. So, when the system has to revert back to this version from the current one, the system will track down all the files where changes were made, and then undo all the changes.

## 3.2 Version Control in Flakes

The Flakes blockchain not only provides provenance, it enables the user to retrieve previous versions. In this section we discuss the process of version control in Flakes.

Two way commute between the versions is allowed in Flakes. For example, if the current version is version 6, and system is reverted to version 2, then the new image of the system will

be tagged version 7, which will be an identical copy of version 2. Thus, the system can be reverted to version 6 anytime, making two way commute possible.

### 3.2.1 Version Control Process

Flakes provides version control in two layers, *General* which is a version control for the whole system, and *Specific*, which maintains versions for each file in the system.

- **General** A block in the Flakes blockchain may contain more than one entries. Each entry is specific for any change in a single file. So when a miner creates a block, it might contain several changes owing to multiple entries. After these changes are committed, the version number of the system is updated by 1. In a way, the current version number of the system is equal to the height of the blockchain. Whenever an user wants to retrieve an earlier version, they can access all the intermediate differentials from the blocks, and undo them in order. So if the current version is $n$, undoing $i$ differentials will yield version $n - i$.

- **Specific** If a user wishes to retrieve an older version of a single file in the system, resetting the whole system for it is inefficient and impacts other files as well, which is not desired. So each entry in Flakes has a specific version number for a file, and its own differentials, which is stored in the metadata of the file. If the user wishes to set the file to an older version, they can undo all the differentials of only that file, and achieve the result.

Retrieving an older version (of a system or a file) is considered as an action and is noted in the blockchain. The new state of the system and the file, although identical to a previous version, is assigned a new version number, for the flow to continue. For example, if the current version is $x$ and the system is reset to $x - i$ where $i <= x$, then version $x + 1$ will be identical to version $x - i$.

## 3.3 Working Model of Flakes

This sections shows how the different modules of the system interact with each other to deliver provenance and maintain versions. Pseudocodes of the important methods are shown here.

Figure 3.3: Flakes Class Diagram

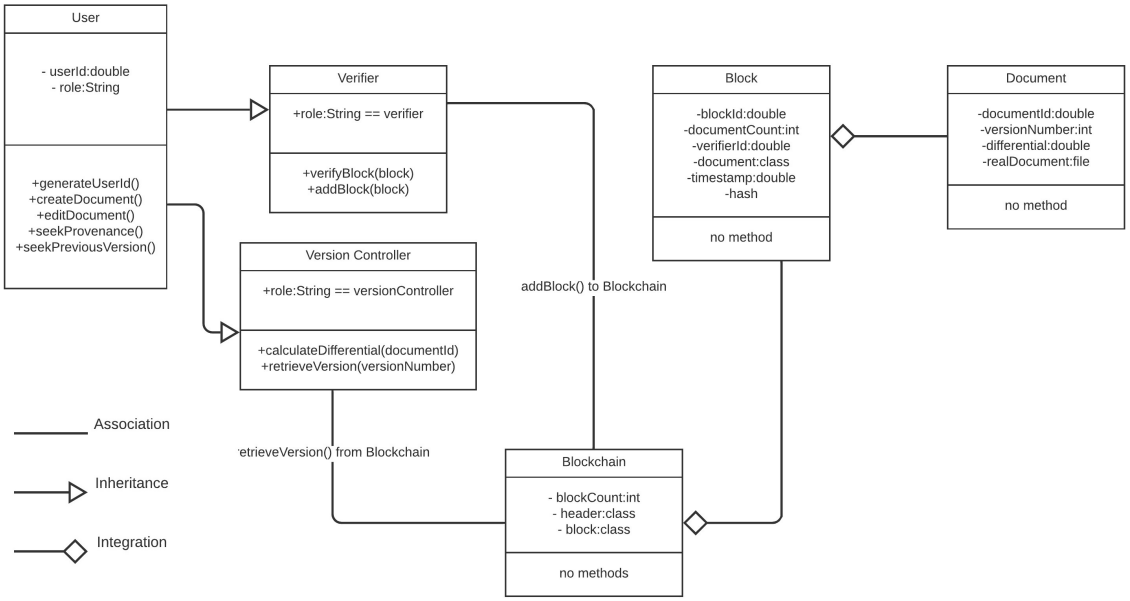



Figure 3.3 describes the Flakes architecture in classes.

### 3.3.1 User

The User class would be defined as follows:

```
public class User{
    double userId;
    String role;
    //constructor
    User(){
        userId = System.nanoTime();
        role = *role*;
    }
    public void createDocument(){
        Document d = new Document();
    }
```

```
public void editDocument(d){
    id = d.documentId;
    openFile(id);
    *edits*
    closeFile(id);
}
public Provenance seekProvenance(d){
    Provenance p = new Provenance();
    p.id = p.previous.id + 1;
    p.creator = d.author;
    for entry e::entryList{
        if e.documentId == d.documentId{
            p.version.next = e.version;
            p.user.next = e.userId;
            p.change = e.differential;
            p.time = e.timestamp;
            }
    }
    return p;
}
public File seekVersion(d,v){
    for entry e::entryList{
        if e.documentId == d.documentId
            docEntry[].next = e;
    }
    docEntry[] = docEntry[].reverse();
    for entry e::docEntry{
        diff[] = e.differential;
```

```
        }
        count = currentVersion - v;
        copy = copyFile(d);
        for i = count:1{
            openFile(copy);
            *subtract diff[i]*
            closeFile(copy);
        }
        return copy;
    }
}
```

### 3.3.2 Verifier

The verifier class would be defined as follows:

```
public class Verifier{
//constructor
    Verifier(){
    userId = System.nanoTime();
    role = 'verifier';
    }

    public boolean verifyBlock(entryList){
        boolean flag = 0;
        for entry e::entryList{
            if isValid(e.userId)
                flag = 1;
            if e.signature==user.signature
                flag = 1;
```

```
            if e.timestamp < currentTime
                flag = 1;
        }
        return flag;
    }

    public Block addBlock(){
        if flag == 1{
            blockId++;
            blockSize = entryCount();
            timestamp = System.time();
            diff=d.content−d.prev.content;
            currentHash = Hash(prevHash,
            blockId,blockSize,timestamp,
            diff);
            Block b = new Block(blockId,
            blockSize,timestamp,diff,
            currentHash);
            return b;
        }
    }
}
```

### 3.4 Evaluation Metrics

The system we are proposing integrates some of the functionality of Git (e.g. version control) and some of the functionality of Karma and Komadu (e.g. secure provenance ). We are taking the key features of each system and implementing them with blockchain, and we measure the difference in the performance of the proposed system and the original git system based on the

following metrics. We will define the metrics in this section and discuss how git fares in these fields.

1. **Processing Change**:

   - The Git *commit* command is used to save changes made to a file in a local repository. However, Git needs to be explicitly mentioned which changes are to be included in a commit before running the git commit command. This means that a file won't be automatically included in the next commit just because it was changed [5].

     In our system, every change made would be recorded. No room for mistakes and/or non-repudiation. Also, users wouldn't have to issue any command or send any notification each time they make a change. Reflection of any change would be automatic.

   - The commit command saves the changes indicated in the *local* repository, and not in the main remote server, or other remote branches. This decision is conscious, and merging can be done by several other commands [5].

     In our system, each server/user would have identical copies of the blockchain. Changes made in one would be reflected in others.

2. **Time Needed to Process a Change**:

   - Time to commit depends on the size of the change to be committed. Creation of a completely new file of large size takes more than smaller changes made to an existing file.

     Using blockchain improves the throughput of the system. The exact ratio can be determined only after the system is fully implemented.

3. **Handling Version Control**:

   - Git stores snapshots of the files every time they are committed, and stores these snapshots in a separate database [5].

Our system stores differentials in the blockchain. This not only relieves the system of the overhead of maintaining a separate database, a blockchain is more secure than an encrypted database.

- To fetch back a previous commit or state, one needs to look back into the log and call that version with the version number using the Git *Checkout* command. This removes the subsequent modifications to the file and puts the file back to the previous state. This action *cannot* be undone. Git assumes that the later versions are useless, and deletes it. To avoid complete deletion of the subsequent changes, one needs to create a duplicate branch and work on that [5].

  Our system does not delete any differential. Reverting back to a previous version means creating a new version with the contents of the specified version. For example, if the current version is $x$ then reverting to version $x$ - *7* means creating a new version $x$ + *1* with the contents of version $x$ - *7*. So all the previous versions are intact and it is possible to retrieve any one, say $x$ - *4* or even $x$.

4. **Security**:

   - Git uses SHA1. SHA1 is vulnerable to collision attacks. Our system uses SHA256 for hashing. It is secure and fast [5].
   - To verify at current time the state of a repository at an earlier date/time, the verifier would need the hash of that commit. However, a malicious user could edit the history of the repository and create a new commit with a new hash, and provide this new hash to the verifier [95]. The hashes will match, and the commit would be verified. This is a serious flaw in how git secures its history [96]. This wouldn't be possible if the verifier had the hash generated at the time of the original commit though. The malicious user would still have to prove the time, since the timestamp is included in the commit, but that can be controlled easily by altering the system clock. So git provides a good guarantee that a user had committed some particular text at what the user claims is a certain time [96]. For a guarantee for post-dated

data, we would need to involve an external system that has a trusted clock. We will be using blockchain to solve this precise problem.

5. **Storage of Data**:

   - **Size Restrictions**: Git restricts a single file over 100MB, and repositories over 1GB to be committed at a time [97]. Our system has no such restrictions.
   - **Redundancy**: Git stores the data in repositories. Each branch is stored in a separate repository. Also, the versions of the files are stored in a version database.

     In our system, there would be a *one* repository where all the files and directories would be stored.
   - **Dependencies**: Git handles dependencies using submodules. Submodules allows the user to keep a Git repository as a subdirectory of another Git repository. This lets another repository to be cloned into the original project and keep the commits separate [5].

     In our system we wouldn't need to handle such dependencies.

6. **Proof of Past Data Possession**: Git allows deletion of blobs, unwanted data, and even credentials from the repository history using the git-filter-branch. Also external processes like the BFG can be used to clean the repository [95]. On the other hand, deletion of any metadata is virtually impossible from the blockchain.

Table 3.1 summarizes the performance of Flakes in comparison to the other systems.

| | Karma / Komadu | Git | Scrybe | Flakes |
|---|---|---|---|---|
| Data Provenance | Yes | No | Yes | Yes |
| Version Control | No | Yes | No | Yes |
| Redundancy | N/A | Present | Present | Absent |
| Blockchain | None | None | Present | Present |
| Hash | None | SHA1 | SHA256 | SHA256 |
| PoPDP | Present, Not Secure | Present, Deletable | Present, Secure | Present, Secure |
| Processing Change | Manual | Manual | Manual | Automatic |
| Data Limit | None | 100MB File/ 1 GB Branch | None | None |
| Dependency | Present | Present | Present | Absent |

Table 3.1: Comparison of the Discussed Systems

## 3.5 Research Questions

In this section, we will discuss our hypothesis leading to our research questions, and find their answers.

### 3.5.1 **What are the attributes of an effective provenance system?**

An effective data provenance system should be able to ensure the following:

- The account of authorship of the data.
- The exhaustive chronological account of changes made to the data along with timestamp and identity of the editor.
- The secure preservation of the above information, not allowing a malicious entity to tamper with it.
- To be able to produce these information whenever asked for.

### 3.5.2 **What are the attributes of an effective version control system?**

An effective version control system should be able to ensure the following:

- Secure preservation of every version of data saved by the creator.
- To be able to produce any version from history when asked for.

## 3.6 Summary of the System

Flakes provides two crucial services, **Data Provenance** and **Version Control**. There are no existing systems that provide both. The systems that provide these services separately, have their limitations. In Flakes our goal is to overcome those limitations by using the Blockchain technology to our aid.

- **Flexibility**: The existing data provenance and version control systems provide services to text-based systems only. Flakes can be adapted to provide services to retail industries, utilities industries, and many other instances, where the data doesn't necessarily is in a text format. It could be graphs, images, transactions and even digitized maps. Since all files are necessarily a map of indices with respective values in them, only the differential calculation segment has to be altered to fit the requirement. For a text file, the index is position, and the values are ASCII numbers. For images, the indices are pixel numbers, and the values are the RGB values in the pixels. For videos, the indices are a couplet, i.e. frame number followed by pixel number. This is because a video is essentially a collection of images played over time. Audios are broken in samples, and each sample have some values associated with them. This way any data type can be represented in the index-values format in Flakes.
- **Integrity**: The backbone of Flakes is Blockchain. The differentials we need to fetch back versions, or the provenance history is stored in blocks in the blockchain, protected by hashes. Any attempt to change any information which has already been posted in the blockchain will result in altered and mismatched hashes all the way down the chain, and won't be possible. A non-blockchain based system relies on authorization for security. That means some entities are allowed to perform the edits, whereas some are not. This

leaves room for malicious users spoofing identities of the authorized entities and performing unauthorized actions. However, in Blockchain, since no one can possibly edit the history, unauthorised actions are out of question.

- **Storage Utilisation** Any existing closed system that contains files and allows multiple users to edit any of these files has a database where the files are stored. For existing systems that provide data provenance, there is a separate database for storing the metadata. The existing version control systems keep a database to store snapshots of the versions once committed. Flakes, however does none. Flakes has no overhead for controlling separate databases. This is achieved in the following manner:

  - The creation or the first write of a file means the differentials are calculated with respect to a blank file. So the differentials themselves are the content of the file. Any subsequent edit are also stored as differentials. All these differentials are stored in the blockchain. So there is no need to store the file physically anywhere. If we want the n-th version of the file, we simply merge the first n differentials and present it to the user. The user can then choose to save it. If we want the current version, the control adds all the differentials till the current version.
  - All the provenance information is also stored in the blockchain.
  - If we want the n-th version of the file, we simply merge the first n differentials and present it to the user. The user can then choose to save it.

  This way Flakes saves all the overheads of maintaining different databases and their securities. And since the blockchain is simultaneously maintained in all the nodes/servers, the chance of the blockchain crashing is very low.

# Chapter 4

# Use Cases

This chapter discusses some real life requirements which can be implemented using Flakes.

## 4.1 The Retail Apparel Tracking System

The RFID lab of the Auburn University is currently working with several retail companies like Macy's, Kohl's, and others to track the movement of their apparel shipments. These shipments originate from a factory, pass through several warehouses till they reach a showroom, or are delivered to an address. Each product comes with an RFID tag to uniquely identify them. Each tag is scanned whenever the product is moved. With each scan, the log file gets updated. The following fields are updated.

1. The product ID
2. The time, when the product was moved
3. The transaction ID
4. The status, i.e. "shipped" ,"in transit", "at warehouse" etc

With each update in the log file, the change gets posted in the blockchain.

Figure 4.2 shows the format of the log book as collected and stored by the RFID lab.

