Secure Password Storage on the Bitcoin Blockchain

Nina Polshakova

Contents

Abstract

Modern password managers require both security and persistent storage, but often require a user to trust third-party providers and pay maintenance costs. A blockchain-based password manager would not only provide a secure and persistent way to store credentials, but it would also eliminate the need to trust third-parties with sensitive information and only require a minimum setup cost. A password manager was built on the Bitcoin blockchain to explore the costs and trade offs of different blockchain-based storage methods.

1 Introduction

Persistent storage is the goal of any data storage system, but few systems can maintain persistent storage with minimum setup cost and zero maintenance cost. The problem of low-cost persistence is particularly applicable to password managers. Most modern password managers depend on maintaining a database of credentials on servers, the cloud or in hardware memory. This forces a client to either trust the external provider with their sensitive data or to put their trust in their own local database not getting corrupted. A trusted, incorruptible distributed data storage method would be an better alternative as a password manager than current alternatives.

The Bitcoin Network has been established to be reliable, secure and here to stay. Global acceptance of Bitcoin has increased rapidly since the digital currency first appeared in 2009. The future of the Bitcoin Network as exclusively a ledger to record transaction versus a decentralized data storage platform is still an open question. While the Bitcoin blockchain may have been envisioned as a way to record financial transactions, the applications built on top of the blockchain have promoted Bitcoin and are what ensure its relevance.

Although the Bitcoin Network is not built for data storage it has been used for storing text, links and images in the blockchain. The Bitcoin Network provides several benefits to data storage. No account is necessary for a user to create to store data in the Bitcoin Network unlike password storage with a cloud provider. Bitcoin provides a way to store data that is anonymous, persistent and incorruptible. A Blockchain-based password manager also provides permanent data storage that will be persistent over time which cannot be edited by an adversary. In addition, unlike many modern password managers that have monthly maintanence fees, a Blockchain-based password manager has zero maintenance cost. These factors support exploring if a password manager which stores credentials on the Bitcoin blockchain is feasible in terms of actual implementation, Bitcoin cost and search cost.

A report made by LastPass in 2017 shows that the average business employee must keep track of 191 passwords. A Dashlane analysis of data from more than 20,000 users in 2015 found that the average user has 90 online accounts (Lord 2018). In the United States, Dashlane found that there are an average of 130 accounts assigned to a single email address. Storing over 100 password credentials requires a password manager that is not only persistent and secure but easy to search. It also gives an estimate of how many passwords a Blockchain-based password manager should be able to handle on average. The cost of storing data on the Bitcoin blockchain depends on the Bitcoin transaction fee which has remained relatively stable over time. With a focus on the Bitcoin cost of storing data securely, different storage methods and data structures need to be analyzed when saving credentials on the blockchain.

The rest of this paper is organized as follows. Section 2 provides an overview of methods for storing data in the Bitcoin blockchain. Section 3 explores the challenges with a Bitcoin password manager. Section 4 will review existing password managers and explore existing blockchain-based and conventional password storage solutions. Section 5 will describe different data storage structures. Section 6 will look at a cost analysis of different data storage structures. Section 7 will focus on future applications and extensions of this project.

2 Methods of Storing Data in the Blockchain

Bitcoin is a cryptocurrency not designed for storing data. Data storage on the network incurs costs on the whole network forever. However in recent years there has been an increase in applications that use the

blockchain to store data. Services such as Proof of Existance certify documents on the Bitcoin Blockchain `https://www.proofofexistence.com/` or CryptoGraffiti `https://cryptograffiti.info/` which allows users to store data permanently on the blockchain by encoding it into a standard Bitcoin transaction. Other applications such as Aperatus `http://apertus.io/` store arbitrary data ranging from text to mp3 files to even videos in the blockchain, providing users with a single transaction id to recover their data.

The Bitcoin Network already has some built-in ways to store data. Coinbase data, the input of a generation transaction, is used by miners to insert ASCII strings such as short messages up to 100 bytes in size or the name of their mining pools. However, coinbase data is not often employed by users to store data. There are several ways to store data in the blockchain as a user which have different trade-offs in terms of Bitcoin cost, permanent storage and future support.

2.1 OP_RETURN

In 2014, the 0.9.0 release of Bitcoin Core added OP_RETURN as a new standard transaction type. This function allows a user to store data in the OP_RETURN script, creating a provably unspendable UTXO for a user-defined sequence of up to 80 bytes. When a transaction containing a challenge script with an OP_RETURN function is mined into a block, the accompanying byte sequence enters the block chain. P2Fk and P2FKH require miners to track them, but OP_RETURN doesn't. This method is currently used by services such as CryptoGraffiti to store data (Sward 2018).

The downside of using OP_RETURN to store data is that it is not necessarily permanent. Since the OP_RETURN data is excluded from the UTXO set, miners are not required to track this data and may not be supported in the future. A user may want to avoid storing a full node locally or query the Bitcoin network through an API query, the method of data storage should be accessible regardless of the access-method.

2.2 Pay-to-Fake-Key-Hash (P2FKH)

Pay-to-Fake-Key-Hash (P2FKH) stores the data as a public key hash in PubKeyHash field of script and creates unspendable UTXOs (unspent transaction output). This allows a user to store data in 20 bytes per transaction. This creates the most unspendable UTXO bloat as Bitcoin miners and developers have no way to know if the hash corresponds to a real public key that someone owns or not. This requires miners to keep track of these fake addresses, storing the data in the blockchain forever. This method is currently used by Apertus.io and has been used to store relatively large data such as an image of Nelson Mandela (bitFossil 2013). Unlike the OP_RETURN it also ensures that a user will be able to access the data regardless if they are using a full node or API queries to the network.

2.3 Pay-to-Fake-Key (P2FK)

The Pay-to-Fake-Key (P2FK) stores data as public key also creating an unspendable UTXO. This stores 65 bytes uncompressed or 33 bytes compressed per transaction. Storing data in an uncompressed public key is not commonly used as it has several issues. It is relatively easy for nodes to detect fake public keys, so Bitcoin miners or developers could prevent this approach. Again, this method makes it harder for nodes to detect fake compressed public keys, requiring the nodes to store the data. Using compressed public keys to store data also creates unspendable UTXOs and bloat on the blockchain.

2.4 Pay-to-Fake-Multisig (P2FMS)

The Pay-to-Fake-Multisig (P2FMS) uses a real public key along with 1 or 2 fake keys to store data. The data is store data using multisig transactions. These transactions are spendable and multiple P2FMS outputs are allowed per transaction. Each fake public key can store 65 bytes per transaction. Unlike the other P2FK or P2FKH, these transactions are spendable, so a user can avoid creating UTXO bloat. Since multiple outputs can be stored within a single transaction, a user can have the same real public key in all of them and allow for easy data recovery while decreasing UTXO bloat. Unfortunately, transactions containing a single

OP_CHECKMULTISIG need to be larger than 400 bytes so the cost in fees for spending these UTXOs is greater than the minimum non-dust values used to send them, making this economically inefficient if a user cares about UTXO bloat.

2.5 Pay-to-Script-Hash (P2SH)

Pay-to-Script-Hash (P2SH) store data in the Redeem Script or the portion of the input script before the Redeem Script. This also creates unspendable UTXOs. When creating a P2SH UTXO, a user first needs a Redeem Script, and then can apply the HASH160 algorithm to this script. A user then can spend the UTXO by creating an input script using the Redeem Script and a sequence of script operations that would make the Redeem Script true only after execution. This way data can be stored in the Reedem Script or the sequence of operations preceding the Reedem Script (Sward 2018).

