SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Design, Implementation, and Assessment of an Improved Revocation Mechanism for Verifiable Credentials

Srajit Sakhuja

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Design, Implementation, and Assessment of an Improved Revocation Mechanism for Verifiable Credentials

Design, Implementation und Evaluation eines verbesserten Sperrmechanismus für Varifiable Credentials

Author:	Srajit Sakhuja
Supervisor:	Prof. Dr. rer. nat. Florian Matthes
Advisor:	Felix Hoops, M.Sc
Submission Date:	15.12.2023

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.12.2023 Srajit Sakhuja

Acknowledgments

I wish to begin by thanking my thesis advisor, Mr. Felix Hoops, for the countless hours that he spent brainstorming with me and for always being available - often at very short notice - to answer my queries and to share his inputs. I would also like to extend my gratitude towards Prof. Dr. Florian Matthes and Mr. Felix Hoops for placing their trust in me and allowing me to write my thesis on this new and exciting topic of Verifiable Credentials. I thoroughly enjoyed my conversations with Prashant Singh and Evan Christopher who have also been working towards their respective theses on closely related subjects. My friends Akash, Srivatsa, Lalit, Andrew, and Olga would be glad to see me submitting my thesis and having no more excuses to decline their invitations for social activities. Lastly, I would like to thank my parents, my sister, the two grandparents who I lost this year, and the rest of my family for their unwavering support throughout my academic life and their relentless commitment towards my education.

Abstract

Verifiable Credentials (VCs) are a proven way for real world entities to stake claims about themselves, issuing parties to be assured that this information is tamper-proof, and verifying parties to be sure that those claims are not fabricated. Along with the Verifiable Credentials standard, the World Wide Web Consortium (W3C) has published two other closely related standards on Decentralised Identifiers and Status List 2021. Together these three standards are the cornerstone for implementing self-sovereign identity (SSI) in decentralised cloud ecosystems such as the European Gaia-X framework.

The subjects of VCs may be citizens of countries, students at universities, and employees of companies. These subjects may become ineligible for possessing the VCs issued to them, at any given point, for a myriad of reasons. When such situations arise these VCs need to be revoked and/or temporarily suspended. It is easy to see why the revocation and suspension of Verifiable Credentials is just as important as issuing them. The Status List 2021 standard is the present-day approach for revoking and suspending VCs. GX Credentials is an open-source project which is a partial implementation of the Verifiable Credentials standard and enables the issuance of VCs to companies and their employees.

In the first part of this work, we set forth the design goals for a revocation setup, explore the shortcomings of the status quo, and describe what is missing in a system like GX Credentials to enable the revocation and suspension of VCs. This analysis reveals that the problem statement can be broken down into two sub-problems: (1) Designing a data structure that improves on the shortcomings of Status List 2021 and (2) Proposing architectural changes to GX Credentials to build an end-to-end setup for revoking and suspending VCs at scale.

In the second part, we design, and implement solutions for these two sub-problems. For the revocation data structure, we draw inspiration from TLS and the solutions that have been proposed to solve TLS certificate revocation. The most promising among these solutions is CRLite which we use as a basis to design a revocation data structure for Verifiable Credentials. We conduct experiments with different variants of this data structure which reveals that using an Approximate Membership Query data structure called XOR Filters gives the most promising results and leads to 7.5x lesser space consumption than the status quo. For the architectural changes to GX Credentials, we propose the addition of two components called the Company Management Service (CMS) and the Membership Checking Service (MCS). This achieves a better separation of concerns between the three components - the existing GX Credentials application run by the Trust Anchor, the CMS managed by companies, and the MCS used by verifiers - and redistributes the data across these components on a need-to-know basis thereby achieveing better data isolation and privacy guarantees.

Finally, we conclude the thesis by evaluating the proposed solutions on the design goals that we delineated in the first part.

Contents

1 Introduction

There are several occasions in our everyday lives where we need to prove things about ourselves. Consider the simple case of collecting an e-commerce delivery from a delivery station. We need to present an identity document that proves that we are the same individual who placed the order or a person authorized by them to collect the order on their behalf. To prove our identity or details about our identity such as our age, nationality, or place of residence, we rely on documents. In countries such as Estonia [1] and India [2], it is becoming increasingly common for citizens to possess and present digital identity cards rather than physical ones. Whether we use physical documents or digital ones, a drawback of the existing systems is that the databases powering them are largely centralized. A student submitting a university transcript for PhD applications requires that the university's centralised server can confirm that this document was issued by them and that it is currently not in a revoked/suspended state. This centralisation of identity gives enormous power as well as responsibility to these centralised Identity Providers (IdPs). IdPs have precise information about the Relying Parties (RPs) that the holders of these documents are presenting them to. Additionally, the IdPs bear the responsibility of securing these large repositories of user data that may become attractive targets for hackers and digital miscreants.

