`b_verify`: Scalable Non-Equivocation for Verifiable Management of Data

by

Henry Aspegren

B.S., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

Author . . . .
Department of Electrical Engineering and Computer Science
September 1, 2018

Certified by . . . .
Neha Narula
Director of Digital Currency Initiative at the Media Lab
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

`b_verify`: Scalable Non-Equivocation for Verifiable Management of Data

by

Henry Aspegren

Submitted to the Department of Electrical Engineering and Computer Science
on September 1, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Equivocation allows attackers to present inconsistent data to users. This is not just a problem for Internet applications: the global economy relies heavily on verifiable and transferable records of property, liens, and financial securities. Equivocation involving such records has been central to multi-billion-dollar commodities frauds and systemic collapses in asset-backed securities markets. In this work we present `b_verify`, a protocol for scalable and efficient non-equivocation using Bitcoin. `b_verify` provides the abstraction of multiple independent logs of statements where each log is controlled by a cryptographic keypair and equivocation is as hard as double spending Bitcoin. Users in `b_verify` can add a statement to multiple logs atomically, even if the users do not trust each other. This abstraction can be used to build applications without requiring a trusted party. `b_verify` can implement *publicly verifiable registries*, and if no participant can double spend Bitcoin, guarantees the consistency and security of the registry. Unlike prior work, `b_verify` can scale to provide efficient non-equivocation for *millions* of applications at little cost. `b_verify` can produce *thousands of new log statements per second* at a cost of *fractions of a cent per statement*. `b_verify` accomplishes this by using an untrusted server to commit many new log statements with a single Bitcoin transaction. Users in `b_verify` maintain small proofs of non-equivocation which require them to download only kilobytes of data per day. We implemented a prototype of `b_verify` in Java and tested its ability to scale. We then built a mock registry application for tradeable commodity receipts on top of our prototype. Our initial proof-of-concept application runs on a mobile phone and can scale to a registry with *millions* of users and *tens of millions* of receipts.

Thesis Supervisor: Neha Narula
Title: Director of Digital Currency Initiative at the Media Lab

Acknowledgments

This research is the product of over a year and a half of work that has involved many different people and organizations. I was introduced to the Digital Currency Initiative (DCI) at the MIT Media Lab by Mark Weber in January of 2017. Mark showed me how damaging weak property rights and exclusionary institutions are and got me thinking about how technology might be able to help. I thank Mark for motivating much of this work and for being a fantastic research partner. I would like to thank Neha Narula for advising me and leading me through the wilderness of applied cryptography and distributed systems. This thesis would not have been possible without her. In particular I would like to thank her for shepherding this work along a fairly unconventional path. I would also like to thank the InterAmerican Development Bank (IADB) for providing funding for my research and contributing domain expertise. I hope that the IADB can use `b_verify` to make create more inclusive economies. Avery Lamp, Christina Lee and Binh Le contributed to this research as UROPS by helping to create a mock `b_verify` application. It was a pleasure to work with this exciting and fun group. Alin Tomescu, Natalie Gil, and Mykola Yerin provided thoughtful discussion that I drew on over the course of this research.

Working at the DCI has been a pleasure and I would like to thank Tadge Dryja, Robleh Ali, James Lovejoy, and Alin Dragos for being fantastic colleagues. I hope the DCI continues to help this field reach its full potential. I also hope that the 3pm union tea breaks and long debates about cryptocurrencies and politics will continue. Finally I would like to thank my friends, my parents Lucy and Lindsay, and my siblings Audrey and Charles for helping me through a challenging year. Research is hard but rewarding, and it is the people around you that make it worthwhile. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Sound protocols for managing public data are critical to building secure systems. However designing these protocols is hard because of the problem of equivocation. Equivocation allows an attacker to present inconsistent data to users. For example, in a public key infrastructure a certificate authority can *equivocate* by signing multiple certificates containing different public keys for the same user. An attacker can exploit exploit this to impersonate the user without detection or to perform a Man-In-The-Middle attack [22, 42].

Inconsistent or omitted data is a significant threat to users. In a domain name system equivocation allows an attacker to direct Internet requests to his own machine [25]. Equivocation of tor directory servers can deanonymize users [4], and in bittorrent, equivocation of the trackers can be used to control what a user downloads and from which peer [37]. Equivocation of cloud storage systems can cause applications to display incorrect information to users [12]. Equivocation is also a problem in our economic systems, where it can result in fraud and creates systemic risks. Omission of data led to billions of dollars of fraudulent lending in the commodities markets [11]. In asset-backed securities markets, a lack of transparency and consistency can create risks for the entire financial system (cite).

Unfortunately equivocation is impossible to prevent without a trusted party [15]. Instead systems can only guarantee *fork* consistency and often rely on the users to *detect* equivocation after it has happened [27, 35, 24]. Recent work has proposed using Bitcoin or other public ledgers to *prevent* equivocation [41, 10, 9]. These systems rely on a network of computers to make equivocating difficult. This approach is attractive because it does not rely on a single trusted party and uses existing infrastructure already proven to be effective [21, 9]. However the current systems are expensive, hard to use for building applications, and do not scale. For example if a thousand applications were to use Catena, a Bitcoin witnessing scheme, they would collectively consume almost half of the entire Bitcoin network's transaction capacity [41]. Furthermore building efficient applications is challenging. For example Blockstack, a decentralized domain name system based on the non-equivocation provided by Bitcoin, requires users to download and replay large amounts of data to resolve a domain name query [1].

1.2 `b_verify`

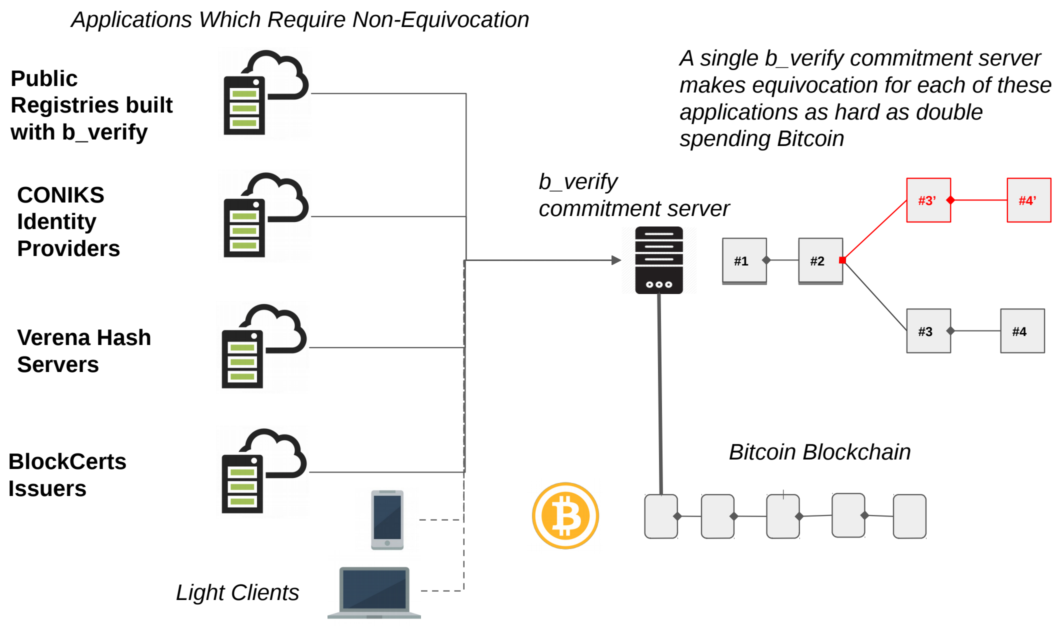
Applications Which Require Non-Equivocation
Public Registries built with b_verify
CONIKS Identity Providers
Verena Hash Servers
BlockCerts Issuers
Light Clients
A single b_verify commitment server makes equivocation for each of these applications as hard as double spending Bitcoin
b_verify commitment server
#1
#2
#3'
#4'
#3
#4
Bitcoin Blockchain

Figure 1-1: Overview of `b_verify`.

We have created a new protocol called `b_verify` for scalable non-equivocation using Bitcoin. `b_verify` provides the abstraction of multiple independent logs of statements. Each log in `b_verify` is controlled by a cryptographic public key and equivocation is as hard as double spending Bitcoin. `b_verify` allows users to add statements to multiple logs simultaneously, even if the logs are controlled by users who do not trust each other. This abstraction can support a number of different applications without the need for a trusted party.

`b_verify` scales by using an untrusted server to commit many new log statements using a single Bitcoin transaction with no loss of security. Batching increases overall throughput and lowers cost by amortizing Bitcoin transaction fees. One consequence is that users in `b_verify` must now download proofs from the server to prove they have not equivocated. This architecture requires careful design to avoid overwhelming the network bandwidth of the server. In `b_verify` the proofs for log statements are logarithmic in the total number of logs and are downloaded from the server in the form of concise updates. A single `b_verify` server can provide non-equivocation to millions of applications.

`b_verify` can be used to directly build *publicly verifiable registries*. A registry is a key/value store in which entries can only be modified by a specific set of users. A publicly verifiable registry is a registry in which the membership of an entry can be verified by anyone. Under the assumption that no participant can double spend Bitcoin, `b_verify` guarantees the consistency and security of the registry. For demonstration and testing, we have developed a registry application for trade-able commodity receipts that uses `b_verify`. This use-case was informed by consultation with the InterAmerican Development Bank and the Government of Mexico. We implemented an initial proof-of-concept application that can run on a mobile phone and scale to millions of users and tens of millions of receipts.

1.3 Overview

The contributions of this thesis are the following:

1. An open-source protocol for scalable, low-cost and efficient non-equivocation using Bitcoin.
2. An API for building applications without a trusted party.
3. A design for publicly verifiable registries on `b_verify`.
4. An example application to demonstrate and evaluate the aforementioned properties.

The remainder of the paper is structured in the following format. Related work is presented in Chapter 2. The design of the core protocol is presented in Chapter 3. Chapter 4 defines publicly verifiable registries and shows how they can be implemented with `b_verify`. An example public registry application for commodity receipts is presented in Chapter 5. The implementation of a `b_verify` prototype in Java is described in Chapter 6 and an evaluation is conducted in Chapter 7. Finally possible improvements and future work are presented in Chapter 8.

Chapter 2

Background and Related Work

This section consists of a short overview of Bitcoin to provide the background necessary to understand `b_verify`, an analysis of how `b_verify` compares to related systems and an examination of the systems which could be improved by `b_verify`.

