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 . . . .
Katrina LaCurts
Chair, 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 new protocol for scalable and efficient non-equivocation using Bitcoin. `b_verify` provides the abstraction of multiple independent logs of statements in which each log is controlled by a cryptographic keypair and makes equivocating about the log as hard as double spending Bitcoin. Clients in `b_verify` can add a statement to multiple logs atomically, even if clients do not trust each other. This abstraction can be used to build applications without requiring a central trusted party. `b_verify` can implement a *publicly verifiable registry* and, under the assumption that no participant can double spend Bitcoin, guarantees the security of the registry. Unlike prior work, `b_verify` can scale to *one million application logs* and commit *1,112 new log statements per second*. `b_verify` accomplishes this by using an untrusted server to commit *one hundred thousand new log statements with a single Bitcoin transaction* which dramatically reduces the cost per statement. Users in `b_verify` maintain proofs of non-equivocation which are comparable in size to a Bitcoin SPV proof and require them to download only kilobytes of data per day. We implemented a prototype of `b_verify` in Java to demonstrate its ability to scale. We then built a registry application proof-of-concept for tradeable commodity receipts on top of our prototype. The client application runs on a mobile phone and can scale to one million users and ten million 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 Inter-American Development Bank (IADB) for providing funding for my research and contributing domain expertise. I hope that the IADB can use `b_verify` to 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 is the deliberate presentation of inconsistent data by a participant within a system. 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 this to impersonate the user without detection or to perform a Man-In-The-Middle attack [1, 2].

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 [3]. Equivocation of tor directory servers can tenancies users [4], and in bittorrent, equivocation of the trackers can be used to control what a user downloads and from which peer [5]. Equivocation of cloud storage systems can cause applications to display incorrect information to users [6]. 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 [7]. In asset-backed securities markets, a lack of transparency and consistency can create structural risks for the entire financial system [8].

Many systems rely on the users to detect equivocation after it has happened [9–11]. However this is not ideal because it may not stop an attack in time and places more burdens on users. Recent work has proposed using Bitcoin or other public ledgers to prevent equivocation [12–14]. This approach is attractive because Bitcoin has proven to be reliable and resistant to attack [15]. Furthermore Bitcoin relies on a network of computers with no central trusted party. Unfortunately the current systems for using Bitcoin as a trusted party 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 [12]. 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 [16].

1.2 `b_verify`

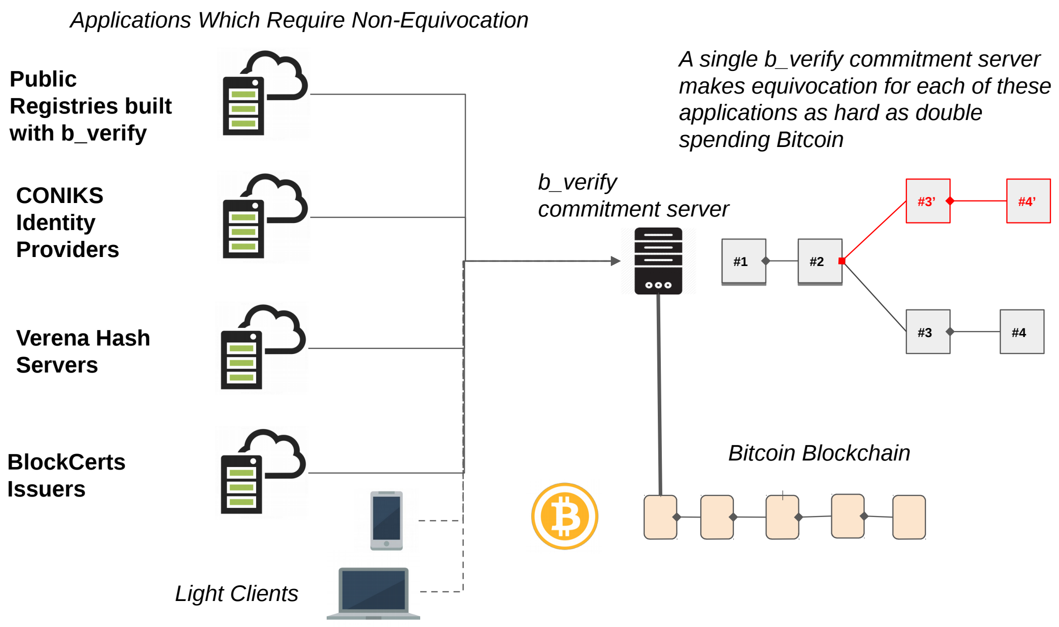

Applications Which Require Non-Equivocation
Public Registries built with b_verify
CONIKS Identity Providers
Verena Hash Servers
BlockCerts Issuers
Light Clients
A single b_verify commitment server makes equivocation for each of these applications as hard as double spending Bitcoin
b_verify commitment 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. Log statements in `b_verify` can be arbitrary bytes. 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 atomically, even if the logs are controlled by users who do not trust each other. `b_verify` allows users to verify the content and non-equivocation and of a log. However `b_verify` does not perform any other verification of log statement data. This must be done by the applications that use `b_verify`. In this thesis we show how this protocol can support a number of different applications without requiring a central party that is trusted by all participants.

`b_verify` scales by using an untrusted server to commit many new log statements using a single Bitcoin transaction. Batching increases overall throughput and lowers cost by amortizing Bitcoin transaction fees. One consequence is that users in `b_verify` must now download additional data 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 proof for a new log statement is logarithmic in the total number of application logs. These proofs are downloaded from the server in the form of concise updates. A single `b_verify` server can provide non-equivocation to millions of application logs.

`b_verify` can be used to directly build *publicly verifiable registries*. A registry is a key/value store in which each key can only be modified by a specific set of users. A publicly verifiable registry is a registry in which the value of a key 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 tradeable commodity receipts that uses `b_verify`. This use-case was informed by consultation with the Inter-American 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 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 [9]. CONIKS currently relies on identity providers to audit each other in order to detect equivocation. `b_verify` instead use Bitcoin to *efficiently prevent all* of these providers from equivocating.

`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 [17]. `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 [18]. 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 [11]. This would prevent malicious users from colluding with the consistency server to present inconsistent versions of data. Finally `b_verify` could be used to improve the security of systems for creating *publicly verifiable* digital credentials. This will be discussed explicitly in Section 4.4.

`b_verify` however cannot determine the correctness of the data in log statements directly. `b_verify` only guarantees that all parties will agree on the contents and non-equivocation of the log. Applications using `b_verify` must have other mechanisms to ensure that log data is correct.

1.4 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 the aforementioned properties.
5. An evaluation showing a single `b_verify` server can scale to a million logs, process over a thousand log statements per second while only requiring clients to download

kilobytes of data per day.

This thesis is structured as follows: we review the background necessary to understand `b_verify` in Chapter 2. We describe the design of the core protocol in Chapter 3. Chapter 4 shows how `b_verify` can be used to implement a publicly verifiable registries. We present an example public registry application for commodity receipts in Chapter 5 and show how `b_verify` improves its security. We discuss our prototype implementation in Java in Chapter 6 and evaluate `b_verify` in Chapter 7. Finally we review related systems in Chapter 8 and discuss future work in Chapter 9.

Chapter 2

Background

This section consists of a short overview of Bitcoin, Catena and authenticated data structures that is necessary to understand the design and motivation of `b_verify`.

2.1 Bitcoin

Bitcoin is a network of peers that run a software protocol to maintain a ledger of ownership for a digital currency [19]. 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, forming a hash chain. The combination of both the blocks and the headers is commonly referred to as the *blockchain*. Peers in Bitcoin follow the valid chain with the largest proof-of-work. This chain 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 are encouraged by a reward system to add valid blocks to the canonical chain. The original Bitcoin whitepaper argues 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 [19]. This is critical for the security of the currency.

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 as of August 1, 2018. 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 [15].

2.2 Catena

Catena is a protocol for an untrusted server to maintain a log of statements [12]. A Catena server is forced to commit to its log using Bitcoin. This is done by constructing a sequence of Bitcoin transactions with 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. The statements in the log are stored in the OP_RETURN outputs. To append a new statement to the log, the server signs and broadcasts a Bitcoin transaction spending the previous transaction's pay to public key hash output and includes the new statement in the OP_RETURN.