The following kinds of changes can happen in the log book:

- A product is shipped from a factory. A new row is created with the product's ID, the status is shown to be "in transit" . The factory's ID, and the time when it was shipped

| Transaction ID | EPC (Unique identifier for each product) | Log Upload Time | Business Step | Event Time | Disposition After Even | Record Time | Facility Location Number |
|---|---|---|---|---|---|---|---|
| 2481e57d669ea26da0cbd0d294affcce33c8b3e46a63115bd0df8ece4a97 | 30342C224005C600012D1372 | 2019-11-06T12:47:00.000000 | urn:epcglobal:cbv:bizstep:shipping | 2019-11-01T18:12:33.819800-04:00Z | urn:epcglobal:cbv:disp:delivered | 2019-11-01T18:12:33.819800-04:00Z | urn:epc:id:sgln:0883661.00001.1 |
| dcfb3b07bb7881b5791a33550057f6b13e40bde8e0cb4e530cd406542b0cb5ea | 30342C224005C600012D13A7 | 2019-11-06T14:45:05.000000 | urn:epcglobal:cbv:bizstep:shipping | 2019-11-01T18:12:33.819800-04:00Z | urn:epcglobal:cbv:disp:delivered | 2019-11-01T18:12:33.819800-04:00Z | urn:epc:id:sgln:0883661.00001.1 |
| fc55ad8df6c36c9b0ed67ae47c7abadef0daa1f34bdaae700cede0ef5f078e23 | 30342C224005C600012D13B2 | 2019-11-06T14:02:45.000000 | urn:epcglobal:cbv:bizstep:shipping | 2019-11-01T18:12:33.819800-04:00Z | urn:epcglobal:cbv:disp:delivered | 2019-11-01T18:12:33.819800-04:00Z | urn:epc:id:sgln:0883661.00001.1 |
| dab2e12d5002898b34617135ffb4aa31ed988902f439121e63e6f47252a42424 | 30342C224005C640012D1434 | 2019-11-06T14:44:37.000000 | urn:epcglobal:cbv:bizstep:shipping | 2019-11-01T18:12:33.819800-04:00Z | urn:epcglobal:cbv:disp:delivered | 2019-11-01T18:12:33.819800-04:00Z | urn:epc:id:sgln:0883661.00001.1 |
| 37d767b81f271a308a232d4c7572ced46906f440459497b795fd138da067a5ed | 30342C224005C640012D147D | 2019-11-06T14:49:57.000000 | urn:epcglobal:cbv:bizstep:shipping | 2019-11-01T18:12:33.819800-04:00Z | urn:epcglobal:cbv:disp:delivered | 2019-11-01T18:12:33.819800-04:00Z | urn:epc:id:sgln:0883661.00001.1 |
| af73211320f498ab5ee922684056d8cf3348765ae228ed3b3e7c9f86a63ff0ba | 30342C224005C640012D1517 | 2019-11-06T14:47:58.000000 | urn:epcglobal:cbv:bizstep:shipping | 2019-11-01T18:12:33.819800-04:00Z | urn:epcglobal:cbv:disp:delivered | 2019-11-01T18:12:33.819800-04:00Z | urn:epc:id:sgln:0883661.00001.1 |
| 6fdba632bea1f86eca4005726b477eff8f45a47ddf013935ed2144491f8ad985 | 30342C224005C640012D1555 | 2019-11-06T14:49:01.000000 | urn:epcglobal:cbv:bizstep:shipping | 2019-11-01T18:12:33.819800-04:00Z | urn:epcglobal:cbv:disp:delivered | 2019-11-01T18:12:33.819800-04:00Z | urn:epc:id:sgln:0883661.00001.1 |
| 346748ec33c63e06f820dca97a7af45af6f269062697a12b6404b1a57a3eff6d | 30342C224005C640012D1568 | 2019-11-06T13:59:56.000000 | urn:epcglobal:cbv:bizstep:shipping | 2019-11-01T18:12:33.819800-04:00Z | urn:epcglobal:cbv:disp:delivered | 2019-11-01T18:12:33.819800-04:00Z | urn:epc:id:sgln:0883661.00001.1 |

Figure 4.1: The RFID Log Book

are noted. The ID of the vehicle is also noted. The number of products is always noted at every step.

- The same product arrives in a warehouse. The status is changed to “arrived at warehouse” . The time of arrival and warehouse’s location ID noted.
- An order is placed for the product. The transaction ID gets updated.
- The specified product gets shipped with multiple others in a combined shipment. The status is changed to “in transit”. The shipment ID is noted. The vehicle ID is updated.

The Tracking system faces the following problems:

- Theft / Fraud: Where the malicious entity changes values in the log to suit their need. For example, the sender of a shipment correctly updated the log to note that there were 50 products in the shipment. The receiver changes the number to 30, indicating that only 30 products were shipped, and they received all of them, thus stealing 20 products. In the retail industry, unaccounted for inventory as a result of theft or fraud is called Shrink, and resulted in a loss of $47 Billion in 2017.
- Misplacement: Claims and Chargebacks occur when shipments are damaged, lost, or inaccurate. In 2017, the retail industry suffered a loss of $36 Billion due to Claims.

The tracking system needs a way to secure the log, i.e. to prevent any alteration in the data already entered in the log. If any erroneous data was entered at any point, it can be


Figure 4.2: Monetary Loss account, data and image courtesy the RFID lab of Auburn University

corrected by entering a fresh set of updated data in the log. The last updated data about a product/shipment/transaction will be considered. The tracking system needs a way to ensure that the log is not corrupted, or deleted, incurring loss of data. This tracking system can benefit from Flakes in the following ways:

- Any update in the log will reflect in the Flakes blockchain. These updates will be immutable, and permanently stored in the Flakes blockchain. So when a malicious entity changes any value in the log, that change will also be recorded. In case of a dispute, the authorities can derive provenance information from the blockchain and prove who changed what data at what time. The data provenance service of Flakes would provide this solution.

- In case the log gets corrupted, i.e. part of its contents become unavailable, or the log gets deleted entirely, the entire log could be re-created from the differentials stored in the blockchain. The version control service of Flakes would provide this solution.

The data flow in the RFID Tracking System using Flakes works in the following way.

1. The retail company/ brand updates the RFID log book.

2. The changes in the log book reflects in the Flakes system. The time of change, identity of the editor, and differentials calculated.

3. The entry created gets posted in the blockchain, along with chained hashes.

4. When provenance information or a previous version is called, the information retrieved from the blockchain.

5. The information is compiled into provenance modules and versions.

6. The provenance information and required versions are returned to the user who sought it.

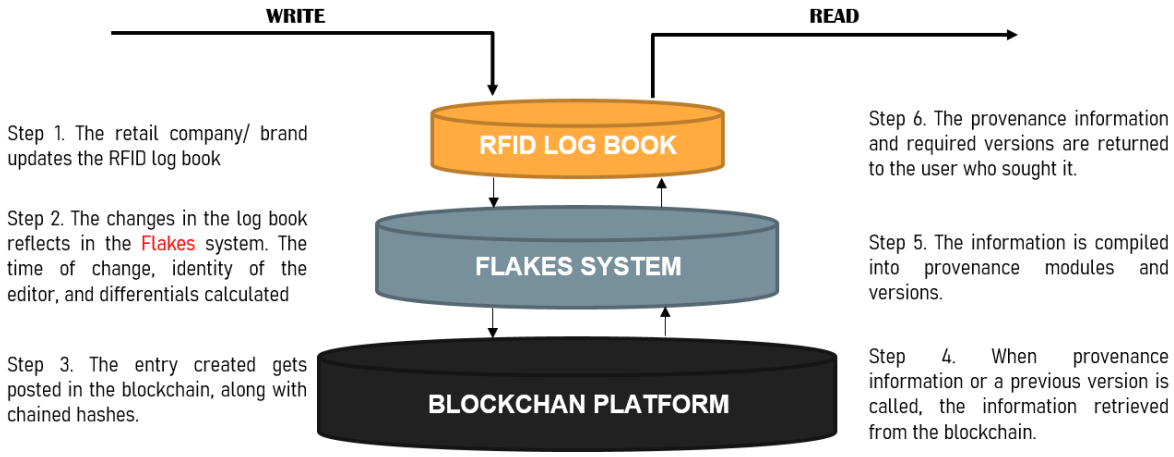



Figure 4.3: Data Flow in the System

Figure 4.3 shows the data flow in this use case.

## 4.2 The Operating Systems Use Case

Developers often use open source operating systems like Arch Linux, and reconfigure them as required. Software developers, including large software projects and operating system developers face multiple challenges when it comes to keeping tracks of changes. Some of these challenges include:

- Keeping track of who made what changes. The teams incur wastage in time and money due to delays in gathering this information. Without proper documentation, companies broadcast emails to large teams and wait for response.

- Mutability in their architecture means changes made in the system are permanent. If the result of these changes are unsatisfactory, then undoing the changes pose as a problem.

- Manual errors in saving changes, and inadvertent deletion causes loss of data/code.
- System malfunctioning can cause loss of data/code.
- Malicious external softwares can tamper/delete useful data/code.

Fedora developed an operating system called Silverblue, in October 2020. Silverblue is an immutable operating system, which means changes made in this system are not irreversible. So when developers use the Silverblue platform to customize operations, they can make changes in the system and configuration files, and decide to keep them, or revert back to a previous version. While Silverblue provides immutability, it does not provide data provenance, i.e. when a large group of developers work on Silverblue, there is no way to track who made what changes.

Sliverblue handles immutability as follows:

- The developers edit the source code by writing in the system/config files.
- Once they are ready, they force an update by the $ rpm – ostree upgrade command.
- ostree is similar to a version control technology but it operates on the entire filesystem trees.
- So whenever the OS is upgraded using ostree, the command creates a new branch, and stores the OS image file in that branch.
- When a package is installed with rpm-ostree, a new OS image is composed by adding the RPM payload to the existing OS image, and creating a new, combined image.
- RPM is redhat package manager, built on ostree.
- Previous versions of the OS can be retrieved, using the rpm-ostree rollback command.

Silverblue encounters the following problems in their immutability handling.

- Silverblue doesn't track the identity of the person/s making the changes.
- It doesn't track what changes were made.

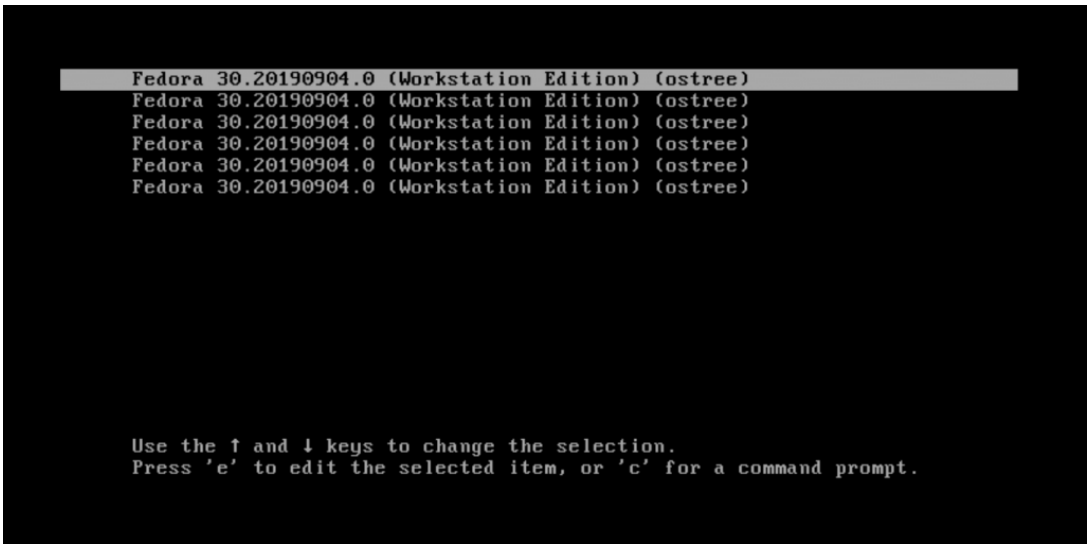



Figure 4.4: Choosing a previous version of the OS image in Fedora

- ostree saves an image of the whole OS at each upgrade which is 1.6 GB if x86 instruction set is used.
- The user has to manually pin a version to be able to access it later.
- Previous versions of individual files cannot be retrieved without undoing the changes made to other files.
- Temporary updates keep the subsequent versions, but permanent updates are one-way and subsequent versions cannot be retrieved.

Using Flakes to handle immutability in Silverblue has the following benefits:

- Flakes is able to keep track of the identity of the developer who makes the change.
- Flakes is able to keep track of what changes were made to which file.
- Flakes is able to retrieve previous version of individual files without disturbing other files.
- Flakes stores the changes in differentials, rather than images of the whole system. The size of these changes depend on the amount of changes made, and will never exceed the size of the image of the whole OS.
- Deletion of files can be undone by adding differentials, and re-creating the file.

- If an external malicious program disrupts the OS by altering system files, the previous working versions can be easily retrieved.

The UML diagram of this use case is shown in figure 4.5.

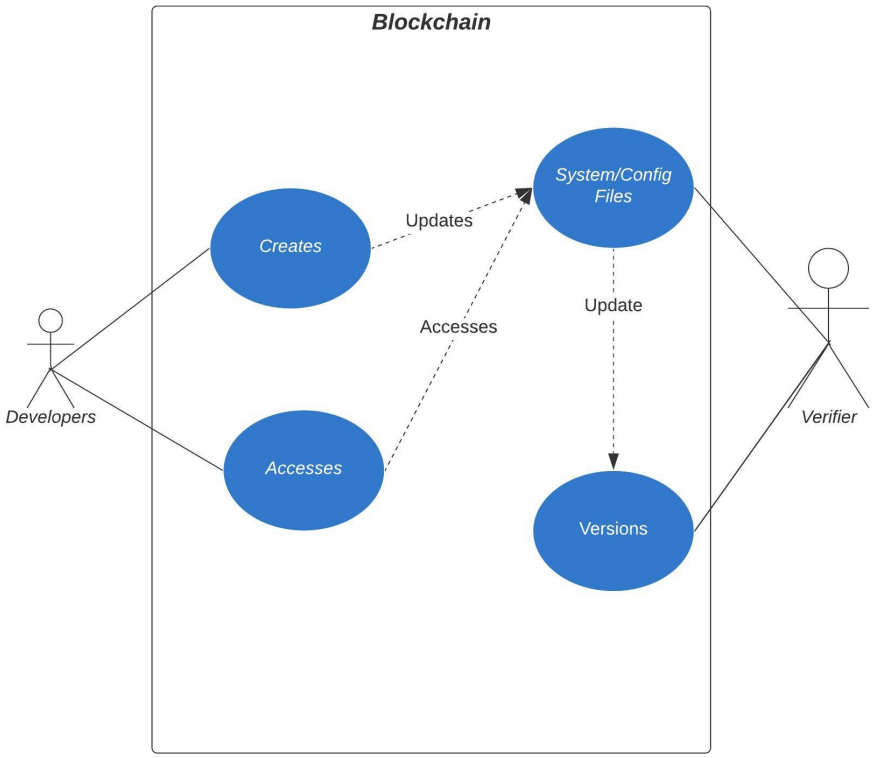



Figure 4.5: The Operating Systems Use Case

## 4.3 Utilities Manager

The three utility services that we have throughout the United States, i.e. the gas lines, electricity, and water lines, do not currently share a common database everywhere. This causes frequent collisions in mapping. For example, the Water Works Board of a town may make a new branch from the main line to a new building and devise a route only to find out it had previously been assigned or acquired by a gas line. The Flakes Blockchain provides an effective and secure solution to this problem. For Flakes to be implemented, we would need to digitized the pipeline grid maps of the town. The sources and the service points, where the utilities are delivered can

be considered as nodes of a graph, and the pipelines can be considered as edges. Only one edge can exist between each source and each service point.