2.1 Bitcoin Background

Bitcoin is a network of peers that run a software protocol to maintain a ledger of ownership for a digital currency [28]. The ledger consists of many individual transactions that transfer ownership of the currency. These transactions are grouped into *blocks*. Each block consists of a Merkle tree of transactions and an eighty byte *block header*. The block headers each contain the Merkle root and the hash of the previous header. The block headers form a hash chain. The combination of both the blocks and the headers is commonly referred to as the *blockchain*. Peers in Bitcoin follow the longest chain of valid blocks, which constitutes the canonical ledger of ownership for the currency. In Bitcoin, a subset of peers called *miners* compete to solve a hash-puzzle which entitles them to add a block to the chain. These peers have encouraged to add only valid blocks to the longest chain. It is argued that as long as the majority of the computational power of the network is controlled by rational actors then the canonical chain will always grow the fastest. This is critical for the security of the currency. A full security analysis of Bitcoin is left to other work [16, 19].

An interesting property of Bitcoin is that it is *permission-less* which means that any new peer can join the system and participate. To participate in Bitcoin, a peer must download and verify the entire Bitcoin blockchain, which is over 180 gigabytes at the time of this writing. However Bitcoin provides support for *Simple Payment Verification*, a process by which a peer can determine if a transaction has been included by downloading and verifying only the block headers and a Merkle inclusion proof. Under this security model, the user assumes the miners of Bitcoin will only produce valid blocks that do not contain double spends. In exchange for this weaker assumption, the user does not verify every transaction or download large amounts of data. The Bitcoin protocol has proven resistant to censorship and attack: Bitcoin consistently functions at Internet scale despite constant attack [43].

2.2 Catena

Catena is a witnessing scheme that uses Bitcoin to prevent equivocation [41]. Catena provides a similar abstraction `b_verify`, but only for single log controlled by a single user. Under the assumption that the user cannot double spend Bitcoin, a proof of non-equivocation can be downloaded directly from the Bitcoin network and verified on a mobile device using Simple Payment Verification. However Catena can be impractical to use because it is slow, expensive and does not scale. To use Catena, the application must set up and fund a Catena server. To make a statement, Catena must spend a Bitcoin transaction. Since each Catena statement requires a Bitcoin transaction, the number of applications Catena can support is fixed. Furthermore getting a transaction included quickly in Bitcoin can require a considerable fee. In January 2018, the average fee for inclusion in the next block was almost forty dollars [6]. High fees translate to increased latency and higher operation costs for applications using Catena. The more applications using Catena, the worse this problem will become.

`b_verify` improves Catena by making it scalable and cheaper without any loss of security. `b_verify` can provide a similar log abstraction to millions of separate applications for an overhead of only hundreds of bytes. `b_verify` uses batching to produce thousands of witnesses and reduces the cost of each witness to fractions of a cent. `b_verify` also provides a richer abstraction and API which supports interactions between logs. Building an application using Catena requires careful thought. For example incorporating Catena into CONIKS efficiently requires redesigning the CONIKS protocol [41]. One goal of `b_verify` is to make it easier to directly build applications that require non-equivocation. `b_verify` supports a Put/Get style interface that can be used to directly design applications such as public registries.

2.3 Open Timestamps

`b_verify` is similar in both spirit and structure to a previous system for producing scalable, trustless timestamps with Bitcoin called Open Timestamps [40]. Many applications take advantage of Bitcoin as a public notary that is hard to coerce [38, 32, 8]. Open Timestamps was motivated by the inefficiency of typical Bitcoin timestamping in which each data item is timestamped in its own Bitcoin transaction. Instead Open Timestamps uses an untrusted server to timestamp many data items as a batch with a single Bitcoin transaction at no loss of security. However unlike `b_verify`, Open Timestamps does not provide non-equivocation.

2.4 Systems That `b_verify` Can Improve

`b_verify` can be used directly by systems that require non-equivocation. For example `b_verify` can be used to prevent equivocation of the identity providers in CONIKS, a public key infrastructure [27]. CONIKS currently relies on identity providers to audit each other in order to detect equivocation. `b_verify` instead relies on Bitcoin to prevent *all* of these providers from equivocating. This design reduces the cost compared to EthIKs, a previous attempt to remove trusted parties from CONIKs using Ethereum to maintain a CONIKS log [10]. This design is also more efficient than using Catena, which would require either many servers and Bitcoin transactions to prevent each provider from equivocating or a redesign of the CONIKS protocol.

`b_verify` can be used to improve secure data management systems. `b_verify` could remove the need for a trusted party in Verena, an end-to-end data management system for building web applications [20]. Verena allows web clients to verify the correctness of queries and computations on the data. However Verena relies on a trusted hash server to store the verification objects. This hash server could be directly implemented as a public registry of verification objects in `b_verify`. However using `b_verify` would require applications to tolerate higher update latency than with a trusted server. Another challenge in using `b_verify` is that it would need to be integrated with existing web frameworks for developers to use. `b_verify` can also be used to implement any number of tamper evident log described by Crosby without requiring the users to always be online [13]. Similarly `b_verify` could be used to ensure that any number of consistency servers in SUNDR, a system for untrusted data management, do not equivocate [24]. This would prevent malicious users from colluding with the consistency server to present inconsistent versions of data.

2.5 Creating Public Records

`b_verify` could be used to improve the security of systems for creating *publicly verifiable* records. Publicly verifiable records can be verified by anyone such that these records could have legal or financial standing. For example a court in China recently used a hash to determine the creation time of a document [44]. BlockCerts is an emerging standard for creating, issuing and verifying digital credentials [9]. The goal of BlockCerts is to reduce the trust required in the issuer and to improve the verifiability of the credential by using a public ledger. BlockCerts credentials are issued in batches by creating a Merkle tree of new credentials and witnessing the root of the Merkle tree in the OP_RETURN of a Bitcoin transaction. The BlockCerts credential contains an additional Merkle proof that references the issuing Bitcoin transaction to prove when a certificate was issued. To verify a certificate, the verifier obtains the witness transaction from Bitcoin and then checks the Merkle proof. If the credential is revocable, the verifier also checks a URL link to determine if the credential has been revoked.

This scheme has a number of shortcomings. It is difficult to determine the set of all credentials an issuer has created because this would require downloading and inspecting the entire Bitcoin Blockchain to find all transactions that could possibly issue credentials. This could allow a malicious issuer to create fake credentials or multiple versions of the same credential without detection or to change the date of an old credential by issuing it again. The system for revoking a credential has even more problems. A compromised URL could simply hide a revoked certificate. Furthermore the issuer can equivocate about which certificates have been revoked. Implementing BlockCerts as a public registry on `b_verify` would solve these problems and prevent the issuer from equivocating about which certificates it has issued or revoked. This will be explored in Section 4.4.

2.6 Authenticated Data Structures

Authenticated data structures are a class of techniques in which a large data set is summarized by a smaller verification object. The verification object can be used to prove properties about the underlying data such as the result of a query or statistical computation. These proofs allow the responder to outsource data management to an untrusted party. We refer the reader to Martel [26] for a description of the standard model and to Tamassia [39] for an overview. Papamanthou [30] and Li [23] provide instructive examples for how authenticated data structures can support membership queries and aggregations on larger data sets.

The commitment server in `b_verify` uses an authenticated data structure similar to Sparse Merkle Trees [14]. Furthermore systems using `b_verify` can leverage authenticated data structures when managing larger data sets. This is done by using `b_verify` to manage verification objects so that storage of data can be outsourced. The commodity receipt application we develop with `b_verify`, discussed in Section 5, will demonstrate how this can be done.

Chapter 3

Design

3.1 System Model

In `b_verify` there are many *clients*, a single *commitment server* and *Bitcoin*. Each client produces a *log* of statements, as shown in Figure 3-1. The goal of `b_verify` is to prevent the clients from equivocating about their logs. `b_verify` accomplishes this by committing new log entries to Bitcoin. This is done efficiently through the use of a commitment server. To simplify the explanation we will model each log as controlled by a single client with a public/private key pair [1].

`b_verify` must support light clients on devices with intermittent network connections and limited storage capacity such as laptops and mobile phones. However the commitment server is assumed to have significant storage capacity, computational resources, network bandwidth and is always available. Finally both the clients and the commitment server are assumed to be able to access the Bitcoin network. All participants in `b_verify` communicate over unsecured Internet connections.

[1]Logs controlled by multiple clients can be implemented by requiring multiple signatures or by using signature aggregation schemes

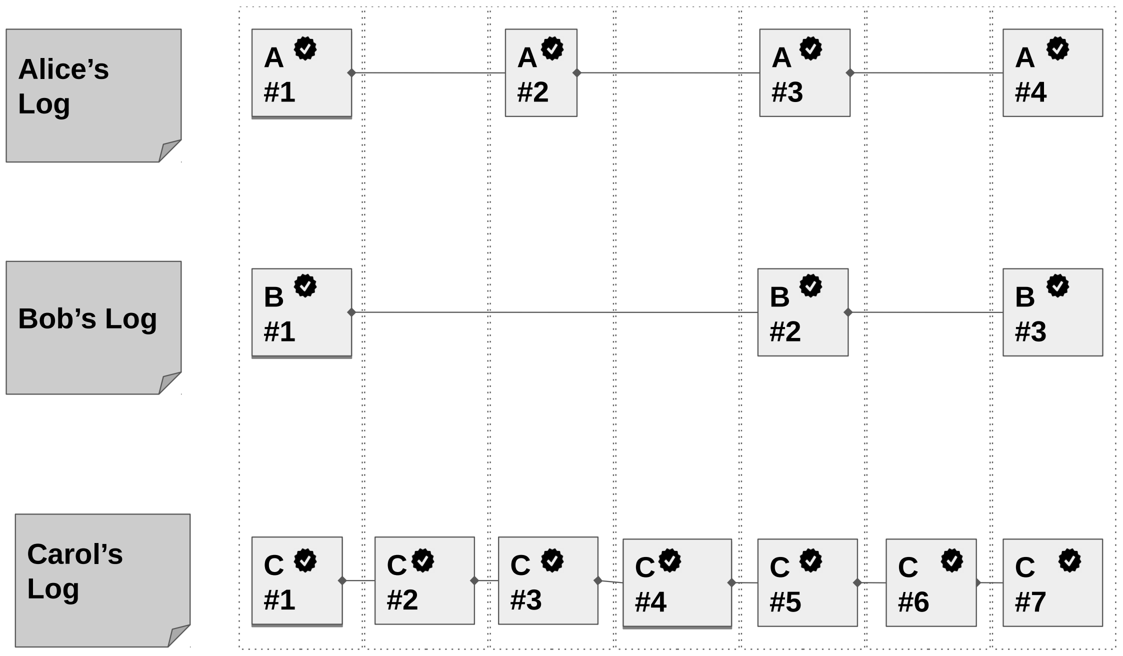

Figure 3-1: Logs of statements for Alice, Bob and Carol.

3.2 Threat Model

Clients in `b_verify` may attempt to equivocate about their log or attempt to add statements to another client's log. Clients can collude with each other or with the commitment server. However all clients are assumed to have the correct `b_verify` software and to trust the operating system on which this software is running. Other techniques, such as code signing can be used to securely distribute source code [34]. The commitment server is not trusted by the clients and can arbitrarily modify the data it stores. However we assume that no participant can collide hash functions or forge digital signatures. Additionally we make the weaker assumption that no participant can successfully execute a double-spend attack against Bitcoin.