By using Bitcoin to commit the log statements, Catena makes it very hard for the server to equivocate about its log. To create remove, change or re-order log statements the server must create and successfully include different Bitcoin transactions that spend the same input. This implies that the server must double spend Bitcoin, which the Bitcoin protocol is designed to prevent. Catena clients can download and verify a Catena server's log using Bitcoin's Simple Payment Verification. This requires the Catena client to download only the Bitcoin headers, the log transactions and Merkle paths proving inclusion of those transactions. If the server cannot double spend Bitcoin, then this constitutes a proof of non-equivocation. This proof is small enough that it can be downloaded and verified on a mobile device.

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 add a new statement to the log, Catena must spend a Bitcoin transaction. Since each Catena statement requires a Bitcoin transaction, the number of applications Catena can support is limited. 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 [20]. 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` attempts to address many of these limitations.

2.3 Authenticated Data Structures

Authenticated data structures are a class of techniques in which a large data set is concisely and completely summarized by a small *verification object*. The verification object can then be used to prove properties about the underlying data. For example given the verification object for an authenticated set, one can prove membership of an element in that set using a short proof. Authenticated data structures decouple the storage of the data from its integrity. We refer the reader to Martel [21] for a description of the standard model and to Tamassia [22] for an overview. Papamanthou [23] and Li [24] 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 a sparse Merkle tree [25]. Authenticated data structures are also relevant becuase they can be leverage by applications to allow `b_verify` to secure larger data sets. This is done by using `b_verify` to store verification objects so that storage of data can be outsourced. The commodity receipt application we develop with `b_verify`, discussed in Chapter 5, will demonstrate how this can be done.

Chapter 3

Design

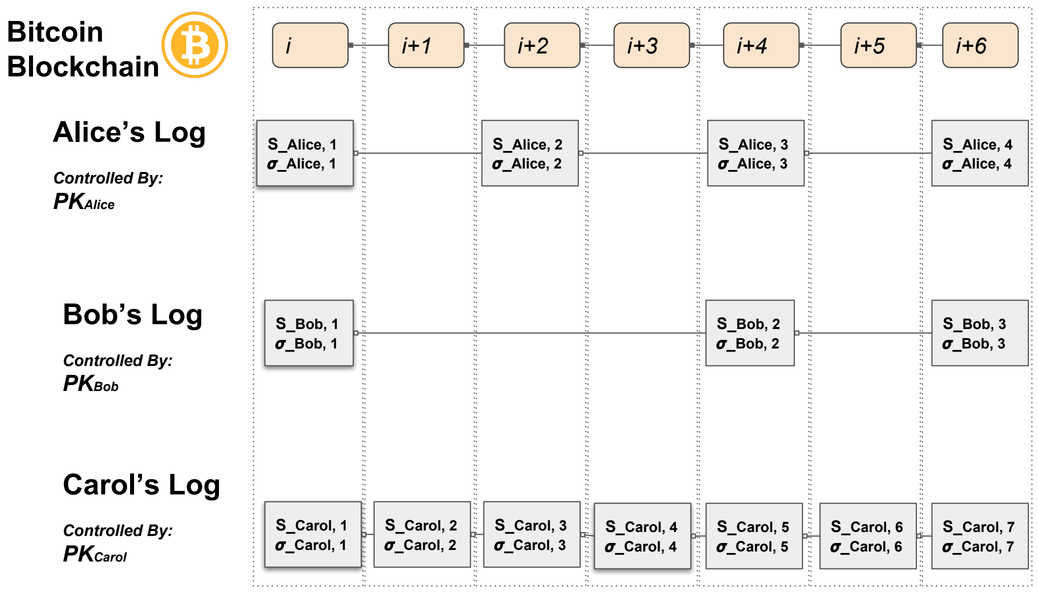


Figure 3-1: Logs of statements for `b_verify` clients *Alice*, *Bob* and *Carol*. Each client's log is identified by his or her public key (PK) and can only be updated with knowledge of the private key. Log are statements (S) are ordered and must be signed (σ) by the client. `b_verify` commits these log statements to Bitcoin, which is shown at the top. In this case the client logs begin at Bitcoin block i. `b_verify` prevents clients from modifying each other's logs and makes equivocating about the log statements as hard as double spending Bitcoin.

3.1 System Model

In b_verify there are many *clients*, a single *commitment server* and *Bitcoin*. Each client produces a *log* of statements. We assume that there is a public key infrastructure such that clients can be identified by public keys. The goal of b_verify is to ensure consistency: all clients must read the same sequence of statements for each log. b_verify does this by witnessing the logs 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] Additionally we describe the design in the case of a single commitment server. The protocol is not limited to a single server. In practice there could be several commitment servers operated by different parties and clients can chose which server to use.

b_verify supports many kinds of clients, including light clients which run on devices with intermittent network connections and limited storage capacity such as mobile phones. However the commitment server is assumed to have significant storage capacity, computational resources, and network bandwidth. Finally both the clients and the commitment server are assumed to be able to access the Bitcoin network.

3.2 Threat Model

Clients in b_verify may attempt to modify each other's logs. A client may also attempt to equivocate about his own log by presenting a different sequence of statements. Clients in b_verify 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 [26].

The commitment server is not trusted by the clients and can attempt to arbitrarily modify the data it stores. However we assume that the commitment server is always available to clients and that all participants can access Bitcoin. Participants communicate over unsecured

[1] b_verify also supports logs that are controlled by multiple clients but this is omitted to keep the explanation clear.

Internet connections. However we assume that no participant can collide hash functions or forge digital signatures. Additionally we make the stronger assumption that no participant can successfully execute a double-spend attack against Bitcoin. All log data in `b_verify` is assumed to be public.

3.3 Example

To make `b_verify` concrete consider a distributed system for monitoring pollution. In this system a network of volunteers operate sensors and report measurements of pollution. We assume that there is some way of identifying volunteers using public keys and we assume that these volunteers report measurements independently of each other. The goal of the system is to ensure that all users see the same readings. However users do not want to rely on a trusted third party to report the sensitive pollution data. The system can use `b_verify` to store the measurements by having each volunteer log the readings using a `b_verify` client. `b_verify` ensures the consistency and integrity of the logs. Using `b_verify` the measurements cannot be retroactively modified, even by the volunteer who created them.

3.4 Building `b_verify` From Catena

`b_verify` is best introduced by starting with Catena and trying to use it to for non-equivocation in a large application. As an example consider using Catena for the monitoring application. Each volunteer could run a Catena server to record measurements. This is desirable because it makes tampering with the log measurements as hard as double spending Bitcoin. Unfortunately it would also require one chain of Bitcoin transactions per volunteer. This is inefficient, expensive and limits the number of volunteers that the system can support. `b_verify` can be thought of as a new protocol that replaces Catena to support applications of this type more efficiently without introducing additional trust assumptions.

In `b_verify` a single server, called the commitment server, is used to manage *many independent client logs* using a *single chain of Bitcoin transactions*. The commitment server stores the client logs in an authenticated data structure and runs a Catena server to witness the data structure's changing roots to Bitcoin. This commits to the history of client logs. Clients in `b_verify` use a Catena client to download the roots and verify that the commitment server has not equivocated. The authenticated data structure allows the server to construct cryptographic proofs about the contents of each client's log. The correctness of these proofs can be evaluated by the clients using the witnessed roots.

This design allows clients to create logs, add statements to logs and efficiently verify the non-equivocation of a log without trusting each other or the server. The primary benefit of this architecture is that now the commitment sever can update many logs in a single Bitcoin transaction, dramatically reducing the cost of each statement and increasing the scalability of the system. For example if the monitoring application uses `b_verify`, then hundreds of thousands of volunteers can run send measurements to the server to be committed as a batch for the cost of one Bitcoin transaction.

This design has an interesting consequence. Clients can download a proof of non-equivocation for the roots from Bitcoin using Catena. However to prove the contents and non-equivocation of a client's log requires additional work. Clients in `b_verify` must periodically download this data from the server. In the context of the monitoring application this means that volunteers must download data form the server to verify that a new measurement has been recorded and to prove they have not equivocated. We will show how `b_verify`'s choice of data structures minimizes the amount of data that needs to be downloaded by clients and keeps the proof of non-equivocation for a client's log small.

