Blockchain for Clinical Decision Support Systems

Martin Hanzel
7221703
mhanz008@uottawa.ca

Cédric Clément
6377949
cclem054@uottawa.ca

Submitted 23 April 2019

Abstract

Blockchain technology has the potential to equip healthcare providers with the tools necessary to offer better care for patients. Some of the limitations in healthcare informatics that blockchain can address are access control [1] and data integration from several different providers [2]. Our objective is to create a data store which provides patients and practitioners with the ability to securely share and revoke access to medical data.

In the first part of this work, we explore the use of blockchain as a data store with Attribute-Based Encryption (ABE) as an access control mechanism. We review shortcomings of ABE and existing blockchain solutions with respect to our use case.

In the second part, we introduce a novel software system, based on blockchain, which enables users to securely create and share data in an untrusted environment. Our system allows for cryptographically-backed, granular access control and revocation, while leaving users in control of their own data. We also discuss the properties of tree-shaped blockchains, specifically in the context of computer security.

Contents

Chapter 1

Introduction

Blockchain is best known for the potential change it brings to the way financial transactions occur, leading to its adoption by some banks [3]. However, blockchain is nothing more than a data structure. The use of blockchain based data stores are on the rise and access to data is increasingly becoming centralized. Blockchain technology has the potential to equip healthcare providers with the tools necessary to offer a better care for mobile patients. For instance, patients transitioning from a pediatrician to a general practitioner must tend to the process of coordinating between the two providers. Additionally, the technology has the potential to connect healthcare providers in a way which is cryptographically secure, and ensures the confidentiality of patient data.

Our objective is to create a data store which provides patients with the ability to securely share and revoke access to their data. Take for example a patient who grants access to an AI system which evaluates the mental state of a patient. In this scenario, patients could allow the system to access their data feed. The patient, or external healthcare professionals, could then be provided with alerts regarding any significant changes to the mental health of this patient. This would allow a team of healthcare providers to address any potential issues at the onset of mental illness while giving the data owner full control of their data.

Additionally, we operate under the following requirements:

- The system must be operational in low connectivity environments. In the event that the system and its users lose connectivity with the Internet, the ability to share and revoke access to data must remain intact. The chosen system must be functional if all connectivity is lost, save that between data owners, data readers, and the local server storing the data.
- The system must provide users with the ability to share data.
- The system must allow users to revoke instances of shared data.
- The system must support a passive access control layer. That is, access

to data cannot be mediated by a proxy or intermediate server due to the low connectivity environment.

- The system must protect the data owners against inference attacks such as in the case where an increase in communication between a patient and healthcare provider could allow for the inference of health-related issues of targeted patients.

- The system must be easy to use in the event that a patient does suffer from a decrease in mental capacities. The cognitive load associated with key management may be too much for patients to manage in the event of a mental health crisis.

In this thesis, we develop a blockchain-based, on-chain data store which enables data owners to share and revoke sets of published information. This information can be anything from social media posts to personal journal entries. We do so in a way which is cryptographically secure and leaves patients in control over access to their data.

We start by introducing Attribute-Based Encryption, an encryption scheme which allows users to encrypt and decrypt data based on a set of attributes. We review the different components and considerations when designing an ABE based crypto-system and why it falls short for our particular use case. Following, we review some of the existing literature regarding ABE-based cryptosystems and how they differ from our use case. We conclude with our introduction of the share tree: a novel blockchain-based data structure which enables users to securely create and share data in an untrusted environment.

Chapter 2

Attribute-Based Encryption

2.1 Introduction

Control over access to resources is an essential consideration in the design of systems which harbour sensitive information. Restrictions on which entities may access a resource is accomplished through the use of access control and authentication mechanisms, such as username and password combinations. These username and password combinations represent an identity. For example, access control can be achieved by website administrators by granting a user and their associated username access to a set of resources and later revoking said rights if need be.

Attribute-Based Encryption (ABE) is the idea that access to a set of data can be granted to any user who possesses a set of attributes. For example, in the context of medical health records, access to a patient's health chart may be granted to anyone satisfying the attributes "Emergency Staff" OR ("Doctor" AND "Immunology"). The important implication to note is that an ABE-encrypted ciphertext can be encrypted today and decrypted by any users in the future who possess the required attributes, *without* any intervention by the encryptor. Additionally, access to specific sets of data is granted by encrypting the data over attributes, which can be non-trivial if a user wishes to share data with multiple users in other PKI crypto-systems. Another advantage of ABE over other access control mechanisms is that ABE schemes do not necessitate the use of servers which approve or deny access to data. The primary reason for this is that the access control mechanisms are created and embedded within either the private keys or ciphertexts during key-creation or data-encryption phases. This provides the data owner with the control over who has access to the data, as opposed to a trusted third party (TTP).

Another reason behind the choice for ABE is that our data storage medium is assumed to be untrusted. Given that ABE has the potential to provide an access control mechanism not mediated by an intermediate third party, and with data stored in an untrusted environment, data owners can publish data without

fear of having their data improperly managed by third parties.

This chapter introduces the various components, types, and design choices to consider when designing a system with ABE as an access control mechanism. We review some of ABE's components as shown in the taxonomical tree presented by Sookhak et al. [4]. We consider how the many features of ABE can coexist in a blockchain environment. Finally, we discuss the shortcomings of ABE in our particular use case.

2.2 Components of ABE

Attribute-based encryption is like a bicycle, in that the choice of components used to build the system shapes its usage. For instance, one would prefer the use of thick tires when riding in the snow over thinner ones which are more efficient over longer distances. The choices available to you when constructing a bicycle lie, in part, in the choice of the frame's material or type of brakes. Much like when building a bicycle, the choice of components must be considered when building a system centred around the use of ABE. The following section covers some of the components and how they relate to our use case.

2.2.1 Policy Location

The first component to consider is the choice over where the access policy is placed. An access policy is created from a universe of attributes and is a boolean expression such as ("Faculty" OR "Staff") AND "University of Ottawa." This boolean expression can be placed in the ciphertext or in the private key. We consider the implications of each.

In Key-Policy Attribute-Based Encryption (KP-ABE) the access policy is encoded in the key. The key is then used to decrypt ciphertext [5]. An example application provided by [5] for KP-ABE is in the management of audit logs; the use of a secret key for encrypting logs is inadequate since encrypting under a single key does not provide the same granularity offered by ABE. Log entries can be tagged as "critical" and "personal information" and only employees having keys which allow them to read entries under both "critical" and "personal information" will be able to decrypt the records. Another use case for KP-ABE provided by the authors of [5] is with regards to broadcast encryption. In this use case, broadcast streams can be tagged with attributes. The streams are encrypted with a symmetric key, and the symmetric key is encrypted with KP-ABE and sent to subscribers. Only the subscribers who have paid for a given package can decrypt and obtain the symmetric key in order to decrypt and read the broadcast stream.

In contrast to KP-ABE, Ciphertext-Policy Attribute-Based Encryption (CP-ABE) is the idea that the ciphertext is encoded with an access policy. The ciphertext can then be decrypted by a key which satisfies the policy [6]. In their paper, Bethencourt et al. stress the fact that in the KP-ABE scheme, the encryptor exerts no control over encrypted data. Instead, a TTP must responsibly

issue keys to users. In contrast to KP-ABE, the decision over who should have access (based on attributes) is moved from the TTP onto the encryptor.

To summarize, Kamp [7] makes the following distinction between the two: in KP-ABE, the key "describes a policy" whereas in CP-ABE, the key is used to "describe a set of attributes." Conversely, in KP-ABE, the ciphertext is "associated with a set of attributes" whereas in CP-ABE it is "associated with a policy." Kamp further reduces the choice as to which scheme to use based on who must define the access policy. When the decision regarding who can see the data is chosen at the time of encryption, CP-ABE is the preferred over KP-ABE because of the granularity offered by CP-ABE at said time. However, this does not seem to account for systems where the key creator (attribute authority) and encryptor are the same user, such as in the case of our system where the data owner's (DO's) control over his or her data is of great importance.

We reiterate that for our use case, DOs must be responsible for the encryption of ciphertexts as well as the creation and distribution of keys to potential data readers (DRs). Our DOs cannot rely on third parties for key issuance since this goes against the idea that our DOs have control over the DRs of their data. Additionally, the low-connectivity requirement could make third parties impractical if they are connected from distant locations. With this in mind, CP-ABE seems like an appropriate choice. However, recall that our ciphertext is placed on a blockchain, an immutable data store. An access policy stored in an immutable data structure cannot be mutated, which leads to issues when considering revocation. If we consider KP-ABE, ciphertexts are instead tagged with attributes. Indirection aside, if attributes are fixed and placed in immutable ciphertexts then KP-ABE is the appropriate choice.