3 Challenges with Bitcoin Password Manager

3.1 Anonymizing Addresses for Storage

Bitcoin is not completely anonymous and if an adversary knows a user's address, blockchain analytics can track the user's entire transaction history. In a blockchain-based password manager, an adversary that knows a user's address will have access to their encrypted credentials and will be able to track when passwords are updated. This requires a way to annonymize the user's address so that a user does not use the same address to send UTXOs every time. One solution is to use Hierarchical deterministic (HD) keys, a type of deterministic bitcoin wallet derived from a known seed. This allows the application to create child keys from the parent key and use these child keys when sending or recieving transactions. Because the child key is created from a known seed, there is an invisible relationship between the child and parent key based on the known key.

This issue is handled using the Bitcoinj library when implementing the password manager. Once a user receives of sends a transaction, a new address is automatically generated. The Bitcoinj library uses the HP protocol (BIP 32) and is able to generate a nearly infinite number of child keys using a deterministically-generated seed. Even if a child key is compromised, the parent key can still continue to be used.

3.2 Anonymizing Network Traffic

Connecting to Tor or using a VPN can prevent an adversary from analyzing network traffic and intercepting requests to search the Bitcoin Network. As some storage methods require multiple requests to the netowork to retrieve data, an adversary has the potential to reconstruct data structures from these requests. This project did not implement any built-in way to anonymize network traffic, but a user can use Tor or a VPN client when storing credentials using the blockchain-based password manager to provide additional security.

3.3 Cost of Write Operations to the Bitcoin Blockchain

The cost of the Bitcoin transaction fee should be relatively inexpensive for a user to store a password. This cost depends on different storage methods and data structure when saving credentials in the blockchain. The cost of storing credentials using a blockchain-based password manager should also be comparable in price to modern, cloud-based password managers (see Section 4.1).

Two factors determine the cost of storing data in the Bitcoin blockchain: the minimum non-dust limit and the transaction fee. Bitcoin dust is the amount of Bitcoin which is lower than the minimum amount for a valid transaction. A transaction can not include outputs that receive less than a third Bitcoin as it would take to spend it in a typical input. Currently dust limit is currently set to 546 satoshi for a P2PKH or P2SH output on Bitcoin Core with the default fee (Guide 2019).

Looking at the average Bitcoin transaction fee in the past couple years in Figure 2, the cost has been relatively stable over time. Although was a spike in average Bitcoin transaction fees in late 2017 to 55.16

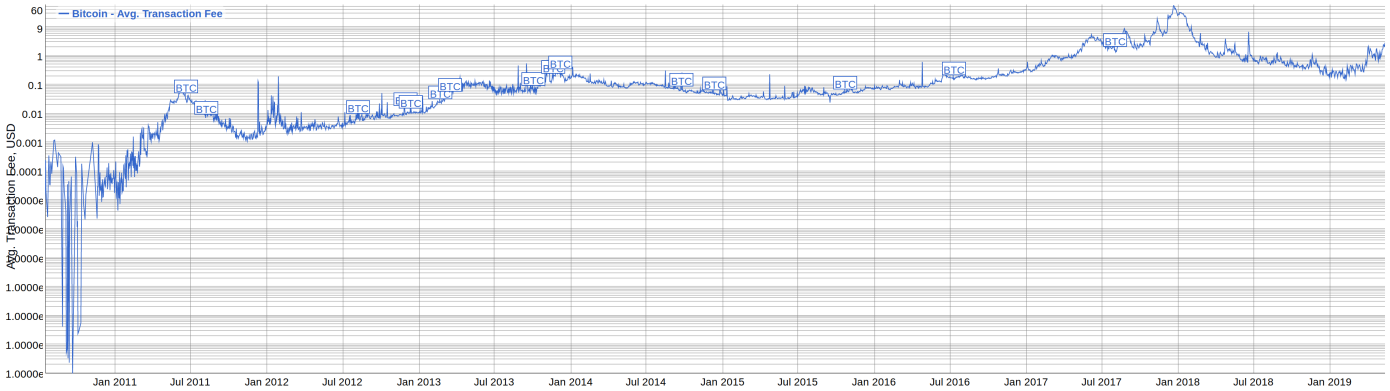

Figure 1: Average Bitcoin Transaction Fee Over All Time (Bitinfocharts n.d.)

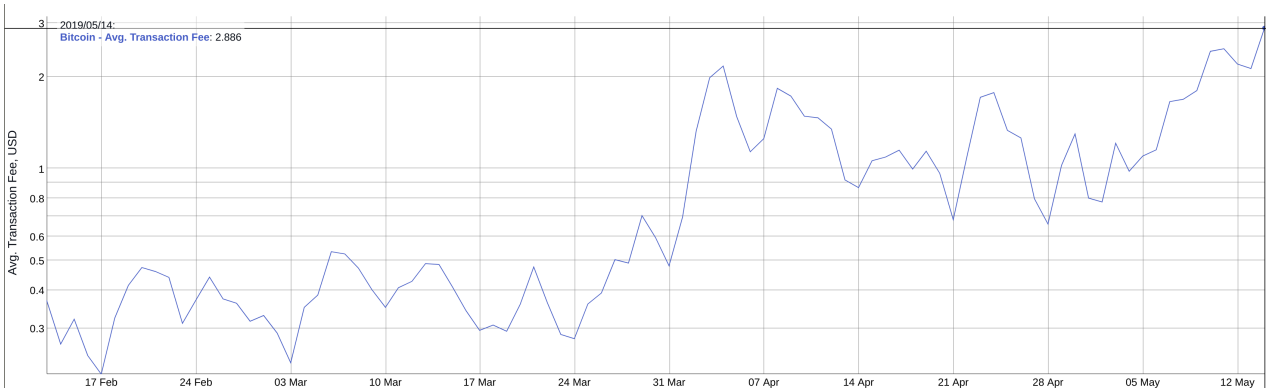

Figure 2: Average Bitcoin Transaction Fee Over Past Several Months (Bitinfocharts n.d.)

USD, the average transaction fee cost quickly reverted to pre-spike costs which on average are less than a dollar per transaction.

3.4 Intermediary Storage of Passwords Locally

Password must be stored securely locally until can be added to blockchain. The time it takes to add a new transaction to the blockchain can vary from anywhere from several minutes to a day. The Bitcoin transaction time depends on two main factors- the amount of network activity and the transaction fee. In order to ensure credentials are not lost, a temporary local storage will need to be implemented. In the current password manager implementation there is no temporary, local storage, but this is a feature for future development.

4 Review of Existing Password Managers

4.1 Cloud-Based Password Managers

Password managers such as 1Password, Dashlane and LastPass store user credentials on their own servers. These tools price ranges are comparable from 1Password being $2.99 a month to $39.99 per year for Dashlane Premium. Many of these services provide a local installation for users along with cloud-sync, allowing passwords to be accessed from multiple devices. Although these services provide the convenience of being able to access credentials from multiple devices, users are entrusting third-party services to store their credentials and are paying maintenance fees for these services. A blockchain-based password manager would still provide a user the ability to access credentials from multiple devices, but remove the need for entrusting a single third-party services or paying maintenance fees.