3.5 API

`b_verify` provides the abstraction of many logs of statements that are each controlled by a single client as shown in Figure 3-1. [2] Each client maintains a proof of the contents and non-equivocation of her log. `b_verify`'s API consists of four methods: **CreateLog**, **AppendStmt**, **VerifyStmt** and **GetUpdates**. These methods are described in Table 3.1. Clients can create logs on demand by calling **CreateLog** on the commitment server with their public key. In the example monitoring application, a volunteer can call this method

[2]Logs controlled by multiple clients can be implemented using multisig schemes.

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 statement *s* and a signature over the statement *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 append a new statement *s* to 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.
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.GetProofUpdates(*pk*) $\rightarrow$ *proof_updates*	Invoked by a client with public key *pk* to get *proof_updates* needed to update 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
S.MultiAppend({*pk*, *pk'*, ..., *s*, {*sig*, *sig'*, ...}) $\rightarrow$ *proof*	Invoked by a group of clients with public keys *pk*, *pk'*, ... to atomically add the statement *s*, to the end of their respective logs. All of the clients must include signatures for this statement. The server returns a proof that the statement has been added to all of the logs.

Table 3.1: `b_verify` API. The first group of methods are similar to the API of Catena but are implemented for multiple logs efficiently. The method **GetProofUpdates** represents the additional data that clients in `b_verify` must download relative to Catena. **MultiAppend** is a new method to support applications by allowing clients to make atomic statements across multiple logs.

to initialize a new log to record measurements. Once the log has been created, new statements can be added to the log as needed using the **AppendStmt** method. Volunteers call use this method to record new measurements. Volunteers can record new measurements independently of one another.

Because clients do not trust the commitment server or one another, these methods return a proof that can be checked by the client using the **VerifyStmt** method to ensure that a log has been created or a statement has been added. This allows the client to maintain a proof of the contents and non-equivocation of his log. Furthermore these proofs are *publicly verifiable* which means that the correctness of the proof can be evaluated by any client. `b_verify` keeps these proofs small so that they can be easily shared with other clients.

`b_verify`'s batching requires clients to periodically download additional data from the server. 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 come online without affecting the security of their log.

In `b_verify` it is possible to atomically append a new statement to multiple logs with the **MultiAppend** method. [3] Critically this method does not require any additional trust assumptions. This method can be used for interaction between multiple logs that are part of the same application or between different applications entirely. For example in a rewards systems, two vendors tracking balances using two `b_verify` clients may agree to credit one account if and only if another account is debited. This could be implemented atomically using **MultiAppend** to add a statement to both logs. We will demonstrate the utility this type of interaction in building applications in Chapter 5.

3.6 Tracking Client Logs

The server tracks the client logs using a Merkle Prefix Trie. This authenticated data structure is a binary prefix trie which commits to a mapping from keys to values. Each client log in `b_verify` has an entry in the map with the key of the hash of the log owner's public key and the value of the current *signed* last statement in their log as shown in Figure 3-2.

The leaves in the trie are the (key, value) pairs and each pair is stored at a location

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

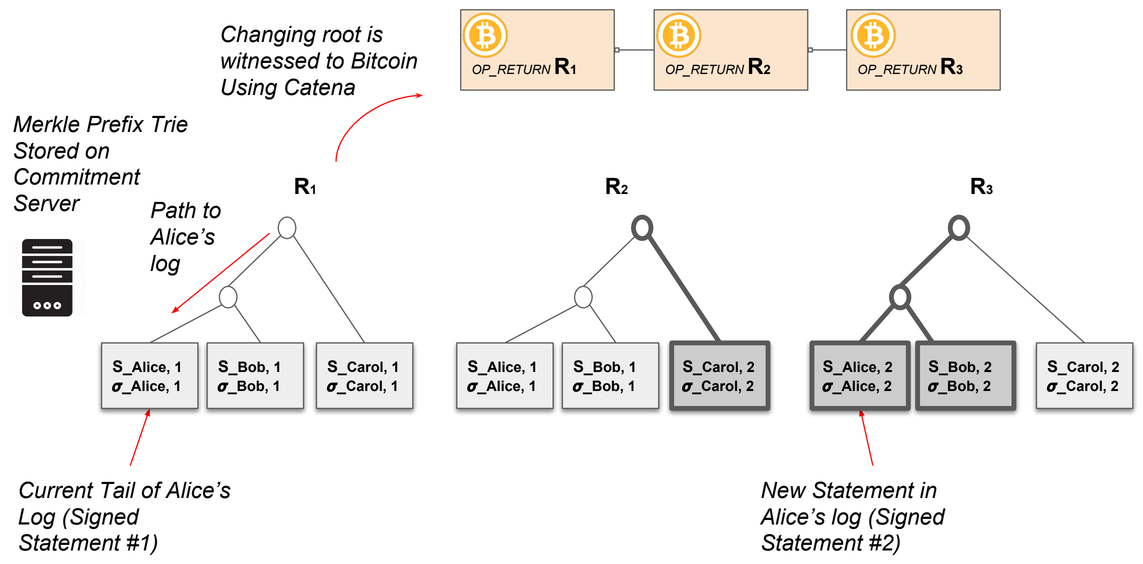


Figure 3-2: Tracking logs using the commitment server. The server stores the last signed statement in each log using a Merkle Prefix Trie. The trie changes over time as new statements are added to the logs. These modifications are shown in bold. The different versions of this trie completely track client logs and the changing roots, R_1, R_2 and R_3 are witnessed to Bitcoin.

determined by interpreting the key as a path in the trie. To commit to the mappings the leaves are recursively hashed to produce a single root hash value. To prove membership or non-membership of a (key, value) pair, one can provide a path from the leaf to the root with the hash values 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 allows clients to verify the statement at the end of a client's log.

This data structure changes over time as clients add statements to their logs. The versions of this trie can be used to prove the entire history of each log. We chose this data structure because membership proofs are small and, we can we can store the versions of the tree efficiently. Using a Merkle Prefix Tire with N entries, the size of a membership 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 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.

This allows the server to store the changing versions of the data structure efficiently. Furthermore the server can send a client new membership proofs efficiently by only transmitting the parts of the proof that have changed from the previous version.

3.7 Appending Statements to a Log

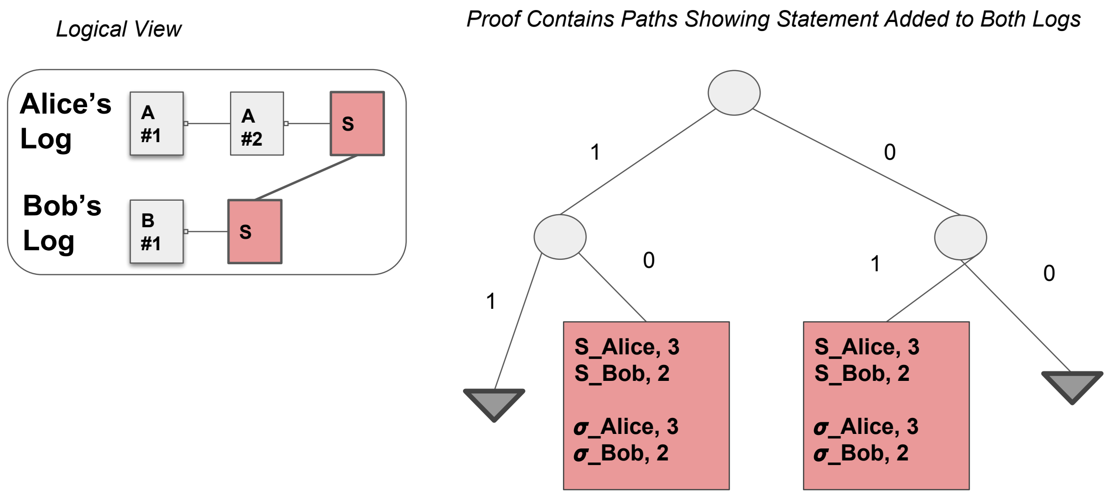


Figure 3-3: The proof returned when calling **MultiAppend** to add a statement S to the logs for Alice (public key $10\ldots$) and Bob (public key $01\ldots$). The statement is added to both logs atomically. The proof demonstrates this by including Merkle paths showing that the statement has been added to the respective logs.