2.2.2 Architecture

Key generation is an important aspect in any crypto-system. In ABE crypto-systems, key generation is typically carried out by an attribute authority (AA). An AA is in charge of verifying users and their identities and distributing attributes in accordance with those identities [7]. For our intents and purposes, we do not want users to have to trust a central AA. Therefore, a decentralized attribute management architecture is the obvious choice. Having made the choice of decentralized AA over a centralized AA, both are introduced for reasons of completion.

ABE schemes utilize attributes as access control components. Different combinations of attributes lead to different access permissions. The declaration of these attributes is done so by attribute authorities. Lewko et al. discuss the issue with ABE with respect to multi-authority systems. Prior to their contribution, the main problem was that multi-AA ABE crypto-systems required that each AA be dependent upon a central authority in order to provide DOs and DRs with encryption and decryption capabilities. The ABE systems at the time were not really decentralized due to their dependence on a central authority. They present a solution which does not require a central authority [8].

Choosing a decentralized ABE architecture is preferable over a centralized given that centralized solutions are weak to corruptible central authorities. In the event that a DO can no longer trust a central AA, his or her control over the attribute issuance process is not ensured. The use of a decentralized ABE crypto-system is suitable for decentralized applications employing blockchain as a data store.

2.2.3 Revocation

While it is true that the act of communication can never truly be revoked, access to stored data *can* be. The importance of revocation cannot be overstated: revocation provides security for users who may otherwise be hesitant to share information, or who engage in other forms of self-censorship. In the context of ABE, modification of attributes leads to revocation of stored data.

This section introduces revocation in ABE systems. We start with a short introduction of the revocation controller, the entity tasked with initiating the revocation mechanism. We then review the different revocation mechanisms and their suitability for blockchain-based clinical decision support systems.

The revocation controller is the entity responsible for initiating the mechanism by which revocation occurs. ABE crypto-systems generally place this responsibility in one of three parties: some authorized authority, a server, or the data owner [4]. As previously mentioned, revocation via server is a poor choice for obvious reasons: a rogue server may decide to ignore revocation requests, or even choose to revoke users who were not originally in a revocation request. Revocation via an authorized authority, other than the original DO, is also a poor choice for the same reasons as choosing revocation via server. Thus, for the purposes of a decentralized blockchain-based data store using ABE as a confidentiality mechanism, the responsibility of revocation controller should lie with the data owner.

In the context of ABE crypto-systems, revocation can occur via attribute revocation or key revocation [9]. Liu et al. refer to the first revocation mechanism as the indirect approach wherein the target data is re-encrypted under a new access policy (or set of attributes in the context of KP-ABE). Following this, all users who have not had their access revoked receive an updated key. This method is called indirect revocation because there is no interaction between the DO and the user with revoked access. The implications of this in our context are significant: a stored data object (SDO) protected via any ABE scheme and published on a blockchain cannot effectively be re-encrypted (and by extension, revoked) due to the innate immutability of blockchain.

The second method is that of the embedded revocation list. The authors refer to this method as the direct approach since ciphertexts are updated with a list of revoked users. Any user wishing to decrypt the ciphertext must, in addition to satisfying the access policy, not be listed in the revocation list. In the context of blockchain systems where SDOs are encrypted using an ABE-scheme (without indirection), the direct revocation approach suffers from the same shortcomings as the indirect approach.

Finally, the third method makes use of a server, used as a first level of decryption for all decryption requests. The second level of decryption occurs on the end of the DR. If a DO wishes to revoke the capacity for a given DR to access their SDO, the request is sent to the server. When the revoked DR wishes to decrypt the SDO, the server does not execute the first decryption phase thus providing revocation.

Of the three revocation mechanisms, the server approach is the only method which we can consider for our use case, albeit it is not perfect. Liu et al. indicate that, in this naïve approach, the multi-step decryption procedure is vulnerable to collusion attacks. This is because a non-revoked user Eve could assist a revoked user Bob by having the server execute the first pass decryption on some SDO. Eve could send the resulting intermediate ciphertext to Bob who would be able to decrypt it. The author's contributions [9] make use of revocation lists, key updates, and expiring keys. The revocation lists are kept short since they only contain users who have unexpired keys; users with expired keys need not be added. As for the revocation mechanism, they add a revocation list in the ciphertext. Additionally, they set a duration for the validity of a ciphertext, effectively making the ciphertext unreadable without being updated. Given our use of blockchain as a data store, the use of this method would leave large portions of the blockchain expired and unreadable by our users.

Having discussed the revocation mechanisms, it is clear that revocation, in the context of our environment, is hard to achieve with ABE. As discussed in other sections, indirection is used by many existing systems. However, in most cases, the indirection is achieved through the use of servers, or other "semi-trusted" entities. For an environment as constrained as our own, revocation is hard to achieve without sacrificing on other requirements such as confidentiality and low-connectivity.

2.3 Shortcomings of ABE in Blockchain Environments

Initially, attribute-based encryption seemed like a viable solution. With it, we could encrypt data on the basis of individual attributes for an object, yielding fine granularity of access. The cognitive load caused by the management of multiple attribute keys was a key factor preventing us from using ABE in our current implementation. A greater aspect of consideration is our choice of data storage medium: blockchain. The crux of the issue regarding an on-chain, blockchain-based data store with ABE as a source of confidentiality and access control is the revocation aspect in conjunction with the immutability offered by blockchain. Data may not be revoked if the data structure prevents their state from being changed.

One of the benefits provided by ABE is that of having an Access Control Structure (ACS) embedded in a ciphertext such as in the case of CP-ABE. This allows DOs to publish data and manage access to said data during publication. However, given that we are publishing these permission-encoded payloads to a blockchain, a data store which ensures immutability and integrity through hash-chains, the method by which revocation can be achieved is unclear. With blockchain as a data store, SDOs cannot benefit from both the fine-grained access control provided by ABE as well as revocation. If the document is published with an embedded ACS, the revocation of any part or whole of the ACS is prevented by the immutability of the blockchain.

2.3.1 Roles

Many of the studied systems had distinct roles for users of their system and do not consider the possibility of a multi-role environment. That is, in traditional patient-healthcare provider environments, the ability to read and manage data is unidirectional; patients produce data that is administered and read by healthcare professionals. In our case, data owners may themselves act as data readers for other data owners in the system. Issues become apparent in ABE systems when assumptions on the trustworthiness of roles are at play. The solution proposed by [1] assumes that attribute authorities are semi-trusted actors and that data owners are completely trusted. Given our multi-role environment, systems which do not consider multi-role actors fall short and are inadequate for our use case.

Some authors present servers as actors in their model. In their 2017 paper, Ma et al. [10] describe the use of expiring attribute-based encryption. Through the use of semi-trusted re-encryption servers, their system creates keys which expire after a given amount of time. Although their model makes use of proxy-re-encryption time servers (which disqualifies their solution given our environmental constraints), they assume some level of trust in the re-encryption server.

2.3.2 Revocation

Much of the current research does not take into account the environmental constraints imposed by our system. External servers are often used as a proxy in the encryption / decryption phases of their schemes ([11], [12]). Additionally, others achieved revocation via a discontinuity in key publishing [13]. While the authors do note that this revocation is partial, our requirements are such that revocation cannot be predicated on the fact that a data owner ceases to provide a revokee with data since "backwards" revocation is required. Likewise, some solutions brought forth the use of revocation lists [9]. Again, our decision to use blockchain as a data store negated the provided solution's ability to revoke access to data since the revocation lists are embedded within the ciphertexts. Thus, it follows that previously published data cannot be revoked given the immutability of our data store.

In conclusion, for the reasons outlined above, and in conjunction with our set of rigid constraints, the use of ABE to protect data stored in blockchain without the use indirection was forgone. Existing solutions, while novel, do sacrifice certain aspects, be it usability due to costs accrued via incentivisation schemes, confidentiality due to assumptions made regarding entity roles, or generally because of their highly distributed nature.

Chapter 3

Existing Systems

A review of existing systems was completed in order to determine what work had already been done. We now introduce a few of the existing blockchain-based solutions which make use of an ABE-scheme for confidentiality.

3.1 Assumptions of Honesty

In the MIStore system [14], blockchain is used to store data in nodes (servers) on the network. Nodes of the system are responsible for storing some subset of published blocks. In contrast to our low-connectivity requirement, if a user goes offline, data becomes unavailable. A workaround to this would be to ensure that all nodes on the network hold a copy of the data. However, doing so would vastly increase storage costs. Additionally, the authors assume that nodes act with some level of honesty in order to provide data confidentiality. Their system also supports the sharing of data belonging to actors (other than the original DO) with other DRs, a function we would like to avoid given our need for DOs to be in complete control over their SDOs.

3.2 Distributivity

Wang et al. [15] describe an architecture where pointers to data on an external filesystem are published to the blockchain. Specifically, pointers reference SDOs located in an interplanetary file system (IPFS). The use of IPFS requires the high availability of storage nodes. Due to our low-connectivity requirement, pointer-based solutions are unsuitable due to their use of remote servers. Additionally, if a system goes offline, pointers reference SDOs that are unavailable. They also make the assumption that nodes operate in good faith, which is problematic in our untrusted environment.