4.2 Local-installations Password Managers

Some applications such as Keeper, KeePassX and Password Safe provide only local-installation password managers. This does not depend on a third-party provider storing a user's passwords, but it also does not allow a user to easily access credentials from multiple devices. These applications are often free and open source, requiring no cost to the user other than providing the memory for storage. A blockchain-based password manager would still have a cost associated with storing credentials, however it would also provide the convenience of being able to access credentials from multiple devices. A blockchain-based password manager would also provide a more permanent storage solution than locally storing credentials. Locally storing credentials poses the risk of local data being corrupted and lost forever.

4.3 Review of open-source Ethereum password manager implementation:

The only existing blockchain-based password manager is an open source experimental password manager built on the Ethereum blockchain (Duong 2018). In this Etherum-based password manager, a user's login credentials are stored in an Ethereum vault. The Ethereum vault is essentially a smart contract. The blockchain-password web frontend allows users to create and edit vaults. The user provides an input vault pass phrase which gets passed through PBKDF2 to generate a hashed key, then this key uses AES from the crypto-js library to encrypt and decrypt the vault contents.

Reading passwords from the vault is free, but all writing operations (storing a new password, deployment, etc) requires a gas cost. The user creates a smart contract with the password manager and pays a gas fee. The user enters a vault password, this will be passed through PBKDF2 to generate a hash key. When a user enters credentials into the form, the password is encrypted with the crypto-js library. The the smart contract has the entered website name, login and encrypted passwords stored with the Ethereum sha-3 (keccak256) hash function.

The credentials are entered into a web form, requiring the user to trust the server not to store their credentials. The password manager can be cloned from Github and run locally to mitigate this concern. The same address is used to send all of the users' credentials. An adversary knowing the password manager's

address will have access to all the users' encrypted credentials. If an attacker knows a specific user's address in addition they will also now know when a user updates their credentials. The smart contract stores all the credentials associated with a user. To retrieve a password, the transaction message is retrieved and decrypted. "Updating" credentials for a specific domain just changes the transaction linked and requires the same gas price as adding a new password.

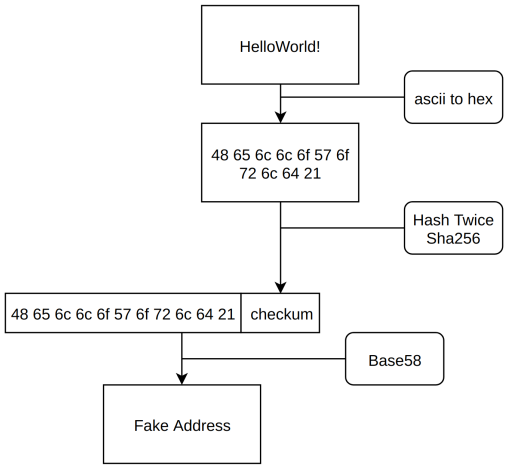

Figure 3: Process of Generating a Fake Address. Image generated by draw.io

5 Design

The Java library Bitcoinj's API was used to store a user's credentials in the blockchain using Pay-to-Fake-Key data storage. Similar to conventional password managers a user enters their credential's domain name, username and password as plaintext. This information is encrypted and labeled and then stored in the blockchain as unspendable transactions to fake addresses. Project code is available on a Github repository at: `https://github.com/npolshakova/pswd-manager`

5.1 Client-Side Storage

The client is required to store several things to recover credentials. The client needs to store the HMAC key in order to recreate the IDs to search. The client must also store one encryption key which is responsible for the encryption and decryption of the stored data. Two keys are required for the encryption scheme (see Figure 4), but hierarchical key derivation can be used to generate a second key from the key the user is already storing. Finally, to recover the data structure, the client must a Bitcoin transaction hash. This transaction hash will serve as an ID to locate the transaction with outputs of fake addresses which store the user's data. The data the user needs to store locally can be stored password-encrypted on a user's device.

5.2 Blockchain Storage

The Pay-to-Fake-Key (see Section 2) method was used in the password manager implementation to store data. This method was chosen mainly due to the multiple methods of access issue to ensure a user would be able to recover credentials regardless of how they chose to search the blockchain. Pay-to-Fake-Key also provides a slightly larger amount of data storage compared to Pay-to-Fake-Key-Hash (33 bytes vs. 20 bytes). Given an input credential string, the ascii string is turned into hex, a checksum is generated and the value is base58 encoded. From the base58 encoding a Bitcoin address is generated (See Figure 3). Data using the password manager can also be stored using the Fake Key Hash (see Section 2), but would require a higher cost. In order to store data in the fake key hash, the credential is stored in the base58 encoding itself instead of the raw key. This requires more payments to fake addresses to store data than the Pay-to-Fake-Key method.

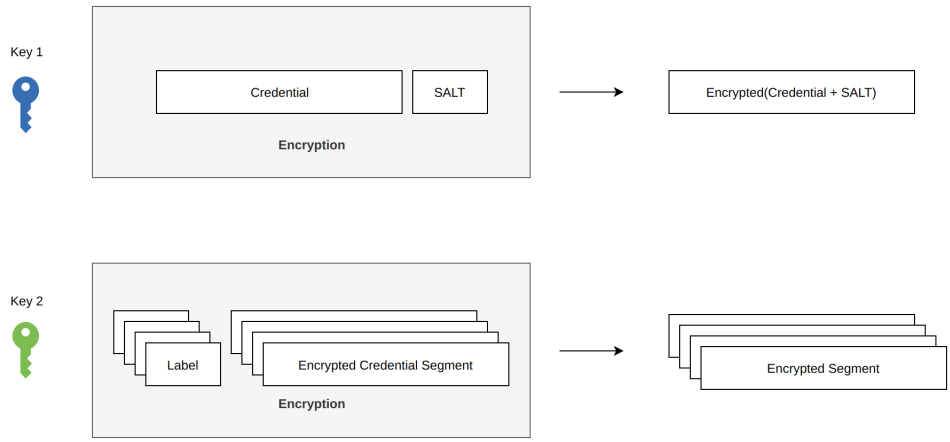

Figure 4: Encryption Scheme for Credentials. Image generated by draw.io

5.3 Data Encoding

The credential domain is hashed using HMAC to generate an ID and this ID will be stored in a transaction along with the value to identify a credential. HMAC provide a secure and compressed way to identify the credential's domain without storing the entire credential. However this requires a user to remember their domain name when recovering a credential.

Long credentials will be split into fake addresses that will get sent in the same transaction, generating a transaction hash. Based on the data structure used to store credentials, values and transaction points are encoded with labels and split to fit into the thirty-three byte address limit.

5.4 Encryption:

The Java Cryptography API allows for encrypting and decrypting data, managing keys and signing messages. The Advanced Encryption Standard (AES) from this library was used to encrypt and decrypt data. A four byte salt was generated using SecureRandom from the Java Security API and combined with the plain text credential. The salt is required to safe guard the plain text password in storage. This prevents dictionary attacks or rainbow table attacks on the credentials stored by the user by extending the length and complexity of the stored credential.

The credential and salt is then encrypted with a private key (Key 1) resulting in an encrypted message. This encrypted message was divided into seventeen character segments and combined with a label (value, child pointer, id, etc.) as described in the Data Encoding section resulting in a thirty-three byte message to store as an address. The resulting thirty-three byte data segment was encrypted using a second private key (Key 2), resulting in encrypted segments that store the identifier label and the encrypted credential piece (see Figure 4).

After a set of addresses is recovered, the addresses are decrypted using a private key (Key 2) resulting in a set of labels and their associated encrypted credential value. These encrypted credential values are reassembled and decrypted with a second private key (Key 1) resulting in the original credential and salt (see Figure 5).