The concept of Self-Sovereign Identity (SSI) stems from the central premise that individuals must be the owners of their identity and control who their private data is shared with and to what extent. Multinational policy frameworks such as the European Gaia-X rely on Verfiable Credentials and their allied technologies to set up an SSI capabilities in a decentralised, user-centric, and secure cloud ecosystem [3].

It is easy to see why revoking/suspending VCs is just as critical a use case as issuing them is. A known offender must not be able to continue possessing and presenting a VC after its issuer decides to revoke/suspend it. The framework around Verifiable Credentials is quite mature and benefits from the existence of other standards and technologies such as Decentralised Identifiers (DIDs) [4] and Zero-knowledge Proofs (ZKPs) [5]. However, the present-day solution proposed for revocation/suspension, called Status List 2021, is simplistic, has several shortcomings, and limits its scope by not recommending an end-to-end mechanism for achieving revocation/suspension. We breakdown this thesis into a series of research questions that we progressively answer in order to improve on the status quo for Verifiable Credential revocations.

In the process of designing this revocation mechanism we take inspiration from solutions that have been devised for TLS certificate revocation and build upon the work done by our colleagues at the Software Engineering for Business Information Systems (SEBIS) Chair on a VC issuance system called GX Credentials.

1.1 Research Questions

The work done in this thesis largely revolves around answering the following four research questions:

RQ1: What are the requirements for revocation/suspension mechanisms for VCs?
Keeping the design of the Verifiable Credentials standard in mind, we wish to identify the design goals that are relevant for a revocation/suspension mechanism.

RQ2: What is the state of the art for data structures that support membership/exclusion operations?
Revocation and suspension are not unique to Verifiable Credentials. The same challenge, for instance, exists in the context of TLS certificates - they also needed to be revoked from time to time. We wish to discover data structures besides Status List 2021 that implement membership/exclusion operations.

RQ3: Designing the Revocation/Suspension data structure
The credentials that are revoked will be stored in a data structure similar to Status List 2021. We wish use our findings from **RQ2** to design different data structures that achieves the design goals we delineate while addressing **RQ1** and benchmark their performance on these design goals.

RQ4: Designing the end-to-end revocation infrastructure
The data structure can not operate in isolation. It must be plugged into a broader VC issuance infrastructure. Here we wish to build on an existing partial implementation of the Verifiable Credentials standard called GX Credentials, proposing modifications and extensions to the implementation to achieve our stated design goals.

2 Background

This thesis is built on the foundation of some key concepts that we introduce in this section. Self-Sovereign Identity (SSI), introduced in section 2.1 is the framework that argues in favour of the user having complete control of their identity and its presentation. Verifiable Credentials (VC), discussed in section 2.2 is the most popular implementation for SSI and relies on various subsystems such as Decentralized Identifiers (DID) and Status List 2021, which we introduce in section 2.2.2 and section 2.2.5 respectively. Finally, in section 2.4, we give an overview of the Interplanetary File System (IPFS) and the Interplanetary Name System (IPNS) which are critical building blocks for the solution and design contributions we thoroughly discuss in chapter 5.

2.1 Self-Sovereign Identity (SSI)

It is critical for users of digital systems to be able to prove who they are and stake claims about themselves. Systems that manage these digital identities are called Identity and Access Management (IAM) systems.

2.1.1 Evolution of IAM Systems and the Need for Sovereignty

In the early days of the internet it was common for every service online to implement its own IAM. The service would be responsible for the issuance of the credentials as well as for verifying their veracity making the IAM service-centric. The details of the credentials were abstracted away from end users who were only responsible for remembering their usernames and passwords. Two internet phenomena soon made this system nearly obsolete - (1) As digital adoption increased, there was a surge in the number of different services that users began to use somewhat rarely (once a week or once a month), (2) A handful of services e.g., email clients like Gmail, Outlook, etc. and social media platforms such as Facebook, LinkedIn, etc. became synonymous with the internet and users began using them very frequently (multiple times everyday).

A natural successor to the service-centric IAM model was to outsource the IAM logic to the services users use very frequently - Facebook, Google, LinkedIn grew to become the most commonly used Identity Providers (IdP) or Issuers. Services online also known as Relying Parties (RP) registered themselves with one or more IdPs and interacted with them through implementations of standards such as OAuth 2.0, SAML, and OpenID Connect (OIDC) [6].