3.3 Incentivization

Another interesting design choice taken by certain solutions [15] is the use of incentives and scarcity, an example being the use of cryptocurrencies such as Ethereum. The use of Ethereum implies a cost associated with the storage of data. This is problematic since (1) low-power systems are likely to be the only ones available in our environment and (2) we do not want to create a system which limits the publication of blocks. Instead, we want users to publish as much data as they can in order to feed clinical decision support systems; we are using blockchain for its cryptographic properties that protect data from alteration.

Additionally, the authors indicate that smaller ABE ciphertexts are preferred due to their reduced publication costs. We do not want create a system where small publications are preferred over larger ones for a few reasons: (1) costs associated with publishing data are likely to discourage the publication of data; and (2) publishing costs modify user behaviour and create a potential for bias in the analysis of data.

3.4 Data-less Chains

The authors of [1] use blockchain as a means to store access policies as opposed to data. Access grants and revocations are tracked via transactions on the blockchain. While their solution does not directly store any data, we mention them since it may be interesting to use their model in a multi-blockchain environment where requests for access to on-chain SDOs are published to a parallel blockchain. This provides an immutable audit log of access grants and revocations, something we discuss in section 4.2.3.

3.5 True Data Ownership

Wang et al. [16] present a distributed system where data producers are hospitals, and data owners are patients. In contrast, in our system, both the DO and data producer are the same. We need a system where there is a single source of data, as opposed to one where multiple stakeholders cooperate in the publishing of data. There is additionally a reliance on a central authority which generates keys for users, something we are trying to avoid due to the potential confidentiality breaches. Furthermore, the data is not stored in the possession of the DO, but in that of the hospital, raising another issue with confidentiality. Their system is one which focuses on the collaboration of multiple healthcare providers, all contributing data for the construction of a patient profile. We are looking for a solution which is more individualistic in the generation and control of user data.

Chapter 4

Our System

The requirements and use case of our application are described earlier in this work. Unfortunately, we found no prior work in technical literature that perfectly matches our requirements. Every existing implementation we considered possessed glaring issues that rendered it unacceptable for our purposes. Therefore, we decided to implement our own system from scratch.

Our system provides integrity of data by storing all data in a single blockchain, called the **master ledger**. The master ledger is the primary, immutable database that stores all user data. Additionally, a data structure called a **share tree** stores access permissions and allows for the secure sharing of data and flexible revocation of permissions.

Our system has the following properties not provided by out-of-the-box blockchain:

- **Confidentiality**. Authors can choose to encrypt their data before publishing, so that only they and authorized users may decrypt it.
- **Authenticity**. Blocks are digitally signed by their authors, so attackers may not forge data or masquerade as other users.
- **Sharing and Access Control**. Information about shared data is stored in a separate data structure, the **share tree**. The share tree defines read permissions and enables users to data that was shared with them.
- **Revocation**. A user may revoke access to previously-shared data by removing the corresponding entry or entries from the share tree.
- **Simple key management**. Users should not need to maintain a potentially large keystore. The only essential information that a user must retain is two keypairs — one for encryption and the other for signing.
- **Ease of use**. Key generation, retrieval, and distribution are completely automated and transparent to the end user.

- **Passivity**. All of the above properties are cryptographically guaranteed; no active access control or authentication layer should be necessary to safeguard data.

Our system implements the client-server architecture, where one or more users run client software that communicates directly with a single server. This approach is convenient for our use case, where we have a small number of users in an isolated environment. The central server enables there to be a single source of truth for the state of the system and data stored within.

We suspect that we can employ consensus algorithms used in existing distributed blockchain systems (like Bitcoin [17]) to scale our system onto a distributed architecture, but that is beyond the scope of this work.

The server component of our system is a single point of failure, from a security standpoint. It is plausible that a user could have physical access to the server, allowing them to install malware or eavesdrop on its network connections. We therefore operate under the assumption that the server is *completely untrusted*. That is, any data stored on the server, any network traffic going to or from the server, and even any machine instructions being executed on the server, are insecure.

Remark 1 (The Untrusted Server) *At any time, the server must be assumed to be compromised, and the state of the server must be assumed to be public knowledge.*

Although this property seems like a draconian restriction, it does have benefits. If the server is assumed to be compromised at any time, we may as well make available all of its data, and subject the server to public scrutiny. This leads to the following observation:

Remark 2 (Public Validation) *If the state of the server is made public, any user may inspect its data for signs of tampering.*

In fact, Remark 2 is one of the core tenets of a public blockchain — if data is public, tampering can be easily detected. A healthy blockchain is frequently validated by several independent parties.

Remark 1 also carries the important implication that any sensitive information must never leave the client's machine. Thus, all encryption, decryption, and signing operations must take place on the client-side. We elaborate more on this observation on page 20.

4.1 Master Ledger

The **master ledger** is a linear blockchain that serves as the primary data store for user data. Since the master ledger is a blockchain, it inherits the cryptographic properties that render data resistant to alteration. In addition, every block in the ledger is digitally signed by the author before publishing, providing authenticity and protecting against forgery and impersonation.

A block in the master ledger typically corresponds to a record that may be shared with other users. The interpretation of a "record" is subjective. Typically, a record is meant to represent a single file, report, post, or piece of content that is published or syndicated at once. The following all are good examples of records:

- A blog post
- A social media post, like a tweet
- A media file
- An article from a journal
- A form, such as a medical self-assessment
- A document, report, memo, or briefing

In this work, we use the term **record** to mean a block in the master ledger that contains user data. In contrast, a **block** is the general data structure that makes up a blockchain, and may not necessarily represent a record.

A good guideline is that records should be shareable atomically. That is, it no part of the record should have different access permissions than another part. This simplifies the process of sharing a record with other users, but also allows creating and enforcing security policies that treat records as the fundamental unit of data. We shall see in Section 5.1 that our system is flexible enough to handle different security policies at one time, all revolving around the concept of records. If a user wishes to make parts of a record have different permissions, they must either split the record into multiple parts, or publish a redacted version of the record with appropriate permissions.

4.1.1 Structure of a Record

Every record has a set of **public attributes**, which are key-value pairs that can be read by anyone. Public attributes may be arbitrarily defined by the author of a record, but each record has at least the following:

- Hash
- Hash of the previous block
- Timestamp
- Author ID
- Signature

Public attributes are essential to maintaining the integrity of the master ledger. Any user may query public attributes and use them to validate the ledger. The **Hash** attribute ensures that the contents of the record have not

been modified after publishing, and that the server completed the publishing procedure reliably. The **Author ID** and **Signature** attributes ensure authenticity and protect against forgery. The **Timestamp** and **Previous Block Hash** attributes ensure consistent ordering of records within the master ledger.

A record may also include any number of **secret attributes**, which are confidential. Before an author publishes a record, they encrypt the set of secret attributes such that they can easily decrypt them. Two possible encryption schemes are discussed below. A special secret attribute, the **payload**, designates the confidential body of the record. For example, if a user wishes to upload a file, the bytes of the file would be stored in the record's **payload** attribute.

4.1.2 Initializing the Master Ledger

When a master ledger is created, it is initialized with an empty **origin block**. The origin block has a null signature, null previous block hash, and a special public attribute that designates it as the origin block. All other blocks in a blockchain are descendants of the origin block.

4.1.3 Creating and Encrypting a Record

Publishing data to the master ledger creates a permanent record that is computationally infeasible to alter. Suppose that Alice wants to publish some data to the master ledger. Creating a record requires two calls to the server:

1. **Request to publish**: Alice creates a lightweight block containing the data she wants to publish, encrypting it if necessary. She sends this block to the server, signaling her intent to create a record.

2. **Sign and publish**: The server appends necessary attributes to the block (such as the timestamp and previous block hash), and sends the updated block back to Alice. Alice signs the updated block and sends back the signature. The server then appends the complete block and signature to the master ledger, creating a record.

Both steps are atomic — they must be executed sequentially, by the same user, to maintain the validity of the master ledger. To see why, suppose that both Alice and Bob wish to publish a block.

1. Alice completes step 1 and receives block B_1 from the server. Let B_0 be the last block in the master ledger. Let $H(B_0)$ be the hash of B_0. B_1 contains $H(B_0)$ in its `previous block hash` attribute.

2. Bob completes step 1 and receives block B_2 from the server. Since Alice's block hasn't been published yet, the previous block hash in B_2 is still $H(B_0)$.

3. Alice signs B_1 and completes step 2. The server publishes the block, and the last block in the master ledger is now B_1.

4. Bob signs B_2 and complete step 2. However, the last block in the master ledger now is B_1, while Bob's block, B_2 still contains $H(B_0)$ as the previous block hash. Publishing B_2 would leave the system in an inconsistent state, so the server rejects B_2.