3.3 API

`b_verify` provides the abstraction of multiple independent logs of statements that are each controlled by a single client as shown in Figure 3-1. Each client log is identified by the client's public key and can only be updated with knowledge of the corresponding private

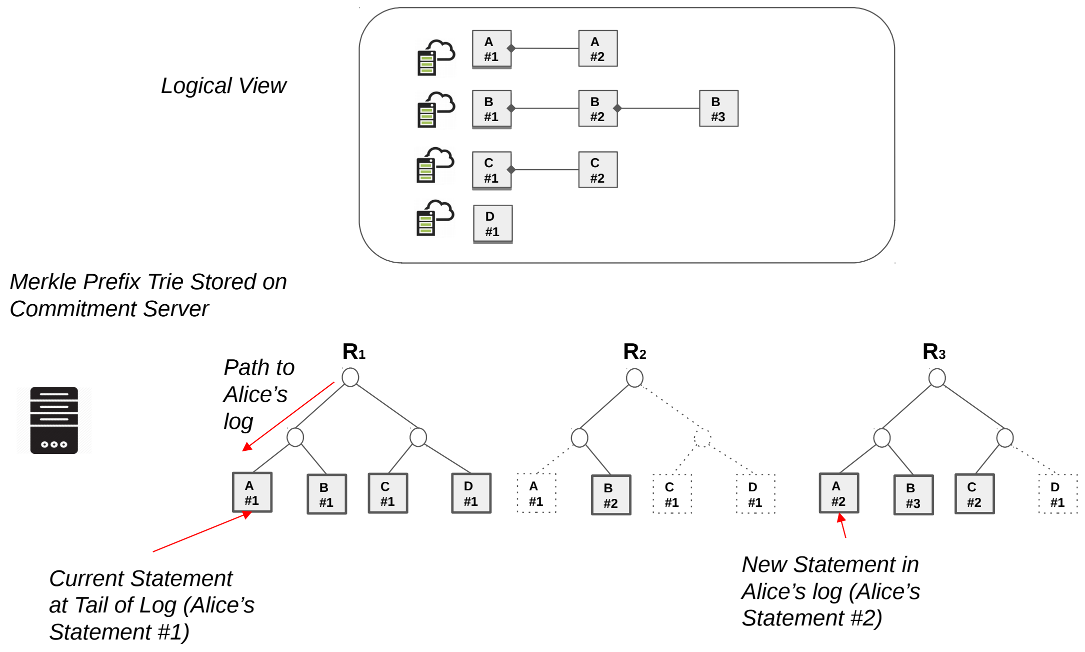
Logical View
A #1
A #2
B #1
B #2
B #3
C #1
C #2
D #1
Merkle Prefix Trie Stored on Commitment Server
Path to Alice's log
R1
R2
R3
Current Statement at Tail of Log (Alice's Statement #1)
New Statement in Alice's log (Alice's Statement #2)

Figure 3-2: Data Structures Stored on the Commitment Server. Entries that have not changed are outlined with a dashed line. The roots R_1, R_2 and R_3 are witnessed to Bitcoin.

key. `b_verify` can be described as a tuple of four methods (**CreateLog**, **AppendStmt**, **GetUpdates** and **VerifyStmt**) which are described in Table 3.1. Clients can create logs on demand by calling **CreateLog** on the commitment server with their public key. Once the log has been created, new statements can be added to the log as needed using the **AppendStmt** method. Because clients do not trust the commitment server, both of these methods return proofs which can be checked by the client using the **VerifyStmt** method to ensure that a log has been created or a statement has been added. These proofs are *publicly verifiable* which means that the correctness of the proof can be evaluated by any client. This allows the client to share a proof for his log with another client to verify its contents and non-equivocation.

New statements can be independently added to logs. The efficiency of `b_verify` comes from committing many new statements as a batch using a single Bitcoin transaction. The need for batching motivates the introduction of the commitment server. However multiplexing log updates has an interesting consequence: the clients whose logs have *not* changed must download updates to their proofs. This is done using the **GetProofUpdates** method. Clients call this method after the server has committed new updates. Note that this does not require clients to always be online. Clients can invoke this method whenever they

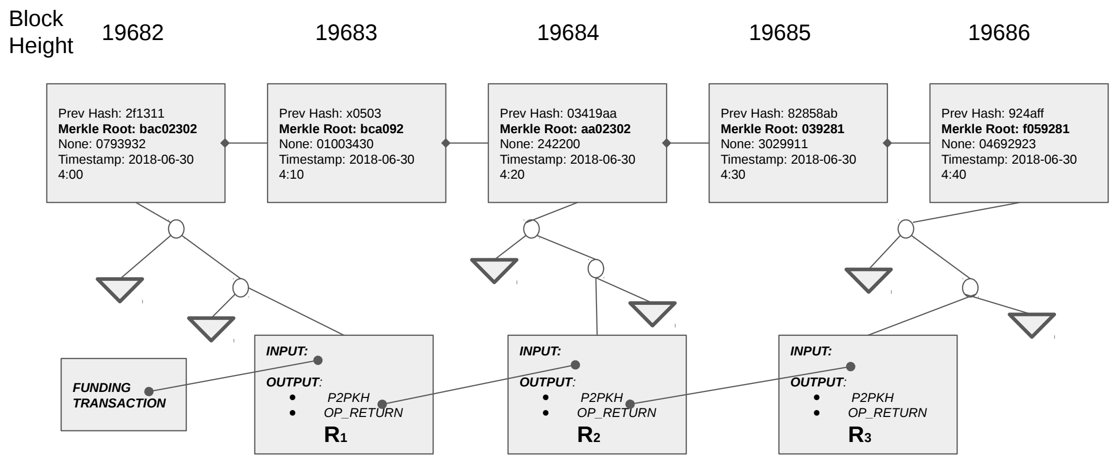

Figure 3-3: The `b_verify` witnessing scheme. The commitment server has witnessed three Merkle Prefix Trie roots R_1, R_2 and R_3 to Bitcoin. This diagram shows the data that is downloaded by a client to verify non-equivocation of the commitment server. The client must download the Bitcoin headers, the witness transactions containing roots R_1, R_2 and R_3, and the Merkle paths to these transactions from the block header.

come online without affecting the security of their log.

`b_verify` allows interaction between logs without requiring a trusted party. In `b_verify` it is possible to atomically append a new statement to multiple logs with the **MultiAppend** method [2]. Critically this method does not require any additional trust assumptions. This method can be used to support interaction between multiple logs that are part of the same application or between different applications entirely. We will show the utility of this method for building applications in Chapter 5.

3.4 Data Structures

Clients in `b_verify` maintain a proof of non-equivocation for their own log. The commitment server is responsible for committing new log statements in batches and distributing proofs. The commitment server does this by using a Merkle Prefix Trie. A Merkle Prefix

[2]This method strictly subsumes the **AppendStmt** method, but is presented separately to simplify explanation of the API.

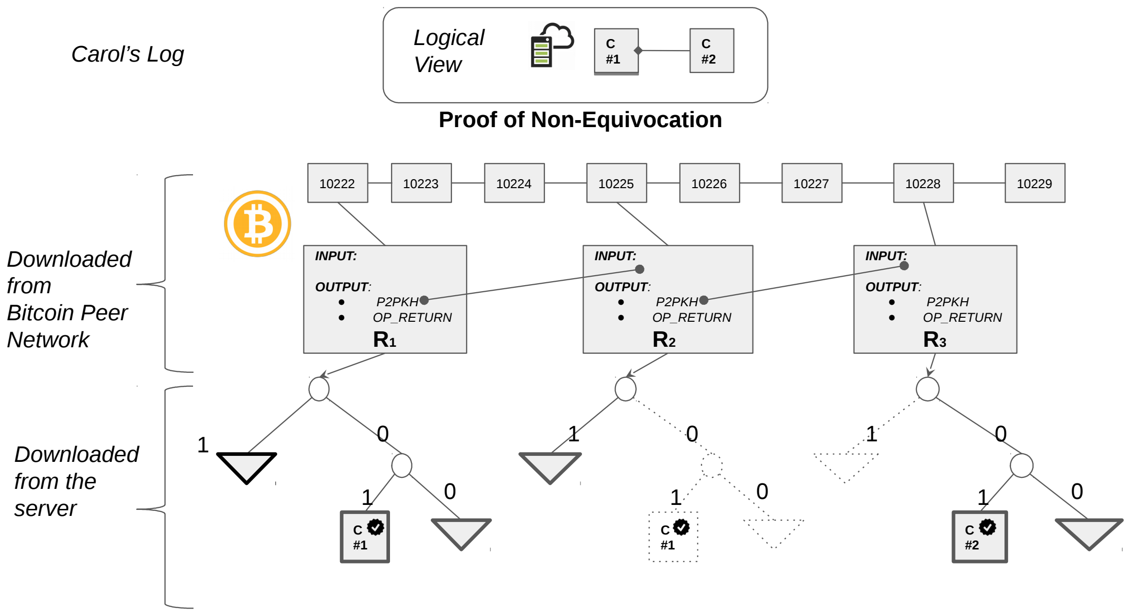

Figure 3-4: Proof of Non-Equivocation for Carol's Log. Hashes are represented by triangles. Only the portions outlined in dark borders are sent by the client directly. Repeated values are outlined with a dashed border and can be inferred when checking the proof.

Trie is a binary prefix trie that stores a map from keys to values. Each log in `b_verify` has an entry in the map with the key the hash of the log owners public key and the value the *signed* last entry in the log as is shown in Figure 3-2.

The leaves in the trie are the (key, value) pairs and each pair is stored at a location determined by interpreting the key as a path. To commit to the mappings the leaves are recursively hashed to produce a single root hash value. The changing root values are witnessed by the server to Bitcoin. To prove that an entry is in the map, one can provide a path from the leaf to the root with the pre-images for all nodes on the co-path. The proof is checked by re-hashing the nodes on the path and checking if the result matches the root hash.