5.5 Single Node Storage:

The simplest method of storing credential data is in a single transaction node. In this storage method each credential will be stored as a set of fake address in the output of one single transaction and the user will then

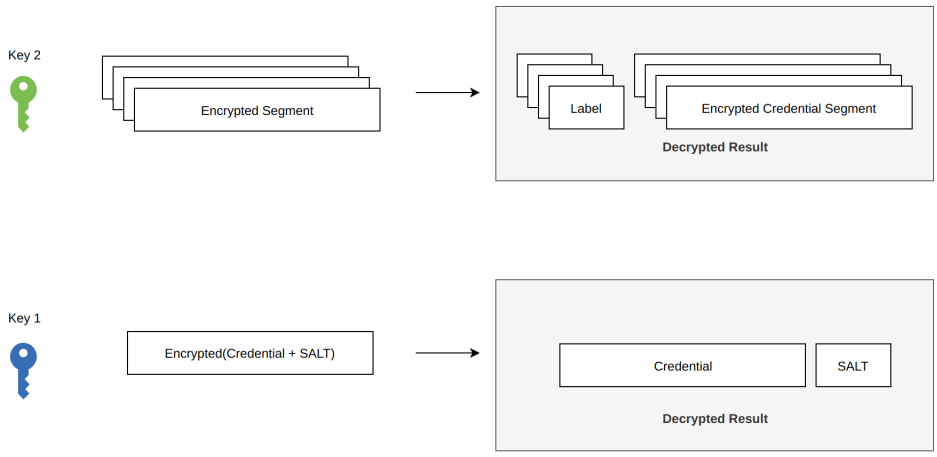

Figure 5: Decryption Scheme for Credentials. Image generated by draw.io

store this transaction hash locally to retrieve credential values. Since multiple credentials are stored in one node, each fake address needs the ID and index count of the credentials to pair and reassemble the original value. The credential is split into smaller segments to combine with the ID and count. As the number of outgoing addresses increases, the total cost will also increase as each address has a minimum dust value. Along with the total cost for sending each fake address in the node Bitcoins, there is also a transaction fee that increases as the transaction size increases.

Inserting using a Single Node data structure is computationally simple, but requires a relatively large Bitcoin cost as the node size increases. Since all credentials are stored in one node, a new transaction is generated each time a user wants to insert, update or delete a credential. This makes the cost for storing a large number of credentials increasingly expensive.

Searching the Single Node data structure also depends on the number of credentials being stored in the structure. As the number of credentials stored increases, the search time will also linearly increase. However as all credentials are stored in one transaction, this only requires one request to the Bitcoin network (see Figure 6). Once the credentials are retrieved from the Bitcoin network, the credentials are all decrypted and a specific credential is located from the node by linear search.

5.6 List Storage:

One possible improvement to the single node is to store data in a linked list made up of smaller transaction nodes. Unlike a single storage node, storing credentials in a linked list structure would link smaller nodes of credentials together, decreasing the cost to store a new credential, as well as the bitcoin fee. Since each node in the list stores a set number of credentials, the linked list structure does not incur as high transaction or fee cost as the single node when saving data.

Multiple credentials are stored in one code, which requires both the ID of the credential, an index value and the password. A node size is specified to limit the size of a node. After the max size is reached, a new node is created and a pointer to the new node is added to the old node creating a linked list (See Figure 7). The addresses store data with the same encoding as in the Single Node Storage, however an additional set of addresses are added to store the transaction hash of the next node. This transaction hash will be parsed based on the index count of the transaction shard stored in the label.

Using a linked list of transactions, the Bitcoin cost of inserting, updating and deleting can be improved significantly. Each node in the linked list represents a transaction with multiple outgoing addresses. A new credential will either be inserted into the top node of the linked list or a new node will be created, pointing

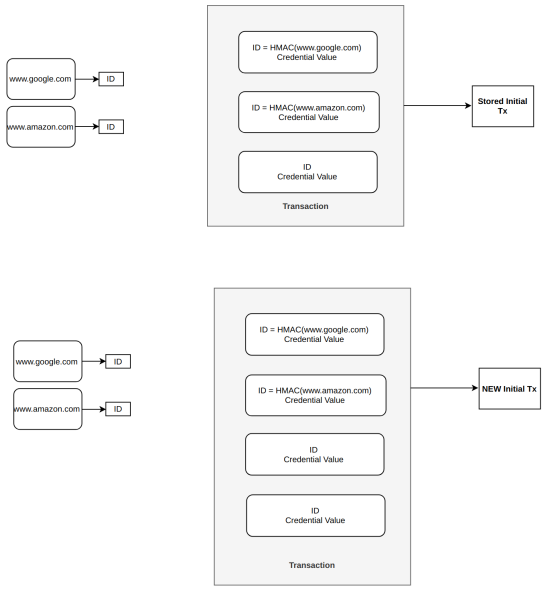

Figure 6: Single Node Storage and Insertion. Image generated by draw.io

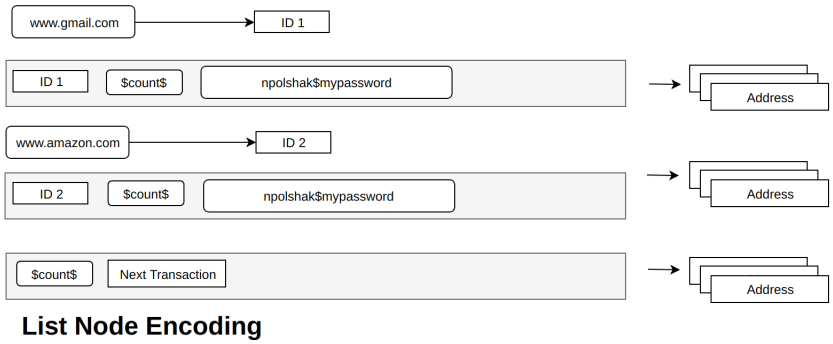

Figure 7: Linked List Encoding as Fake Addresses. Image generated by draw.io

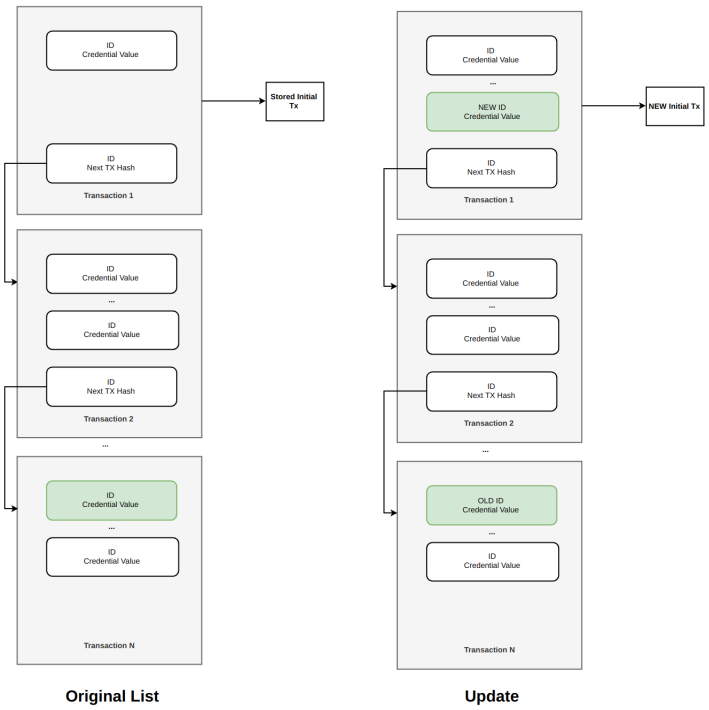

Figure 8: Linked List Storage with update. The old value still remains in the list while the new value is inserted in the top node. Image generated by draw.io

to the original head of the list. When searching the linked list, nodes are recovered one by one and searched linearly until the credential is found. This avoid recovering a large set of addresses from one transaction as in the single node case, but requires multiple requests to the Blockchain server to recover multiple transactions. Updating using a Linked List structure can be implement by inserting into the list and avoiding updates and deletions completely (see Figure 8). In the Linked List structure, credential values are never deleted. Any value inserted is permanently added to the chain of credentials and is considered valid until a more recent entry with the same ID is inserted. Since search would return the first occurrence of the credential, only one new transaction would ever be generated, lowering the Bitcoin cost. This would also store a history of all the user's past passwords.