Thus, the process of creating a record is a synchronous process that requires two calls to the server. Users are blocked from creating records while one user has not terminated the procedure. This presents an easy surface for denial-of-service attacks, and so the system must be protected by limiting the frequency of calls from any particular user.

Proof of Work

In public, distributed blockchains such as Bitcoin [17], there exists a procedure known as **proof of work** that prevents users from forging data. For instance, the Hashcash mechanism (employed by Bitcoin) only accepts blocks whose hash begins with a certain number of zero bits [18]. A block is updated with a random nonce attribute until its hash satisfies this condition, at which point the block is accepted by the system and is added to the blockchain. For an attacker to alter or forge data, they must recalculate or otherwise derive a hash that satisfies the proof of work for the chosen block, as well as for every other following block (since the `previous block hash` attribute in the following block would be wrong otherwise). Naturally, brute-forcing a suitable hash using Hashcash requires a great deal of computing power and time, and so altering a block quickly becomes computationally infeasible.

In our system, it is important for users to be able to publish data immediately, which disqualifies the above procedure. Instead, we use digital signatures in place of proof of work. If a malicious user compromises the server containing the master ledger, they cannot alter or forge data without being able to generate their victim's signature.

A malicious user who compromises the server *can* replace the signature of the target record (and every following record) with their own signature. However, this publicly identifies them as an attacker to other users, and puts the trustworthiness of their own records in question. Further, if other users retain their own copy of the master ledger (which is allowed and encouraged), they can identify any altered signatures. This reveals another vital property of our system over existing blockchain implementations:

Remark 3 (Attackers are Stakeholders) *Every user has a mutual stake in the trustworthiness of the system, discouraging would-be attackers from tampering with other users' data.*

Proof of work also ensures that the security of records in the master ledger will increase with time. If an attacker has access to the server and wishes to alter the final record in the master ledger, they may alter the record's content on disk and replace the signature with their own, making the record appear valid. If the attacker does this before any other user has had the chance to make their own

copy of the master record, this action may be undetectable. However, to alter a record in the "middle" of the ledger, the attacker must replace the signature on that record and every subsequent record. This behaviour appears highly suspicious and is nearly guaranteed to be detected by other users, especially if data are noticed to be missing.

Remark 4 (Security Increases with Time) *The security of a record in the master ledger increases the longer it has been in the ledger. Specifically, for a given record R, every record published after R increases the chance that a modification of R will be detectable.*

Encrypting and Decrypting Blocks

A record may contain any number of secret attributes, one of which is the payload (the primary content of the record, such as a document or media file). Before a record is published, those secret attributes must be encrypted. In this procedure, all secret attributes and the payload are concatenated into a single byte string, padded with a random nonce to protect against chosen-ciphertext attacks, then encrypted using a suitable algorithm. The resultant ciphertext is then appended to the block as a public attribute.

We explored two schemes of encrypting secret attributes:

- Symmetric, in which the secret attributes are encrypted under a randomly-generated symmetric key S. S is then encrypted under the author's public key, and the result is appended to the block as a public attribute.
- Asymmetric, in which the secret attributes are encrypted under the author's public key.

The symmetric scheme performs faster and simplifies the process of sharing data, as we shall see later. Even though this method introduces at most one new symmetric key per published record, keys are stored as attributes on the record and are only retrieved when needed. The user needs only to remember their encryption and signing keypairs. In fact, we have demonstrated in a simple reference implementation that key management in the symmetric scheme can be made completely transparent to the user.

Encrypting a record under the symmetric scheme adds the following public attributes:

- Symmetric key S encrypted under the author's public key
- Ciphertext of secret attributes

If a block contains no secret attributes, S will not be generated.

To decrypt a block, the author first decrypts S using their private key, decrypts the ciphertext using S, and reconstructs the original set of secret attributes.

It is essential to note that all encryption and decryption operations are performed by clients. This property makes it possible for our system to operate

in spite of Remark 1. Since the server at no point handles plaintexts or private keys, its operation has little bearing on the confidentiality of user data.

Remark 5 (Client-Side Encryption) *All encryption and decryption operations are the responsibility of clients. The server never handles any plaintext or private keys, thus, it cannot eavesdrop on confidential data.*

4.1.4 Retrieving a Record

A core feature of blockchains, and our system, is that every record should be retrievable and validatable by any user. A user can look up a record by its unique hash.

Retrieving a record returns its complete set of public attributes, and the user may decrypt the record to retrieve its set of secret attributes. Of course, decryption is only possible if the user possesses the correct key, either from authoring the block or receiving the key via a share.

4.1.5 Validating a Record

Having retrieved a record, a user may compute its hash. All public attributes, including ciphertext of secret attributes, are considered when computing the hash, except the `Hash` and `Signature` attributes. Verifying the hash allows a user to ensure integrity of the block.

It is important that the user's record-hashing algorithm is the same as the one employed by the server. In particular, the order in which attributes are hashed, byte-encoding of strings, endianness of numbers, method of encrypting secret attributes, etc. must match the server's record-hashing algorithm.

Given that the author ID is a public attribute, it is possible to look up the author's public key and verify the record's signature against the computed hash. Verifying the signature allows a user to ensure integrity and authenticity of the block.

If either the hash or signature do not match the computed hash of a retrieved block, the data in the block must have been altered, and can no longer be trusted.

4.1.6 Validating the Master Ledger

A blockchain is essentially a reverse-linked list. Blocks contain pointers to the previous block in the chain, all the way up to the origin block, whose previous block hash is null. Since the previous block hash is a public, hashable attribute on every block, a user can validate the entire master ledger by starting at the final block and recursively validating from the previous block, as in the following algorithm:

```
function validateRecursive(block):
    if block == originBlock:
        # Exit condition
```

```
        return true

    assert isValidHash(block)
    assert isValidSignature(block)
    prevBlock = getBlock(block.previousBlockHash)
    assert prevBlock.timestamp < block.timestamp
    return validateRecursive(prevBlock)
```

4.1.7 Implementation Notes

In our reference implementation, the master ledger stores records in a hashmap keyed by the hash, allowing lookups in `O(1)` time. To enable fast lookups on public attributes, indexes may be implemented on select attributes, similar to a database management system. For example, an index on the author ID attribute would be useful for users to quickly retrieve their own records.

4.2 Share Tree

Public attributes of a record in the master ledger are readable by all users, but secret attributes are, by default, decryptable only by the author. There must exist some mechanism for sharing secret data with other users. We have particular requirements for such a mechanism:

- It must be possible to share secret data with any number of users, groups of users, or no users at all.
- It must be possible to revoke a user's access to data at any time.
- It must be possible to manage access permissions at either coarse or fine granularities.
- The process of sharing must not alter data stored within the master ledger.
- Access control must be enforced cryptographically, with no active access control layer. That is, any security policies or permissions must hold even if the server is compromised and its safeguards disabled.

To achieve these goals, we extended the concept of a linear blockchain into a tree-like structure, called a **blocktree**. In a blocktree, each block contains the hash of its parent block, except for the root of the tree, which is the origin block. We use a blocktree to store access permissions separately from the master record — we call this structure the **share tree**.

The share tree contains a subtree rooted at the origin block for each user. Each user's subtree represents a capability list for that user — access to a record may be granted to a specific recipient by inserting appropriate information into the recipient's subtree. Because the share tree is based on a blockchain, it also

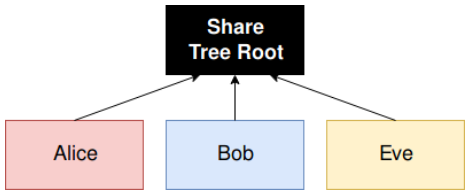

Figure 4.1: A blank share tree, containing nodes for Alice, Bob, and Eve. Each one of these nodes is the root of that user's share subtree.

inherits the same properties of the master ledger that keep its contents safe. Particularly, Remarks 1, 2, 3, and 5 also apply to the share tree.

Blocks in a user's subtree can represent a permission granted to that user. We call these permission-containing blocks, **"shares"**. Not all blocks need to be shares. In fact, we will see that using non-share blocks becomes incredibly useful for organizing permissions, allowing granular management of permissions, and auditing.

4.2.1 Sharing

Suppose that Alice wants to share a confidential document with Bob, and Alice used the symmetric scheme to encrypt the document. Alice has already published a record containing the document in the master ledger.

Let:

- R be Alice's record in the master ledger.
- H_R be the hash of R.
- S_R be the symmetric key used to encrypt R.
- H_B be the hash of the root block of Bob's share subtree.

Recall, the encryption of S_R under Alice's private key is stored in R, so that only Alice can retrieve S_R.

To share the confidential document, Alice must create a block under Bob's share subtree containing S_R. She follows a procedure similar to publishing to the master ledger:

1. **Retrieve information**: Alice retrieves the record she wishes to share and decrypts S_R.
2. **Request to publish**: Alice creates a lightweight block containing H_B as the parent block hash, and H_R and S_R as secret attributes. She encrypts them using the same procedure as before, but she uses Bob's public key instead of her own. She sends the block to the server.