This data structure was selected because it has short proofs and, most importantly, when values for only a few mappings change, these proofs can be updated efficiently. Using a Merkle Prefix Tire with N entries, the size of the proof for a single entry is a random variable with expectation $O(\log(N))$. Crucially for `b_verify`, when only a few mappings in this data structure are changed, most of the co-path nodes in the proof for each entry do not

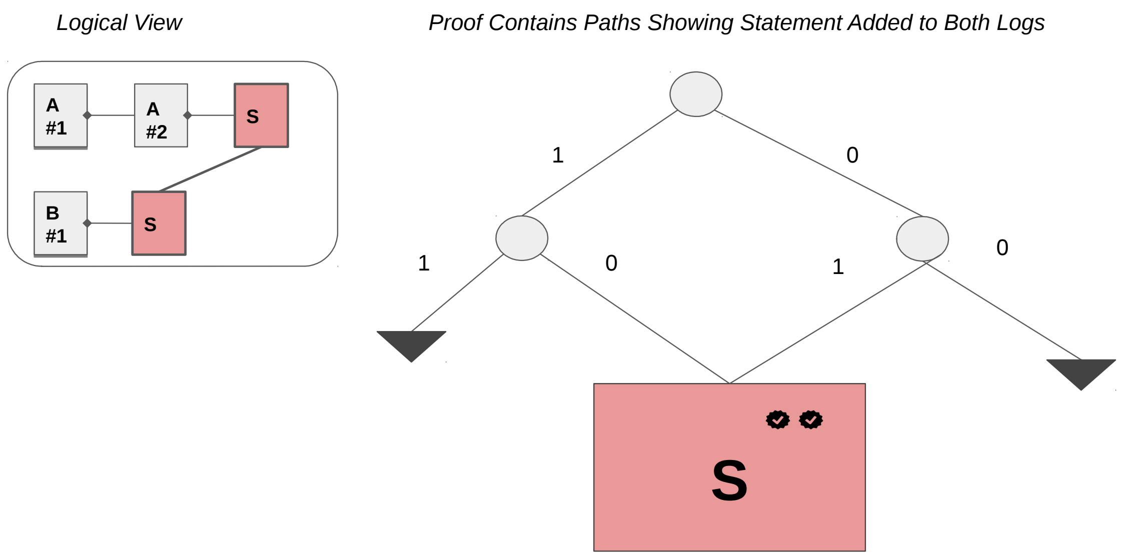

Figure 3-5: The proof returned when calling **MultiAppend** to add a statement S to the logs for Alice (public key $10\ldots$) and Bob (public key $01\ldots$)

change. For example if a single mapping is updated, $O(\log(N))$ hashes in the data structure change (in expectation), but only *one* co-path node in the proof for any given (key, value) mapping changes. If there are updates to U mappings, then updating each proof requires transmitting $O(\log U)$ hashes in expectation. If $N = \Omega(U)$ then transmitting updates to clients is asymptotically more efficient than re-transmitting the entire path.

3.5 Bitcoin Witnessing

The commitment server uses Bitcoin to witness the changing roots of the Merkle Prefix Trie. To do this the server creates a chain of Bitcoin transactions. This is shown in Figure 3-3. Each witness transaction has two outputs: a pay to the hash of a public key and an OP_RETURN. The OP_RETURN opcode is an unspendable output used to embed up to 80 bytes of data in Bitcoin without polluting the pool of unspent outputs. Each new commitment is witnessed by creating a new transaction spending the previous transaction's pay to public key hash output and including the new Merkle Prefix Trie root in the OP_RETURN. To equivocate, the commitment server would have to produce two different transactions that spend from the same input. This would correspond to double spending a Bitcoin output, which the Bitcoin protocol is designed to prevent. There is the possibility that blocks in the Bitcoin may be re-ordered (so called *reorganizations*), either due to network latency between miners or a malicious miner. However if the majority of the mining hash power is honest it is believed that this becomes exponentially less likely with time [28]. As a result clients in `b_verify` may wait for several block *confirmations* before accepting a transaction as committed.

If clients assume the server cannot double spend Bitcoin, then this witnessing scheme can be viewed as a proof of non-equivocation. Clients can retrieve the witnesses and download a proof of non-equivocation by using Bitcoin's Simple Payment Verification. To do this clients sync the block headers from peers in the Bitcoin network along with Merkle proofs for each of the witness transactions. The clients are relying on the Bitcoin network to prevent the server from equivocating by ensuring each output is spent by exactly one input.

3.6 Appending Statements to the Log

When a client wants to add a statement to the end of his log he invokes the **AppendStmt** method on the commitment server and provides the new signed statement. The commitment server checks the signature and then updates the log entry in the Merkle Prefix Trie to contain the new signed statement and re-calculates the root hash of the Merkle Prefix Trie. The server then commits to the new log entry by witnessing the new root hash in Bitcoin. The server may batch many new log statements into a single commitment for increased throughput and to amortize Bitcoin transaction costs. Batching also reduces the total number of hashes that must be re-calculated by the commitment server to update the Merkle Prefix Trie.

Since the server is not trusted, the server must also return a proof to the client that the new log entry has been committed. This proof consists of the path in the Merkle Prefix Trie to the new statement. Clients do not consider a statement committed until a new root has been witnessed in Bitcoin and the server has returned a valid proof.

The procedure for **MultiAppend** is similar, but required updates to multiple entries in the Merkle Prefix Trie. The proof returned by the server contains one path for each log, showing that the new signed statement has been added. An example is given in Figure 3-5. Clients are protected from partial application because the message they sign references *all* of the logs which must add the statement. The statement is only considered valid if it is included in *all* of the logs.

3.7 Proof of Non-Equivocation

Under the assumption that clients cannot double spend Bitcoin, forge digital signatures or collide cryptographic hash functions, clients in `b_verify` can prove that they have not equivocated about their logs. Consider the example shown in Figure 3-4 for Carol's log. Her log contains two statements $C\#1$ and $C\#2$, and the commitment server has witnessed three roots R_1, R_2 and R_3 in Bitcoin. The witnesses in Bitcoin can be considered a proof that the commitment server has not equivocated. This proof can be checked using Bitcoin SPV, as discussed in 3.5.

Showing that Carol has not equivocated about her log requires an additional proof consisting of the Merkle paths from R_1 and R_2 to her first signed log statement $C\#1$, and from R_3 to her second signed log statement $C\#2$. Note that her log did not change when the server witnessed R_2, but her proof includes a path to this witness showing that no new statement was issued. This is a necessary consequence of multiplexing logs. If the client did not include this proof, then it would be possible for the client to equivocate by simply not including entries in its log. When the server witnesses a new commitment, clients whose logs have not changed must download a proof from the commitment server using the **GetProofUpdates** method.

The Merkle paths in the proof may contain many of the same co-path nodes, as shown in the Figure 3-4 in dashed lines. This has two implications for `b_verify`. First it allows `b_verify` to reduce the size of the proofs by avoiding transmitting the same co-path pre-image multiple times. Proofs contain each pre-image exactly once and clients simply infer the pre-images for co-path nodes that have not changed. The second implication is that clients can more efficiently request proof updates from the server by transmitting only of the nodes on the client's Merkle path that have changed rather than the full path. This reduces the bandwidth requirements of the system.

3.8 Non-Repudiable Proof of Equivocation

If the commitment server equivocates then there exist two signed transactions spending from the same output with two different statements. This is a non-repudiable proof that the server has equivocated. A similar proof exists if a client equivocates about his log, but for this to occur *both* the commitment server *and* the client must collude.

However the commitment server is free to arbitrarily manipulate the data it stores. The commitment server can replay a previously signed statement in a client's log, partially apply an update to multiple logs or simply commit arbitrary data. If the server replays a log, or partially applies an update a non-repudiable proof of this behavior exists. This proof consists of the Merkle path to the partially applied update or replayed statement. However if the server commits arbitrary data, it will not be able to return any valid proofs when clients call **GetProofUpdates**. In this case the misbehavior of the server can be shown interactively.

3.9 Security Argument (Sketch)

For the commitment server to equivocate, it would need to create a different chain of transactions and successfully include this chain in Bitcoin. This requires executing a double spend attack in Bitcoin which is prohibited by assumption. The full argument and security analysis is presented in other work [41].

We now show that non-equivocation of the commitment server implies non-equivocation of each client's log. For a client to equivocate about his log he would need to produce valid proof from the same witnessed roots to different sequence of statements. This requires producing Merkle proofs from the same root hash to two different statements which requires finding a collision in a cryptographic hash function. This is prohibited by assumption. Also note that each client can only produce new statements for his log. This is because all log statements are signed, which requires knowledge of a private key. For the server or some other client to produce a new statement would require them to forge a digital signature which is prohibited by assumption.

The commitment server cannot equivocate or forge log entry proofs, but it can arbitrarily modify the data it stores. This is discussed in Section 3.8 and can be detected and proven. Other steps can be taken by an application to prevent replay attacks. These will be discussed in Chapter 5. The commitment server can however simply chose not to apply updates or to go offline. This could allow it to censor client applications. In practice we expect these risks to be mitigated by a desire to protect the reputation of the party operating the commitment server. If participants are unhappy, they can also chose to start running a new commitment server.

Function	Explanation
S.CreateLog(*pk*, *s*, *sig*) $\rightarrow$ *proof*	Invoked by a client with public key *pk* on the commitment server to initialize a new log. The client includes an initial message *s* and a signature over the message *sig*. The commitment server returns a *proof* that the log has been created
S.AppendStmt(*pk*, *s*, *sig*) $\rightarrow$ *proof*	Invoked by a client with public key *pk* on the commitment server to issue a new statement *s* at the end of its log. The client includes *sig*, its signature over the message. The server returns a *proof* that the update has been applied to the client.
S.GetProofUpdates(*pk*) $\rightarrow$ *proof_updates*	Invoked by a client with public key *pk* to get *proof_updates* for updating the proof of non-equivocation proof for her log. Clients must call this method on the server to update their proofs after the commitment server has committed new updates
C.VerifyStmt(*pk*, *s*, *i*, *proof*) $\rightarrow$ *proof*	This method is invoked on the client to verify that the client with public key *pk* has the statement s in the *i*th entry of its log. Returns true if the statement was made and false otherwise. Must be invoked in the order $i = 0, 1, 2, ...$
S.MultiAppend({*pk*, *pk'*, ...}, *s*, {*sig*, *sig'*, ...}) $\rightarrow$ *proof*	Invoked by a group of clients with public keys *pk*, *pk'*, ... to simultaneously add the statement *s*, to the end of their respective logs. All of the clients must sign. The server returns a proof that the statement has been added to all of the logs.

Table 3.1: `b_verify` API

Chapter 4

Publicly Verifiable Registries

4.1 Definition

A registry is a key/value map in which each entry is controlled by a specific set of users. For example a public key infrastructure is a public registry mapping plain-text identifiers (e.g. Google.com) to cryptographic public keys (e.g. 559436F4F4416C7AB8D21...) in which each entry can only be modified by a designated certificate authority. Registries process updates to entries and perform look-ups. It is critical that the registry is *consistent* for all users. Many systems such as Verena's hash server, bittorrent trackers, and BlockCerts issuers can be modeled as registries.

Building a registry without a trusted party is difficult and requires solving multiple challenging problems. The registry must prevent equivocation. A registry must also support *efficient* look-ups that allow a user to *verify* if an entry is in the registry. A publicly verifiable registry is one in which verification can be done by anyone. Ideally even light clients with intermittent connections should be able to verify if an entry is in the registry. This is rarely the case: for example replaying a Blockstack log to determine a name mapping cannot be done on a mobile device. Finally public registries must support updates to entries, and ideally allow users to update multiple entries simultaneously.