Depending on the length of the linked list and the number of credentials stored, the search time for credentials can be computationally expensive. In addition to multiple queries for different transaction hashes, the linked list structure also never removes old credentials, potentially creating a large unused credential bloat.

5.7 Binary Tree Storage:

One data structure that could potentially improve the search cost compared to the linked list structure is a binary tree. However, storing credentials in a binary tree on the blockchain requires more addresses to save transaction hashes of child nodes, but lowers cost for insertions, updates and deletions.

In the binary tree storage design, the data is encoded by breaking credentials, transaction hashes and IDs into thirty-three bytes and if necessary padded to be the correct length. To identify the data a value tag is appended to credential value addresses, ID tag is appended to IDs and left or right tag are appended to transactions to identify the left and right children. The children of the node are represented by transaction hashes that store the node with that child node's data similar to the pointer transaction hashes used for the

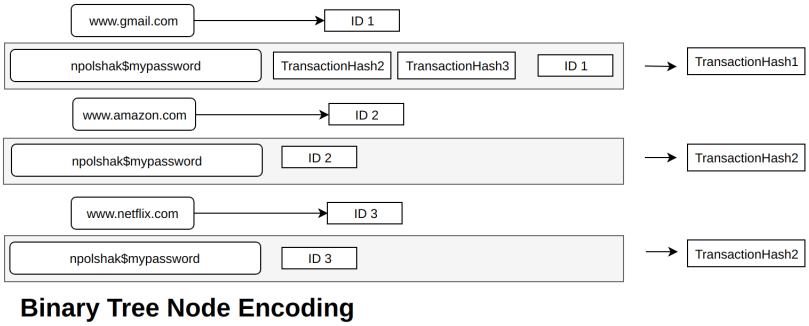

Figure 9: Binary Tree Encoding as Fake Addresses. Image generated by draw.io

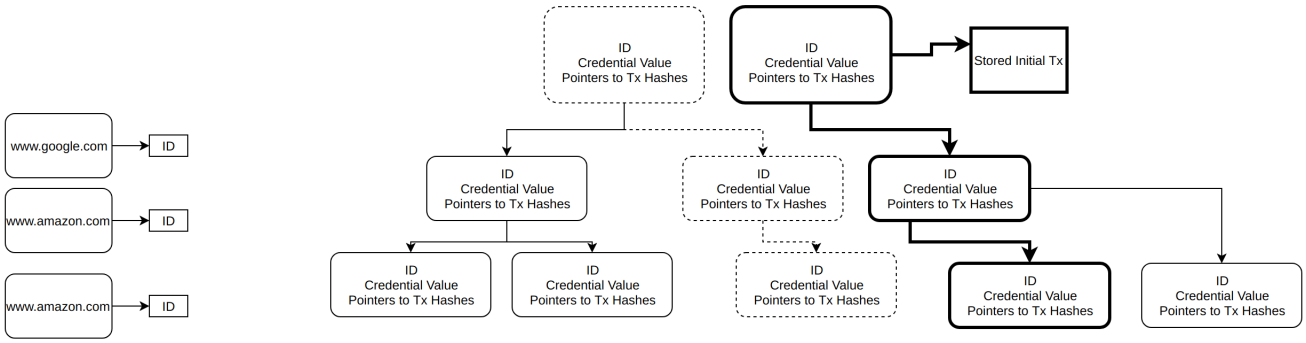

Figure 10: Inserting a Credential in Binary Tree Storage. Image generated by draw.io

linked list (See Figure 9).

Inserting a new node or deleting a node is similar to updating and also only requires a subset of transaction hashes to be updates. The binary tree is not rebalanced when inserting a new credential and instead relies on the domain addresses HMAC generating random IDs to ensure the tree is relatively balanced. When inserting a new node, the ID of the new node is compared to the binary tree's nodes. A new transaction is generated for the new node, and then the path to the new node is updated to store the transaction hash of the new node (See Figure 10).

Search time is also improved as not all transaction in the binary tree need to be recovered in order to locate a credential. Since updating a Binary Tree is logarithmic, updating a Binary Tree stored in the Bitcoin Blockchain requires updating only a subset of the transaction hashes. The value to be updated is searched for in the Binary Tree, recording the path to the node. After the node is found, a new transaction is generated for the updated node. This new transaction hash is the pointer of the node's parent, so each node in the path must be updated (See Figure 11).

Searching the Binary Tree stored in the Bitcoin Blockchain requires less requests to the Bitcoin Network compared to the linked list. The linked list stores credentials sorted by most recently added. The binary search tree stores credentials based on their ID. Although the binary tree is not rebalanced to save costs, the IDs are generated from the credential's domain HMAC, making the distribution random. This ensures the binary search tree structure will be relatively balanced.

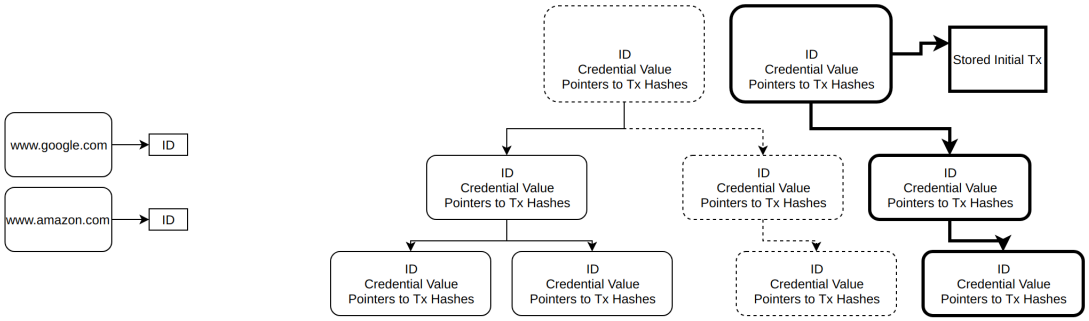

Figure 11: Updating a Credential in Binary Tree Storage Image generated by draw.io

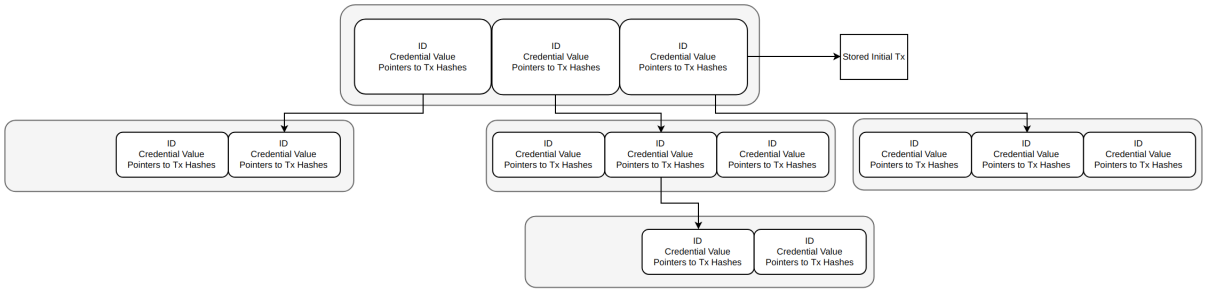

Figure 12: BTree Storage with Size Three. Image generated by draw.io

5.8 Modified B Tree Storage:

Storing credentials on the Blockchain with a modified BTree provides the benefits of a BTree allowing searches, sequential access, insertions, and deletions in logarithmic time. Each BTree node has multiple credential entries stored and multiple pointers to children nodes based on the modified B Tree size parameters. Each pointer to a child node is the transaction hash of the child node. The root of the BTree's transaction hash is stored locally to recover the tree (see Figure 12).