The users in this paradigm only had to remember a few usernames and passwords and RPs could avoid reinventing the (IAM) wheel. The IdP-centric model is fraught with problems - (1) IdPs tend to become silos for highly sensitive and private information. This makes them

prone to data breaches - both accidental and deliberate, (2) Users have no control over what portion of their data the IdP shares with an RP, (3) The IdP-centric model also has self-evident privacy concerns because the IdP has an auditable record of every online service the user uses, (4) IdPs are not immune to the most commonly observed problem with centralized systems - they become single points of failure and any unavailability of the IdP blocks the end users from using services of an RP, (5) Users have no control over their identity and can abruptly lose access to multiple services if an IdP decides to disable their account - this is especially relevant in case of IdPs that also function as social media platforms, such as Facebook, LinkedIn, and Twitter, which have a culture of banning users who violate the platform's social media norms.

The oligopolistic nature of the IdP-centric model underlines the need to give the user more sovereignty over their digital identities which is the core idea behind Self-Sovereign Identity (SSI) [7].

2.1.2 SSI Design Principles

Defining SSI as a paradigm that keeps the user central to the administration of identity, *C. Allen* published the 10 principles of SSI [8] [9]. The three principles most relevant to this discussion are as follows:

1. Control - users must be able to control their identities without relying on a third-party such as an IdP. This principle eliminates single points of failure, privacy risks, and the existence of the sensitive-information silos which in turn reduces the risk of accidental or deliberate leaking of private data.

2. Persistence - user identities must be long-lived. This principle reduces platform arbitration over which users can be banned or abruptly denied an identity.

3. Consent and Minimization - the user must be able to share the minimum subset of its claims with an RP. Cryptographic techniques such as Zero Knowledge Proofs for Set Membership [10] and Zero Knowledge Range Proofs [5] allow users to prove to the RP that a claim is true without actually revealing the contents of their credentials. A popular use case here is for a user to prove to an RP that they are above the age of consent without actually disclosing their date of birth (and certainly not other details such as their address which may be enclosed in their credential).

2.2 Verifiable Credentials (VC)

Real world entities such as people and organizations need to make claims about themselves almost everyday. University students need to present their student IDs while writing exams, cross-border commuters need to present their passports at border checkpoints, and drivers around the world need to present their driving licenses whenever they make traffic violations. All these documents broadly contain two categories of information - (1) claims about the

real world entity possessing them - such as people's names, their pictures, the titles of their bachelor's theses, medical records, etc., and (2) signatures such as holograms, bar codes, watermarks to assert the document's authenticity. Despite the sophisticated design of these documents and improvements in forgery detection methods, document counterfeiting is still widespread. Verifiable credentials aim to provide a foolproof way for real world entities to stake claims about themselves, issuing parties to be assured that this information is tamper-proof, and verifying parties to be sure that those claims are not falsehoods.

2.2.1 Roles in the VC Ecosystem

The Verifiable Credentials W3C standard [11] specifies the following actors/roles as being relevant to the VC ecosystem. These roles and their synonyms are frequently used in the rest of the thesis especially in sequence diagrams that describe the interactions between these actors and to give real-world examples.

1. Holder: The actor who possesses the VC and generates Verifiable Presentations (VP) from them. Common examples include students, employees, citizens, etc.

2. Subject: The actor about whom the claims are made. In many cases the subject would be the same as the holder, but in several cases the holder may be different from the subject. For example, parents being holders of VCs that pertain to their children (the Subjects), or humans being holders of VCs containing claims about their pets (the Subjects).

3. Issuer: The actor who performs the due diligence about the relevant claims of the subject. Common examples include universities, city halls, governments, corporations, etc.

4. Verifier: The actor who wishes to check if a certain claim about a subject is true or not. Common examples include security personnel, passport control officers, drug store operators, etc.

2.2.2 Decentralized Identifiers (DID)

Decentralized Identifiers (DIDs) are an important prerequisite to understanding how VCs work. VCs are issued by issuing parties and DIDs can be seen as a way to identify those parties and to obtain information about them. A DID is a Uniform Resource Identifier (URI) and the resource being identified by a DID is its DID Document. The process of a DID resolving to its DID Document is powered by a data infrastructure known as Verifiable Data Registries (VDR) [11]. Examples of VDRs include distributed ledgers, peer-to-peer networks, decentralized file systems, etc. DIDs conform to the syntax [4] described in figure 2.1.

The following are some examples of DIDs

```
did:btcr:xz35-jzv2-qqs2-9wjt
did:ethr:0xE6Fe788d8ca214A080b0f6ac7F48480b2AEfa9a6
```

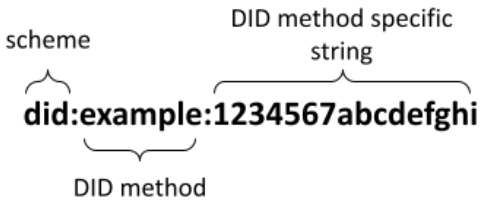


Figure 2.1: Syntax of Decentralized Identifiers (DID)

There are tens of DID methods and each has its own specification. For instance, the two DIDs listed above belong to the btcr (Bitcoin) and the ethr (Ethereum) method names respectively [12]. The specification of a given DID method specifies the syntax of its DID method specific string and the location for obtaining its DID Document. For instance, the two DIDs listed above will lead a DID resolver to a DID document stored on the Bitcoin and Ethereum blockchains respectively.