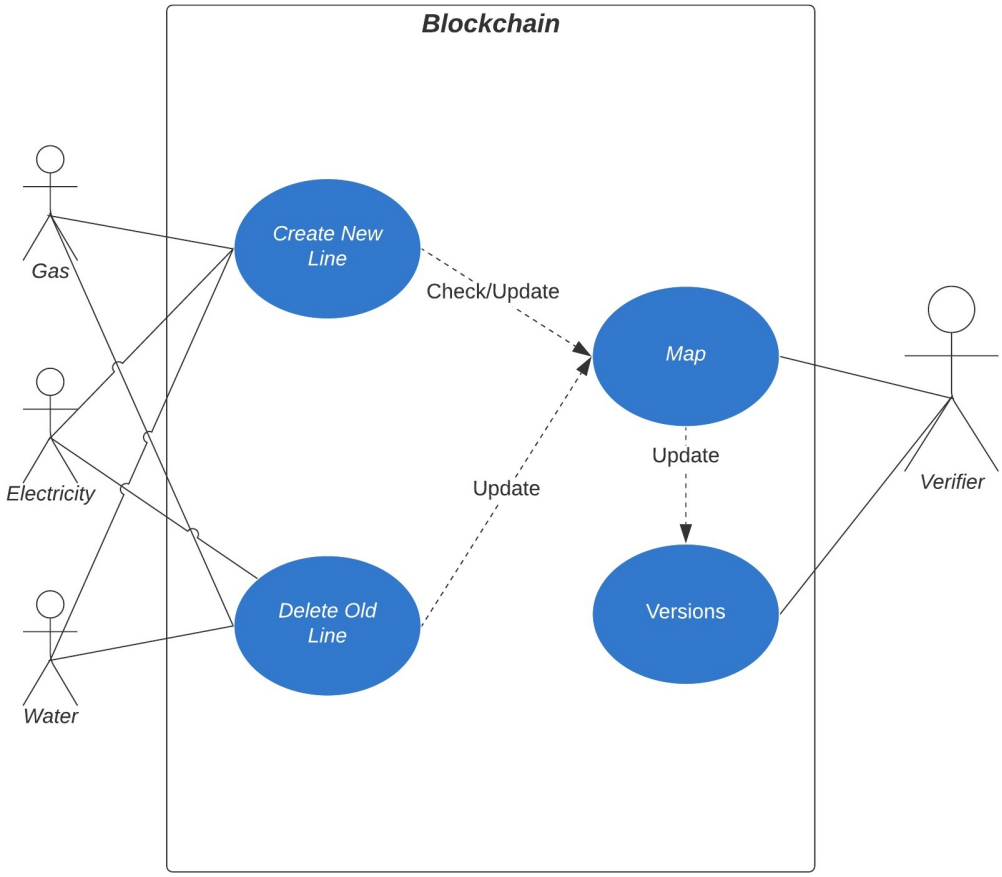



Figure 4.6: The Utilities Manager Use Case

The distributed system would have three sets of users, the Gas, the Electricity, the Water. Each user can perform the following actions:

- Create a new utility line.
- Remove an existing line.
- Modify the route of an existing line.
- Bring a new service point into the map.
- Run a simulation for new layouts, and then decide to implement it, or reject it.

In this use case, the maps would be converted to graphs. For every city, the area code is the PURL for the file. Which is also the primary key, as per Flakes. The content of each file is the map in the format

$$E(N_i, N_j) = \{e, g, w\}$$

. Here,

$$N_s = Nodes\ and\ Es = Edges$$

0 means no edge between those two nodes, and e,g, or w means an edge exists and it belongs to electricity, gas, or water, respectively. Each household/apartment complex/factory/junction would be a node, and edges would be a line (water, gas or electricity). So it will be a matrix. Every time a utility board makes a change, i.e. changes a 0 to a letter, or a letter to a 0, the ID of the board will be included.

Here is an example:

PURL : 36849 (Zip code for Auburn University)

| | N0 | N1 | N2 | N3 | N4 |
|---|---|---|---|---|---|
| N0 | - | 0 | 0 | e | w |
| N1 | | - | 0 | g | 0 |
| N2 | | | - | w | g,e |
| N3 | | | | - | 0 |

Table 4.1: Example of a Utilities Map

Table 4.1 shows an example of the format of the contents of a file in the Utilities use case. This will be a text based format, and entirely applicable using Flakes current version.

Whenever a new line is created, the action is posted in the corresponding document by updating the edge between the concerned nodes. Removal of an existing line is accordingly posted in the document, and the concerned edge is removed. Addition of a service point means addition of a new node in the map. Thus any change in the map corresponds to a change in the document, and any change in the document is posted in the blockchain. The blockchain maintains a secure and chronological order of the log of events and actions that take place. Figure 4.7 shows the workflow of the use case.

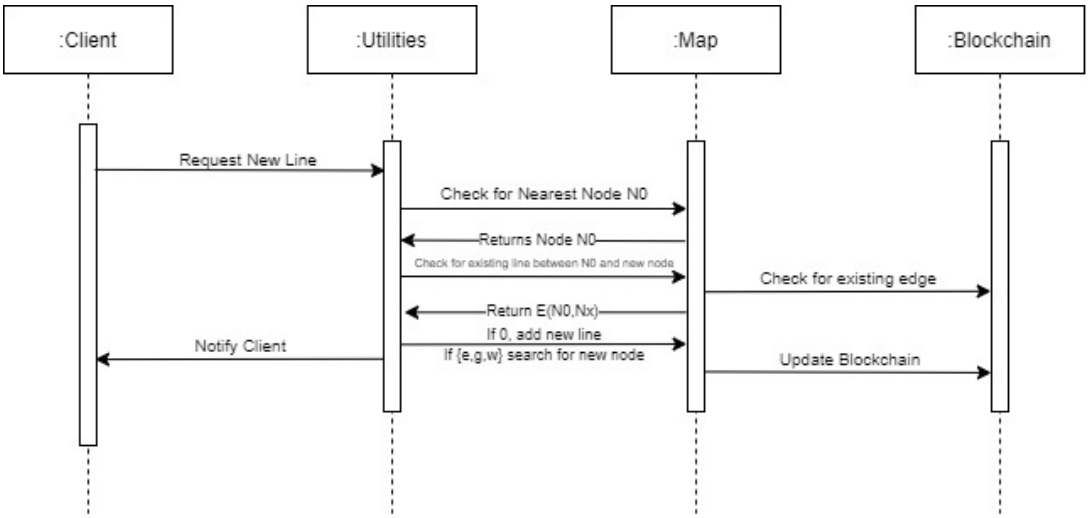



Figure 4.7: The Workflow Diagram

### 4.3.1 Efficiency

Here we will discuss how the use of Flakes can improve the performance of the management.

- Currently, the utility companies do not communicate with each other in a distributed shared network. Flakes would bring transparency in their communication.
- Using Flakes, these companies can avoid collision. Very often a utility company digs up to install a line only to find an existing line from another company. Flakes would completely solve that problem.
- Entire history of the activities of the companies would be preserved, for research and conflict resolution purposes.
- These utility providers can run simulations on the system by creating virtual lines and estimating the throughput.
- The companies can actually plan the networks before implementing them using Flakes. In this process, if they want to revert back to a previous, more efficient version, they can use the version control feature of the Flakes Blockchain.

## 4.4 Experiment Parameters Manager

In different experimental studies, such as in Chemistry or Biology, often there are experiments where there are huge numbers of parameters which need to be altered to find a combination which provides the optimum result. Flakes can be used as a provenance collection and version control device in those experiments. Figure 4.8 shows the architecture of this use case.

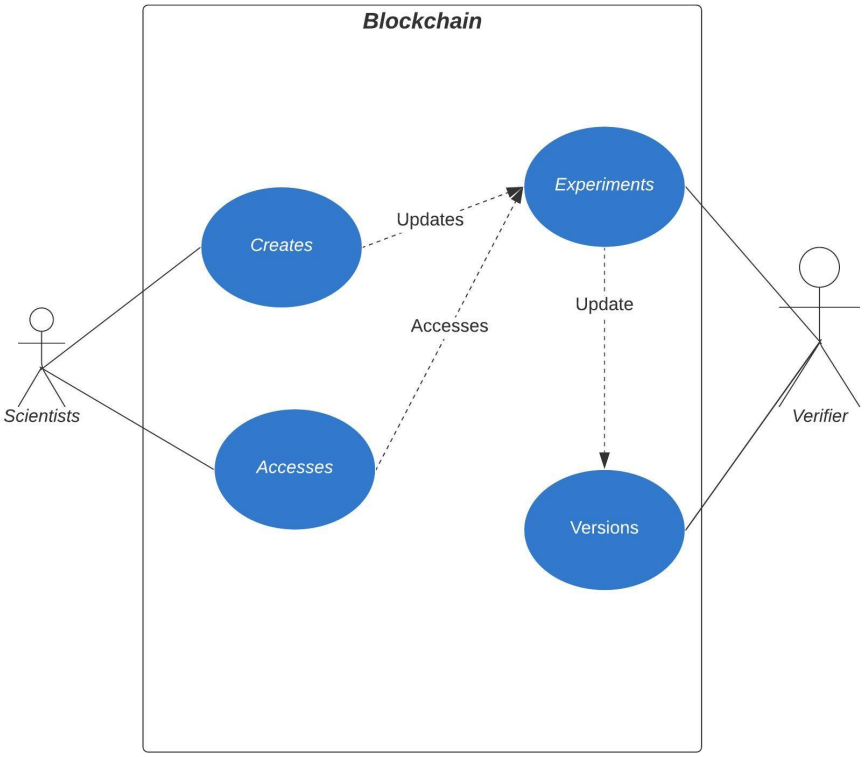



Figure 4.8: The Experiment Parameter Manager Use Case

As an example, let's say there is a drug combination response experiment. There are n number of drugs, each with varying dosages. Each distinct dosage combination is a version. After conducting a trial of the experiment with a set of dosages, the dosage values will be updated in the document. After completion of the trial, the success percentage will also be recorded. All the changes made to the document will be recorded in the Flakes blockchain.

| Drugs | D1 | D2 | D3 | D4 | Result |
|---|---|---|---|---|---|
| Dosage 6 | 2 | 18 | 0 | 22 | 14% |

Table 4.2: Example of a Dosage Combination of a Drugs Experiment

Table 4.2 shows an example of the format in which details of a trial will be recorded in Flakes.

### 4.4.1 Efficiency

In this section we will discuss how the use of Flakes makes record keeping of the drug trials more efficient.

- Often, when multiple scientists work on an experiment, preservation of history of each scientist's action is necessary to avoid confusion and for accountability. Flakes blockchain will secure the ID of the user and times at which they conducted a particular trial.

- Since these are optimization processes, tracking changes is essential for backtracking. Say Dosage (n-7) had a 90% success, but to improve upon it, the scientists changes a few dosages, but the result was not satisfactory. They want to go back to the dosage with the maximum success. Each time a dosage of a drug is altered, the file is edited, and the change is locked in blockchain with respective version numbers. This way with Flakes version control, the scientists can easily revert back to the required version/dosage number.

A similar idea works with Gene Expression Analysis. Scientists vary different DNA/RNA sequences to achieve the desired result. Each sequence would be a new version.

In both these cases, provenance will keep track of who made the changes, what changes were made, and at what time. Version control will allow reverting back to any previous version, without losing any subsequent data.

## 4.5 Assignment Manager

The Flakes architecture can be conveniently used as a provenance tool in an assignment/home-work/exams management in schools and universities. This section provides a detailed illustration on how Flakes can facilitate an efficient Assignment Manager.

An Assignment Manager would have two groups of users,

1. The Instructors.
2. The Students.

The users can perform the following actions:

1. The Instructors can create exams, assignments, and home-works. For our convenience, I will refer to each of these as assignments.
2. The Instructors can set time limits and availability of these assignments.
3. The Students can access the assignments within the allotted time limits.
4. The Students can submit more than one versions of any particular assignment, i.e., they can make changes in an already submitted assignment, delete a previous version of an assignment, or submit a whole new one.
5. The Instructors can access and review each submission made by the student for a specific assignment.

The Flakes architecture can be used to implement such an Assignment Manager.

1. The assignments and the submissions will be treated as individual files in the system, and the introduction, deletion, or modification to any file will be recorded in the Blockchain with a timestamp and the author's digital signature, and the version number.
2. Since there are timestamps assigned to each assignment, and consequent submission, restricting the availability of each assignment can be automated. Also, the time of submission for each submission will also be recorded, allowing the instructors to check whether the submissions were late or on time.

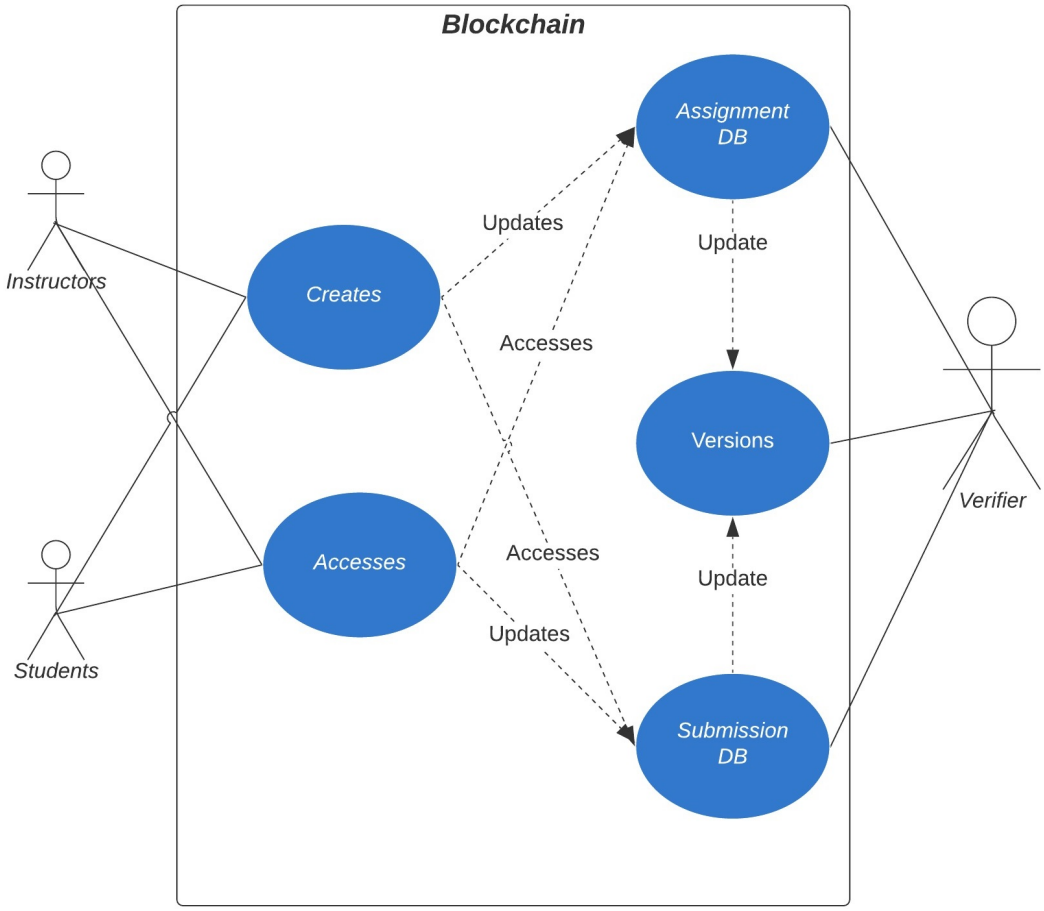



Figure 4.9: The Assignment Manager Use Case

3. Since each assignment/submission will be stored with a version number, a student can make changes freely in the assignments, and revert back to a previous version if needed. The instructors can also review each submission, and accept the last submission, or the best one, according their policy.

# Chapter 5

## Implementation and Discussion

Flakes is a system that provides data provenance and version control as services. Flakes uses blockchain as a tool to provide that service. In a closed system, Flakes has multiple authenticated users who can access multiple files. The users can create a file, edit it, or delete it. They can ask for provenance information on any file, or seek an earlier version of a file.

Whenever a change to the system is made, the change is verified by the servers, and are posted in the blockchain. The Flakes blockchain is a distributed ledger, public to all the users of the system. The metadata of the changes made to the system are collected into a data structure called Entries, which contains information like the signature of the author, the timestamp when the changes were committed, and the differentials marking the changes. Multiple verified entries are then collected into a data structure called a block. While posting a block in the blockchain, a hash is calculated, whose arguments contain all the elements of the block and the hash of the previous block. This cumulative hashing creates a cryptographically linked chain, which continues from the genesis block to the current block, making it impossible for a malicious user to tamper with any data inside the blockchain.

When a new file is introduced to the system, and first changes submitted, the differential is calculated against a blank file which is essentially the contents of the file. Any further edits are differentials with respect to the previous state. So the contents of the file are stored in various stages in the blockchain under the same PURL primary key and a unique version number. Therefore there is no need to physically store the file anywhere, reducing maintenance overhead. Whenever a version of a file is required, a blank file is assumed, and all the differentials till that version are added to get the final state.