Similar to the binary tree, when updating, inserting or deleting a node in the modified BTree only a specific path needs to be updated, decreasing the total number of transactions needed (See Figure 13). When inserting a value, the node is first filled, and then the children are added by comparing the new credential's ID to the entry IDs in the node. The transaction hash for the new node is generated and then the path to the new node is updated with this new transaction hash.

When deleting a value from the modified BTree, the value is not removed to avoid restructuring the tree and incurring additional Bitcoin costs (See Figure 14). Instead the entry's credential value is invalidated. This avoids the cost of restructuring the tree while keeping the tree still functional for insertions, updates, deletes and search operations.

In the modified BTree tree storage design, similar to the Binary Tree design, the data is encoded by breaking credentials, transaction hashes and IDs into thirty-three bytes and if necessary padded to be the correct length. To identify the data a value tag is appended to credential value addresses, id tag is appended to IDs and children tags are appended to transactions to represent pointers to children nodes. The children

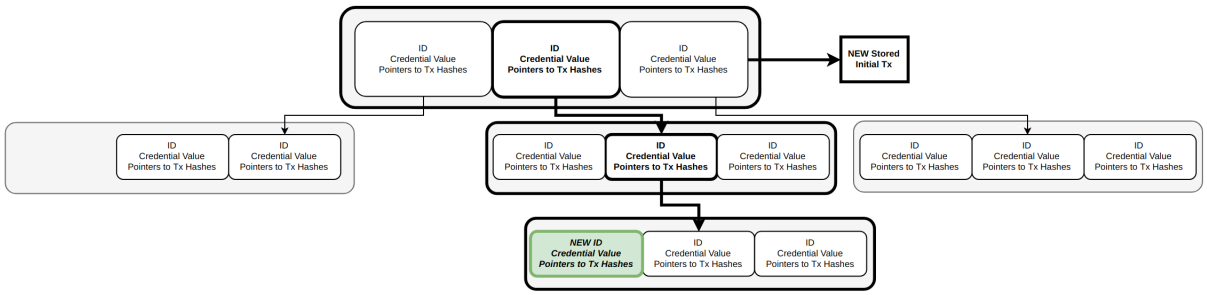

Figure 13: BTree Storage Insertion. Image generated by draw.io

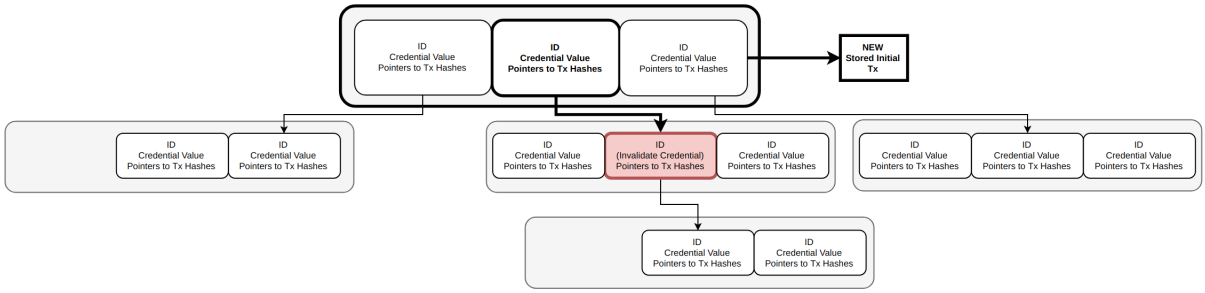

Figure 14: BTree Storage Deletion. Image generated by draw.io

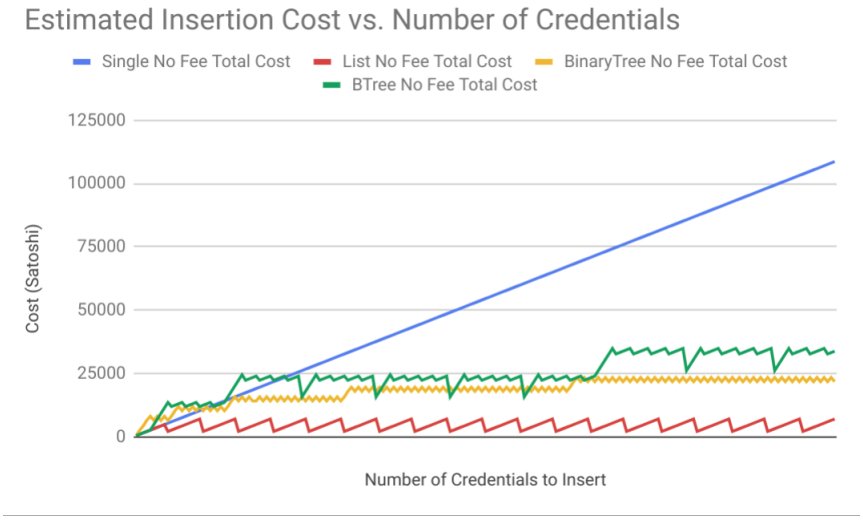

Figure 15: Additional Insertion Cost versus Number of Credentials

tag contain both the index of the transaction hash segment as well as the which entry the child belongs to.

6 Data Storage Cost Analysis:

Analyzing a set of fake credentials, the cost, time and search times were analyzed and recorded by storing data on the Bitcoin Testnet. Estimated costs were made based on the expected no-fee minimum cost for the different data structures to insert the nth value into the structure. The calculated costs were based inserting a set of fake credentials using the password manager implementation and calculating the total cost. Since Bitcoin fees fluctuate the calculated costs are not guaranteed to be using the exact same fee, but the fee for each test time was on average comparable based on data from Bitinfocharts (Bitinfocharts n.d.).

6.1 Estimated Costs

When determining the trade offs between the different data structures, the estimated Bitcoin costs can be calculated to approximate the storage cost. The estimated time was calculated based on a fixed minimum non-dust cost of 546 satoshi to store one fake address. A Linked List node size of ten was used to calculate the estimated cost for this structure and a modified BTree size of 5 was used in the estimated calculations. The estimated cost for credential storage without no fee shows that the Linked List storage method provides the lowest estimated cost (See Figure 15). Taking an approximated Bitcoin fee of 0.001BTC/kB, the estimated cost will be proportionally higher for all data storage methods, but the linked list should still perform relatively well as the Bitcoin fee will be limited based on the maximum node size. Estimating the total cost it would take to insert, the linked list still has the lowest total cost (See Figure 16).