3. **Sign and publish**: The server appends necessary attributes to the block, and sends the updated block back to Alice. Alice validates the updated block and generates a signature Sig. She encrypts Sig under Bob's public key and sends the ciphertext to the server. The server adds the complete block and encrypted signature to the share tree.

This process has three important differences from the master ledger publishing procedure.

1. Alice must manually specify the parent block hash. Whereas the master ledger appends to the end of the chain without exception, a sharer specifies the parent block hash to indicate the identity of the recipient, or the parent context of the share (described in the next section).

2. Alice encrypts her signature under Bob's public key. Encryption is necessary to prevent leaking the sharer's identity through their signature. This has the unfortunate tradeoff that the share cannot be validated by anyone but Bob. However, Alice may choose to not encrypt her signature, allowing public validation of her share and potentially increasing security, at the cost of revealing her as the sharer.

3. There is no longer a critical section around steps 2 and 3. Multiple users may create shares concurrently.

This procedure results in a new share being inserted into Bob's subtree, illustrated by Figure 4.2. The new share points to Alice's shared record in the master ledger, and contains the symmetric key S_R needed to decrypt the record. Bob can decrypt the share using his private key, then decrypt Alice's record in the master ledger using S_R.

Sharing using the Asymmetric Scheme

The above procedure described how to grant access to a record encrypted using the symmetric scheme. What if a record were encrypted purely with Alice's (the sharer's) public key? There exists no way, short of Alice revealing her private key, for other users to decrypt the data. In this instance, Alice must create a duplicate of the record's secret attributes, re-encrypt them under Bob's (the recipient's) public key, then publish the ciphertext to the Bob's subtree. This effectively grants Bob a read permission to a copy of the original record.

With the decrypted secret attributes in hand, Bob may reconstruct the original record using its public attributes, obtained from the master ledger. He may then encrypt the secret attributes under Alice's public key and compare the ciphertext to the original record's ciphertext to ensure integrity. If they match, Bob can be confident that the share has not been tampered with.

The asymmetric encryption scheme poses several problems:

- Secret attributes are duplicated every time a record is shared. If a record is shared with multiple users, the storage costs rise accordingly. This is a

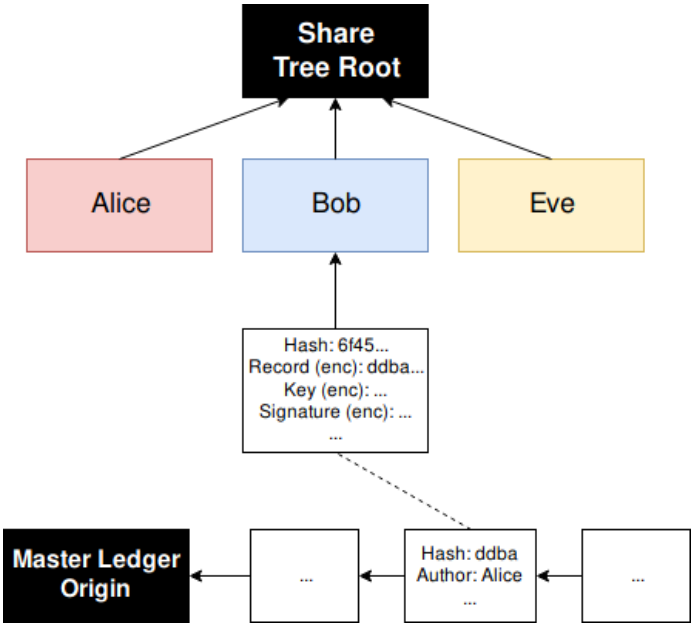

Figure 4.2: The blank share tree from Figure 4.1 after Alice shares a record with Bob. Note how the share references Alice's record from the master ledger. Secret attributes are annotated with `(enc)`. Not all attributes are shown.

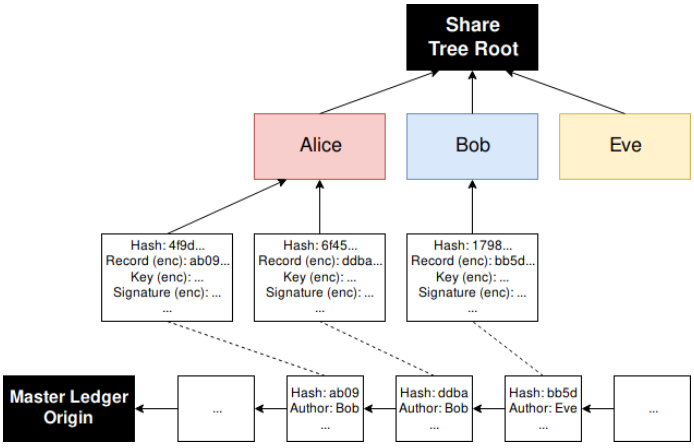

Figure 4.3: A more complex share tree, after Bob shared two records with Alice and Eve shared one record with Bob.

significant issue for two reasons. First, if users share large files, like videos or high-resolution documents, the storage and time cost of sharing may become prohibitive. Second, in remote or performance-limited environments (like spacecraft, or embedded systems), the size of the share tree may eventually exceed the environment's storage capacity. As the cost of high-density storage decreases, however, we anticipate that storage costs will become less of a bottleneck.

- Asymmetric algorithms are slower and require larger key sizes than equivalent symmetric algorithms. Efficient hardware accelerators for cryptographic operations exist, such as as dedicated platforms by Intel [19] and the Intel AES instruction set for x86 and x64 CPUs [20]. However, since all cryptographic operations are performed on clients, it may not be reasonable to expect all clients to have sufficient performance, especially in the case of embedded systems.

- Sharing using the asymmetric scheme is considerably more complex, and requires additional encryption steps in order to validate shared records. Much of the additional complexity is on the client side, which may discourage third parties from developing their own client software, hindering adoption.

The only major benefit of the asymmetric scheme is that it does not generate an additional symmetric key for every new record. At first glance, the asymmetric scheme appears to simplify key management. In practice, we found that managing keys in the symmetric scheme is quite painless and completely transparent to the end user. That said, since encryption is the responsibility of clients, it is possible to use either scheme with no change to the server component, or even use both schemes at the same time.

4.2.2 Granularity and Contexts

We can use the hierarchical nature of the share tree in order to organize permissions. One of our system requirements was that permissions relating to related records (for example, a user's blog posts) should be managed together. We can achieve this by adding blocks into a user's share tree that do not contain data, but serve as containers for future shares. We call these blocks, "**context roots**". The tree rooted at a context root is a **context**, or a group of logically related shares.

Suppose that Alice generates daily reports that she wishes to share with Bob. Alice wants to keep these shares logically grouped. Further, she wishes to organize these reports by year. Alice would create a context root under Bob's share subtree, labelled *Reports*. She would then create additional context roots under *Reports* for every year. Whenever Alice publishes a report, she would create a share in the appropriate context for that year, granting Bob read access to the report.

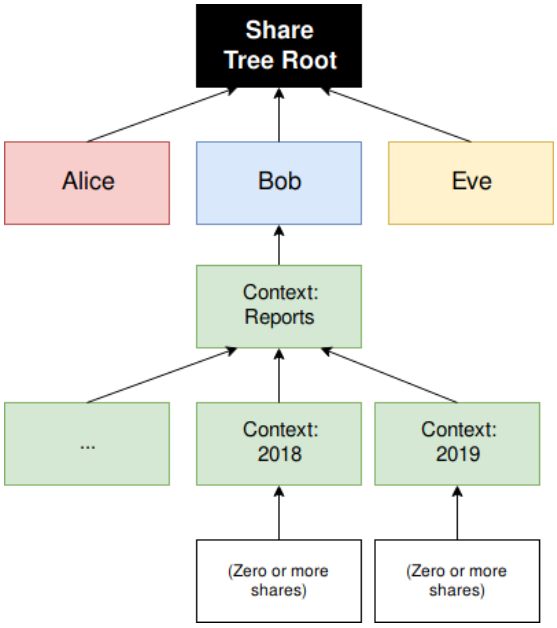

Figure 4.4: A share tree showing a hierarchy of contexts for the daily report example. Several shares may exist as leaves under a context. Green nodes represent context roots.

If Alice decides not to share a single report, she would simply not create a share in Bob's share subtree. Thus, Alice still has control over which records she wishes to share, regardless of the record's membership in a particular context. In fact, Alice can share the same record with another user in an entirely different context, or share it several times under different contexts. Contexts apply to the shares, and not the records themselves.

4.2.3 Revocation

Revocation of data access permissions is an essential part of any access control system. Permissions granted to a user should be revoked if the user leaves the organization, has their private keys compromised, or becomes malicious themselves. Further, in a high-security system, users should not retain access to data they no longer need, and unnecessary permissions should be regularly revoked (this is the *Principle of Least Privilege* [21]).