When a client wants to add a new 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.

Since the server is not trusted by the client, the server returns a proof that the log statement has been committed. This proof consists of the path in the Merkle Prefix Trie to the new statement along with a new root. The procedure for **MultiAppend** is similar, but the proof includes one path for each log, showing that the new signed statement has been added. An example is given in Figure 3-3. 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.

Using the membership proofs the client can verify the contents of the log over time. However there is a significant problem: to check these proof clients must have the correct root values. A malicious server could potentially present a different history of root values to clients to allow clients to equivocate about their logs. For example a volunteer could collude with the server by singing a different pollution measurement and retroactively inserting this measurement into his log.

`b_verify` prevents this by witnessing the roots of the Merkle Prefix Trie to Bitcoin with Catena. Once a root has been witnessed it cannot be modified and all clients will see the same roots. Clients do not consider a log statement to be committed until they have both a Merkle proof to the issued statement and an SPV proof that root has been witnessed 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.

3.8 Proof of Non-Equivocation

The proof of non-equivocation consists of the witnessed roots in Bitcoin and membership proofs in the Merkle Prefix Trie for the log statements. Consider the example shown in Figure 3-4 for Carol's log. Her log contains two statements $S_{Carol,1}$ and $S_{Carol,2}$, 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 downloaded from Bitcoin SPV.

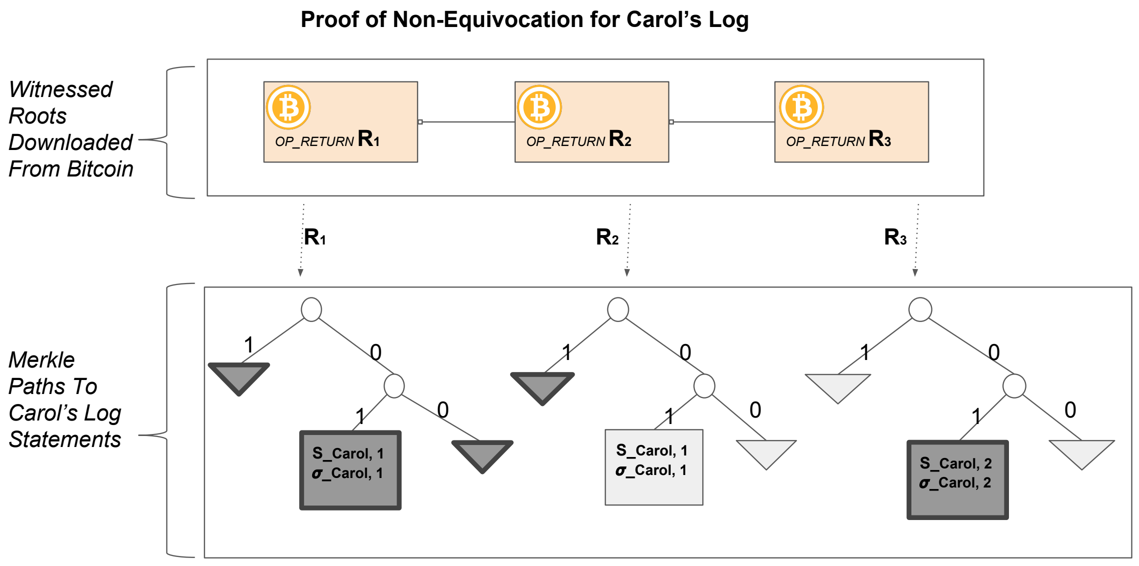

Proof of Non-Equivocation for Carol's Log
Witnessed Roots Downloaded From Bitcoin
OP_RETURN R1
OP_RETURN R2
OP_RETURN R3
R1
R2
R3
Merkle Paths To Carol's Log Statements
1
0
S_Carol, 1
σ_Carol, 1
S_Carol, 2
σ_Carol, 2

Figure 3-4: The complete proof of Non-Equivocation for Carol's Log. The top portion can be downloaded directly from Bitcoin using Catena. The bottom portion is downloaded initially from the commitment server and consists of Merkle paths. Hash values are represented by triangles. Only the portions in the dark borders are sent by the client directly. Repeated hash values can be inferred when checking the proof. Once Carol has this proof she can share it on a client-to-client basis.

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 $(S_{Carol,1}, \sigma_{Carol,1})$ and from R_3 to her second signed log statement $(S_{Carol,2}, \sigma_{Carol,2})$. 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 can be thought of as a non-inclusion proof and is a necessary consequence of managing multiple logs on the server. If the client did not include this proof, then it would be possible for the client to equivocate by omitting entries in its log. When the server witnesses a new commitment, clients whose logs have not changed still 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 hashes, as shown in the Figure 3-4. 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.

3.9 Server Misbehavior

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.

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 partially applies an update a non-repudiable proof of this behavior exists. This proof consists of the Merkle path to the partially applied update. We leave detecting and resolving replay attacks up to the clients. One useful approach will be discussed in Chapter 4. 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.

3.10 Handling Bitcoin Re-organizations

In Bitcoin the canonical blockchain may change in a so called *reorganization* that removes blocks from the chain. This can be caused by network latency between miners or because of a buggy or malicious miners. During a re-organization, the commitment server needs to re-broadcast the Bitcoin witness transactions. The Bitcoin peer network does this automatically by dumping these transaction back into the pool of unconfirmed transactions whenever a re-organization is detected. However a malicious client could still broadcast other transactions containing different log statements. The Bitcoin protocol makes the likelihood of a transaction being removed from the canonical chain decrease with each additional block (these additional blocks are often called *confirmations*). To guard against this clients in `b_verify` may wait for several blocks before accepting a new root as committed.

3.11 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 [12].

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 a 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 3.9 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.

3.12 Cost

The server must pay high fees to get its transactions included in the next Bitcoin block. On August 1st, 2018 this feed would be 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.

3.13 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. Improving the fault tolerance of `b_verify` is left to future work.

3.14 Privacy and Legal Risks

Privacy was not a design goal for `b_verify`. The protocol does not provide any guarantees about the privacy of the data and applications must use other techniques if privacy is a concern. However `b_verify` does provide a form of weak privacy. Clients in `b_verify` only need to reveal a hash of the log statements to the server rather than the actual log statement. Therefore the operator of the server does not necessarily know the contents of logs he stores. This is an acceptable level of privacy for many applications and may reduce the legal risks associated with operating a commitment server.

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 the public key for a user 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 critical systems can be modeled as registries. For example the hash server in Verena [17], a system that allows web clients to verify the correctness of queries and computations on data, [17] can be viewed as registry of verification objects. Similarly tor directory servers can be viewed as registries of tor nodes.

Building a registry without a trusted party is difficult and requires solving multiple challenging problems. The registry must prevent equivocation while supporting *efficient* look-ups that allow a user to determine the value of a key in the registry. Users should be able to *verify* if the value is correct. A *publicly verifiable* registry is one in which this verification can be performed by anyone. Ideally even light clients with intermittent connections and limited resources should be able to look up and verify the value for a key in the registry.

This is often not the case: for example replaying a Blockstack log to determine a name mapping cannot be done on a mobile device because it requires downloading Gigabytes of data. Finally public registries must support updates to the value of a key, and ideally allow users to update multiple keys simultaneously. The registry must ensure that only the appropriate user(s) can update the value of a given key.

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 allows light clients to verify lookups. `b_verify`'s model of a public registry could be applied to any of the applications discussed previously and can also build new kinds of registries. The application described in Chapter 5 will serve as an example.