Another metric to consider is search time when retrieving passwords. Although this operation does not have a Bitcoin cost, different storage structures have different trade offs when calculating data queries. If a user is not storing a full node locally, they will need to make API queries to the Bitcoin network to retrieve transaction information. This is a potentially slow operation and is a useful metric when considering tradeoffs between different storage structures.

Looking at Figure 17, the single node has the lowest search cost based on queries. This cost is constant since only one node is storing all the credentials. The binary tree has a high search query cost as the design of this structure stores a single credential per node. However these estimated costs are based on retrieving

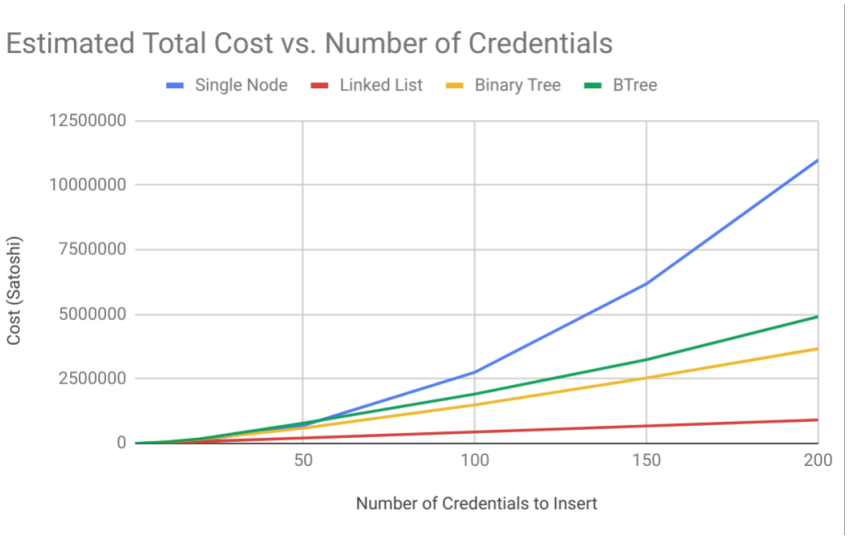

Figure 16: Sum of Insertion Cost versus Number of Credentials

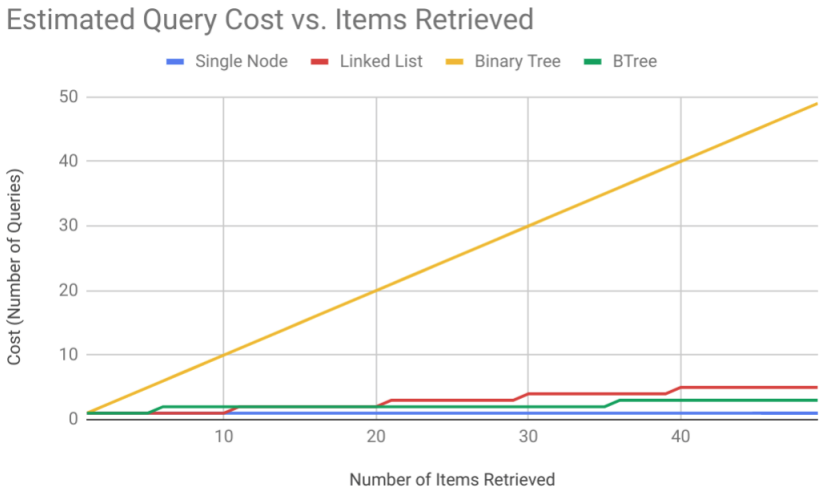

Figure 17: Search Cost versus Number of Credentials

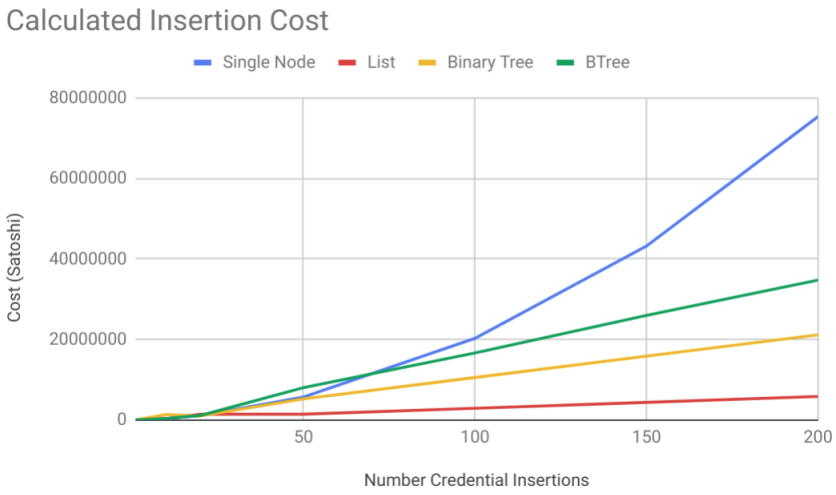

Figure 18: Calculated Cost versus Number of Credentials Stored

Number Credentials	Single Node	List	Binary Tree	BTree
1	23046	23046	41692	23046
10	394130	409030	1381752	409030
20	1154360	1472120	1059384	1169260
50	5721150	1472120	5293800	8074432
100	20307300	2944240	10587600	16668320
150	43202550	4416360	15881400	25992840
200	75341400	5888480	21175200	34765640

Figure 19: Table of Calculated Costs

n credentials. The binary tree performs better than the single node and linked list when searching for a specific credential as search would be in logarithmic time instead of linear search time.

6.2 Calculated Cost

To calculate the actual cost of storing a credentials on the Bitcoin blockchain, the Bitcoin Testnet was used to calculate the real storage costs for different structures. When calculating the estimated cost, transaction fees were ignored in the calculations and then estimated with a fixed cost of 0.001 BTC/byte. The calculated costs reflect the actual costs of inserting credentials at fixed points (1,10,20,50,100,150,200 credentials). Using the Bitcoinj Java library, fake credentials were inserted in sequence in each structure. Testnet bitcoins were generously obtained from the Bitcoin Testnet Faucet `https://testnet-faucet.mempool.co/` and the change in wallet balance for storing the fake credentials was recorded.

As expected, the single node cost was the highest as the number of insertions increased. As with the estimated cost, the linked list had the lowest calculated cost out of all the different data structures. The binary tree and modified btree calculated costs were relatively high and although they may perform better for larger quantities of credentials, this shows they are not cost-effective for this use case. The average cost of updating a credential in the linked list structure was 7,740,936 satoshi (about $443 USD as of May 5th 2019) for one hundred passwords. This cost is relatively high compared to alternative modern password managers, even though there is no maintenance fee. As discussed in Section 4, cloud based password storage services such as Dashlane Premium cost about $39.99 per year and provides an unlimited number of password storage space. The Bitcoin-based password manager would only be cost-effective in the long-term.

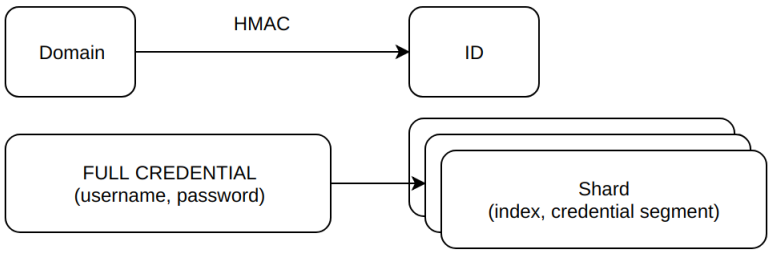

Figure 20: Shard Construction. Image made using draw.io

Assuming an average user has about 100 online accounts with unique passwords and does not change these passwords frequently, the Bitcoin password manager would only be cost effective after ten years. Alternative data storage methods should be further explored if the feasibility of using the Bitcoin blockchain to store credentials is to be seriously considered.