When using an append-only, immutable data store such as a blockchain, it may seem difficult to revoke access to data once granted. After all, revoking a permission involves mutating the state of an existing permission, and mutation of data is expressly forbidden in blockchains. Part of the share tree's purpose is to avoid this restriction. By storing permissions in a structure separate from the immutable master ledger, we can allow mutating the share tree while keeping the data records themselves safe.

Revocation is handled by removing the appropriate share from the share tree. For example, if Alice shares a record with Bob by mistake, she can request that the server remove the offending share. After revocation, Bob will no longer see that share in his share tree, and so will not be able to decrypt Alice's record.

This method of removing shares is particularly powerful when combined with contexts. Continuing the daily report example above, if Alice decides to revoke Bob's access to all reports, she can revoke his access to the entire *Reports* context. In this way, permissions can be revoked in bulk. Removing a branch from the share tree does not compromise the structure of the blockchain if and only if all descendants of a removed node are removed as well.

While the process of revocation alters the share tree, it is impossible to alter the shares themselves, since each share is protected by a hash and signature. Any modification to a share can be detected by the recipient of the share, who can choose not to trust the share.

Revocation by pruning a branch from the share tree is illustrated in Figure 4.5.

Authentication

There must exist some mechanism that prevents users from revoking shares that they did not create. Otherwise, a malicious insider can revoke any share as soon as it is created, and deny service to legitimate users. Moreover, this mechanism must not leak the sharer's identity to other users

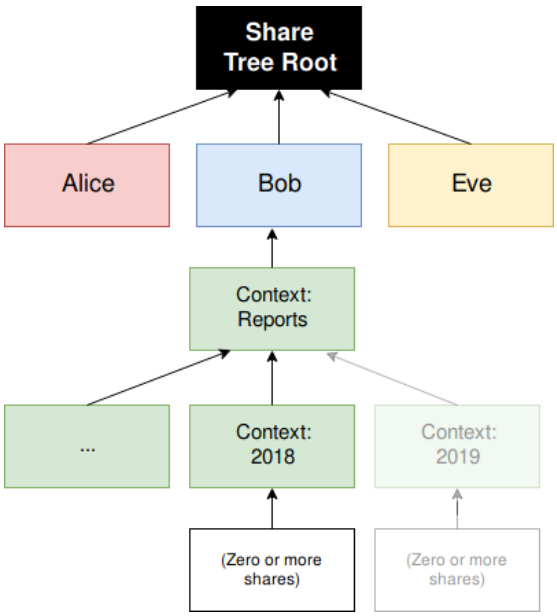

Figure 4.5: A share tree with several contexts, showing how a branch can be pruned without compromising the structure of the blocktree. The faded *2019* context shall be revoked. It is equally possible to revoke individual shares, or the top-level *Reports* context.

One suitable mechanism involves using one-time passwords. Whenever Alice creates a share, she generates a random token T and appends the hash of it, $H(T)$, to the share as a public attribute. Later, if Alice chooses to revoke the share, she supplies T, that only she knows. The server processes the revocation only if the hash of the supplied token matches the $H(T)$ in the share.

This approach requires the generation and storage of a token for every newly created share. This is an acceptable tradeoff, for two reasons:

1. A user's client software must somehow keep track of all shares that they created. Otherwise, they must crawl the entire share tree in order to retrieve them, which is inefficient. It would be trivial to store a small token along with this information.

2. If it is not trivial or if it is inconvenient to store this token on the client side, the token can be encrypted under the sharer's public key and stored as a public attribute on the share, similar to how symmetric keys are stored on records in the master ledger.

Authentication is an active process — the server must deliberately ensure that revocations are performed by authorized users. Understandably, an attacker who compromises the server may bypass this procedure and arbitrarily delete shares from disk. However, the share tree contains no data that cannot be easily recreated by users — lost permissions may be granted again with little trouble.

Auditing

It may be useful to track when a share was revoked, and by whom. The share tree supports this style of auditing by replacing a revoked share with a placeholder block that signifies a revoked permission. Such a block might contain (some of) the following attributes:

- Date and time that the original permission was granted
- Date and time of revocation
- Identity of the revoker
- Hash of the corresponding record(s) in the master ledger
- Reason for revocation

By recording all revocations, an audit log of grant and revoke operations can be constructed by reading the timestamps on each block. To avoid polluting the share tree with audit blocks, a separate blockchain may be employed exclusively for storing audit information.

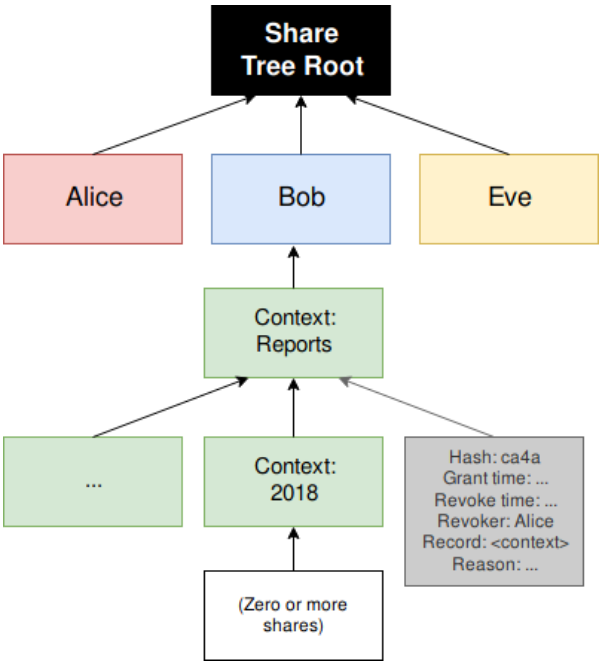

Figure 4.6: The share tree from Figure 4.5 after the *2019* context has been revoked and replaced by a placeholder block containing information about the revocation.

The Revocation Problem

Once any data is shared with other users, there is no guarantee that their access can be successfully blocked. Nothing prevents a malicious or careless user from saving the symmetric key, or the corresponding decrypted record, on their local machine.

This problem can be somewhat mitigated with well-written client software. Ideally, for every access to a shared record, the client should always query the share tree for the required share, and not cache the share locally. Otherwise, the client is saving the symmetric key required to decrypt the shared record on the recipient's machine, with no guarantee that it will be safely or timely deleted once revoked. Well-behaved clients can protect against careless users, but are powerless against deliberate attackers.

In our view, this is an unsolvable problem that also plagues conventional access control systems. Once access is granted to a resource, the sharer must trust the recipient to not handle the resource inappropriately.

One might wonder, if the revocation problem cannot be perfectly solved, and if compromised data cannot be removed from a blockchain, is revocation entirely necessary? It absolutely is. In fact, revocation is extremely important in a circumstance where a legitimate and responsible user (say, Alice), who does not cache shares or keys, is compromised by an attacker (say, Eve). Even if Eve steals Alice's private keys and can masquerade as Alice, Eve will not be able to access records for which Alice's access was revoked, precisely because those permissions were removed from Alice's share subtree. If the system did not support revocation, Eve would have full access to any records shared with Alice in the past.

The previous point illustrates the importance of the Principle of Least Privilege. If permissions to access a record are no longer needed, they should be revoked. Revoking a share prevents any future attackers from accessing the data contained within the share.

4.2.4 Role-Based and Group-Based Access Control

So far, we explored how records can be shared with single users via the share tree. What if Alice wants to share a record with several users, say, Bob and Charlie? One option is to create two shares — one for Bob and one for Charlie — but this is inefficient and requires Alice to revoke two different shares down the line.

Share trees support the creation of user roles or user groups by creating a subtree for each role or group. In this case, a subtree for the Bob-Charlie group would hold shares that are accessible by either Bob or Charlie.

To allow any member of the group to decrypt shares, shares may be encrypted in several ways:

- Conventionally, by generating a random symmetric key S and encrypting S under each member's respective public key.

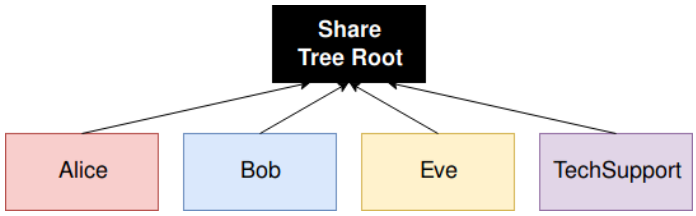

Figure 4.7: An empty share tree, similar to Figure 4.1, with an extra subtree root for the `TechSupport` group. From the perspective of the share tree, this group is treated as a regular user, and receives no special treatment.

- Using an attribute-based encryption scheme on attributes of users (department, salary, security clearance, etc.)
- Using a secret-sharing scheme such as ones proposed by Shamir [22], Blakley [23], or Mignotte [24], and distributing key fragments among members of the group.

A detailed analysis of role- and group-based access control using share trees was beyond the scope of this work, but it remains a topic that we are eager to explore in the future.