In this section we show how `b_verify` can be used to build a *publicly verifiable* registry with these properties. Registries built with `b_verify` support updates to multiple entries and look-ups can be verified on a light client. `b_verify`'s model of a public

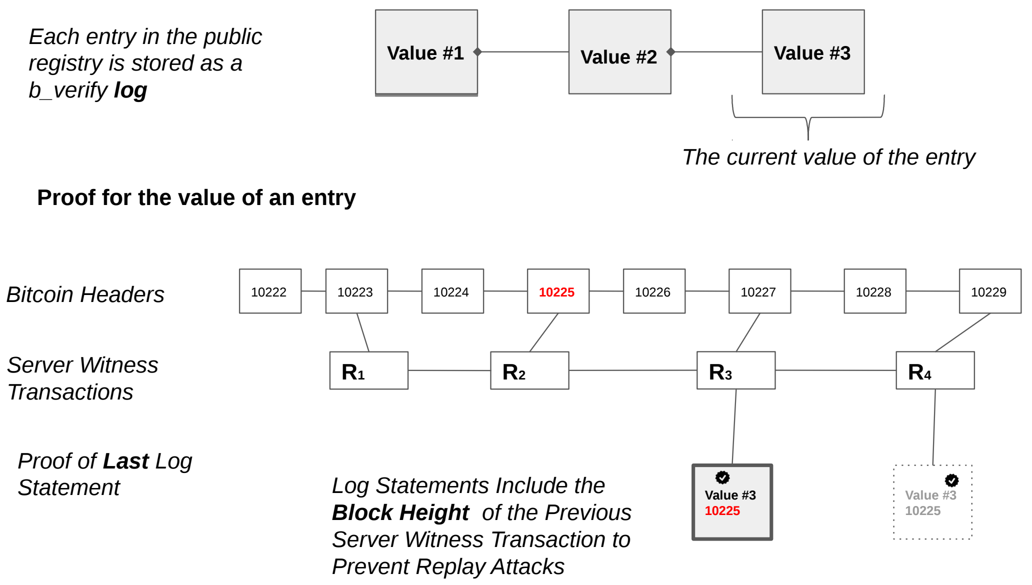
Each entry in the public registry is stored as a b_verify **log**
Value #1
Value #2
Value #3
The current value of the entry
Proof for the value of an entry
Bitcoin Headers
10222
10223
10224
10225
10226
10227
10228
10229
Server Witness Transactions
R1
R2
R3
R4
Proof of **Last** Log Statement
Log Statements Include the **Block Height** of the Previous Server Witness Transaction to Prevent Replay Attacks
Value #3 10225
Value #3 10225

Figure 4-1: Proof for the current value of an entry in the registry.

registry could be applied to any of the applications discussed previously and can also build new kinds of registries, such as the application described in Chapter 5.

4.2 Model and API

We model a registry as a map from keys to values. Each entry in the map is controlled by a set of cryptographic public keys. The registry is a pair of methods: (**Put**, **Get**). The **Put** method is used to update the values of one or more entries, and requires signatures of the public keys that control the modified entries. The **Get** method returns the value for a specific key and verifies that the value is correct. Note that in a registry users do not necessarily care about intermediate states. We assume that users only require the registry to not equivocate and to ensure that **Get** always returns the result of the latest **Put**.

4.3 Design

`b_verify` can store each entry in the registry as its own *log* controlled by a set of cryptographic public keys (the log identifier may commit to these public keys for example:

$id = H(\{pk_{Alice}, pk_{Bob}\}))$. The log contains a history of the values for the entry, with the last statement in the log as the current value. In `b_verify` the log statements must be signed by the public key(s) that control the log. However to prevent the server from replaying a previously signed entry, the log statements in a registry also include the block height of the server's previous commitment. Updates to multiple entries are implemented by adding a new statement atomically to the end of multiple logs using the **MultiAppend** method.

Clients verify that an entry is in the registry using a proof as shown in Figure 4-1. This proof has two components: first a proof that the commitment server has not equivocated, as described in Section 3.5, and second a proof that a specific statement is located at the end of the log. This second proof is just the last portion of the log proof as described in Section 3.7. Note that the proof does *not* contain all the statements in the log. The intuition for this is the following: if the client knows the server has not equivocated then no log it stores could have equivocated. Furthermore the client only cares about the *last* statement so we can omit the proofs for any previous statements.

A registry must also ensure that an attacker cannot authenticate a stale entry. In `b_verify` a malicious Bitcoin peer can simply hide new witness transactions from a light client. This could allow the client to erroneously accept an old value. To improve the freshness registries in `b_verify`, the commitment server can agree to broadcast witness transactions at a predefined interval, for example at a specific sequence of block heights. Thus the client knows which blocks should contain new witnesses and the software can alert the user if it did not receive a transaction from the peer network.

4.4 Implementing BlockCerts As A Publicly Verifiable Registry

BlockCerts, the emerging standard for creating verifiable credentials, could be implemented using a publicly verifiable registry. In this design the registry stores the credentials. Each BlockCerts issuer has an entry in the registry. The value of an issuer's entry is the set of all currently valid credentials [1]. This set will grow or shrink as new credentials are issued or revoked. To prove that a credential is authentic, a client would keep a *changing* proof that his credential is in this set by using the **Get** method of the public registry.

This scheme has a number of advantages. The issuer cannot equivocate about which credentials have been issued or revoked. Verification of a credential can be done efficiently using the **Get** method. The `b_verify` log also contains a timestamped history, which can be used to prove when a certificate was issued or revoked. Furthermore this allows a single `b_verify` commitment server to support *many* BlockCerts issuers, lowering the cost of issuing and revoking credentials. The fundamental trade-off in this design is that now clients must periodically update the proof for their credential. However this is necessary in any system with revocable certificates. Using `b_verify` is better than using a URL because certificates can be revoked securely without relying on the integrity of an external website.

[1]More precisely it would contain the *verification object* for the set of credentials, for example the root of a Merkle tree of certificates.

Chapter 5

Building New Kinds of Registries With `b_verify`

`b_verify` can be used to build public registries which require operations involving multiple users such as the transfer of a digital asset. To demonstrate this we have developed an application for issuing, redeeming and trading commodity receipts. This application does not require users to trust each other or rely on a trusted party. Our design solves several technically interesting problems that exist in current receipt management systems used in the developing world. The problem choice and design was informed by collaboration with the Inter-American Development Bank and the Government of Mexico.

5.1 Commodity Receipts

Commodity receipts, also known as warehouse receipts, are used around the world to track ownership of physical commodities such as agricultural products. For example in many countries a farmer drops his produce off at a warehouse and is issued a warehouse receipt. This receipt entitles the bearer to remove the goods from the warehouse. Warehouse receipts are frequently traded on international secondary markets and are used as collateral for loans. These records are an important part of global supply chains and tracks billions of dollars worth of goods [29, 2, 3]. Despite their importance, warehouse receipts are usually tracked using paper documents or in a centrally managed database. Unfortunately this not transparent and has led to fraud. Receipt and loan records can be manipulated, often by insiders with high level access. In one such incident, a warehouse in China was able to issue multiple loans backed by the same collateral [11]. The crucial technical problem that enabled this fraud was that the warehouse could present different sets of receipts and loans without detection.

Overall warehouse receipts have several interesting challenges that are difficult to resolve with traditional approaches.

1. The integrity and security of the data is critical: all participants must see the same records and it should be difficult to tamper with these records.

2. Participants do not trust each other and it is difficult to find a mutually trusted third party [1].

3. Participants frequently collude or equivocate.

We have designed an application for commodity receipts that addresses these problems by using `b_verify` to securely manage receipt and loan data. While the design is specific to this use case, the attributes that make warehouse receipts interesting generalize to other assets such as stock ownership or bonds.

5.2 Application Design

The warehouse receipt application we have developed allows users to securely issue, redeem, and transfer receipts. The application also allows a bank to issue loans that use the receipt as collateral. This architecture is public and permission-less - anyone can join and participate by downloading the mobile or desktop versions of the application and anyone can verify the ownership and integrity of a receipt. We assume that a public key infrastructure exists so that participants can be identified by public keys. The receipts in the application are JSON objects that contain details about the type of good, the quantity, the quality, etc. The application supports *warehouses* around the world that store goods and issue receipts

[1]This is particularly true in the developing world, which suffers from a lack of quality institutions [33].

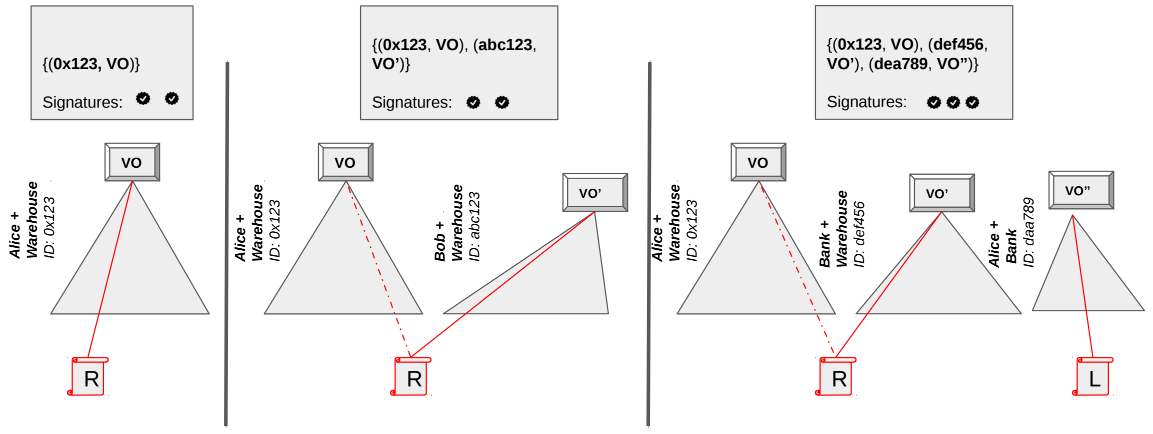
{(0x123, VO)}
Signatures:
VO
Alice + Warehouse ID: 0x123
R
{(0x123, VO), (abc123, VO')}
Signatures:
VO
Alice + Warehouse ID: 0x123
Bob + Warehouse ID: abc123
VO'
R
{(0x123, VO), (def456, VO'), (dea789, VO")}
Signatures:
VO
Alice + Warehouse ID: 0x123
Bank + Warehouse ID: def456
VO'
Alice + Bank ID: daa789
VO"
R
L

Figure 5-1: Application operations implemented using `b_verify`. In the first panel Alice is issued a receipt, R, by the Warehouse. In the second panel Alice transfers a receipt, R to Bob. Finally Alice uses a receipt, R, as collateral for a loan, L, from the Bank. The changes to the data structures are shown, with addition represented by a red solid line and removal represented by a red dashed line. The lines also represent Merkle proofs of how the data structure is changed. Finally the new signed log statement is provided at the top of each panel.

to *depositors* - the farmers, traders and small businesses interested in the goods. There are also *banks* that seek to verify the ownership of receipts and create loans against them. The application uses `b_verify` to manage all receipt and loan data, but this is hidden behind the user interface.