The Flakes architecture has been implemented using Java. In this architecture, each document and its meta-information is stored in a file class. Each changes made to the document, including creation, edit and deletion, along with the metadata is stored in an Entry class. Once an Entry is verified, multiple validated Entries can be stored in the Block class. And when a Block is formed, it is uploaded to the Blockchain along with the chained hash. The Differential class has methods to calculate the differentials which we use to roll back to previous versions. The provenance class returns the history of a file when called.

## 5.1 File

The file class has in-built methods to calculate the differentials each time an edit is done, and to roll back to a previous version by subtracting the differentials, each time it is called for.

Figure 5.1 shows the calculate differential method, which calculates the change in values at each index where a change has been made. When a file is first created, all consecutive indices, starting from the first index, are changed. When a file is deleted, all the indices are re-set. So even after a file ceases to exist, its contents are always stored securely in the Blockchain.

```
@Override
public IDifferential calculateDifferential(IFile newFile){
    var sbCurrent = new StringBuilder(this.fileContent);
    var sbNew = new StringBuilder(newFile.getContent());
    var diffHashmap = new HashMap<Integer, Integer>();

    sbCurrent.setLength(newFile.size());

    for (int i = 0; i < sbNew.length(); i++){
        var charDiff = sbNew.charAt(i) - sbCurrent.charAt(i);
        if (charDiff != 0){
            diffHashmap.put(i, charDiff);
        }
    }
    return new Differential(diffHashmap, newFile.size() - this.size());
}
```

Figure 5.1: The Calculate Differential Method

The roll-back method, as shown in figure 5.2, subtracts the differential from all the indices where a change has occurred.

```
class File implements interfaces.IFile {
    private String pURL;
    private String fileContent;


    @Override
    public IFile rollback(IDifferential diff){
        var sb = new StringBuilder(fileContent);
        var diffMap = diff.getDifferentialMap();
        var fSizeDiff = diff.getFileSizeDiff();
        var newSize = this.size() - fSizeDiff;
        sb.setLength(newSize);
        for (var x : diff.allIndices()){
            sb.setCharAt(x, (char)(sb.charAt(x) - diff.get(x)));
        }
        return new File(this.pURL, sb.toString());
    }
```

Figure 5.2: The Roll Back Method

## 5.2 Entry

Whenever a file is edited, the purl of the file, the differentials generated, the identity of the user and timestamp are collected to form an entry. Figure 5.3 shows the constructor to the Entry class.

```
public Entry(Long entryId, Long userId, Long versionNumber,
             IDifferential differential, String linkedFile,
             Long userSignature) {
    this.entryId = entryId;
    this.userId = userId;
    this.versionNumber = versionNumber;
    this.differential = differential;
    this.linkedFilePURL = linkedFile;
    this.userSignature = userSignature;

    this.timeStamp = System.currentTimeMillis();

    recalculateHash();
}
```

Figure 5.3: Contents of an Entry

After all the fields are filled, the method to calculate hash is called. It concatenates all the fields and calculates the SHA256 hash of the contents, as shown in figure 5.4.

```
@Override
public void recalculateHash() {
    var strToHash = new String();
    var diffsHashCode =  differential.getDifferentialMap().entrySet()
            .stream().map(x -> x.getKey().toString() + x.getValue().toString())
            .collect(Collectors.joining());

    strToHash.concat(String.valueOf(entryId));
    strToHash.concat(String.valueOf(userId));
    strToHash.concat(String.valueOf(versionNumber));
    strToHash.concat(String.valueOf(userSignature));
    strToHash.concat(String.valueOf(timeStamp));
    strToHash.concat(diffsHashCode);

    try {
        this.hash = Commands.getSHA256Hash(strToHash);
    } catch (Exception ex){
        //HANDLE EXCEPTION HERE
    }
}
```

Figure 5.4: Calculating Hash

## 5.3 Block

At least one, or more than one entries are verified, and then stored in a block to be introduced into the blockchain. Figure 5.5 shows the contents of a block. It contains the height or ID of the block, the ID and the signature of the verifier, the list of entries, the timestamp of when the block was creates, and a hash of all these information and the previous block's hash combined.

```
public Block(int blockId, Long verifierId, ArrayList<IEntry> entries) {
    this.blockId = blockId;
    this.verifierId = verifierId;
    this.entries = entries;

    this.timestamp = System.currentTimeMillis();

    recalculateHash();
}
```

Figure 5.5: Contents of a Block

Before the block is uploaded to the blockchain, a hash is taken with all these contents and the hash of the previous block, to create the literal chronological chain, as shown in figure 5.6.

```
class Block implements interfaces.IBlock {
    private int blockId;
    private Long verifierId;
    private ArrayList<IEntry> entries;
    private Long timestamp;
    private byte[] hash;

    @Override
    public void recalculateHash() {
        var strToHash = new String();
        var entriesHashCode =  entries.stream().map(x -> x.getHash().toString())
                .collect(Collectors.joining());
        strToHash.concat(String.valueOf(blockId));
        strToHash.concat(String.valueOf(verifierId));
        strToHash.concat(String.valueOf(timestamp));
        strToHash.concat(String.valueOf(entriesHashCode));

        try {
            this.hash = Commands.getSHA256Hash(strToHash);
        } catch (Exception ex){
            //HANDLE EXCEPTION HERE
        }
    }
```

Figure 5.6: Hash Calculation in a Block

## 5.4 Version Handling

When a new version is called, the differentials would have to be merged with the current version of the file to retrieve the version sought for, as shown in figure 5.7. This is achieved by traversing the file by indices, and for each index, adding all the corresponding differentials from the first version to the version asked for.

```
public IDifferential mergeWith(IDifferential diff){
    var mergedDiff = new HashMap<Integer, Integer>();

    var newSizeDiff = this.getFileSizeDiff() + diff.getFileSizeDiff();


    for (var i : this.allIndices()) {
        mergedDiff.put(i, this.get(i));
    }

    for (var i : diff.allIndices()) {
        mergedDiff.put(i, mergedDiff.getOrDefault(i, 0) + diff.get(i));
    }
    return new Differential(mergedDiff, newSizeDiff);
}
```

Figure 5.7: The merge method

## 5.5 Provenance Handling

When the provenance history of a file is asked to be shown, the control traverses the blockchain for all the blocks that have entries that contains information about that file. The Provenance method returns all the pertinent entries along with the version numbers and timestamps as shown in figure 5.8

```
class Provenance implements interfaces.IProvenance {
    private Long creator;
    private ArrayList<Long> versions;
    private ArrayList<Long> userIds;
    private ArrayList<Long> timestamps;
    private ArrayList<IDifferential> changes;

    @Override
    public void addEntry(Long version, Long userId, Long timestamp, IDifferential change){
        this.versions.add(version);
        this.userIds.add(userId);
        this.timestamps.add(timestamp);
        this.changes.add(change);
    }

    @Override
    public void addEntry(IEntry entry){
        this.versions.add(entry.getVersionNumber());
        this.userIds.add(entry.getUserId());
        this.timestamps.add(entry.getTimeStamp());
        this.changes.add(entry.getDifferential());
    }

    public Provenance(Long creator) {
        this.creator = creator;

        this.versions = new ArrayList<Long>();
        this.userIds = new ArrayList<Long>();
        this.timestamps = new ArrayList<Long>();
        this.changes = new ArrayList<IDifferential>();
    }
```

Figure 5.8: The Provenance Class

## 5.6 Blockchain

The blockchain class adds a new block whenever it is created, and always shows the updated chain. So the blockchain is essentially an immutable data structure, which can be thought of an array of blocks, with the most recent array on top.

```
class Blockchain implements interfaces.IBlockchain {
    private ArrayList<IBlock> blocks;
    private int nextBlockId = 0;

    @Override
    public void addBlock(IBlock newBlock){
        this.blocks.add(newBlock);
        nextBlockId++;
    }

    //SINGLETON PATTERN
    private static final IBlockchain instance = new Blockchain();
    static IBlockchain getInstance() {
        return Blockchain.instance;
    }
    private Blockchain() {};
    ///

    @Override
    public ArrayList<IBlock> getBlocks() {
        return blocks;
    }

    @Override
    public int getNextBlockId() {
        return nextBlockId;
    }
```

Figure 5.9: The Blockchain Class

## 5.7 User Actions

A Flakes user can ask for provenance information on a file or the system, ask for previous versions of a file, or add a block in the blockchain. The user can also create, edit or delete a file. The methods are shown in figures 5.10 and 5.11.

```
public class UserActions {
    private final IBlockchain blockchain;
    private User user;

    public UserActions(User user) {
        this.user = user;
        this.blockchain = Blockchain.getInstance();
    }

    private ArrayList<IEntry> getAllEntries() {
        var blocks = this.blockchain.getBlocks();
        ArrayList<IEntry> entries = new ArrayList<IEntry>();

        for (var block : blocks) {
            entries.addAll(block.getEntries());
        }

        entries.sort(Comparator.comparingLong(IEntry::getVersionNumber));
        return entries;
    }

    public IProvenance seekProvenance(IEntry inputEntry) {
        IProvenance provenance = new Provenance(this.user.getUserId());

        var matchingEntries = getAllEntries().stream()
                .filter(x -> x.getLinkedFilePURL() == inputEntry.getLinkedFilePURL());

        matchingEntries.forEach(x -> provenance.addEntry(x));

        return provenance;
    }
```

Figure 5.10: Seek Provenance

```
public void addBlock(ArrayList<IEntry> unfilteredEntries) {

    Function<IEntry, Boolean> verificationConditions = (IEntry x) ->
                    ((isValid(x.getUserId())) &&
                    (x.getUserSignature() == this.user.getSignature()) &&
                    (x.getTimeStamp() < System.currentTimeMillis()));

    var filteredEntries = unfilteredEntries.stream()
            .filter(e -> verificationConditions.apply(e))
            .collect(Collectors.toCollection(ArrayList<IEntry>::new));

    var newBlockId = this.blockchain.getNextBlockId();
    var verifierId = this.user.getUserId();

    var newBlock = new Block(newBlockId, verifierId, filteredEntries);

    this.blockchain.addBlock(newBlock);
}
```

Figure 5.11: Add Block

## 5.8 Front End

In this section, we will take a cursory look at how Flakes would appear to a user. The front end was developed using websockets and is available at *https://lazy-alpaca-98436.herokuapp.com/*.

The following figures show each action as seen by the user.

- A user needs to sign in using their User ID and a password, which would be used as their digital signature as shown in figure 5.12.
- Once logged in, the user can view the existing files in the system as shown in figure 5.13.
- The user can choose to create a new file, or edit an existing file as shown in figures 5.14 and 5.15.
- The user can view the current state of the blockchain as shown in figure 5.16.
- The user may want to check the provenance information of a file as shown in figures 5.17 and 5.18.
- The user may want to retrieve any previous version of the file as shown in figures 5.19, 5.20 and 5.21.