4.2.5 Time to Live and Denial of Service

One important consideration regarding the sharing of data is the length of time that a share should stay valid. If shares never expire, this opens up the possibility of denial-of-service attacks by polluting the share tree with bogus shares that are never cleaned up.

To avoid this situation, the server must periodically check for expired shares, and remove them from the share tree. Naturally, there must exist a mechanism for users to renew legitimate shares if needed, requiring active participation by the sharer.

Imposing a time to live on shares also enforces the Principle of Least Privilege, as unused shares are automatically revoked and become unavailable to their recipients after a certain time.

4.2.6 Implementation Notes

We were able to implement a basic share tree in our reference implementation of the software. We demonstrated that records can be created, shared, retrieved, and decrypted by the recipient with very little user intervention. Though theoretically possible in our implementation, we did not demonstrate context management or group-based access control. Revocation remains a pending feature.

Chapter 5

Blocktrees

While developing the share tree model for data sharing, we discovered that blocktrees have several properties that make them attractive in computer security applications. In this chapter, we list some of these properties, possible applications, and the significance they have on the security of a data storage system.

5.1 Different Security Policy Property

The first property that we discuss is that different branches on a blocktree may have different security policies imposed on them. From an organizational standpoint, this is extremely useful in a multi-user environment where users may have different roles or belong to different departments in a company. Each role or department would have its own branch in a blocktree, and may enforce whichever policies they see fit on their own branch.

The advantage of using a multi-branched blocktree instead of deploying several different blockchains is one of transparency and auditing. A blocktree shared across many departments encourages good data-publishing practices, and gives every user a stake in the integrity of the system, as per Remark 3. If every department used a distinct blockchain, users could tamper with other departments' blockchains with no consequence to the trustworthiness of their own data.

Further, a blockchain or blocktree system with many users is inherently safer, since more users are publishing data. Blocks which have many descendants are more resistant to tampering, by virtue of Remark 4. However, since an attacker with access to the server can delete data from disk or overwrite signatures with their own, appropriate intrusion detection measures must remain in place.

5.1.1 Immediate-Deferred Blocktrees

We can exploit the Different Security Policy property to fix an important limitation of the master ledger-share tree model. This limitation relates to a possible network snooping attack when sharing records. If an eavesdropper, Eve, is listening to the network while Alice shares a record with someone, Eve will be able to see traffic going to the server. Since all shares in the share tree are public and timestamped, Eve can cross-reference shares with the time of the transmission and determine the recipient and public attributes of the incoming share.

A similar vulnerability exists in the master ledger. The public timestamp of records, while essential to the ability for a user to validate the blockchain, can leak information about the record's contents. If Eve knows Alice's publishing habits (for instance, she may know that Alice publishes an encrypted medical evaluation at 7 PM every day), she can infer the contents of Alice's records by the timestamp.

A possible solution would be for the server to defer publishing of records until a known time in the future, or until a sufficient number of records has been submitted. However, if the server queues records, it will be unable to handle urgent records that must be published immediately (because an urgent record must know the hash of the latest record, which may be queued).

A blocktree can address this problem by having two branches: an **immediate** branch and a **deferred** branch. Records published in the immediate branch are made public right away, at the expense of possible inference attacks. Records published in the deferred branch are queued on the server and are not added to the ledger until a known point in the future. A user may choose to publish in either branch.

Note that, in using two different branches in a blocktree, we can apply two conflicting security policies (immediate and deferred publishing) in the same data structure.

5.2 Recursive Access Control Property

In the share tree model, a user, Alice, must explicitly share every record to a set of users, regardless of the shares' membership in a context. Using blocktrees, however, there is a method for Alice's records to be implicitly shared if they belong to a shared context.

Let us revisit the daily report example in Section 4.2.2. If Alice makes a report, she must first publish it to the master ledger, *then* share the record with every appropriate user or group. She must also ensure that she shares the appropriate context.

Suppose that Alice creates a context in Bob's share subtree containing a **context key** for that context. She can then do the following:

- Add a public attribute A to each report record, indicating the record's membership in the *Reports* context.

- Encrypt the report such that the context key is sufficient to decrypt it.

Bob can, at any time, query the master ledger for records with attribute A. Since Bob has access to the context key, he can decrypt those records. Additionally, since the context key lives in Bob's share tree, Alice can revoke Bob's access at any time.

Here, we have imposed access control to a set of child objects (the records) based on access to their parent object (the context). This procedure can be applied recursively to create a filesystem-like hierarchy of objects where a grant on an object also implies a grant on all descendants.

This recursive access control model can be extended to allow multiple inheritance, where an object may have more than one parent. Access control could be configured such that, in order to have implicit access to an object, a user must have access to all, one, or a subset of an object's parents. This model requires the use of an appropriate multiple encryption or secret sharing scheme, which is beyond the scope of this work.

5.3 User Management and PKI

A blocktree can be combined with PKI to implement a robust user management framework based on a chain of trust. Such a framework would have several attractive features:

- Resistance to tampering, in particular against the creation of fake identities
- Complete transparency to internal users and the public
- Auditability
- Soft revocation (revocation of a user's permissions)
- Hard revocation (soft revocation, plus recursive revocation of all permissions granted to others by the user)

A blocktree-based user management system has several components:

- The root of the tree represents a trust anchor, typically the developer or distributor of the system.
- Leaves of the tree represent end users, who have read-only access to the blocktree.
- All other nodes represent administrators, who have read-append access to the blocktree.

During the deployment process, the software distributor, acting as a trust anchor, nominates the system owner as an administrator by creating a block in the blocktree containing the owner's public key certificate and capability list. The distributor signs the block, vouching for its legitimacy. The new administrator may now apply the same process themselves, creating new end users and administrators.

Like the chain of trust in PKI, a user's credentials may be independently validated by verifying each certificate and signature up to the trust anchor. Because blocktrees are resistant to alteration, it is possible to store capability lists within them and remain confident that users may not alter them.

This system is a data structure only — it does not enforce any access control. Rather, policy enforcement software would use it to authenticate a user and determine whether they are authorized to perform a certain task, then carry out enforcement.

Since blocks cannot be altered, revocation must be handled using a revocation list, which may exist either as a known branch in the blocktree, or as a separate ledger. During the process of authentication, every user up to the trust anchor is checked against the revocation list. If a match is found, the policy enforcement point can choose to deny access.

The idea of using blockchain in PKI and internet authentication is not new [25]–[28]. If blockchain PKI can be combined with the other concepts we explored in this work, it may yield a powerful, secure, and transparent user management and data sharing system that goes beyond what PKI offers today.

We formerly envisioned using this kind of framework to implement user management in our system, but it fell out of scope for this work. We intend to explore user management via blocktrees in the future.

Chapter 6

Conclusion and Future Directions

In this work, we explored different software approaches of handling sensitive data using blockchain, particularly in the realm of healthcare. We provided an overview of attribute-based encryption (ABE), why it appears attractive, and why it fundamentally fails in a restricted and untrusted environment. We analyzed several existing blockchain-based data stores, and concluded that all compromise in at least one essential area. Finding no suitable existing solution, we devised one from scratch, the master ledger and share tree system, that appears to fulfill all of our requirements: tolerance to low connectivity, tolerance to insecure or untrusted environments, ability to share data at coarse and fine granularity, ability to revoke access permissions, ease of use, and cryptographic guarantee of security. We spent considerable effort discussing the security properties of this system. Finally, we presented additional properties and applications of blocktrees as they relate to our system.

We found that many existing solutions do not consider environments where data stores are held locally or in environments where communication may be intermittent. Many existing solutions using blockchain technology assume a higher level of connectivity. The original white paper on Bitcoin [17] describes a purely peer-to-peer system which allows for an alternative form of online payments; Much of the research on blockchain is in the context of highly distributed cryptocurrency systems like Bitcoin. However, more research is required into the properties of blockchain as a medium for data storage.

We identified specific design considerations for our system and any future solution:

- **Low-connectivity environments**: Blockchains may be chosen as the data-storage medium for their ability to highlight incorrect or altered data.
- **Multi-role environments** Existing solutions make assumptions regarding the trustworthiness of the actors but do not account for cases where

responsibilities (such as data creation, ownership, and consumption) are undertaken by the same actor actor.

- **Resistance against inference**. Inferring the contents of a record based on its metadata opens up an attack surface, even if data is encrypted.
- **Untrusted environments**. A system must operate securely even if the network or hardware is insecure or compromised.
- **Low cognitive load**. A system must be usable for non-technical users or users suffering from a mental disability.
- **Flexible permissioning**. Data must be able to be shared with other users. Access must be managed with coarse or fine granularity, and be able to be revoked at will.
- **Data ownership**. Users must be in full control of their own data. Users should be able to decide with whom they share their data.

In the future, we hope to develop our system into a robust platform for secure data storage, and make it available to the community as open-source software. We intend on exploring role- and group-based access control, implementing user management via blocktrees, and scaling our system into a distributed architecture.