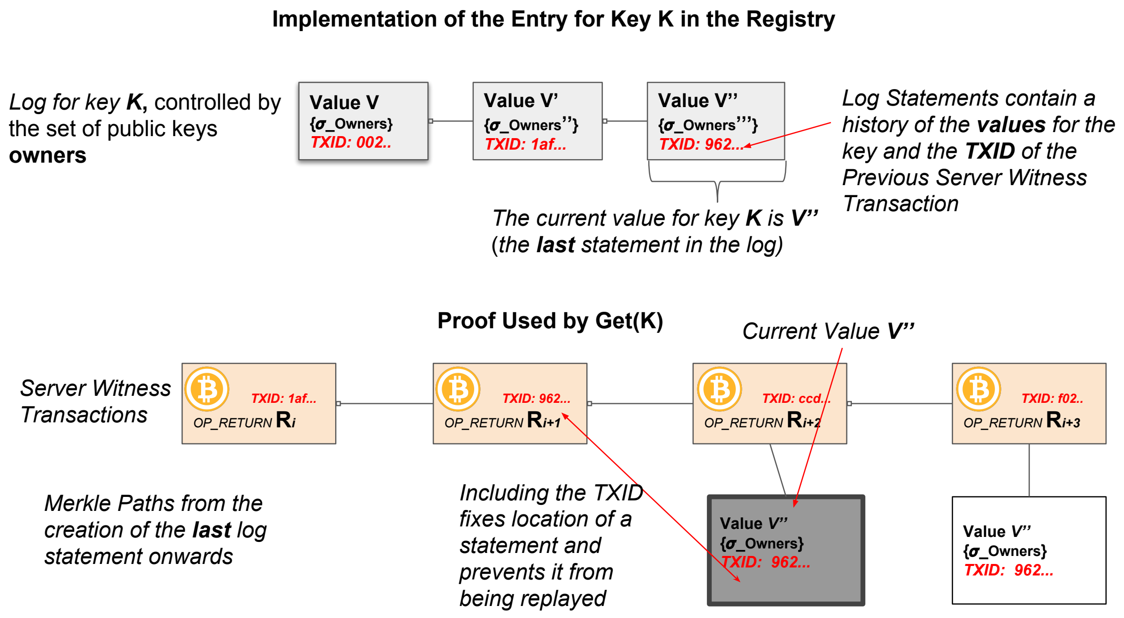
4.2 Model and API

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 keys, and requires signatures from the public keys that control the modified entries. The **Get** method returns the value for a key and checks that this value is correct. Behind the scenes **Put** and **Get** create and evaluate proofs using `b_verify`. 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. We assume that clients have some scheme for determining which log stores a given key and determining the public keys which must sign updates. This could be calculated dynamically, distributed as part of the source code of an application or be part of the client PKI. The log contains a history of the values for the entry, with the last statement in the log holding the current value.

A **Put** is implemented by adding a statement containing the new value to the log. Puts to multiple keys can be implemented atomically using the **MultiAppend** method. The **Get**



Implementation of the Entry for Key K in the Registry
Log for key K, controlled by the set of public keys owners
Value V {σ_Owners} TXID: 002..
Value V' {σ_Owners''} TXID: 1af...
Value V'' {σ_Owners'''} TXID: 962...
Log Statements contain a history of the values for the key and the TXID of the Previous Server Witness Transaction
The current value for key K is V'' (the last statement in the log)
Proof Used by Get(K)
Current Value V''
Server Witness Transactions
TXID: 1af... OP_RETURN Ri
TXID: 962... OP_RETURN Ri+1
TXID: ccd... OP_RETURN Ri+2
TXID: f02.. OP_RETURN Ri+3
Merkle Paths from the creation of the last log statement onwards
Including the TXID fixes location of a statement and prevents it from being replayed
Value V'' {σ_Owners} TXID: 962...
Value V'' {σ_Owners} TXID: 962...

Figure 4-1: Overview of the design of a registry using `b_verify`. The registry stores key/value maps using `b_verify` logs. The setup for a single key **K** is shown along used by the method **Get(K)** to verify the value of the key.

method should return the value stored in the last statement in the log. The proofs for these operations are based on log proofs in `b_verify`.

Using `b_verify` guarantees the consistency of the log and ensures that only statements signed by the correct public keys can be added. There is still a potential problem: a malicious server might attempt to replay a previously signed log statement. `b_verify` lets client decide how best to prevent replay attacks. The simplest way would be to have clients number their log statements. If clients expect to mostly read entire logs this works just fine. However for clients in a public registry this is not optimal because clients only care about the *last* statement in the log. Clients do not want to replay the entire log just to determine the last statement.

Our design solves this problem by including the Bitcoin transaction id of the prior server commitment in the log statements. This fixes the location of the statements without requiring the client to replay the log. We implement **Get** by checking a proof of the last log statement 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 3.4, and second a proof that a specific statement is located at the end of a client's log. This second proof is just the last portion of the log proof as described in Section 3.8.

4.4 Improving BlockCerts Using a Public Registry

BlockCerts is an emerging standard for creating, issuing and verifying digital credentials [27]. For example BlockCerts has been used to issue degrees to MIT students [28]. The goal of BlockCerts is to reduce the trust required in credential issuers and to improve the verifiability of digital credentials by using a public ledger. BlockCerts has many different implementations that use various public ledgers. The hope is that using public ledgers can improve trust in the records. One encouraging example is that a Chinese court recently accepted this method to legally determine the creation time of a document [29].

However one potential problem for BlockCerts is revoking credentials. Currently BlockCerts is investigating how this could be done and has a current proposal using Ethereum [30]. This would require light clients to download the Ethereum block headers which as of August 1st, 2018 are 3 GB. We show how BlockCerts could alternatively implement revocable credentials using a publicly verifiable registry with `b_verify`. This only requires light clients to download about 40 MB of data. In this design 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 a credential is in this set by using the **Get** method of the public registry.

This scheme has a number of nice properties. 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 credential 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

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

must periodically update the proof for their credential. Other systems may not provide an easy way to do this securely. For example determining a certificate has been revoked using `b_verify` is better than using a URL because it does not depend on the integrity of an external website.

Chapter 5

Building New Kinds of Registries With `b_verify`

`b_verify` can be used by applications 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 central trusted party. Our design solves several technically interesting problems that exist in current systems for managing receipts 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. Commodity 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 [31–33]. Despite their importance, commodity receipts are usually tracked using paper documents or in a centrally managed database. Unfortunately this often lacks transparent and has led to fraud. Receipt and loan records can be manipulated 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 [7]. 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 secure receipt and loan data. While the design is specific to this use case, the attributes that make warehouse receipts interesting generalize to other assets.

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. 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 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.

[1]This is particularly true in the developing world, which suffers from a lack of quality institutions [34].

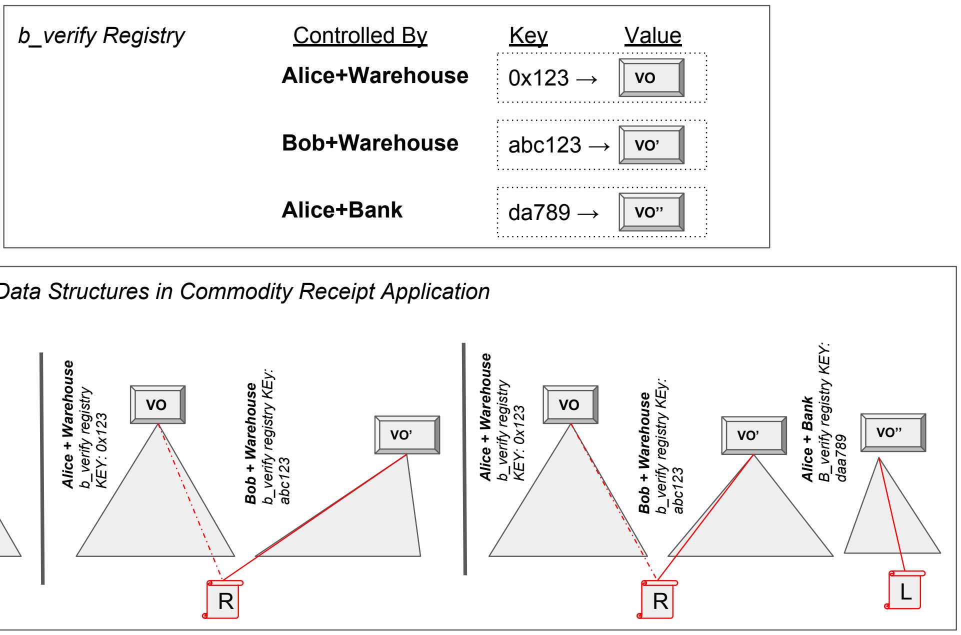

b_verify Registry
Controlled By
Key
Value
Alice+Warehouse
0x123 →
vo
Bob+Warehouse
abc123 →
vo'
Alice+Bank
da789 →
vo''
Operations and Data Structures in Commodity Receipt Application
Alice + Warehouse b_verify registry KEY: 0x123
Bob + Warehouse b_verify registry KEY: abc123
Alice + Bank B_verify registry KEY: daa789
vo
vo'
vo''
R
L

Figure 5-1: Overview of the design of the commodity receipt application. The application uses a `b_verify` public registry to store verification objects as shown in the top panel. The application uses additional authenticated data structures to store the data and implement the operations as shown in the bottom panel. In the bottom left panel Alice is issued a receipt, R, by the Warehouse. In the bottom center panel Alice transfers a receipt, R to Bob. Finally in the bottom right panel 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.