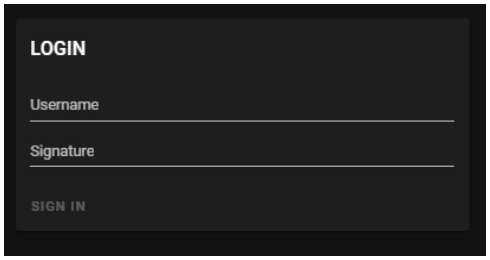



Figure 5.12: Credentials

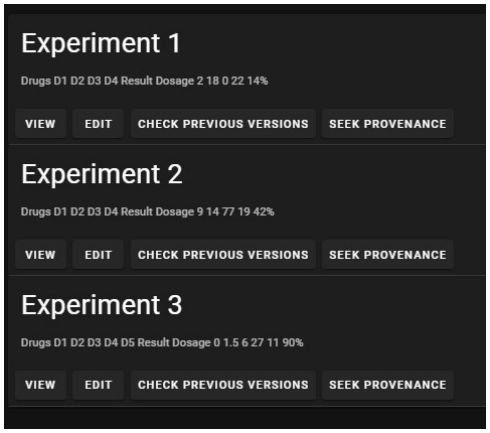



Figure 5.13: Existing Files

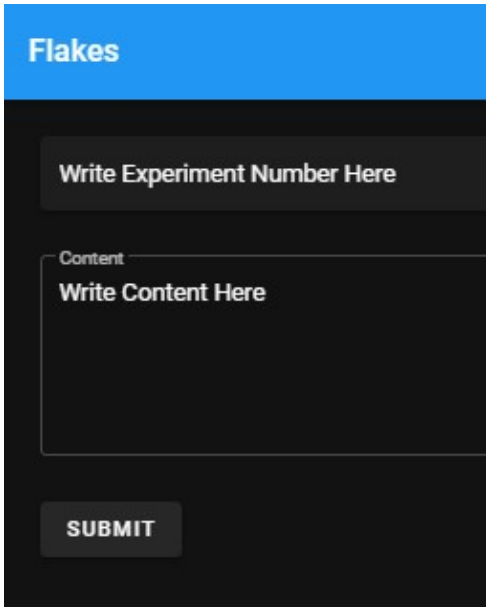



Figure 5.14: Create a new file

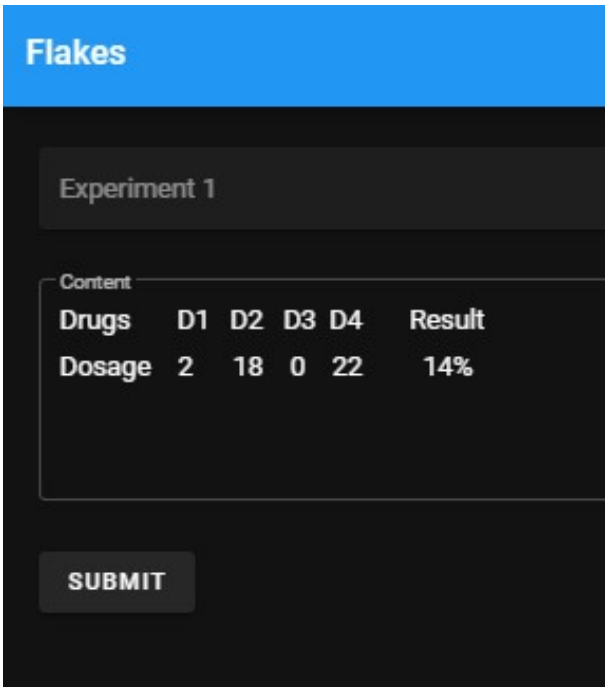



Figure 5.15: Edit an existing file

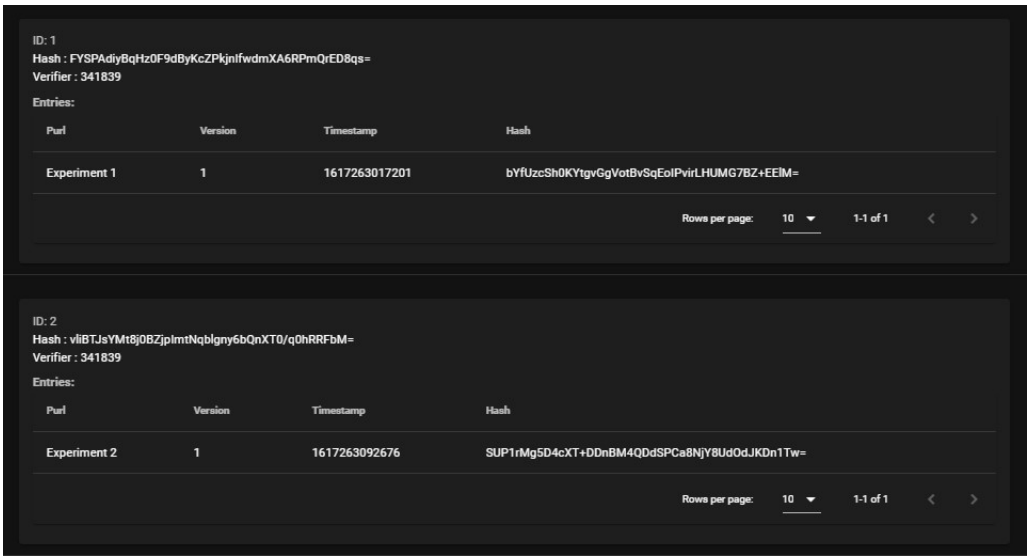



Figure 5.16: The Current Blockchain

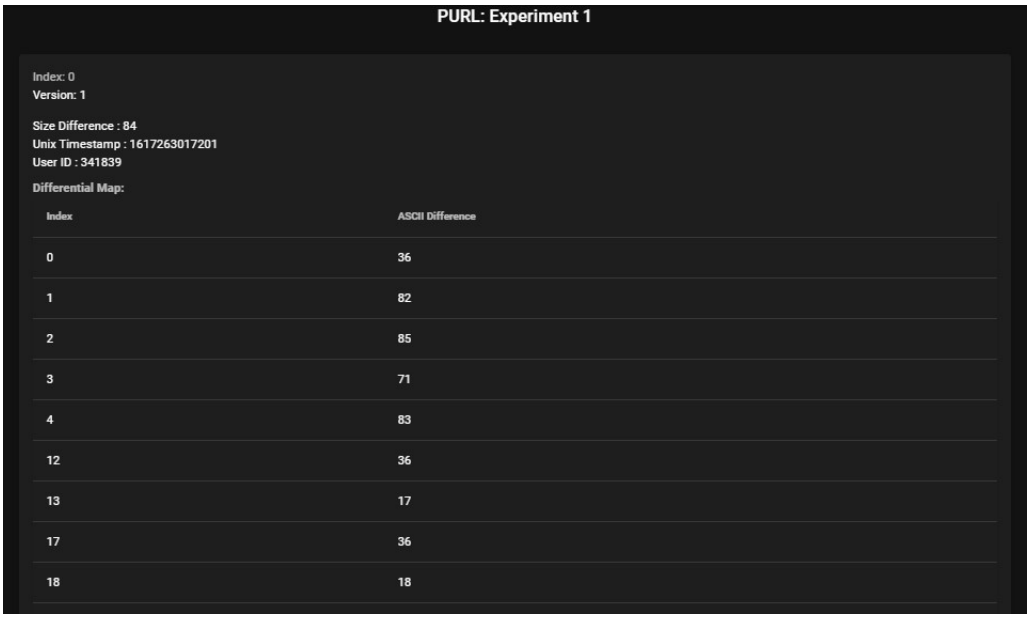



Figure 5.17: Provenance of a file part 1

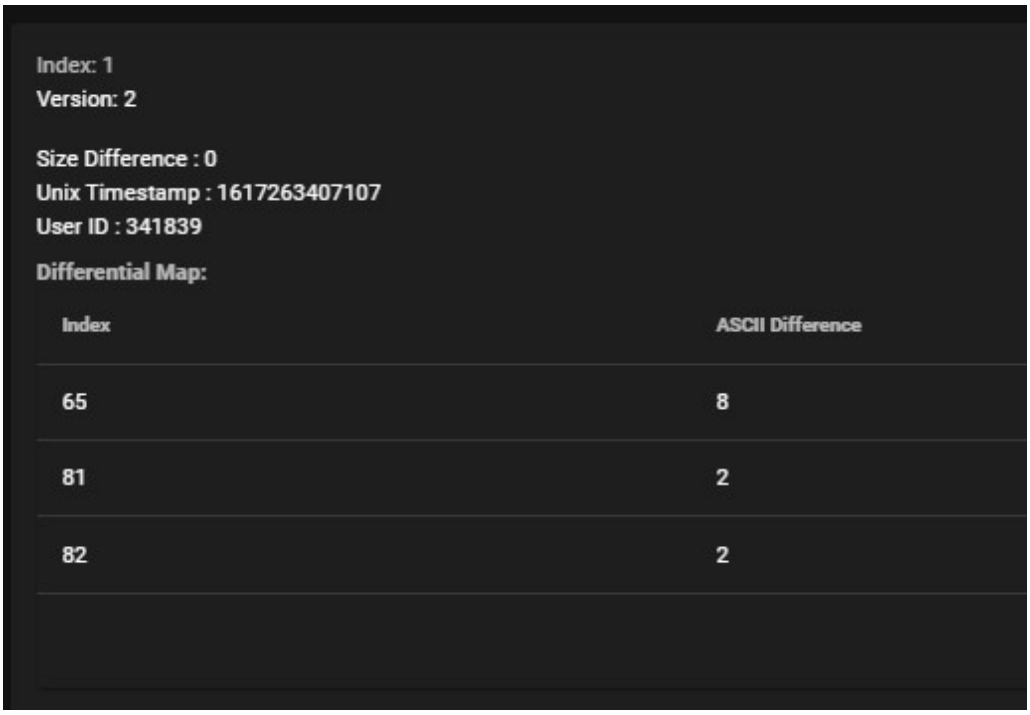



Figure 5.18: Provenance of a file part 2

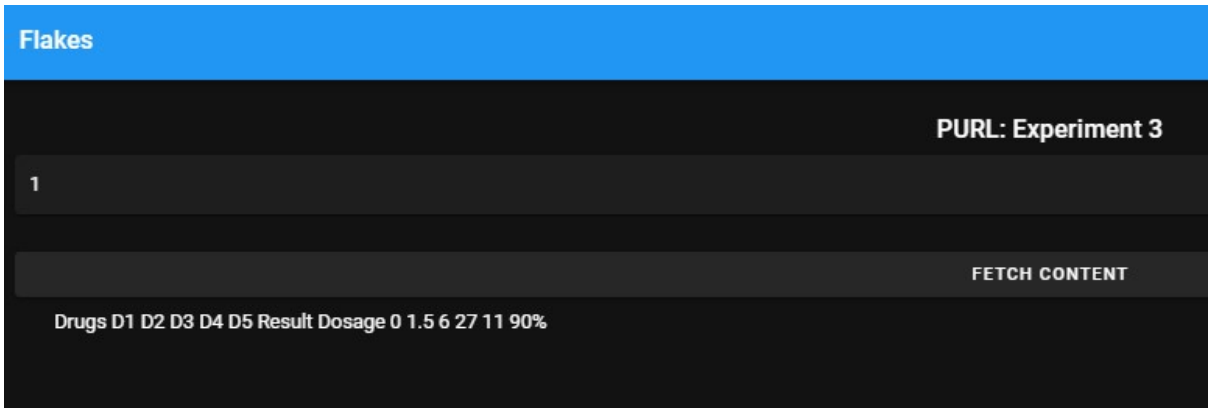



Figure 5.19: Versions of a file part 1

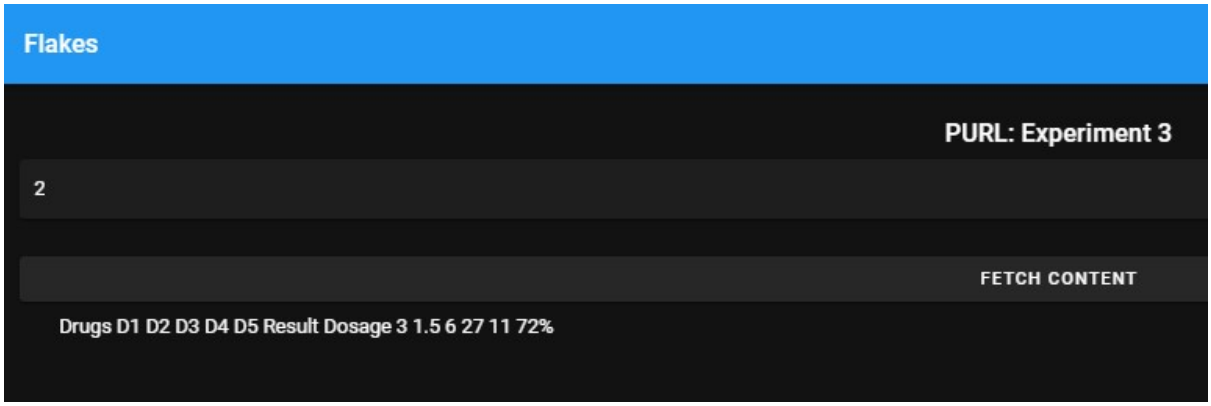



Figure 5.20: Versions of a file part 2

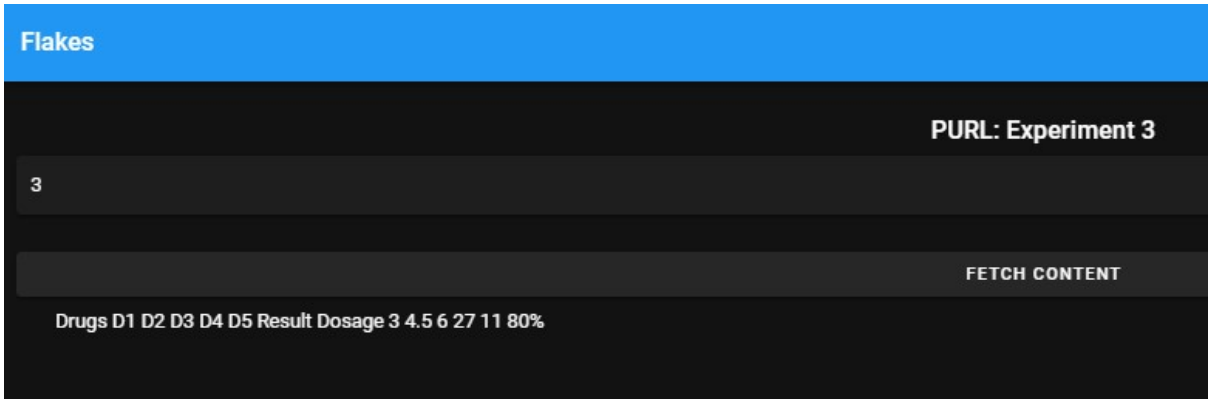



Figure 5.21: Versions of a file part 3

## 5.9 Proof of Correctness

This section discusses the proofs of correctness of the Flakes system. To determine the proof that the system produces the intended output, we built test cases to check. To do that, we had to isolate all the outputs the system was producing, and writing inputs for which the outputs were known, feed the inputs to the system and check if the outputs matched the known outputs.

- PURL: The system cannot have multiple identical PURLs. Since the PURLs are file identifiers, and used as primary keys, each PURL needs to be unique. So we checked for the uniqueness, i.e. trying to create a file with an existing PURL. Also, we checked the mapping between the mapping of a file and the contents of a file. Figure 5.22 shows the code for the test case.

- Entry Hash: The Entry Hash should be unique and include all the fields present in an entry. Figure 5.23 shows the code for the test cases to check the correctness of the hash.

- Block Hash: Like Entry hash the system should be able to correctly calculate the hash of the contents of the block. It should contain the hash of the previous block as well and should be unique. Figure 5.24 shows the code for the test case.

- Provenance Signature: This section tests the validity of the signature of the user provided in the provenance information. Figure 5.25 shows the code for the test cases.

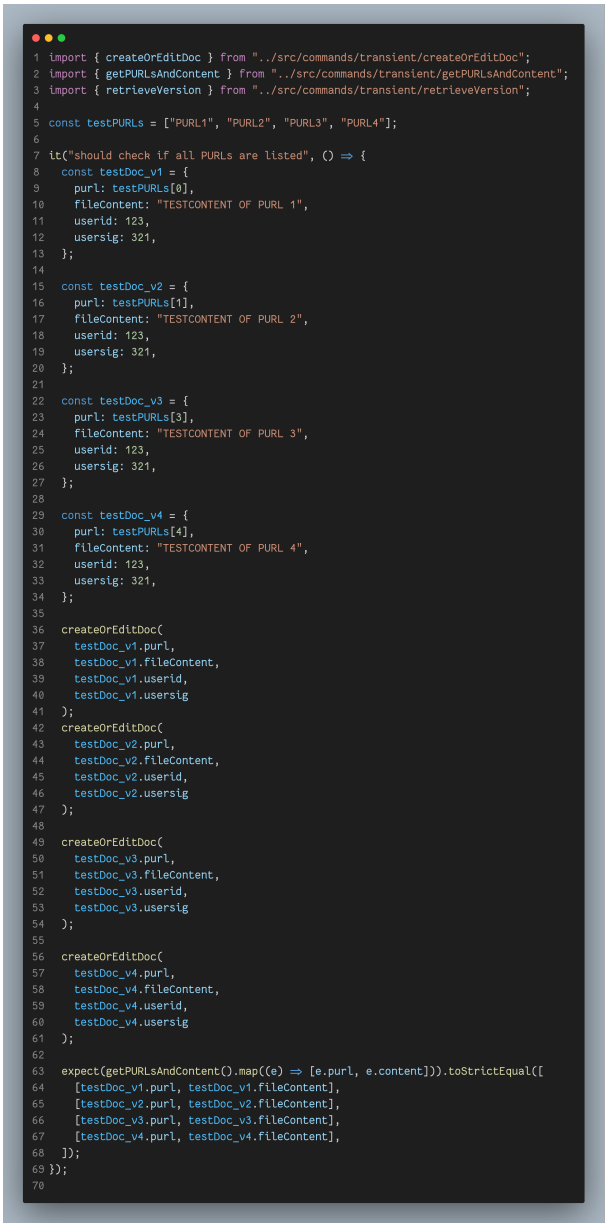
- Differential: This section tests the accuracy of the differentials calculated when a change is made in the document, and the difference in size between two successive edits. Figure 5.26 shows the code for the concerning test cases.

- Version Control: This segment checks the roll back method, i.e. if a previous version can be successfully restored from the current version using the differentials. Figure 5.27 shows the code for the test cases.

- Creation of a Document: This segments tests whether the event of creating a document reflects in the blockchain. Figure 5.28 shows the corresponding code for the test cases.

- Editing a Document: This segment tests whether the event of editing an existing document is reflected properly in the Blockchain. Figure 5.29 shows the code for the corresponding test cases.

- Lastly, figure 5.30 shows the result of all the test cases, which were successfully completed.