We believe that our system has great potential in the real world, since it addresses a niche that is not filled by any other solution, to the best of our knowledge.

Glossary

- **Access Control Structure (ACS)** An access structure is a boolean expression built from a set of attributes. Also referred to as a policy, or access policy.
- **Attribute Authority (AA)** An entity which declares the universe of attributes with which keys or ciphertext are created.
- **Attribute-Based Encryption (ABE)** An encryption scheme wherein attributes dictate a user's ability to decrypt ciphertext.
- **Block** In a blockchain, a block is an atomic data structure that stores data. Many blocks are cryptographically tied together in a linked list to construct a blockchain.
- **Blocktree** A blockchain, arranged in a tree-like structure, where every node (except for the root) is cryptographically linked to its parent node.
- **Ciphertext-Policy Attribute-Based Encryption (CP-ABE)** An Attribute-Based Encryption scheme wherein ciphertexts contain an access policy, and secret keys contain attributes.
- **Context** In a share tree, a group of shares having to do with related records.
- **Context Root** In a share tree, a block that is the root of a subtree which forms a context.
- **Data Owner (DO)** An entity which produces a set of data.
- **Data Reader (DR)** An entity which is granted read rights on a set of data.
- **Interplanetary File System (IPFS)** A distributed file system.
- **Key-Policy Attribute-Based Encryption (KP-ABE)** An Attribute-Based Encryption scheme wherein secret keys contain an access policy, and ciphertexts contain attributes.

- **Master Ledger** In our system, the primary blockchain that permanently and immutably stores user data.
- **Origin Block** The first block in a blockchain, created when the blockchain is initialized.
- **Payload** A special secret attribute on a record that contains the main, confidential body of the record. For example, the bytes of a file would go in the payload attribute.
- **Public Attribute** A key-value pair on a block or record that is readable by any user.
- **Record** In the master ledger, a block containing user data and its associated metadata. A record typically corresponds to a document, report, post, media file, or other resource that can be shared with others as a whole.
- **Secret Attribute** A key-value pair on a block or record that is encrypted before publishing, and readable only by the author or authorized users.
- **Share** In the share tree, a block representing a granted permission on a record, and containing information on how to access that record.
- **Share Tree** In our system, a blockchain-like data structure, in the shape of a tree, that stores access permissions and allows records in the master ledger to be shared with other users.
- **Stored Data Object (SDO)** A document published to a data store. In our context, an SDO is encrypted via an ABE scheme, published and stored on the blockchain.
- **Trusted Third Party (TTP)** An actor entrusted with information or data belonging to a user.

Bibliography

[1] G. Bramm, M. Gall, and J. Schütte, "Bdabe - blockchain-based distributed attribute based encryption," in *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications - Volume 1: SECRYPT*, Jan. 2018, pp. 99–110. DOI: 10.5220/0006852600990110.

[2] X. Yue, H. Wang, D. Jin, M. Li, and W. Jiang, "Healthcare data gateways: Found healthcare intelligence on blockchain with novel privacy risk control," *Journal of medical systems*, vol. 40, no. 10, p. 218, 2016.

[3] H. Son. (2019). Jp morgan is rolling out the first us bank-backed cryptocurrency to transform payments business, [Online]. Available: https://www.cnbc.com/2019/02/13/jp-morgan-is-rolling-out-the-first-us-bank-backed-cryptocurrency-to-transform-payments--. Accessed: 22.04.2019.

[4] M. Sookhak, F. Richard Yu, K. Khan, Y. Xiang, and R. Buyya, "Attribute-based data access control in cloud computing: Taxonomy and open issues," *Future Generation Computer Systems*, vol. 72, Aug. 2016. DOI: 10.1016/j.future.2016.08.018.

[5] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06, Alexandria, Virginia, USA: ACM, 2006, pp. 89–98, ISBN: 1-59593-518-5. DOI: 10.1145/1180405.1180418. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180418.

[6] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 321–334. DOI: 10.1109/SP.2007.11.

[7] T. Kamp, "Combining abcs with abe: Privacy-friendly key generation for smart card based attribute-based encryption," Master's thesis, University of Twente, 2014.

[8] A. Lewko and B. Waters, "Decentralizing attribute-based encryption," in *Annual international conference on the theory and applications of cryptographic techniques*, Springer, 2011, pp. 568–588.

[9] J. K. Liu, T. H. Yuen, P. Zhang, and K. Liang, "Time-based direct revocable ciphertext-policy attribute-based encryption with short revocation list," in *Applied Cryptography and Network Security*, B. Preneel and F. Vercauteren, Eds., Cham: Springer International Publishing, 2018, pp. 516–534, ISBN: 978-3-319-93387-0.

[10] S. Ma, J. Lai, R. H. Deng, and X. Ding, "Adaptable key-policy attribute-based encryption with time interval," *Soft Computing*, vol. 21, no. 20, pp. 6191–6200, Oct. 2017, ISSN: 1433-7479. DOI: 10.1007/s00500-016-2177-z. [Online]. Available: https://doi.org/10.1007/s00500-016-2177-z.

[11] I. Youcef, A. Lounis, and A. Bouabdallah, "Revocable attribute-based access control in multi-authority systems," *Journal of Network and Computer Applications*, vol. 122, Aug. 2018. DOI: 10.1016/j.jnca.2018.08.008.

[12] Y. Zhao, M. Ren, S. Jiang, G. Zhu, and H. Xiong, "An efficient and revocable storage cp-abe scheme in the cloud computing," *Computing*, Jun. 2018, ISSN: 1436-5057. DOI: 10.1007/s00607-018-0637-2. [Online]. Available: https://doi.org/10.1007/s00607-018-0637-2.

[13] R. H. Deng and H. Cui, "Revocable and Decentralized Attribute-Based Encryption," *The Computer Journal*, vol. 59, no. 8, pp. 1220–1235, Aug. 2016, ISSN: 0010-4620. DOI: 10.1093/comjnl/bxw007. eprint: http://oup.prod.sis.lan/comjnl/article-pdf/59/8/1220/8039693/bxw007.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/bxw007.

[14] L. Zhou, L. Wang, and Y. Sun, "Mistore: A blockchain-based medical insurance storage system.," *J. Medical Systems*, vol. 42, no. 8, 149:1–149:17, 2018. [Online]. Available: http://dblp.uni-trier.de/db/journals/jms/jms42.html#ZhouWS18.

[15] S. Wang, Y. Zhang, and Y. Zhang, "A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems," *IEEE Access*, vol. 6, pp. 38 437–38 450, 2018, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2851611.

[16] H. Wang and Y. Song, "Secure cloud-based ehr system using attribute-based cryptosystem and blockchain," *Journal of medical systems*, vol. 42, no. 8, p. 152, 2018.

[17] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf.

[18] A. Back, "Hashcash - a denial of service counter-measure," Tech. Rep., 2002.

[19] I. Corporation. (2013). Integrated cryptographic and compression accelerators on intel architecture platforms, [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/integrated-cryptographic-compression-accelerators-brief.pdf. Accessed: 20.04.2019.

[20] S. Gueron, *Intel® Advanced Encryption Standard (AES) New Instructions Set*. May 2010. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[21] W. Stallings and L. Brown, *Computer Security: Principles and Practice*, 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014, ISBN: 9780133773927.

[22] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979, ISSN: 0001-0782. DOI: 10.1145/359168.359176. [Online]. Available: http://doi.acm.org/10.1145/359168.359176.

[23] G. R. Blakley, "Safeguarding cryptographic keys," in *Managing Requirements Knowledge, International Workshop on*, Los Alamitos, CA, USA: IEEE Computer Society, Jun. 1979, p. 313. DOI: 10.1109/AFIPS.1979.98. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/AFIPS.1979.98.

[24] M. Mignotte, "How to Share a Secret," in *Cryptography*, T. Beth, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 371–375, ISBN: 978-3-540-39466-2.

[25] L. M. Axon and M. Goldsmith, "PB-PKI: a privacy-aware blockchain-based PKI," vol. 6, SCITEPRESS, 2016.

[26] A. Yakubov, W. M. Shbair, A. Wallbom, D. Sanda, and R. State, "A blockchain-based pki management framework," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2018, pp. 1–6. DOI: 10.1109/NOMS.2018.8406325.

[27] M. Al-Bassam, "Scpki: A smart contract-based pki and identity system," in *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, ser. BCC '17, Abu Dhabi, United Arab Emirates: ACM, 2017, pp. 35–40, ISBN: 978-1-4503-4974-1. DOI: 10.1145/3055518.3055530. [Online]. Available: http://doi.acm.org/10.1145/3055518.3055530.

[28] N. Alexopoulos, J. Daubert, M. Mühlhäuser, and S. M. Habib, "Beyond the hype: On using blockchains in trust management for authentication," in *2017 IEEE Trustcom/BigDataSE/ICESS*, Aug. 2017, pp. 546–553. DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.283.