Receipt ownership is tracked using authenticated data structures. Each $warehouse \times depositor$ and $bank \times depositor$ is mapped to an authenticated set, a Merkle Prefix Trie, holding receipts or loans respectively. Restrictions on transferring and loaning receipts are enforced by requiring multiple signatures to update the data. The verification objects for these sets are stored in a `b_verify` public registry. This allows the verification objects to be retrieved or updated using `b_verify`'s Put/Get interface. Critically `b_verify` support simultaneous updates to multiple verification objects that are required to implement the higher level application operations. An overview of these operations is given in Figure 5-1 and the specific details are provided below.

Issuing Receipts: to issue a receipt, the warehouse employee types in the details on his laptop. Once the details are entered, the receipt is sent to the depositor's application, which is running on a mobile device. If the depositor approves, both the warehouse and depositor add the hash of the receipt to the authenticated set and compute the new verification object independently. The applications exchange signatures and update the verification object by submitting a **Put** request to the registry on the commitment server. The commitment server then updates the verification object and returns the proof to the warehouse, who shares it with the client. At this point both parties have a proof that the receipt was issued and that the depositor currently owns it.

Transferring Receipts: the transfer of a receipt involves three parties: the current owner who is sending the receipt, the new owner who is receiving the receipt, and the warehouse that has issued the receipt. During a transfer the receipt is moved from the sender's authenticated set to the recipient's authenticated set. Since the current owner and new owner do not have each other's data, they use verification objects to construct proofs about how the data will be updated. To perform the transfer, the sender must provide a proof showing that he has changed his account by removing the receipt and the recipient must provide a proof he has changed his account by adding the receipt. All parties check these proofs and then update *two* verification objects simultaneously with a **Put** request on the commitment server.

Loaning Against Receipts: to loan against a receipt, the receipt is moved from the depositor's account at the warehouse to the bank's account at the warehouse and the bank adds a record of the loan to the depositor's account at the bank. This involves changing the verification objects for three data structures simultaneously and the bank, depositor and warehouse exchange proofs showing the data structures were updated correctly. All parties then sign updated verification objects and submit a **Put** request to the commitment server. Once the update has been applied the bank has control of the receipt and the client has a record of the loan agreement. It is not possible for the client to retroactively hide the record of the loan once it has been created.

Chapter 6

Implementation

To evaluate `b_verify`, we implemented a generic prototype of the commitment server and client. We then customized this generic prototype for the warehouse receipt application. All codes and ongoing development are open source [1]

6.1 Commitment Server

We implemented a generic commitment server along with its core data structures in 3,033 lines of Java, excluding serialization code for messages and proofs. The Merkle Prefix Trie implementation uses SHA-256 for the cryptographic hash function. The prototype uses ECDSA for digital signatures on the scep256k1 curve. The witnessing for server commitments is provided as a separate library with a modified version of Catena as the default choice. However the server can optionally use other witnessing schemes for different or multiple public blockchains interchangeably. The prototype server API is exposed through Java Remote Method Invocation [31]. All proofs and update messages are serialized using code pro-grammatically generated by Google Protobufs [17]. Google Protobuf provides small serializations that can be sent and received in multiple languages without having to write serialization code manually.

[1]`www.github.com/b-verify`

6.2 Warehouse Receipt Application

The warehouse receipt application is implemented as a desktop client written in Java and as a mobile client written in Android Java. The desktop client is 3,238 lines of Java, but this includes the code required for the user interface. The android client is 35,362 lines of Java, but nearly all of this code is from the android Bitcoin wallet used as a base.

The application stores the underlying data and maintains proofs for verification objects. The applications also keep the private keys for the user which are necessary to sign updates to data. All of this is hidden by the application from the user. The desktop and mobile clients use Catnea/BitcoinJ [7] and an open source android Bitcoin wallet [5] respectively to communicate with Bitcoin and obtain the Bitcoin commitments over SPV. Android Java does not currently support Java RMI so the commitment server for this application was modified to use gRPC [18]. gRPC is an RPC framework built on top of Google Protobuf which provides convenient inter-interoperability between languages. In our deployment some clients were behind NATs, and as a result not all clients were able to connect to each other directly over IP. As a practical solution, the commitment server was modified by adding an additional API endpoint to forward application messages between clients. The commitment server for this application can optionally be configured to keep a back up of all of the underlying data. The cost of using this option is that doing so requires transmitting all application data to the server. Since clients can transmit data on a client-to-client basis it is not strictly necessary. However doing so enables the server to provide the data to clients whenever all the clients currently storing the data are off-line. Note that this does not affect the security of the system since clients already assume an unreliable network and do not trust each other.

Chapter 7

Evaluation

In this section we evaluate the design, empirically measure its performance, and present our experiences from building an application with `b_verify`.

7.1 Setup

We evaluate `b_verify` using a mock deployment of the commodity receipt application. To test the system at scale we use a deployment large enough to manage the entire commodity receipt system for all of Latin America (citation). Our deployment has 10^7 different receipts and loans, which are collectively controlled by 10^6 different users. As discussed in Chapter 5, the application uses `b_verify` as public registry of verification objects. The registry in this deployment tracks the verification objects for 10^6 different authenticated sets, which collectively contain 10^7 receipts and loans. Recall that each entry in the registry is uses its own log of statements, so this deployment has 10^6 individual logs. Modifications to verification objects are done by adding new statements to the logs.

The mock deployment used a commitment server running on an AMD Ryzen 7 1700 Eight-Core Processor with 31 gigabytes of RAM running Ubuntu 18.04. All micro benchmarks and simulations were done on the commitment serve. Micro benchmarks are based on the single shot times for an operation to complete from a cold start over 100 trials, as measured by the Java Micro Benchmarking Harness [36]. As a result micro benchmarks likely understate performance in a real system. For simulations, mock clients were created

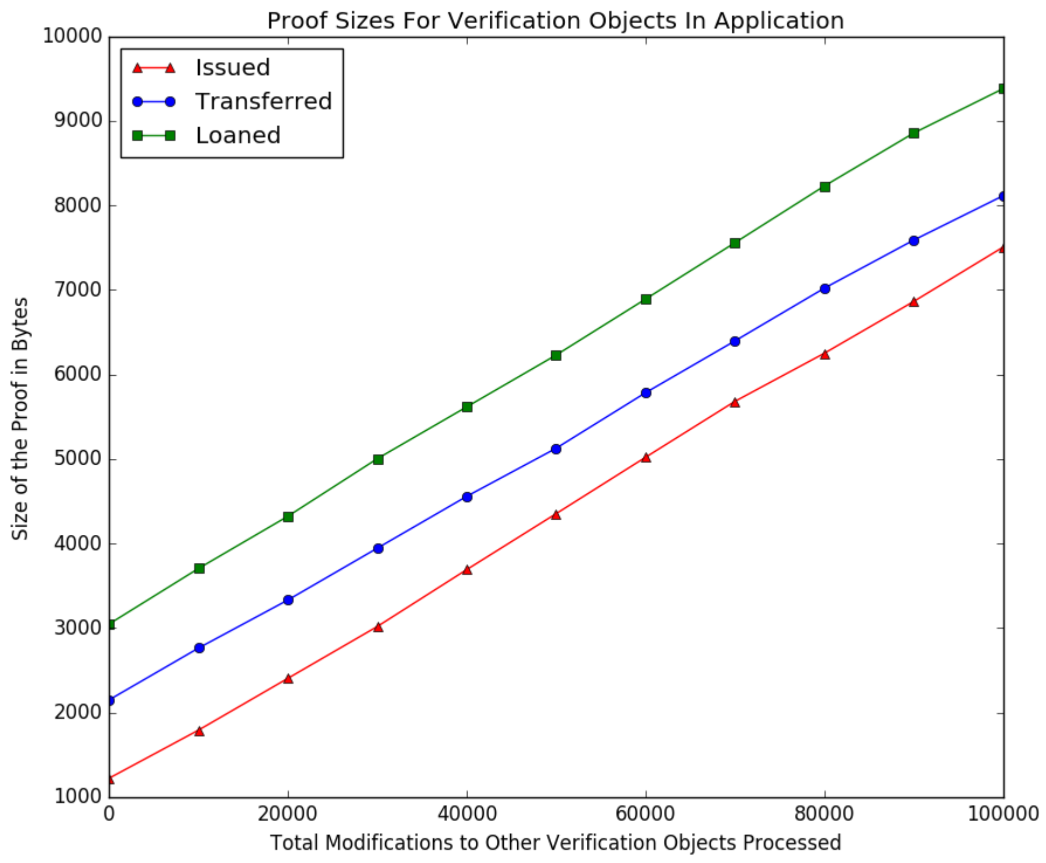

Figure 7-1: Size of the proof for the verification object in the warehouse receipt application. Transferred, issued and loaned correspond to the verification object after the respective operation.

using 500 Java threads running on the commitment server concurrently with the server application. These threads simultaneously requested thousands of operations through the server API to create heavy load and contention for resources on the server.

7.2 Proof Size

In `b_verify`, clients maintain a proof of non-equivocation for their own logs. These proofs can be shared without involving the commitment server. However the client must download an initial log proof from the commitment server and keep it up to date by periodically downloading proof updates. If the size of the proof or the updates is sufficiently

Measurement	Issued	Transferred	Loaned
Number of Verification Objects Changed	1	2	3
Number of Logs Modified	1	2	3
Number of Signatures Required	2	3	3
Size of Statement (bytes)	219	366	440
Size of Signatures (bytes)	141	213	213
Merkle Proof for Statement (bytes)	994	1774	2598

Table 7.1: Breakdown of the size of the proof for the verification object of receipts in the application. Transferred, issued and loaned correspond to the verification object after the respective action has occurred.

large then the time and bandwidth required for the commitment server to transmit this data to the clients will become a bottleneck on the entire system. For example sending a one megabyte proof to each client in this deployment would require the commitment server to send a terabyte of data.

To evaluate the demands `b_verify` places on the network we measure the size of the proof for the current verification object in the commodity receipt application [1]. Recall that the current verification object is the last statement in the log, so this proof represents the total amount of data the commitment server must transmit to the client for each statement in its log. We measure the size of this proof as the system continues to commit updates to other verification objects, as would be the case in a real deployment. Recall that to update a verification object the server has to commit a new statement to the log.

The proof sizes for the verification objects are shown in Figure 7-1 and a breakdown is given in Table 7.1. As shown in the figure, the proof becomes larger with time. This is because as log statements are added to other logs, the client must download a proof update to show it has not added a new statement to its log. The amount of data that the client downloads depends on which other logs are updated. We plot the average size of the proof update as a function of the number of other logs updated in Figure 7-2. If the commitment server adds ten thousand log statements to other logs, the average proof update is only a couple hundred bytes.