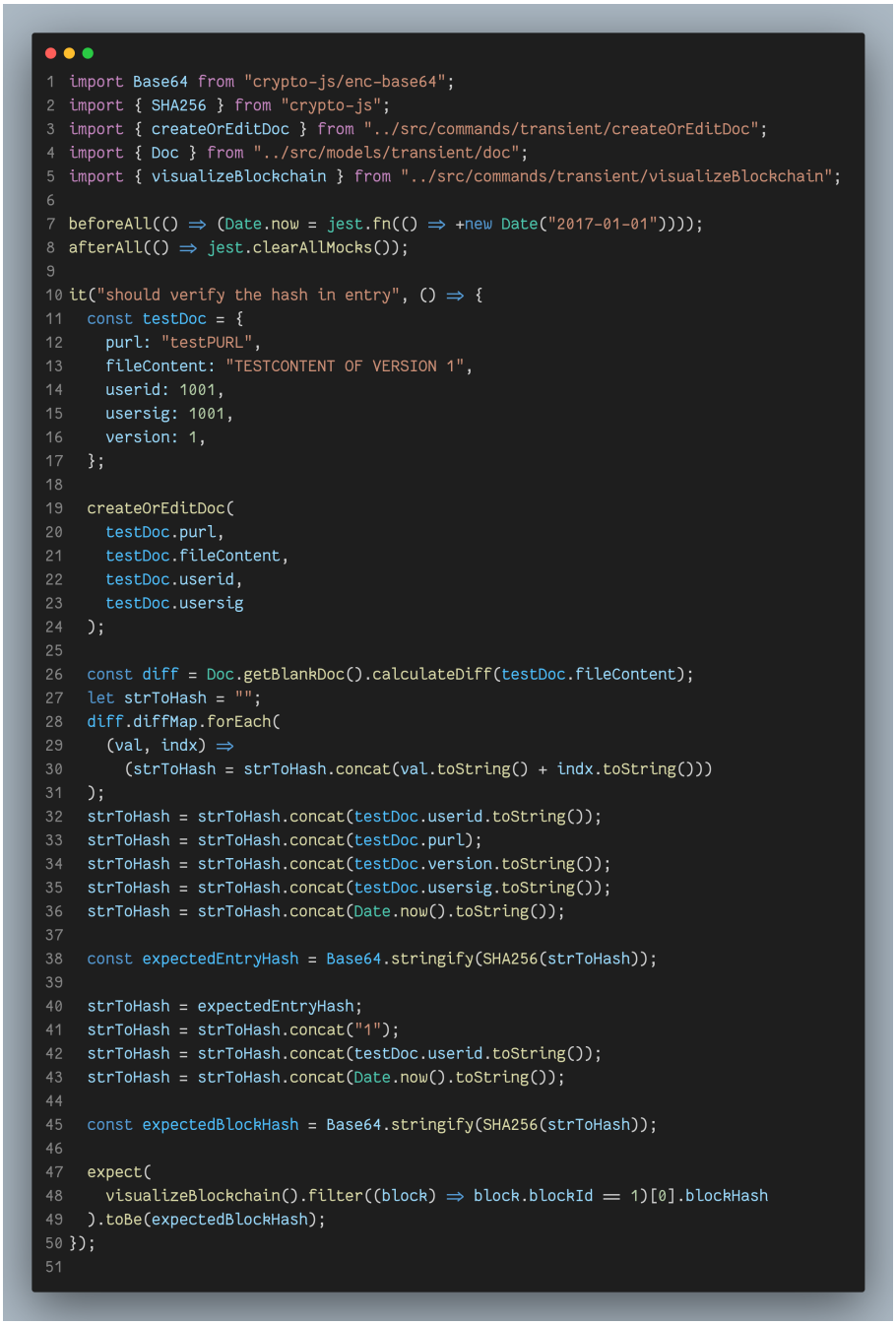


```
import { createOrEditDoc } from "../src/commands/transient/createOrEditDoc";
import { getPURLsAndContent } from "../src/commands/transient/getPURLsAndContent";
import { retrieveVersion } from "../src/commands/transient/retrieveVersion";

const testPURLs = ["PURL1", "PURL2", "PURL3", "PURL4"];

it("should check if all PURLs are listed", () => {
  const testDoc_v1 = {
    purl: testPURLs[0],
    fileContent: "TESTCONTENT OF PURL 1",
    userid: 123,
    usersig: 321,
  };

  const testDoc_v2 = {
    purl: testPURLs[1],
    fileContent: "TESTCONTENT OF PURL 2",
    userid: 123,
    usersig: 321,
  };

  const testDoc_v3 = {
    purl: testPURLs[3],
    fileContent: "TESTCONTENT OF PURL 3",
    userid: 123,
    usersig: 321,
  };

  const testDoc_v4 = {
    purl: testPURLs[4],
    fileContent: "TESTCONTENT OF PURL 4",
    userid: 123,
    usersig: 321,
  };

  createOrEditDoc(
    testDoc_v1.purl,
    testDoc_v1.fileContent,
    testDoc_v1.userid,
    testDoc_v1.usersig
  );
  createOrEditDoc(
    testDoc_v2.purl,
    testDoc_v2.fileContent,
    testDoc_v2.userid,
    testDoc_v2.usersig
  );

  createOrEditDoc(
    testDoc_v3.purl,
    testDoc_v3.fileContent,
    testDoc_v3.userid,
    testDoc_v3.usersig
  );

  createOrEditDoc(
    testDoc_v4.purl,
    testDoc_v4.fileContent,
    testDoc_v4.userid,
    testDoc_v4.usersig
  );

  expect(getPURLsAndContent().map((e) => [e.purl, e.content])).toStrictEqual([
    [testDoc_v1.purl, testDoc_v1.fileContent],
    [testDoc_v2.purl, testDoc_v2.fileContent],
    [testDoc_v3.purl, testDoc_v3.fileContent],
    [testDoc_v4.purl, testDoc_v4.fileContent],
  ]);
});

```

Figure 5.22: Test Cases for checking PURLs

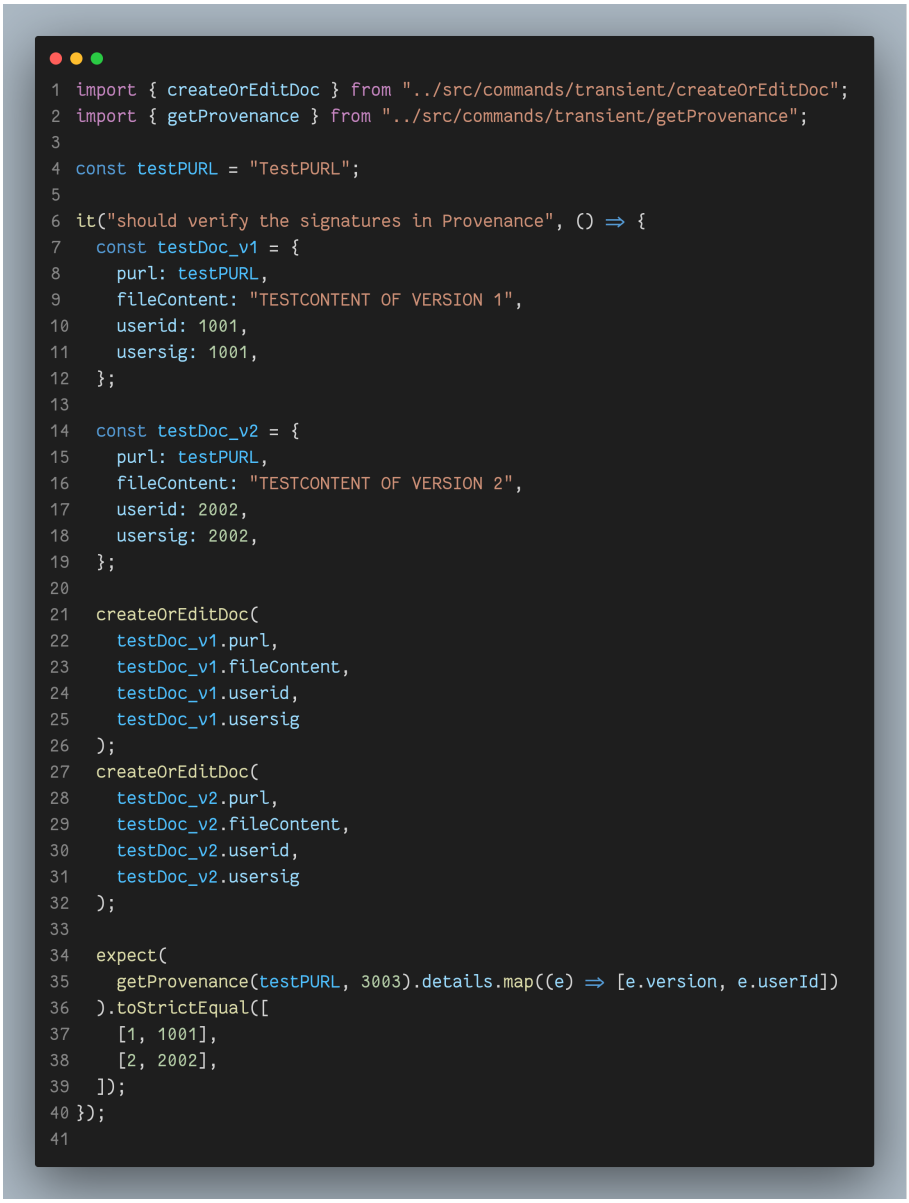
```
import Base64 from "crypto-js/enc-base64";
import { SHA256 } from "crypto-js";
import { createOrEditDoc } from "../src/commands/transient/createOrEditDoc";
import { getProvenance } from "../src/commands/transient/getProvenance";
import { Doc } from "../src/models/transient/doc";

beforeAll(() => (Date.now = jest.fn(() => +new Date("2017-01-01"))));
afterAll(() => jest.clearAllMocks());

it("should verify the hash in block", () => {
  const testDoc = {
    purl: "testPURL",
    fileContent: "TESTCONTENT OF VERSION 1",
    userid: 1001,
    usersig: 1001,
    version: 1,
  };

  createOrEditDoc(
    testDoc.purl,
    testDoc.fileContent,
    testDoc.userid,
    testDoc.usersig
  );

  const diff = Doc.getBlankDoc().calculateDiff(testDoc.fileContent);
  let strToHash = "";
  diff.diffMap.forEach(
    (val, indx) =>
      (strToHash = strToHash.concat(val.toString() + indx.toString()))
  );
  strToHash = strToHash.concat(testDoc.userid.toString());
  strToHash = strToHash.concat(testDoc.purl);
  strToHash = strToHash.concat(testDoc.version.toString());
  strToHash = strToHash.concat(testDoc.usersig.toString());
  strToHash = strToHash.concat(Date.now().toString());

  const expectedHash = Base64.stringify(SHA256(strToHash));

  expect(
    getProvenance(testDoc.purl, 100).details.filter(
      (e) => e.version == testDoc.version
    )[0].hash
  ).toBe(expectedHash);
});

```

Figure 5.23: Test Cases for checking the hash of an Entry



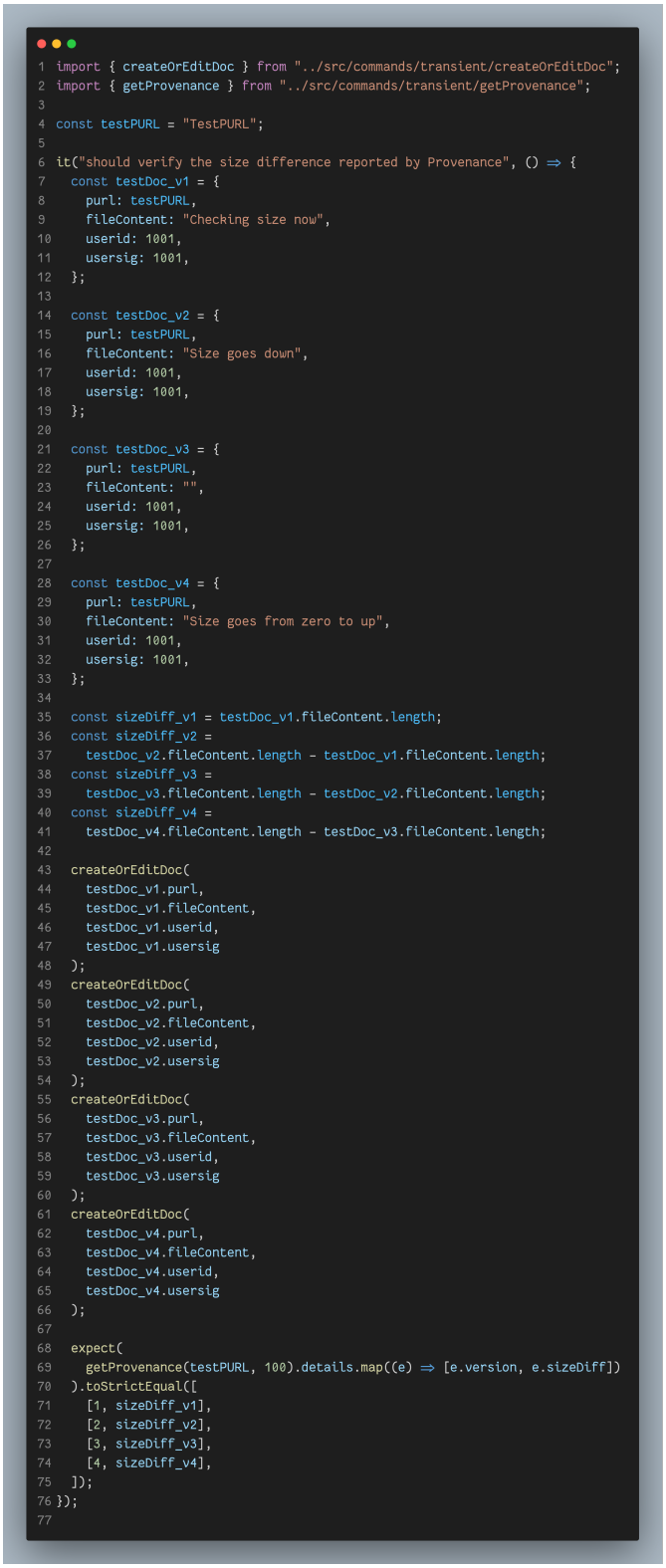
```
import Base64 from "crypto-js/enc-base64";
import { SHA256 } from "crypto-js";
import { createOrEditDoc } from "../src/commands/transient/createOrEditDoc";
import { Doc } from "../src/models/transient/doc";
import { visualizeBlockchain } from "../src/commands/transient/visualizeBlockchain";

beforeAll(() => (Date.now = jest.fn(() => +new Date("2017-01-01"))));
afterAll(() => jest.clearAllMocks());

it("should verify the hash in entry", () => {
  const testDoc = {
    purl: "testPURL",
    fileContent: "TESTCONTENT OF VERSION 1",
    userid: 1001,
    usersig: 1001,
    version: 1,
  };

  createOrEditDoc(
    testDoc.purl,
    testDoc.fileContent,
    testDoc.userid,
    testDoc.usersig
  );

  const diff = Doc.getBlankDoc().calculateDiff(testDoc.fileContent);
  let strToHash = "";
  diff.diffMap.forEach(
    (val, indx) =>
      (strToHash = strToHash.concat(val.toString() + indx.toString()))
  );
  strToHash = strToHash.concat(testDoc.userid.toString());
  strToHash = strToHash.concat(testDoc.purl);
  strToHash = strToHash.concat(testDoc.version.toString());
  strToHash = strToHash.concat(testDoc.usersig.toString());
  strToHash = strToHash.concat(Date.now().toString());

  const expectedEntryHash = Base64.stringify(SHA256(strToHash));

  strToHash = expectedEntryHash;
  strToHash = strToHash.concat("1");
  strToHash = strToHash.concat(testDoc.userid.toString());
  strToHash = strToHash.concat(Date.now().toString());

  const expectedBlockHash = Base64.stringify(SHA256(strToHash));

  expect(
    visualizeBlockchain().filter((block) => block.blockId == 1)[0].blockHash
  ).toBe(expectedBlockHash);
});

```

Figure 5.24: Test Cases for checking Block Hash



```
import { createOrEditDoc } from "../src/commands/transient/createOrEditDoc";
import { getProvenance } from "../src/commands/transient/getProvenance";

const testPURL = "TestPURL";

it("should verify the signatures in Provenance", () => {
  const testDoc_v1 = {
    purl: testPURL,
    fileContent: "TESTCONTENT OF VERSION 1",
    userid: 1001,
    usersig: 1001,
  };

  const testDoc_v2 = {
    purl: testPURL,
    fileContent: "TESTCONTENT OF VERSION 2",
    userid: 2002,
    usersig: 2002,
  };

  createOrEditDoc(
    testDoc_v1.purl,
    testDoc_v1.fileContent,
    testDoc_v1.userid,
    testDoc_v1.usersig
  );
  createOrEditDoc(
    testDoc_v2.purl,
    testDoc_v2.fileContent,
    testDoc_v2.userid,
    testDoc_v2.usersig
  );

  expect(
    getProvenance(testPURL, 3003).details.map((e) => [e.version, e.userId])
  ).toStrictEqual([
    [1, 1001],
    [2, 2002],
  ]);
});