The overview of our design is shown in Figure 5-1. Receipt ownership is tracked by the application using authenticated data structures. As discussed in Section 2.3, an authenticated data structure commits to a set of data with a concise *verification object*. It is possible to construct short proofs about the data that can be checked using the verification object. Each $warehouse \times depositor$ and $bank \times depositor$ is mapped to an authenticated set, a Merkle Prefix Trie, holding receipts or loans respectively.

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. Restrictions on transferring and loaning receipts are enforced through shared ownership of the entries in the registry. Updating the value in the registry requires the depositor and the bank (or respectively the depositor and the warehouse) to sign a new value.

Critically for this application, `b_verify` supports atomic updates to multiple keys in the registry which will be used to implement transfer operations.

5.2.1 Application Operations and Implementation

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 for the *new* verification object value. The warehouse submits the signed value by using a **Put** request for the registry on the commitment server. The commitment server then updates the verification object in the registry 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.

5.3 Analysis

By using `b_verify`, only the owners of a key in the registry can modify its value and **Get** will return the same value to all clients. We show how our design uses this primitive to mitigate potential attacks.

A devious depositor could try to issue himself receipts or a nefarious warehouse could try to modify a previously issued receipt, for example by reducing the amount. Both of these actions would require updating the verification object in the registry. Our design prevents this because the warehouse and the depositor share control the entry in the registry and therefore *both* must cryptographically sign new values.

Users in the application must be able to determine who owns a receipt. In Bitcoin, each peer must replay every transaction to determine the validity of a new transaction. Our design tackles this problem a different way by requiring the warehouse to approve the result of every transfer of receipt. It is up to the warehouse to do this correctly. From the perspective of users, as long as the receipt is in the registry and signed by the required parties it is valid. The benefit of this choice is that unlike Bitcoin, users in our application no longer need to replay entire logs to determine who owns a receipt.

Another problem is that a depositor could attempt to transfer the receipt to two different parties simultaneously by signing multiple new verification objects (a *double spend* in crpytocurrency terminology). Our design prevents a client from successfully doing so by guaranteeing the consistency of the registry. Because the registry contains the verification objects, all clients will see the same set of ownership records.

This design does have some limitations. Both depositors must sign to transfer a receipt. This requires both depositors to be online when a transfer occurs. This is inconvenient. By requiring the warehouse to also sign we open up the possibility that the warehouse could extort or censor depositors. However we believe that for a receipt, which has a clear real world dependency on the issuing warehouse, this is an acceptable risk.

Chapter 6

Implementation

To evaluate `b_verify`, we implemented a prototype of the commitment server and client. We then implemented a proof-of-concept for the warehouse receipt application that uses this prototype. All codes are open source. [1]

6.1 Commitment Server

We implemented a 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 [35]. All proofs and update messages are serialized using code programmatically generated by Google Protobufs [36]. 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 application also keeps 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 [37] and an open source android Bitcoin wallet [38] 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 [39]. gRPC is an RPC framework built on top of Google Protobuf which provides convenient inter-operability 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 trut the server.

Chapter 7

Evaluation

7.1 Goals and Methodology

In this section we seek to test the following hypothesis: `b_verify` can scale to many client logs and be used to build practical publicly verifiable registries. To evaluate our hypothesis we need to test the following:

1. Log proofs are small enough to be downloaded and verified on a mobile device.

2. Clients only need to download small amounts of data per day.

3. The commitment server can handle heavy client load.

4. The registry can provide an acceptable experience for users.

To answer the first two questions we analyze how log proof size and the amount of data clients must download per day scales theoretically in terms of the number of client logs and the number of updates. We then measure the actual sizes for a system with one million client logs using our prototype. To determine if the commitment sever can handle a large system we use microbenchmarks of the limiting code paths to identify the current performance bottlenecks. We then use simulations to measure the performance of the system under heavy client load. Finally to evaluate `b_verify`'s ability to build a practical registry we analyze the theoretical size of the proof for an entry. We then measure actual sizes and evaluate the user experience for our prototype for the commodity receipt application with 10^6 different users and 10^7 different receipts and loans.

7.1.1 Test Setup

The test machine is an AMD Ryzen 7 1700 Eight-Core Processor with 31 gigabytes of RAM running Ubuntu 18.04. All micro benchmarks and simulations were done with a commitment server running on the test machine. 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 [40]. As a result micro benchmarks likely understate performance in a real system. For simulations, mock clients were created 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

7.2.1 Theoretical Analysis

The size of the proof for a log depends on several factors. Let N be the total number of client logs, and let S be the number of server Bitcoin transactions and U be the number of new log statements in each server commitment. The size of the log proof up to constant factors is described by the following equation:

$$\log(N) + S \times \log(U)$$

The first term in the sum is the length of the path to the first log statement in the initial Merkle Prefix Trie. The second term in the sum is the amount of information required to update this path in the subsequent versions of the MPT. As discussed in 3.6, this requires sending data logarithmic in the number of leafs changed in the Merkle Prefix Trie, which is $\log(U)$. In `b_verify`, the server makes at most one transaction per Bitcoin block, so S grows with time at a rate comparable to the Bitcoin headers. Therefore over time the proof

Updates	Data Downloaded Per Day	Proof Size (One Month)	Proof Size (One Year)
10^3	5 KB	159 KB	1.91 MB
10^4	7 KB	212 KB	2.56 MB
10^5	8.8 KB	265 KB	3.18 MB
10^6	10.6 KB	318 KB	3.82 MB

Table 7.1: The approximate size of the proof and the amount of data a client must download per day for a log in `b_verify` to support a given number of updates per hour. This assumes that the server makes exactly one Bitcoin transaction per hour to commit the updates and uses 32 byte cryptographic hashes (e.g SHA-256). This does not include the portion of the proof that is downloaded from Bitcoin SPV.

is dominated by the second term in the formula. As long as $\log(U)$ is small then the entire proof is comparable in size to a Bitcoin SPV proof, and thus we expect it to be verifiable on a cell phone.

The amount of data that clients need to download per day is determined by the rate at which the server includes Bitcoin transactions in new blocks. However there are only about 144 Bitcoin blocks per day, so the amount of data that a client must download per day scales with the logarithm of the number of updates. As long as $\log(U)$ is small this is reasonable for a mobile phone client.

To make this analysis concrete we have calculated the approximate sizes for the proofs and the amount of data that clients must download per day in terms of the number of updates the server can support per hour. The results are given in Table 7.1

7.2.2 Empirical Measurement

To corroborate our theoretical analysis we measured the data a client must download per server commitment in our prototype with one million total client log and varying numbers of updates. The results are shown in Figure 7-1. As expected, the amount of data the client must download scales as the logarithm of the number of updates.