Note that the proof for the transferred receipt's verification object in Figure 7-1 is larger

[1]Note that proving ownership of a specific receipt requires an additional proof using the verification object, but we do not include this in our analysis because the size of this proof is determined by the authenticated data structure and not the design of `b_verify`.

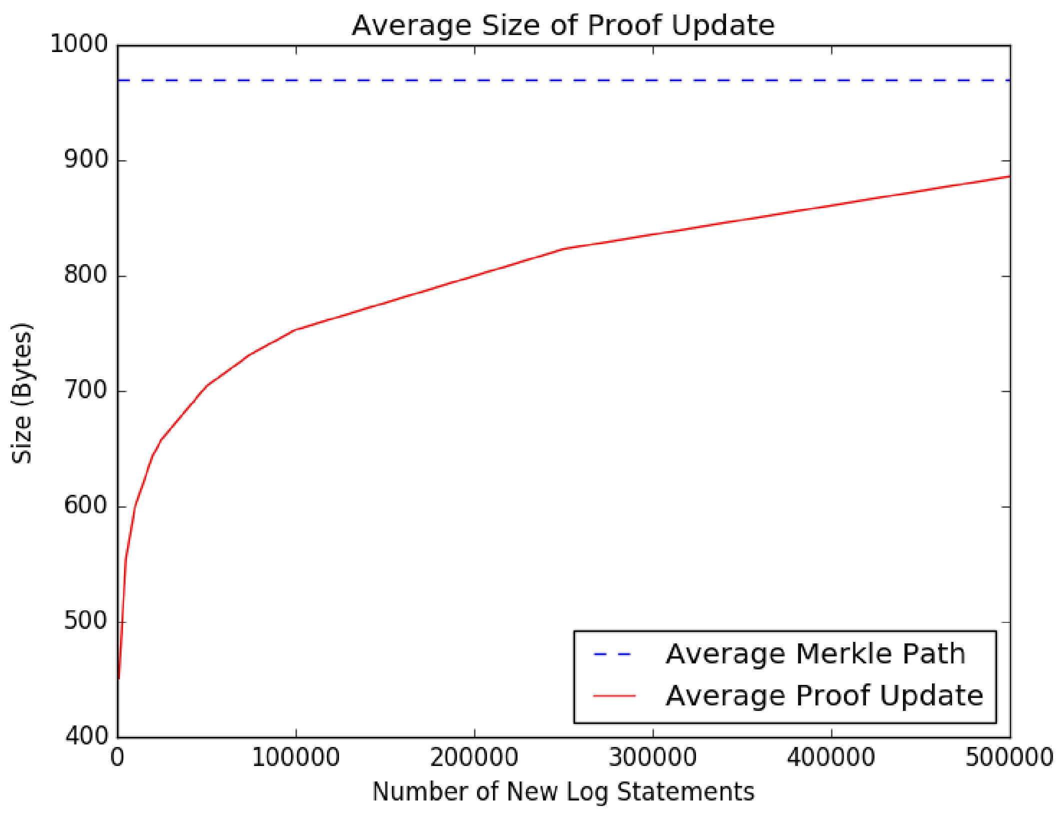

Figure 7-2: Average size of proof updates that must be downloaded by each client in the warehouse receipt application. The average size of a full Merkle path in the Merkle Prefix Trie is provided for comparison purposes.

than for an issued receipt's verification object. This is because to transfer a receipt, a statement must be added to two logs, one to the sender's log and one to the recipient's log whereas to issue a receipt a statement only needs to be added to a single log. In general `b_verify` allows for the same statement to be simultaneously added to as many logs as desired, but adding the same statement to more logs will increase the size of the proof for that statement.

If a log is not updated, the proof for the last statement becomes dominated by the portion that must be added after each commitment. To make this as small as possible, we only include the value of the nodes on the co-path that have changed. To measure the impact of this optimization we compare the total bandwidth needed to transmit the proof with and without it.

The results are shown in Figure 7-3. The optimization reduces the size of the proof and

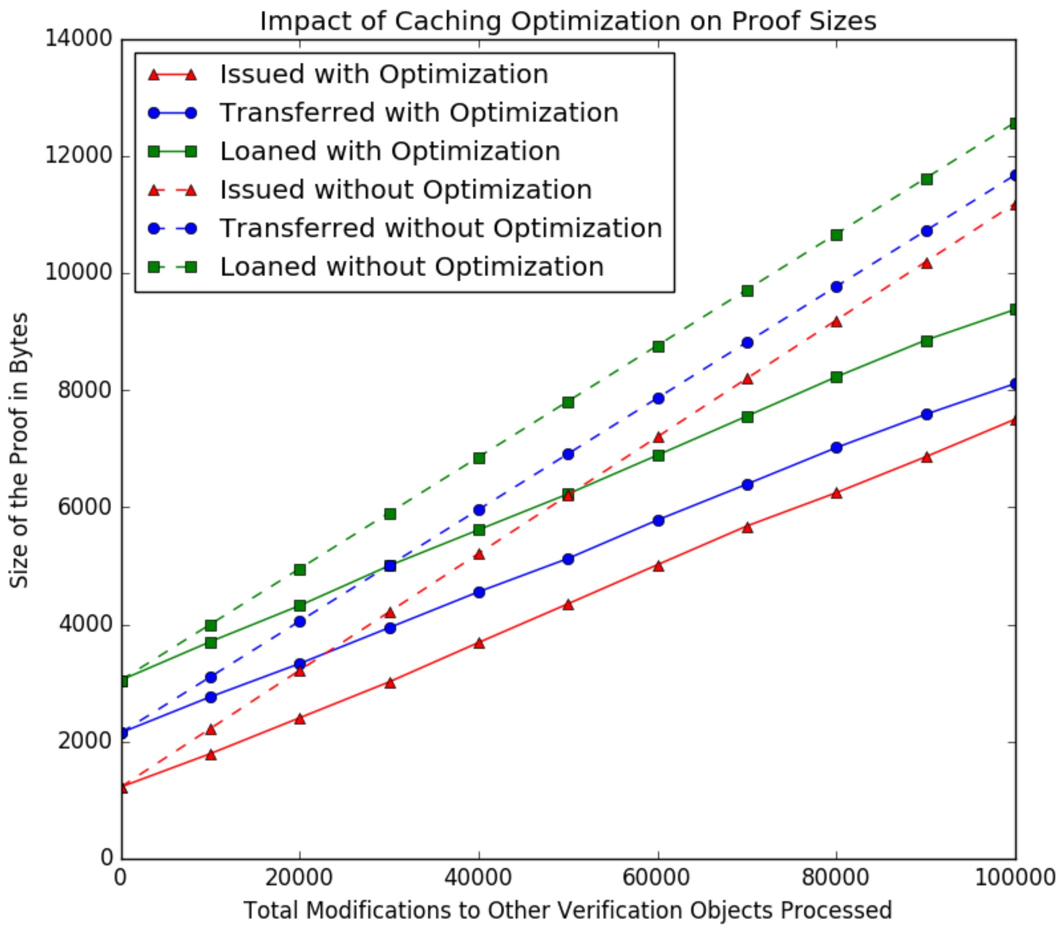

Figure 7-3: Comparison of proof size using full Merkle paths versus omitting repeated nodes.

the bandwidth required by nearly 50%.

7.3 Throughput

In `b_verify` the commitment server must be able to handle many concurrent requests to commit new log statements and to produce proofs. The commitment server must do this quickly to avoid becoming a bottleneck on the entire system. We first evaluate the commitment server via micro benchmarks of the limiting code paths and then use simulations of a server under heavy client load to get a measurement of throughput under realistic circumstances.

Operation	Time (Milliseconds)	Std
Check Two Signatures	4.730	0.587
Single MPT Update	0.026	0.010
Batch Commitment	12.205	4.142
Proof Updates Generation	0.528	0.447
Full Proof Generation	2.382	4.053

Table 7.2: Micro benchmarks of the commitment server. The first group of operations are the steps to commit a new log statement and the next two groups are generations of proofs, which are there own operations.

To commit a new log statement, the server must determine the owner(s) of the log, verify that the statement has been signed by the required parties (which can involves multiple signature checks if the log is controlled by more than one client). If the statement is signed then the commitment server updates the log entry in the Merkle Prefix Trie and schedules it to be committed. To commit, the server must re-calculate the hashes that have changed in the Merkle Prefix Trie. Since updates are typically committed as a batch, the commitment micro benchmark is the time required to issue new statements for 10^4 logs or equivalently to update 1% of all logs. In the micro benchmark, committing this batch requires re-calculation of $12,312$ of the $2,885,976$ total SHA-256 hashes in the Merkle Prefix Trie.

The server also needs to generate two types of proofs: complete proofs of non-equivocation for a log and periodic proof updates. For proof updates we select a log at random and then add statements to 1% of the other logs. The micro benchmark is the time required to calculate and send the proof updates for the selected log. For the complete proofs we select a log at random and then add statements to 10% of the remaining logs in 10 batches of equal size. The micro benchmark is the time required to calculate the entire proof of non-equivocation for the selected log.

The micro benchmarks are given in Table 7.2. These results indicate that the critical path for committing a new log statement is dominated by signature verification and re-calculation of the Merkle root to be witnessed in Bitcoin. From these benchmarks we can see that proof generation is relatively fast and can be done in milliseconds. Observe that checking signatures and generating proofs can be done parallel. The current implementation parallelizes both of these operations to get higher overall throughput.

Simulation	Time (Seconds)	Average Throughput (Operations/Second)
Commit New Log Statements	97	1,112
Generate Non-Equivocation Proofs for Logs	56	17,770

Table 7.3: Simulation of commitment server facing heavy load. In these simulations large number of clients request operations on the server simultaneously. The test measures the amount of time required to respond to client requests, and the average throughput of the commitment server in performing these operations.

To measure server throughput under heavy load we use simulations. First we simulate lots of requests to commit new log statements and measure overall throughput. In this simulation, mock clients submit 5×10^4 requests to issue receipts and 5×10^4 requests to transfer receipts, both of which require committing new log statements. We measure the time required for the server to verify the request, perform and commit the new statements and create proofs for the clients. In the second simulation we simulate lots of clients requesting proofs from servers. In this simulation we update 10% of the logs in 10 batches of equal size. After committing the new statements mock clients request a proof for each of the logs stored on the server, requiring the sever to generate 10^6 proofs in total.

The results of the simulations are shown in Table 7.3. The throughput of committing new log statements in the simulation is higher than the micro benchmark would suggest, because signature verification is parallelized. However the throughput of proof generation is slower than the single threaded micro benchmark. This is probably due to contention for resources and locks. Overall the simulation results indicate that the server can process thousands of new log statements and create tens of thousands of proofs per second while under load and contention for resources.