```

Figure 5.25: Test Cases for checking Provenance Signature



```
import { createOrEditDoc } from "../src/commands/transient/createOrEditDoc";
import { getProvenance } from "../src/commands/transient/getProvenance";

const testPURL = "TestPURL";

it("should verify the size difference reported by Provenance", () => {
  const testDoc_v1 = {
    purl: testPURL,
    fileContent: "Checking size now",
    userid: 1001,
    usersig: 1001,
  };

  const testDoc_v2 = {
    purl: testPURL,
    fileContent: "Size goes down",
    userid: 1001,
    usersig: 1001,
  };

  const testDoc_v3 = {
    purl: testPURL,
    fileContent: "",
    userid: 1001,
    usersig: 1001,
  };

  const testDoc_v4 = {
    purl: testPURL,
    fileContent: "Size goes from zero to up",
    userid: 1001,
    usersig: 1001,
  };

  const sizeDiff_v1 = testDoc_v1.fileContent.length;
  const sizeDiff_v2 =
    testDoc_v2.fileContent.length - testDoc_v1.fileContent.length;
  const sizeDiff_v3 =
    testDoc_v3.fileContent.length - testDoc_v2.fileContent.length;
  const sizeDiff_v4 =
    testDoc_v4.fileContent.length - testDoc_v3.fileContent.length;

  createOrEditDoc(
    testDoc_v1.purl,
    testDoc_v1.fileContent,
    testDoc_v1.userid,
    testDoc_v1.usersig
  );
  createOrEditDoc(
    testDoc_v2.purl,
    testDoc_v2.fileContent,
    testDoc_v2.userid,
    testDoc_v2.usersig
  );
  createOrEditDoc(
    testDoc_v3.purl,
    testDoc_v3.fileContent,
    testDoc_v3.userid,
    testDoc_v3.usersig
  );
  createOrEditDoc(
    testDoc_v4.purl,
    testDoc_v4.fileContent,
    testDoc_v4.userid,
    testDoc_v4.usersig
  );

  expect(
    getProvenance(testPURL, 100).details.map((e) => [e.version, e.sizeDiff])
  ).toStrictEqual([
    [1, sizeDiff_v1],
    [2, sizeDiff_v2],
    [3, sizeDiff_v3],
    [4, sizeDiff_v4],
  ]);
});

```

Figure 5.26: Test Cases for checking Accuracy of Differentials

```
import { createOrEditDoc } from "../src/commands/transient/createOrEditDoc";
import { retrieveVersion } from "../src/commands/transient/retrieveVersion";

const testPURL = "TestPURL";

it("should verify the contents of all versions of an edited Doc", () => {
  const testDoc_v1 = {
    purl: testPURL,
    fileContent: "TESTCONTENT OF VERSION 1",
    userid: 123,
    usersig: 321,
  };

  const testDoc_v2 = {
    purl: testPURL,
    fileContent: "TESTCONTENT OF VERSION 2",
    userid: 123,
    usersig: 321,
  };

  createOrEditDoc(
    testDoc_v1.purl,
    testDoc_v1.fileContent,
    testDoc_v1.userid,
    testDoc_v1.usersig
  );
  createOrEditDoc(
    testDoc_v2.purl,
    testDoc_v2.fileContent,
    testDoc_v2.userid,
    testDoc_v2.usersig
  );
  expect(retrieveVersion(testPURL, 1).fileContent).toBe(testDoc_v1.fileContent);
  expect(retrieveVersion(testPURL, 2).fileContent).toBe(testDoc_v2.fileContent);
});

```

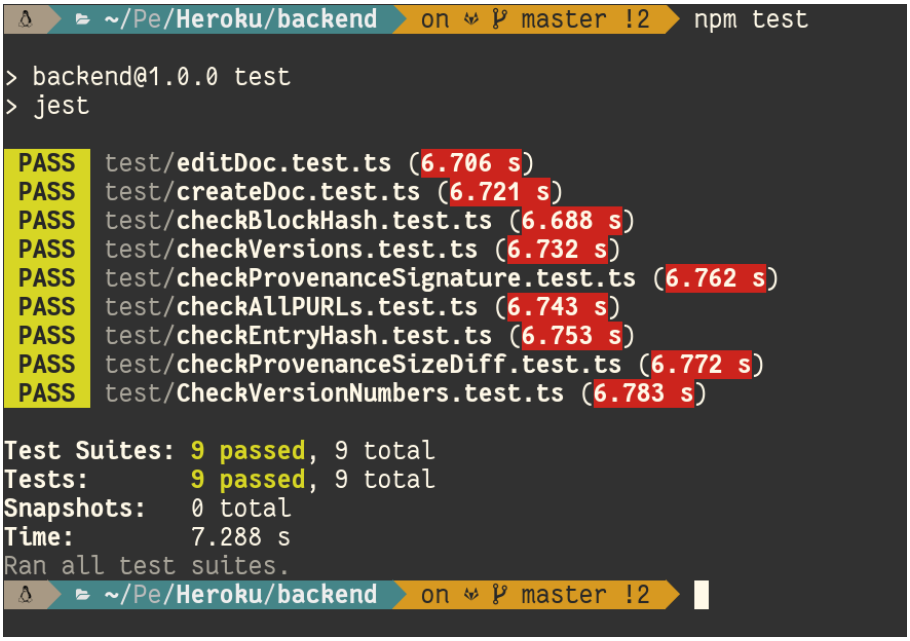
Figure 5.27: Test Cases for checking Roll Back Method

```
import { createOrEditDoc } from "../src/commands/transient/createOrEditDoc";
import { getDoc } from "../src/commands/transient/getDoc";

it("should verify the content of new Doc", () => {
  const testDoc = {
    purl: "TestPURL",
    fileContent: "TestContent",
    userid: 123,
    usersig: 321,
  };
  createOrEditDoc(
    testDoc.purl,
    testDoc.fileContent,
    testDoc.userid,
    testDoc.usersig
  );
  expect(getDoc(testDoc.purl).fileContent).toBe(testDoc.fileContent);
});

```

Figure 5.28: Test Cases for checking the event register of creating a document

```
import { createOrEditDoc } from "../src/commands/transient/createOrEditDoc";
import { getDoc } from "../src/commands/transient/getDoc";

it("should verify the content of new Doc", () => {
  const testDoc = {
    purl: "TestPURL",
    fileContent: "TestContent",
    userid: 123,
    usersig: 321,
  };
  createOrEditDoc(
    testDoc.purl,
    testDoc.fileContent,
    testDoc.userid,
    testDoc.usersig
  );
  expect(getDoc(testDoc.purl).fileContent).toBe(testDoc.fileContent);
});

```

Figure 5.29: Test Cases for checking the event register of editing a document



```
~/Pe/Heroku/backend on master !2 npm test

> backend@1.0.0 test
> jest

 PASS  test/editDoc.test.ts (6.706 s)
 PASS  test/createDoc.test.ts (6.721 s)
 PASS  test/checkBlockHash.test.ts (6.688 s)
 PASS  test/checkVersions.test.ts (6.732 s)
 PASS  test/checkProvenanceSignature.test.ts (6.762 s)
 PASS  test/checkAllPURLs.test.ts (6.743 s)
 PASS  test/checkEntryHash.test.ts (6.753 s)
 PASS  test/checkProvenanceSizeDiff.test.ts (6.772 s)
 PASS  test/CheckVersionNumbers.test.ts (6.783 s)

Test Suites: 9 passed, 9 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        7.288 s
Ran all test suites.
~/Pe/Heroku/backend on master !2
```

Figure 5.30: Result for the Test cases

# Chapter 6

# Conclusions

In this chapter we will overview the relevance and the efficiency of Flakes, the blockchain based tool which provides data provenance and version control as a service. We will discuss why we need a service like Flakes, and how it is better than the existing services.

## 6.1 Relevance

In this age of digital awareness, more stress is given to privacy and security. More importance given to restrict someone to access and edit information and not so much on preserving history of the information. There are a few existing data provenance tools which try to preserve action history, but they have some intrinsic weaknesses. In 2008, Bitcoin provided the world with the revolutionary blockchain technology which is a simple yet elegant way of securely preserving a log. Although initially blockchain was used only for cryptocurrency transactions, later various enterprises started using blockchain for creating and maintaining smart contracts and other value-based transaction data. We realised blockchain could be immensely powerful in storing data history. This provides a two way benefit. It allows the user to seek provenance at any point of time, and since it keeps a track of the changes made in the system, it could be used to traverse through versions of the system. In absence of a tool that provides both of these services and ensures utmost security, Flakes is a very relevant system in the world of data security.

## 6.2 Analysis

In this section, we will look at how Flakes holds up to the challenges faced by data provenance and version control services and blockchain as a tool.

- **Provenance Threat Modelling**

  The existing provenance tools store their provenance information in a database as a link to the documents, or as graphs. These databases are encrypted with only authorized entities allowed to edit them. These tools experience mainly two sorts of threats, i) Communication Hijacks and ii) Database Attacks [80]. In the former, when a change is committed in the system, a notification needs to be sent out for the system to respond, and process the changes. An attacker can block or disable the notifications and hence stop the event be noted in the database. In the latter, malicious entities can tamper with a database using SQL Injection attacks or other means.

  In Flakes, all the servers or nodes process the changes simultaneously and attain a consensus about the decision. The servers check the system for changes after every pre-fixed time period, and does not require notifications to alert them. Hence Communication Hijacks are countered. All the provenance information is stored in the blockchain, which is immutable, countering the Database Attacks.

- **Version Control Threat Modelling**

  The existing version control tools save the versions of the files as snapshots in a separate database. This means that multiple copies of the same file, with major or minor edits are being stored. The database itself is prone to the attacks discussed in the previous point. In Git, when a previous version is restored, all the subsequent versions are deleted, assuming they won't be needed. Hence there is a loss of history and version information.

  In Flakes, instead of storing separate versions, the differential between two successive versions are stored after each edit. When a previous version has to be restored, the subsequent differentials, and hence the contents still remain in the blockchain, so there will be no loss of provenance or content.

- **Blockchain Threat Modelling**

  Worldwide cryptocurrencies rely on blockchain for their functioning. While the core security of blockchain has never been challenged, the policies these cryptocurrencies implemented came under various attacks. Since blockchain is a consensus based tool, these problems could have affected Flakes as well. But instead of a competitive majority based consensus in cryptocurrencies, Flakes relies on unanimity. While unanimity gets rid of the 51% attack, it leaves the system vulnerable to denial of service attacks, where a malicious entity wilfully disagrees with the rest of the peers. In Flakes, we have dealt with the problem by having the servers show reasons behind their decisions. This can be easily done by making them point out which field/fields they think are invalid. So a double verification sorts the denial of service problem.

In conclusion, in this thesis we present a detailed understanding of blockchain and cryptocurrencies; provenance, and various tools which provide the service; version control and git. We have proposed and developed a system which uses blockchain as a tool to provide data provenance and version control as a service. We have discussed how the proposed system is better than the existing ones, and which real life scenarios might benefit from a system like Flakes.

# Chapter 7

# Research Questions

Answering the following high level research questions provide relevance and scope of this dissertation.

## 7.1 **What problem are you trying to solve?**

Develop a distributed data provenance and version control architecture that will:

1. Track every change to every document in the system.
2. Keep a secure, robust log of all those changes.
3. Prevent attackers from tampering with those changes.
4. Provide information about those changes on demand.
5. Allow retrieving a previous version of any individual document, or the system as a whole.

## 7.2 **What will you know when you are done?**

1. The benefits of using blockchain as a distributed ledger for tracking actions on data.
2. The benefits of using blockchain to achieve version control.
3. How the system so developed fares against relevant attacks and similar systems.

## 7.3 How will you know when you are done?

1. When the working architecture deliverables are achieved, along with proof of correctness and proof of security.

## 7.4 How original is this?

1. There are no present version control systems that use blockchain to ensure secure management of history.

2. There are no existing systems that provide both data provenance and version control.

## 7.5 Who will care? What's the impact?

1. The need for data provenance is universal and significant.

2. This system will be useful in maintaining Academic Integrity. Collaborative researchers will benefit from this.

3. The system will impact Digital Forensics, where data comes from various sources, and are sensitive.

4. In general this system will be useful to any users who need to maintain sensitive data from various sources.

## 7.6 Why did you select this problem among many?

1. Handling a multiparty system with lots of data calls for secure management of not only the data, but also the edits performed on them. Knowing the source and history of data in a system is very important for preserving the integrity of the system. Present data provenance softwares have lots of limitations which this dissertation project aims to curb.

## 7.7 Why will this be a significant contribution to the literature in your area?

1. This dissertation will provide a novel approach towards data provenance introducing blockchain.

2. This will contribute not only to data provenance, but also to selective and general version control.

# Chapter 8

# Future Works

This section talks about various updates within Flakes, and how Flakes can be utilized in some other scenarios.

## 8.1 Flakes Mining

To stop fraudulent information from getting into the blockchain, there is a need for verification of the information. Cryptocurrencies verify a transaction through a process called mining, where multiple miners compete to verify a block to obtain a reward. To prevent them from taking fraudulent actions, they are required to perform a resource intensive task, and show a proof that the task has been completed. This proof can be proof of work, or proof of stake. In Flakes, there would be no reward for verification. An unanimous consensus can be achieved to perform verification.

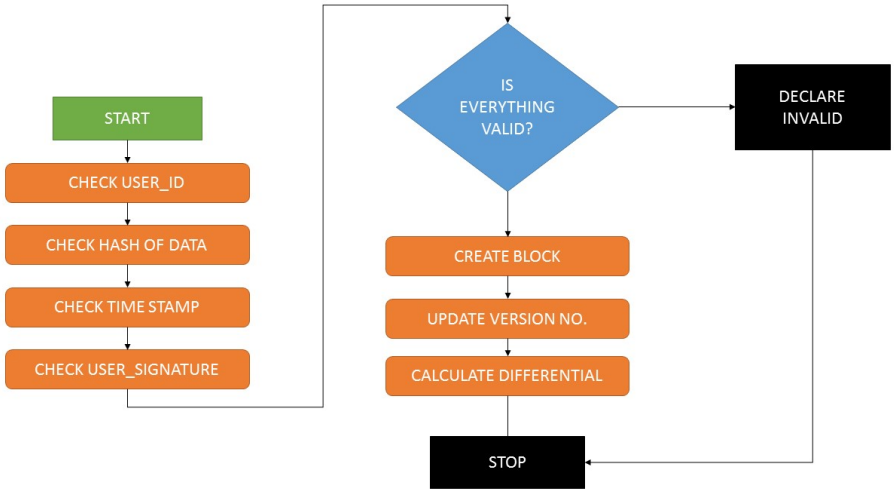
### 8.1.1 Verification

The process of verification of any action on any item in the system includes the following steps:

- The User-ID is verified if it exists within the permissioned pool.
- The hash of the data is checked
- The timestamp is compared with the current time to assert that it be lesser than the current time.
- The User Signature is verified for consistency with the User ID

- If any of the above fields turn out to be invalid, the change is not updated, and is declared invalid.
- If everything turns out to be valid, then a block is created with a new hash. The Version No is updated, and the Differential is calculated.

Figure 8.1: The Algorithm of Mining in Flakes




### 8.1.2 Consensus

Cryptocurrencies achieve consensus by making it mandatory by the miners to provide a proof of work, or proof of stake. However, they are resource intensive, and the miners are awarded financially in return. In Flakes, mining is lightweight, and the system does not include financial transactions. Scrybe uses a randomized selection process to select a miner from the servers. This randomization of the process keeps the selection process fair, and prevents collusion. However, it is not entirely risk free, and adds an unnecessary overhead to the system. Flakes uses a simpler process to achieve consensus. Since every server has a copy of the blockchain, whenever a change is made in the system, all the servers are notified simultaneously. All the servers individually verify the the validity of the change and updates the blockchain with the corresponding metadata. Even if only one of the servers are uncompromised, the updates wouldn't go through as they would be out of sync.

In case of a server being nonoperational for a certain period of time, its copy of blockchain will be updated with all the changes made during that time period, and the server can verify each of them when it starts operating again.

The process of mining in Flakes is shown in figure 8.1.

### 8.1.3 Threat Analysis

In this section We will look for potential vulnerabilities, and how the system counters them.

#### 51% Vulnerability

51% Attack[98] is a known vulnerability in blockchain based systems, specially in cryptocurrencies. The core principle of a blockchain based system is distributed consensus [98]. However, if a miner or a group of miners collude to take hold of more than 50% of the hashing power, they control the outcome of the transactions, since they will be the majority in the consensus. Bitcoin counters this by making the mining process computationally expensive through proof of work, which reduces the profit-worthyness of obtaining 51% hashing power[99]. Some other cryptocurrencies use proof of stake to mitigate this attack [99]. But it is possible for miners to attain 51% stake in a system.