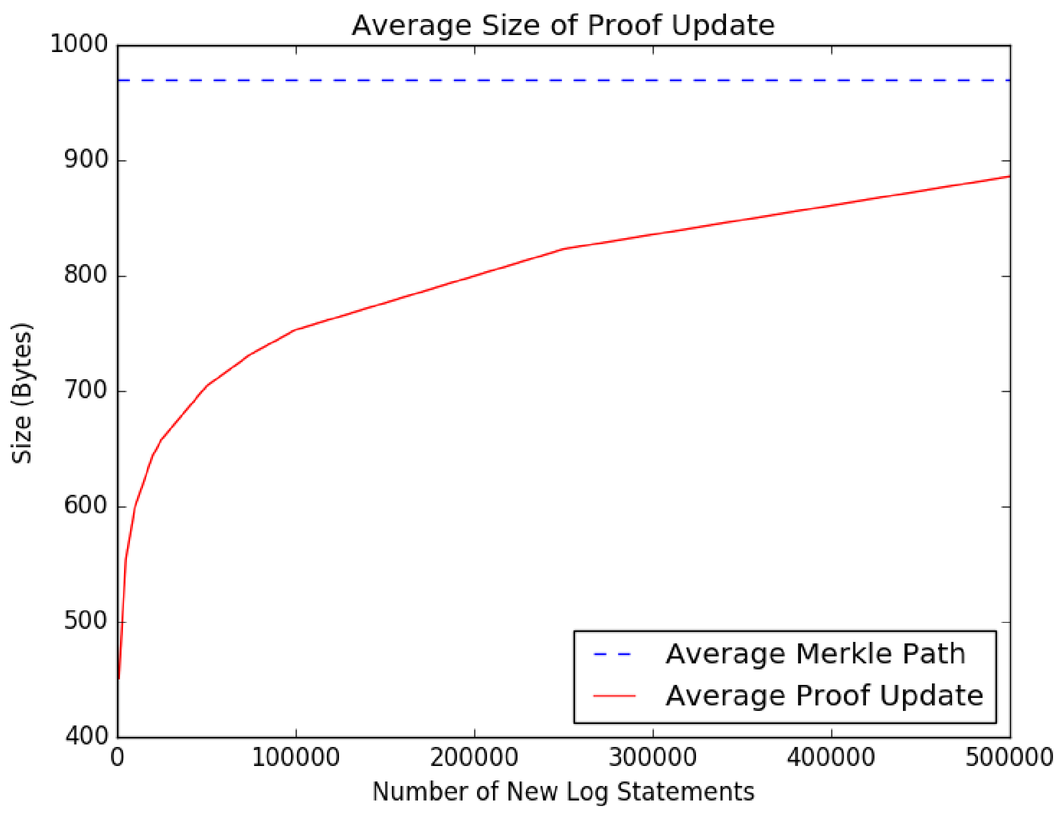


Figure 7-1: Average size of the data that must be downloaded by each client in terms of the number of new log statements. The average size of a full Merkle path in the Merkle Prefix Trie is provided for comparison purposes.

7.3 Performance of the Commitment Server

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 throughput via micro benchmarks of the code paths on the server to determine the limiting operation and then use simulations of a server under heavy client load to get a measurement of throughput under realistic circumstances.

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 involves multiple signature checks if the log is controlled by more than one owner). If the statement is signed then the commitment server updates the log entry in the Merkle Prefix Trie and schedules it to be

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.

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 add statements to 1% of the logs in a system with 10^6 total client 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 for generating proof updates is the time required to calculate and send the proof updates for the selected log. To benchmark the time to generate complete log proofs we select a log at random and then add statements to 10% of the remaining logs in 10 batches of equal size. The resulting micro benchmark is the time required to calculate the entire proof of non-equivocation for the selected log.

The micro benchmarks results 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.

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 commit 10^5 new log statements. We measure the time required for the server to verify the request, perform and commit the new statements and create proofs

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.

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 Evaluating Public Registries Built With `b_verify`

In a registry, the most important operation is **Get**. As discussed in Chapter 4, we have optimized our design to make the proof for a lookup as small as possible. Let N be the total number of entries in the registry and let S be the total number of Bitcoin transactions by the server. Consider a single entry in the registry. Let S_{old} be the number of server Bitcoin transactions *before* the last update to the registry entry and S_{new} be the number of updates *after* such that $S = S_{old} + S_{new}$. Let U be the number of changes to the registry committed in each server transaction. The size of the proof for the entry in the registry is given by the following formula:

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 7.4: Breakdown of the size of the **Get** 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.

$$S_{old} + \log(N) + S_{new} \times \log(U)$$

The first term in this formula is the cost of verifying the initial server witnesses. The second term is the path to the last statement in the log. Finally the last term is the additional proof that the statement is at the end of the log. Observe that if $S_{new} << S_{old}$, this is much smaller than the proof for the entire log of statements. Furthermore clients can periodically re-insert the entry in the registry to reset S_{new} to zero. In this case the size of the proof is:

$$S + \log(N)$$

Furthermore once the client has obtained the root witnesses and verified non-equivocation of the server, each look up requires a proof of only $\log(N)$. This indicates that `b_verify` is particularly amenable to creating public registries.

7.5 The Commodity Receipt Application

To show that `b_verify` is practical we evaluate the commodity receipt application using our prototype.

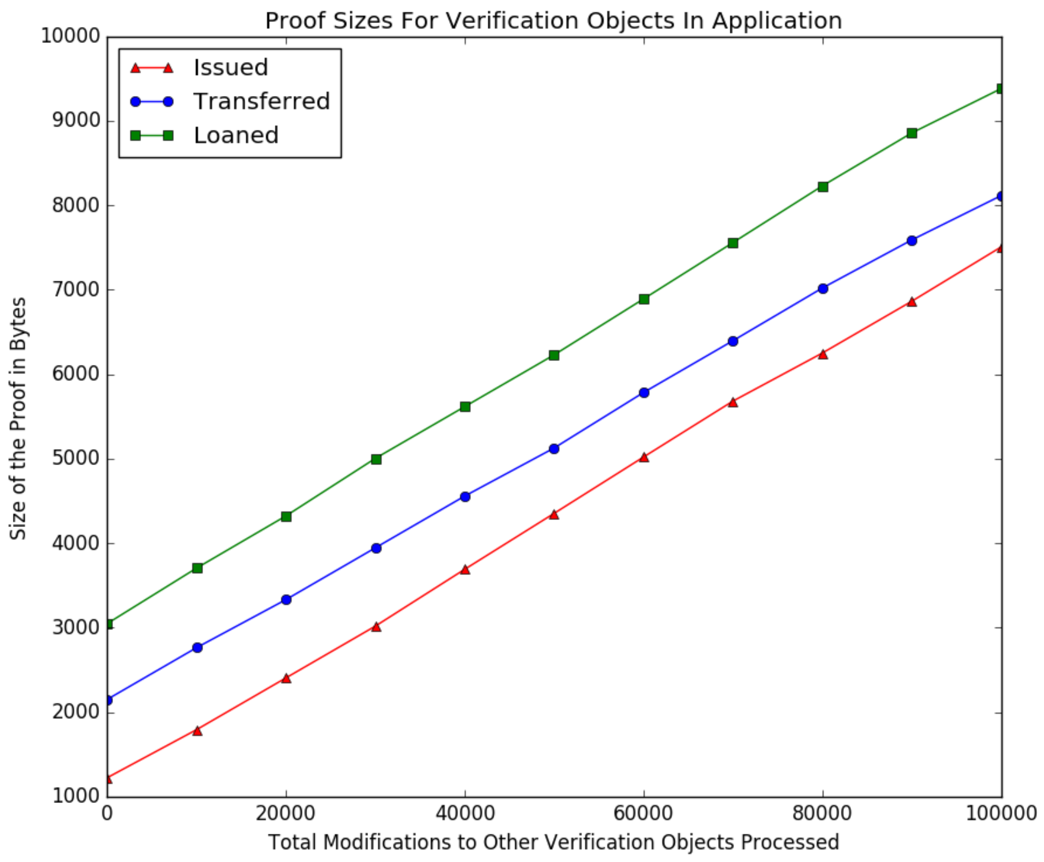


Figure 7-2: Size of the **Get** proof for the verification object in the warehouse receipt application. Transferred, issued and loaned correspond to the verification object after the respective operation.

7.5.1 Size of Proofs

Recall that in this application, verifying ownership of a receipt requires a **Get** operation on the public registry to retrieve the verification object. [1] We measure the size of the proof needed for this **Get** using our prototype.