7.4 Latency

The latency of the warehouse receipt application was measured and the results are given in Table 7.4. To start using `b_verify` the client must first obtain the server commitments from Bitcoin. This requires downloading the chain of Bitcoin block headers, the transactions

Operation Description	Time
Initial Application Starting Time	10-20 minutes
Verify Ownership of a Receipt (Read)	*block generation time* + 5 milliseconds
Changing Ownership of a Receipt (Write)	*block generation time* + 8 milliseconds

Table 7.4: Latency measurements from the warehouse receipt application. These measurements were collected using Bitcoin's Testnet and *block generation time* represents the time from the time the operation was requested until the next block is generated by the network.

witnessing the commitments, and Merkle proofs showing that these transactions were included. To verify the server commitments the client must check the work done on the headers and check the inclusion proofs. At the time of this writing this involves downloading 40 MB worth of data and computing around six hundred thousand SHA-256 hashes. Using BitcoinJ's SPV implementation, this process takes several minutes, but can be comfortably done on a mobile device. After downloading and verifying the data, everything except for the server commitments and the last few headers may be discarded. At this point the client only needs to verify new server transactions which can be done incrementally whenever the client comes online.

The read latency experienced in the commodity receipt application is determined by the time required to check that a verification object is correct. This involves sending the verification object along with the proof over the network. The client must then check the proof using the server commitments obtained from Bitcoin. Checking the proof requires verifying several signatures and calculating dozens of SHA-256 hashes. This can be done quickly. Clients may also cache verification objects and proofs to speed this up. Once a client has the correct verification object, the time to actually check a specific receipt depends on the underlying authenticated data structure. In our experience read latency in the application was small enough to avoid impacting user experience.

The update latency in the commodity receipt application is the time required to issue, transfer, loan or redeem receipts. This is primarily determined by the time the commitment server must spend waiting for a new Bitcoin transaction to be included. In Bitcoin, blocks are mined probabilistically with a target average of one block per ten minutes. Applications that require additional security may choose to wait several additional blocks or confirmations before considering an update to have committed. Thus even if the commitment server witnesses a new root every block, the latency of `b_verify` is substantial. Applications using `b_verify` must be able to tolerate high update latency. One practical approach to dealing with latency is to have the application display the results of unconfirmed transactions, but graphically mark this information as still tentative to inform the user. This was the approach taken by our warehouse receipt application.

7.5 Cost

The server must pay high fees to get its transactions included in the next Bitcoin block. Currently these fees are several dollars. If the demand for Bitcoin transactions or the price of Bitcoin increases then these transactions will become more expensive in dollar terms. However note that in `b_verify`, a batch of many statements is committed with a single Bitcoin transaction so the cost is amortized. Therefore in `b_verify` most log statements will cost only thousandths of a cent. This is a small price to pay for removing the need for a trusted party. In practice we expect the server operator to fund the Bitcoin transactions or to charge the clients for new statements.

7.6 Fault Tolerance

If the commitment server goes offline, clients will not be able to commit new log statements. However if clients have fresh proofs, these proofs are still valid and can continue to be used. If the server has produced a new commitment, but has not yet provided proofs then clients may only have stale proofs. Resolving this situation is more challenging. If the server has become permanently unavailable then the clients can come together and pool their logs to create valid proofs by reconstructing the Merkle Prefix Trie stored on the server. Unfortunately if a client simply chooses to not participate then reconstructing the proofs becomes computationally impossible. Applications using `b_verify` can choose to run multiple shadow commitment servers that maintain replicas of the commitment server's data structures for fault tolerance.

7.7 Privacy

All application data in `b_verify` is assumed to be public and strict privacy was not a design goal. The protocol does not provide any guarantees about the privacy of the data and applications must use other techniques if stronger privacy guarantees are needed. However the system does provide some limited obfuscation of the underlying data that is probably an acceptable level of privacy for many applications. For example in the warehouse receipt application the log statements contain verification objects which do not reveal the underlying receipts. In this application the commitment server does not need to be aware of any of the actual data. This is a desirable form of weak privacy for users and reduces the legal risks for the operator of the server.

Chapter 8

Conclusion

8.1 Future Work

There are many aspects of the design, implementation and practicality of `b_verify` that can be improved. The current implementation uses BitcoinJ which has not been optimized for `b_verify`. Future work could reduce the latency of `b_verify` by optimizing its interaction with Bitcoin, perhaps using new RPC calls introduced to the Bitcoin protocol. One barrier to using `b_verify` to prevent equivocation in existing applications is the difficulty of integrating it. Future work could implement `b_verify` as a part of commonly used frameworks or libraries. For example, `b_verify` could be implemented as an in-browser web extension that is extensible and shared between applications. Doing this would also amortize the overhead of using `b_verify` across many applications.

Currently `b_verify` has only been deployed on Bitcoin. We selected Bitcoin because it is the oldest and in our opinion the most secure cryptocurrency. Additionally Bitcoin's SPV can be run on a mobile phone, allowing us to support light clients. `b_verify` however could be adapted to other distributed ledgers. This would make the system more flexible and allow it to exploit the different properties of these ledgers.

Another area for future work is figuring out what new kinds of applications can be built with `b_verify`. As we have shown, efficient non-equivocation can be used to build applications that do not require a trusted party. Our commodity receipt application is just one example. Future work needs to be done to explore what other applications can be built with `b_verify`.

8.2 Recap

Many different systems require non-equivocation for security. `b_verify` makes it easy and inexpensive for applications to prevent equivocation by using Bitcoin. By solving this hard problem, `b_verify` enables new applications which do not require a trusted party. Our design makes it easy to build these applications by providing an extensible API.

`b_verify` contributes to a growing body of research into how distributed public ledgers can be used. In this thesis we have shown how `b_verify` can provide a base protocol for building new applications and can be used to improve the security of existing systems for managing public data.

Bibliography

[1] Muneeb Ali, Jude C Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *USENIX Annual Technical Conference*, pages 181–194, 2016.

[2] World Bank. How warehouse receipts can improve lives. `https://www.ifc.org/wps/wcm/connect/news_ext_content/ifc_external_corporate_site/news+and+events/news/how+warehouse+receipts+can+improve+lives`. Accessed 2018-07-11.

[3] World Bank. Project appraisal document on a proposed loan in the amount of usd 120 million to the united mexican states for the grain storage and information for agricultural competitiveness project. `http://documents.worldbank.org/curated/en/263681490714537272/text/Mexico-PAD-main-03072017.txt`, March 2017. Accessed 2018-07-11.

[4] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Low-resource routing attacks against tor. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pages 11–20. ACM, 2007.

[5] Bitcoin wallet for android. `https://github.com/bitcoin-wallet/bitcoin-wallet`. Accessed: 2018-06-30.

[6] Bitcoin transaction fees. `https://bitcoinfees.info/`. Accessed: 2018-07-08.

[7] Bitcoinj: Java bitcoin library. `https://bitcoinj.github.io/`. Accessed: 2018-06-30.

[8] Bitcoin notary. `https://notary.bitcoin.com/`. Accessed 2018-07-11.

[9] Blockcerts. `https://www.blockcerts.org/`. Accessed 2018-07-08.

[10] Joseph Bonneau. Ethiks: Using ethereum to audit a coniks key transparency log. In *International Conference on Financial Cryptography and Data Security*, pages 95–105. Springer, 2016.

[11] Manila Bulletin. After port fraud, china's vast warehouse sector under scrutiny, Jun 2014.

[12] Christian Cachin, Idit Keidar, and Alexander Shraer. Trusting the cloud. *Acm Sigact News*, 40(2):81–86, 2009.

[13] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.

[14] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees. In *Nordic Conference on Secure IT Systems*, pages 199–215. Springer, 2016.

[15] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[16] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[17] Google protobuf. `https://github.com/google/protobuf`. Accessed: 2018-06-30.

[18] Google rpc. `https://grpc.io/`. Accessed: 2018-06-30.

[19] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium*, pages 129–144, 2015.

[20] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 895–913. IEEE, 2016.

[21] Keybase. `https://keybase.io/`. Accessed 2018-07-08.

[22] Neal Leavitt. Internet security under attack: The undermining of digital certificates. *Computer*, 44(12):17–20, 2011.

[23] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for aggregation queries. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):32, 2010.

[24] Jinyuan Li, Maxwell N Krohn, David Mazieres, and Dennis E Shasha. Secure untrusted data repository (sundr). In *OSDI*, volume 4, pages 9–9, 2004.

[25] Andreas Loibl and J Naab. Namecoin. *namecoin. info*, 2014.

[26] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[27] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. Coniks: Bringing key transparency to end users. In *USENIX Security Symposium*, volume 2015, pages 383–398, 2015.

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[29] Gideon Onumah. Improving access to rural finance through regulated warehouse receipt systems in africa. In *United States Agency for International Development–World council of credit unions conference on paving the way forward for rural finance: an international conference on best practices. Washington, DC, June*, pages 2–4, 2003.

[30] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 437–448. ACM, 2008.

[31] Esmond Pitt and Kathy McNiff. *Java. rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[32] Proof of existence. `https://poex.io/`. Accessed 2018-07-11.

[33] Dani Rodrik, Arvind Subramanian, and Francesco Trebbi. Institutions rule: the primacy of institutions over geography and integration in economic development. *Journal of economic growth*, 9(2):131–165, 2004.

[34] Aviel D Rubin. Secure distribution of electronic documents in a hostile environment. *Computer Communications*, 18(6):429–434, 1995.

[35] Mark Dermot Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*, 2014.

[36] A Shipilev. Openjdk jmh project. *URL https://web. archive. org/web/20160119005244/http://openjdk. java. net/projects/code-tools/jmh*, 2016.

[37] Jung Ki So. *Defending against Malicious Behaviors in BitTorrent Systems*. PhD thesis, North Carolina State University, 2012.

[38] Entrust the blockchain to notarize proof of ownership of any digital creation. `https://stampd.io/`. Accessed 2018-07-11.

[39] Roberto Tamassia. Authenticated data structures. In *European Symposium on Algorithms*, pages 2–5. Springer, 2003.

[40] Peter Todd. Opentimestamps: Scalable, trust-minimized, distributed timestamping with bitcoin. `https://petertodd.org/2016/opentimestamps-announcement`, 2016. Accessed: 2018-06-30.

[41] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via bitcoin. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 393–409. IEEE, 2017.

[42] Fake turkish site certs create threat of bogus google sites. `https://www.cnet.com/news/fake-turkish-site-certs-create-threat-of-bogus-google-sites/`. Accessed: 2018-07-02.

[43] Marie Vasek, Micah Thornton, and Tyler Moore. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In *International Conference on Financial Cryptography and Data Security*, pages 57–71. Springer, 2014.

[44] Wolfie Zhao. Blockchain can legally authenticate evidence, chinese judge rules. *Coin Desk*, July 2018.