Flakes does not rely on the majority rule. Unlike the cryptocurrencies, Flakes requires decisions to be unanimous, i.e. all the servers has to settle on the same outcome as stated by the rules. Even if only one of the servers remain benign, a false action can be prevented.

#### Denial of Service Attack

Countering 51% vulnerability by unanimity gives way to denial of service attacks. In this case a server can be manipulated to reject a valid block and hence stopping it from going through.

This problem can be solved by having each server mention why they are declaring a block to be invalid, i.e. which field is inconsistent. If the inconsistency does not match the decision, the decision is overruled. A server with a mismatch is penalized by a strike through counter. After 3 consecutive strikes or 5 overall strikes, that server is suspended from making validations.

When a verifier declares a block invalid, they reset the validity flag to 0. When the flag is 0, the next field is checked, which contains the reason of the declaration. Figure 8.2 shows the updated fields of a Flakes block.

Figure 8.2: Flakes Class Diagram

| | |
|---|---|
| 1 | Block ID |
| 2 | Block Size |
| 3 | Entry Count |
| 4 | Previous Hash |
| 5 | Current Hash |
| 6 | Entry List |
| 7 | Timestamp |
| 8 | Differential |
| 9 | Validity = 0 |
| 10 | Reason |
| 11 | Verifier Signature |

When a verifier declares a block valid, they set the validity flag to 1, and the next field is skipped.

## 8.2 Digital Forensics Use Case

Presently, Flakes has been coded to handle text-based use cases only. However, with minor adjustments in formatting, it can be applied to track provenance for any type of documents, like images, audio, or video. For text-based format, we treat a text file as a large character

array, where every index has an ASCII value stored in it. For images, the pixels will be treated as indices, and the RGB values considered for differentials. Videos are essentially a list of frames which are nothing but images. So the index would be a $Frameno., PixelNo.$ couple. For digital audio files, the indices would the sample numbers, and each sample number will have some value attached to it. So there will be no difficulty in calculating differentials. Once that is achieved, Flakes can be useful to the following use case.

The Flakes architecture has an effective use case in digital forensics. A Forensics evidence preservation system is a complex model, where there are three main entities involved,

- The crime lab technician, who collects and performs tests and analysis on the evidence.
- The attorneys, who seek validity of the decisions and the provenance of the evidence, and
- The court, who is a read only entity.

Whenever an evidence is introduced to the system, it can be in any format, the event is recorded in the blockchain. Any tests performed on them or edits made is also recorded in the blockchain in chronological order with timestamps and technicial signatures. While such tests, if any evidence is altered from its original state, it can be reverted back to that state through the differentials. Since all activities are recorded securely in the blockchain, when an attorney seeks provenance on any evidence, it is readily available. Figure 8.3 shows the UML diagram of such a system.

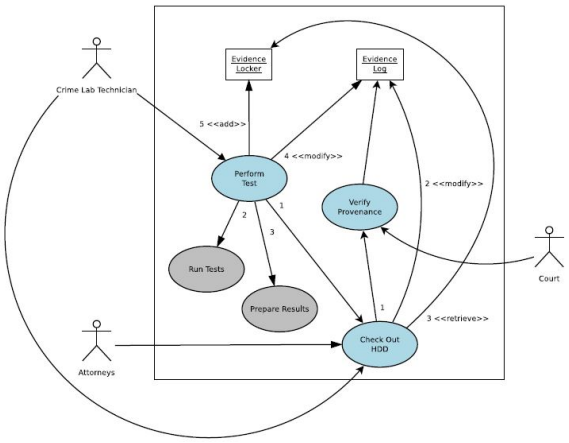



Figure 8.3: The Digital Forensics Use Case

References


[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[2] Ken Shirriff. Mining bitcoin with pen and paper. http://www.righto.com/2014/09/mining-bitcoin-with-pencil-and-paper.html, 2014.

[3] M. Stephen. Forking in a blockchain. https://bitcoin.stackexchange.com/questions/35719/is-there-a-fixed-order-for-all-transactions-in-the-block-chain, 2015.

[4] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. *available at ietf.org*, 2015.

[5] Scott Chacon. Git internals. *Pro Git*, pages 223–250, 2009.

[6] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Data provenance: Some basic issues. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 87–93. Springer, 2000.

[7] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, page 9. ACM, 2008.

[8] Investopedia. Blockchain. http://www.investopedia.com/terms/b/Blockchain.asp, 2016.

[9] Bitcoin Wiki. Bitcoin block. https://en.bitcoin.it/wiki/Block, 2016.

[10] Melanie Swan. *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[11] Bitcoin Wiki David Perry. Bitcoin mining. `https://en.bitcoin.it/wiki/Mining`, 2014.

[12] Hal. Double spending. `https://en.bitcoin.it/wiki/Double-spending`, 2015.

[13] BitBucket. What is git. `https://www.atlassian.com/git/tutorials/what-is-git`, 2016.

[14] Diomidis Spinellis. Version control systems. *IEEE Software*, 22(5):108–109, 2005.

[15] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[16] Jennifer Vesperman. *Essential CVS: Version Control and Source Code Management*. " O'Reilly Media, Inc.", 2006.

[17] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. *Version control with subversion*. " O'Reilly Media, Inc.", 2004.

[18] Janos Gyerik. *Bazaar version control*. Packt Publishing Ltd, 2013.

[19] Diomidis Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.

[20] Vincent Driessen. A successful git branching model. *URL http://nvie. com/posts/a-successful-git-branching-model*, 2010.

[21] Justin Pinnix, Brian Harry, Michael Sliger, Christopher Antos, and Thomas McGuire. Version control system, October 19 2006. US Patent App. 11/107,145.

[22] Stefan Otte. Version control systems. *Computer Systems and Telematics*, 2009.

[23] Ryan Farell. An analysis of the cryptocurrency industry. *available at repository.upenn.edu*, 2015.

[24] Jason Teutsch, Sanjay Jain, and Prateek Saxena. When cryptocurrencies mine their own business, 2013.

[25] Nicolas Sklavos and Odysseas Koufopavlou. Implementation of the sha-2 hash family standard using fpgas. *The Journal of Supercomputing*, 31(3):227–248, 2005.

[26] Shaik Shakeel Ahamad, Madhusoodhnan Nair, and Biju Varghese. A survey on crypto currencies. In *4th International Conference on Advances in Computer Science, AETACS*, pages 42–48. Citeseer, 2013.

[27] David Chaum. David chaum on electronic commerce how much do you trust big brother? *IEEE Internet Computing*, 1(6):8–16, 1997.

[28] Thomas R Eisenmann and Lauren Barley. Paypal merchant services. *Available at hbs.edu*, 2006.

[29] William Jack and Tavneet Suri. Mobile money: The economics of m-pesa. Technical report, National Bureau of Economic Research, 2011.

[30] N Gregory Mankiw. *Principles of macroeconomics*. Cengage Learning, 2014.

[31] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International Workshop on Fast Software Encryption*, pages 371–388. Springer, 2004.

[32] Wiki. Bitcoin. Bitcoin address. `https://en.bitcoin.it/wiki/Address`, 2015.

[33] H Dobbertin, A Bosselaers, and B Preneel. "ripemd160, a strengthened version of ripemd," fast software encryption 1996. *Lecture Notes in Computer Science*, 1039.

[34] Bitcoin Wiki. Bitcoin wallet. `https://en.bitcoin.it/wiki/Wallet`, 2017.

[35] Bitcoin Wiki. Proof of work. `https://en.bitcoin.it/wiki/Proof_of_work`, 2014.

[36] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[37] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 475–490. IEEE, 2014.

[38] Nember A. Poi. http://blog.nem.io/tag/proof-of-importance/, 2014.

[39] Nember A. Nem. http://www.nem.io/, 2014.

[40] Dean R. Proofofresource. http://cryptorials.io/tag/proof-of-resources/, 2014.

[41] Artefact2 (Pseudonym). Limit.

http://bitcoin.stackexchange.com/questions/161/how-many-bitcoins-will-there-eventually-be#comment7700_274, 2014.

[42] Bitcoin Wiki. Target. https://en.bitcoin.it/wiki/Target, 2017.

[43] Bitcoin Wiki. Difficulty. https://en.bitcoin.it/wiki/Difficulty, 2017.

[44] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, pages 1–16, 2009.

[45] Bitcoin Wiki. Proof of stake. https://en.bitcoin.it/wiki/Proof_of_Stake, 2016.

[46] JP Buntinx. What is proof of importance. https://themerkle.com/what-is-proof-of-importance/, 2015.

[47] Bitcointalk Forum. Nem technical reference. https://www.nem.io/NEM_techRef.pdf, 2015.

[48] Henri Gilbert and Helena Handschuh. Security analysis of sha-256 and sisters. In *Selected areas in cryptography*, pages 175–193. Springer, 2003.

[49] Nicolas T Courtois, Marek Grajek, and Rahul Naik. Optimizing sha256 in bitcoin mining. In *Cryptography and Security Systems*, pages 131–144. Springer, 2014.

[50] João Neto. Ethash. `https://github.com/ethereum/wiki/wiki/Ethash`, 2014.

[51] Orr Dunkelman and Dmitry Khovratovich. Iterative differentials, symmetries, and message modification in blake-256. In *ECRYPT2 Hash Workshop*, volume 2011. Citeseer, 2011.

[52] Daniel J Bernstein. Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8, 2008.

[53] Dean. Hashx11. `http://cryptorials.io/glossary/x11/`, 2014.

[54] Mohamed El-Hadedy, Martin Margala, Danilo Gligoroski, and Svein J Knapskog. Resource-efficient implementation of blue midnight wish-256 hash function on xilinx fpga platform. In *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, pages 44–47. IEEE, 2010.

[55] Bernhard Jungk, Steffen Reith, and Jürgen Apfelbeck. On optimized fpga implementations of the sha-3 candidate groestl. *IACR Cryptology ePrint Archive*, 2009:206, 2009.

[56] Hongjun Wu. The hash function jh. *Submission to NIST (round 3)*, page 6, 2011.

[57] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak sha-3 submission. *Submission to NIST (Round 3)*, 6(7):16, 2011.

[58] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. *Submission to NIST (round 3)*, 7(7.5):3, 2010.

[59] Christophe De Canniere, Hisayoshi Sato, and Dai Watanabe. Hash function luffa: specification. *Submission to NIST (Round 2)*, 2009.

[60] Daniel J Bernstein. Cubehash specification (2. b. 1). *Submission to NIST*, 2008.

[61] Eli Biham and Orr Dunkelman. The shavite-3 hash function. *Submission to NIST (Round 2)*, page 113, 2009.

[62] Stefan Tillich, Martin Feldhofer, Mario Kirschbaum, Thomas Plos, Jörn-Marc Schmidt, and Alexander Szekely. High-speed hardware implementations of blake, blue midnight wish, cubehash, echo, fugue, gröstl, hamsi, jh, keccak, luffa, shabal, shavite-3, simd, and skein. *IACR Cryptology ePrint Archive*, 2009:510, 2009.

[63] Martin Schläffer. Subspace distinguisher for 5/8 rounds of the echo-256 hash function. In *International Workshop on Selected Areas in Cryptography*, pages 369–387. Springer, 2010.

[64] Tuomo Nieminen Antonio M. Juarez Seigen, Max Jameson. Cryptonight. `https://cryptonote.org/cns/cns008.txt`, 2013.

[65] Ujan Mukhopadhyay, Anthony Skjellum, Oluwakemi Hambolu, Jon Oakley, Lu Yu, and Richard Brooks. A brief survey of cryptocurrency systems. In *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*, pages 745–752. IEEE, 2016.

[66] Adam Back. The hashcash proof-of-work function. *Draft-Hashcash-back-00, Internet-Draft Created,(Jun. 2003)*, 2003.

[67] Karl J O'Dwyer and David Malone. Bitcoin mining and its energy footprint. In *Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET*, pages 280–285. IET, 2013.

[68] John Polkinghorne and Michael Desnoyers. Application specific integrated circuit, March 28 1989. US Patent 4,816,823.

[69] Krzysztof Okupski. Bitcoin developer reference. *Available at http://enetium. com/resources/Bitcoin. pdf*, 2014.

[70] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[71] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[72] David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, page 5, 2014.

[73] C Lee. Litecoin, 2011.

[74] CoinPursuit. Sha2 and scrypt. `https://www.coinpursuit.com/pages/bitcoin-altcoin-SHA-256-scrypt-mining-algorithms/`, 2014.

[75] Monero. Monero. `https://bitcointalk.org/index.php?topic=583449.0`, 2014.

[76] Nicolas van Saberhagen. Cryptonote v 2. 0. *HYPERLINK https://cryptonote. org/whitepaper. pdf*, 2013.

[77] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[78] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[79] Jean-loup Gailly and Mark Adler. Zlib compression library. 2004.

[80] Oluwakemi Hambolu, Lu Yu, Jon Oakley, Richard R Brooks, Ujan Mukhopadhyay, and Anthony Skjellum. Provenance threat modeling. In *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*, pages 384–387. IEEE, 2016.

[81] Data to Insight Center. Karma provenance, 2012.

[82] Bin Cao, Beth Plale, Girish Subramanian, Ed Robertson, and Yogesh Simmhan. Provenance information model of karma version 3. In *Services-I, 2009 World Conference on*, pages 348–351. IEEE, 2009.

[83] Yogesh L Simmhan, Beth Plale, Dennis Gannon, and Suresh Marru. Performance evaluation of the karma provenance framework for scientific workflows. In *Provenance and Annotation of Data*, pages 222–236. Springer, 2006.

[84] Alvaro Videla and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.

[85] Peng Chen and Beth Plale. Visualizing large scale scientific data provenance. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1385–1386. IEEE, 2012.

[86] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504, 2003.

[87] Data to Insight Center. Komadu provenance, 2012.

[88] Isuru Suriarachchi, Quan Zhou, and Beth Plale. Komadu: A capture and visualization system for scientific data provenance. *Journal of Open Research Software*, 3(1), 2015.

[89] Fernando Pérez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[90] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, 41(W1):W557–W561, 2013.

[91] Juliana Freire, Cláudio T Silva, Steven P Callahan, Emanuele Santos, Carlos E Scheidegger, and Huy T Vo. Managing rapidly-evolving scientific workflows. In *International Provenance and Annotation Workshop*, pages 10–18. Springer, 2006.

[92] Shawn Bowers, Timothy Mcphillips, Sean Riddle, Manish Kumar Anand, and Bertram Ludäscher. Kepler/ppod: Scientific workflow and provenance support for assembling

the tree of life. In *International Provenance and Annotation Workshop*, pages 70–77. Springer, 2008.

[93] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor Von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206. IEEE, 2007.

[94] Andrew P Davison, Michele Mattioni, Dmitry Samarkanov, and Bartosz Telenczuk. Sumatra: a toolkit for reproducible research. *Implementing reproducible research*, 57, 2014.

[95] Roberto Tyley. Bfg repo-cleaner, 2018.

[96] Jeff King. Git submodule vulnerability announced, 2018.

[97] Github Help. What is my disk quota?, 2018.

[98] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 107:841–853, 2020.

[99] Huru Hasanova, Ui-jun Baek, Mu-gon Shin, Kyunghee Cho, and Myung-Sup Kim. A survey on blockchain cybersecurity vulnerabilities and possible countermeasures. *International Journal of Network Management*, 29(2):e2060, 2019.

# Appendix A

## Publications

There have been a few publications so far from this dissertation, and a few which have been submitted and under review. Below is the list of published articles.

- **A brief Survey of Cryptocurrency Systems**, published in 14th Annual Conference on Privacy, Security, & Trust (PST), a peer reviewed IEEE Conference.
- **Provenance Threat Modelling**, published in 14th Annual Conference on Privacy, Security, & Trust (PST), a peer reviewed IEEE Conference.
- **Scrybe: A 2nd-Generation Blockchain Technology with Lightweight Mining for Secure Provenance and Related Applications**, published at MALCON 2018.
- **Flakes: A Blockchain based tool for Secure Provenance and Version Control** which is ready to submit.