The proof sizes for this operation are shown in Figure 7-2 and a breakdown is given in Table 7.4. As shown in the figure, the proof becomes larger with time and the slope of this line is $\log(U)$ where U is the total number of updates. Note that the proof for the transferred receipt's verification object in Figure 7-2 is larger than for an issued receipt's verification

[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`.

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

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.

7.5.2 Latency

Latency is an important concern in building a real-time user facing application. The latency of the warehouse receipt application was measured and the results are given in Table 7.5. When a user starts the application for the first time the application must obtain the server commitments from Bitcoin. This requires downloading the chain of Bitcoin block headers, the transactions 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. As of August 1st, 2018 this involves downloading 40 MB worth of data and computing around six hundred thousand SHA-256 hashes. After downloading and verifying this data, everything except for the server transactions and the last few headers may be discarded. From here on the application only needs to verify new server transactions which can be done incrementally.

The read latency experienced by users 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. 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. Therefore the write latency of the application is considerable. Our approach to prevent this latency from impacting user experience was have the application display the results of unconfirmed transactions, but graphically mark this information as still tentative to inform the user.

Chapter 8

Related Work

The most comparable system to `b_verify` is Catena [12]. As previously discussed `b_verify` improves on Catena by increasing throughput and lowering costs for clients. The overhead of `b_verify` relative to Catena is the additional data that must be downloaded to prove non-equivocation of a log. For a `b_verify` system with one million logs this amounts to only kilobytes of extra data per day. `b_verify` also provides a richer API than Catena that supports issuing statements atomically to multiple logs.

Previous work has used public ledgers for security. There are several protocols that use Bitcoin for timestamping [41–44]. However these protocols only allow users to check that a statement was made at specific time not whether they have equivocated. BlockCerts uses Bitcoin for creating digital credentials. `b_verify` could be used as a revocation scheme to determine if credentials have been revoked [27]. `b_verify` could be used to address this problem. Colored coins, a protocol for tracking assets using Bitcoin, does rely on Bitcoin to prevent equivocation [45]. However unlike `b_verify` requires one Bitcoin transaction per transfer operation. Blockstack is a domain name system that uses Bitcoin [16]. Blockstack depends on Bitcoin to prevent equivocation, but requires one Bitcoin transaction per domain transfer. Blockstack could also use `b_verify` to reduce the number of Bitcoin transactions. `b_verify` can provide the same system as EthIKS, a protocol using Ethereum to audit a CONIKS log [13], but support light clients.

Previous work has used authenticated data structures to produce a verifiable log [18]. However this work does not leverage Bitcoin, and relies on other methods for preventing equivocation. `b_verify` complements this research by making it possible to prevent equivocation using Bitcoin. Verena [17] and SUNDR [11] both leverage authenticated data structures to build higher level applications. However these systems rely on trusted servers to prevent equivocation. `b_verify` could be used to remove this dependency. `b_verify` is similar to CONIKS, a system for an auditable public key infrastructure [9]. However CONIKS if for a specific application and uses auditors rather than Bitcoin to resolve problems related to equivocation.

Chapter 9

Conclusion

9.1 Future Work

There are many aspects of the design, implementation and practicality of `b_verify` that can be improved. One problem with `b_verify` is that 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. Future work could examine how to improve freshness. One possible scheme is to have the commitment server broadcast witness transactions at predefined 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 Bitcoin peer network. However additional work is required to deal with re-organizations. Another weak point in `b_verify`'s design is fault tolerance. Future work could investigate improving fault tolerance by distributing `b_verify`'s commitment server or use multiple different commitment servers.

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. 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.

As we have shown, efficient non-equivocation can be used to build applications that do not require a central 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`.

9.2 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 central 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. We hope that `b_verify` and its ideas can be used to build more inclusive, transparent and decentralized systems.

Bibliography

[1] N. Leavitt, "Internet security under attack: The undermining of digital certificates," *Computer*, vol. 44, no. 12, pp. 17–20, 2011.

[2] "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.

[3] A. Loibl and J. Naab, "Namecoin," *namecoin. info*, 2014.

[4] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker, "Low-resource routing attacks against tor," in *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pp. 11–20, ACM, 2007.

[5] J. K. So, *Defending against Malicious Behaviors in BitTorrent Systems*. PhD thesis, North Carolina State University, 2012.

[6] C. Cachin, I. Keidar, and A. Shraer, "Trusting the cloud," *Acm Sigact News*, vol. 40, no. 2, pp. 81–86, 2009.

[7] M. Bulletin, "After port fraud, china's vast warehouse sector under scrutiny," Jun 2014.

[8] J. Crotty, "Structural causes of the global financial crisis: a critical assessment of the 'new financial architecture'," *Cambridge journal of economics*, vol. 33, no. 4, pp. 563–580, 2009.

[9] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "Coniks: Bringing key transparency to end users.," in *USENIX Security Symposium*, vol. 2015, pp. 383–398, 2015.

[10] M. D. Ryan, "Enhanced certificate transparency and end-to-end encrypted mail.," in *NDSS*, 2014.

[11] J. Li, M. N. Krohn, D. Mazieres, and D. E. Shasha, "Secure untrusted data repository (sundr).," in *OSDI*, vol. 4, pp. 9–9, 2004.

[12] A. Tomescu and S. Devadas, "Catena: Efficient non-equivocation via bitcoin," in *Security and Privacy (SP), 2017 IEEE Symposium on*, pp. 393–409, IEEE, 2017.

[13] J. Bonneau, “Ethiks: Using ethereum to audit a coniks key transparency log,” in *International Conference on Financial Cryptography and Data Security*, pp. 95–105, Springer, 2016.

[14] “Keybase.” `https://keybase.io/`. Accessed 2018-07-08.

[15] M. Vasek, M. Thornton, and T. Moore, “Empirical analysis of denial-of-service attacks in the bitcoin ecosystem,” in *International Conference on Financial Cryptography and Data Security*, pp. 57–71, Springer, 2014.

[16] M. Ali, J. C. Nelson, R. Shea, and M. J. Freedman, “Blockstack: A global naming and storage system secured by blockchains.,” in *USENIX Annual Technical Conference*, pp. 181–194, 2016.

[17] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun, “Verena: End-to-end integrity protection for web applications,” in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 895–913, IEEE, 2016.

[18] S. A. Crosby and D. S. Wallach, “Efficient data structures for tamper-evident logging.,” in *USENIX Security Symposium*, pp. 317–334, 2009.

[19] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2009.

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

[21] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, “A general model for authenticated data structures,” *Algorithmica*, vol. 39, no. 1, pp. 21–41, 2004.

[22] R. Tamassia, “Authenticated data structures,” in *European Symposium on Algorithms*, pp. 2–5, Springer, 2003.

[23] C. Papamanthou, R. Tamassia, and N. Triandopoulos, “Authenticated hash tables,” in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 437–448, ACM, 2008.

[24] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, “Authenticated index structures for aggregation queries,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 4, p. 32, 2010.

[25] R. Dahlberg, T. Pulls, and R. Peeters, “Efficient sparse merkle trees,” in *Nordic Conference on Secure IT Systems*, pp. 199–215, Springer, 2016.

[26] A. D. Rubin, “Secure distribution of electronic documents in a hostile environment,” *Computer Communications*, vol. 18, no. 6, pp. 429–434, 1995.

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

[28] E. Durant, "Digital diploma debuts at mit." `http://news.mit.edu/2017/mit-debuts-secure-digital-diploma-using-bitcoin-blockchain-technolog` Accessed 2018-08-01.

[29] W. Zhao, "Blockchain can legally authenticate evidence, chinese judge rules," *Coin Desk*, July 2018.

[30] "Verifiable credentials data model." `https://w3c.github.io/vc-data-model/`. Accessed 2018-08-02.

[31] G. 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*, pp. 2–4, 2003.

[32] W. 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.

[33] W. 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.

[34] D. Rodrik, A. Subramanian, and F. Trebbi, "Institutions rule: the primacy of institutions over geography and integration in economic development," *Journal of economic growth*, vol. 9, no. 2, pp. 131–165, 2004.

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

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

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

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

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

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

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

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

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

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

[45] M. Rosenfeld, “Overview of colored coins,” *White paper, bitcoil. co. il*, p. 41, 2012.