6.3 Mempool Limitations

When a Bitcoin transaction is transmitted to the network, it needs to first be verified by all the Bitcoin nodes available. After it is verified, it sit in a node's "Mempool" (memory pool) for unconfirmed transactions. It waits here until a miner picks it up and includes the transaction in the next block. The mempool limitation assumes the user will not be uploading their entire password manager at one time or is willing to pay a delay penalty. If the mempool gets filled, all future transactions will be rejected by the password manager, forcing the user to wait several minutes before being able to store another credential or increase the Bitcoin fee.

To avoid the Mempool limitation, a temporary storage structure for unconfirmed transactions is necessary to keep locally. This temporary structure introduces new parameters to explore to add as features to the password manager. Increasing the Bitcoin fee for a transaction will ensure it is added to the Bitcoin network faster, but this introduces a speed vs. Bitcoin cost trade off. If a user is willing to spend more Bitcoins to ensure their credentials are added to the network faster and accessible not only locally but from other devices, they might be willing to pay a higher fee cost.

7 Future Applications

Secret sharing schemes work best when storing highly sensitive or important information. Since a user wants both confidentiality and reliability when storing credentials, the secret sharing scheme address both of these issues by allow arbitrarily high levels of confidentiality and reliability to be achieved. Credentials can "shredded" and stored in multiple data structures to increase security. This approach is explored as a theoretical future extension to the Bitcoin network password manager project. A credential (domain, username, password) can be split into smaller parts called "shards" and reassembled when a user needs to retrieve a password. Instead of storing an entire credential in a single node, list or tree structure, a user can store credentials in multiple separate structures. This would involve a user storing several transaction hashes locally and a way to reassemble the credential shards.

Although this has a higher Bitcoin cost as well as higher storage cost and search time, there are arguably security benefits to shredding credentials before storing on the blockchain. Creating shards of credentials provides a method of secret sharing by distributing the credentials among separate addresses. Unlike third-party cloud-based storage providers, a blockchain-based password manager allows secret sharing to be easily implemented by the user themselves by duplicating the normal storage method of credentials described

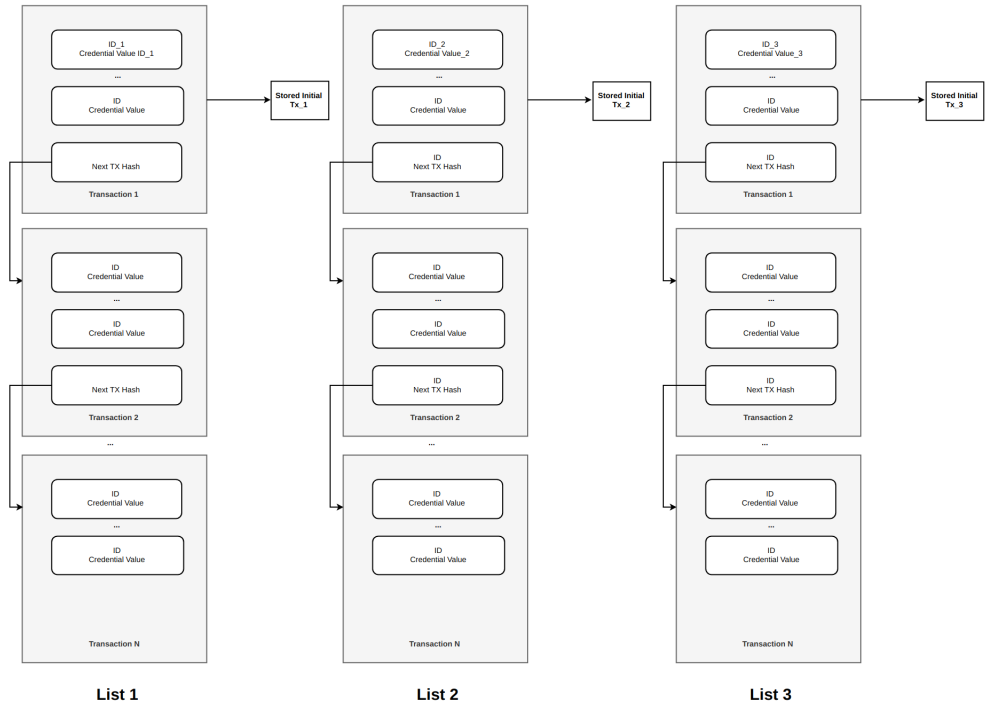

Figure 21: Multipe Linked List Storing Credential Shards

above for each "shard". Each of these shards needs to be kept highly confidential, but it is also critical that each shard should not be lost, otherwise the user loses their credential. Similar secret sharing schemes are already used in cloud computing environments where a key can be distributed over multiple serves and then reconstructed when needed.

In Figure 21, a secret sharing scheme is illustrated using the linked list structure. A credential is shredded into shards as described above. Each shard is then stored with a shard index and encrypted with the same procedure as before for the non-shared credential storage. These shards are then stored in multiple separate structures (List 1, List 2 and List 3). These separate structures result in three separate transaction hashes. All three hashes now need to be stored by the user locally, however if one list is compromised, an attacker will not be able to recover the entire credentials.

8 Discussion

As the number credentials a user has increases and the ease of access become more important, storing credentials on the blockchain provides a secure alternative to third-party cloud storage. Storing information on the Bitcoin blockchain provides distributed, persistent storage with a minimum setup cost and no maintenance cost unlike other services that charge monthly fees. In addition to no maintenance fees, an account is not necessary for a user to create to store data in the Bitcoin Network unlike most password storage with a cloud provider. A client no longer needs to trust and external provider with sensitive data or use up memory in their own local storage.

The Bitcoin Network has been established to be not only reliable and secure, but also permanent enough for a user to ensure long-term use. Although the Bitcoin Network is not built for data storage, users have been stored text, links and images in the blockchain. A password manager was built using the Bitcoinj Java library to store data in Bitcoin transactions. Analyzing the cost of different storage structures, a linked list

structure out-performed the other structures in terms of Bitcoin cost and number of search queries. However the cost for storing even one hundred passwords on the Bitcoin blockchain is still relatively high compared to alternative password manager costs. Future exploration of alternative data storage methods should be explored to see if costs can be lowered. In addition, it may be beneficial to look into using altcoins with lower transactions fees and minimum non-dust costs.

References

bitFossil (2013). URL: http://bitfossil.com/78f0e6de0ce007f4dd4a09085e649d7e354f70bc7da06d697b167f353f115b8e/

Bitinfocharts (n.d.). *Bitcoin Transaction Fees*. URL: https://bitinfocharts.com/comparison/bitcoin-transactionfees.html.

Duong, Alvis (2018). *Blockchain Password*. URL: https://github.com/Margatroid/blockchain-password.

Guide, Bitcoin Developer (2019). URL: https://bitcoin.org/en/.

Lord, Nate (2018). *Uncovering Password Habits: Are Usersfffdfffdfffd Password Security Habits Improving?* URL: https://digitalguardian.com/blog/uncovering-password-habits-are-users-password-security-habits-improving-infographic.

Sward, Andrew (2018). *Data Insertion in Bitcoinfffdfffdfffds Blockchain*. URL: https://ledgerjournal.org/ojs/index.php/ledger/article/download/101/93.