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/ed25519-2020/v1"
  ]
  "id": "did:example:123456789abcdefghi",
  "authentication": [{
    "id": "did:example:123456789abcdefghi#keys-1",
    "type": "Ed25519VerificationKey2020",
    "controller": "did:example:123456789abcdefghi",
    "publicKeyMultibase": "zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
  }]
}
```

Listing 2.1: Example DID Document

2.2.3 Structure of a VC

In this section, we give an example of a basic Verifiable Credential and briefly describe its structure. A Verifiable Credential is a set of key-value pairs where the keys are also called fields.

The `id` field contains the identifier for the VC. The `type` field is useful for a verifier to determine if the provided VC or VP can be used for the intended use case or not. The `issuer`, `validFrom`, `validUntil` fields have self-explanatory semantics. The functional purpose of VCs to assert claims about a given subject. The `credentialSubject` field contains claims about one or more subjects. In the following example, the value for the `credentialSubject` field

indicates that a subject with id `did:example:ebfeb1f712ebc6f1c276e12ec21` has a degree of type `ExampleBachelorDegree`. The `proof` field contains the cryptographic signature created by the issuer of the VC in the nested `proofValue` field. It also contains the metadata to verify the signature e.g., the `verificationMethod` field which specifies the public key for the signature verification and the `type` which specifies the cryptographic suite of the proof e.g., the example below specifies the type as the EdDSA Cryptosuite v2020.

```
{
  "@context": [
    "https://www.w3.org/ns/credentials/v2",
    "https://www.w3.org/ns/credentials/examples/v2",
    "https://w3id.org/security/suites/ed25519-2020/v1"
  ],
  "id": "http://university.example/credentials/3732",
  "type": [
    "VerifiableCredential",
    "ExampleDegreeCredential"
  ],
  "issuer": "https://university.example/issuers/14",
  "validFrom": "2010-01-01T19:23:24Z",
  "validUntil": "2020-01-01T19:23:24Z",
  "credentialSubject": {
    "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
    "degree": {
      "type": "ExampleBachelorDegree",
      "name": "Bachelor of Science and Arts"
    }
  },
  "proof": {
    "type": "Ed25519Signature2020",
    "created": "2023-11-26T23:04:13Z",
    "verificationMethod": "https://university.example/issuers/14#key-1",
    "proofPurpose": "assertionMethod",
    "proofValue": "z3y5D7UNFmKwv...TELYGtizDVnfMMtWDBERd3fPVeJnsLUvoShat1ghXj1E"
  }
}
```

Listing 2.2: Example Verifiable Credential

2.2.4 Verifiable Presentations

The concept of Verifiable Presentations (VP) becomes relevant at verification time. As the name suggests, a VP is a presentation made to a verifier the contents of which are encoded

in a way that the authenticity of the data can be consider trustworthy after cryptographic verification. A VP is derived from one or more VCs and certain types of VPs may not contain the original VC but some data that is synthesized from VCs e.g., in cases of zero-knowledge proof-based VPs. A VP is expected to be very short-lived and is tied to a specific challenge given by a verifier.

The following is an example of a VP which contains an array of VCs stored in the `verifiableCredential` field.

```
{
  "@context": [
    "https://www.w3.org/ns/credentials/v2",
    "https://www.w3.org/ns/credentials/examples/v2"
  ],
  "id": "urn:uuid:3978344f-8596-4c3a-a978-8fcaba3903c5",
  "type": ["VerifiablePresentation", "ExamplePresentation"],
  "verifiableCredential": [{ }],
  "proof": [{ }]
}
```

Listing 2.3: Example Verifiable Presentation

2.2.5 Revoking a VC: Status List 2021

In most cities around the world a passenger vehicle driver needs to commit an average of five traffic violations before getting their driver's license banned for up to a year. After this moratorium period the driver can pass a driving test and get their driver's license reinstated. This problem of revoking a claim document and subsequently reverting a revocation is not unique to drivers' licenses - it is just as applicable to other claim documents such as university degrees, visas, club membership IDs, etc. A VC, once issued, is self-sufficient to prove its authenticity. So we need additional metadata in the VC and infrastructure to achieve these design goals of revoking and temporarily suspending VCs. This challenge is addressed by Verifiable Credentials using the Status List 2021 standard [13].

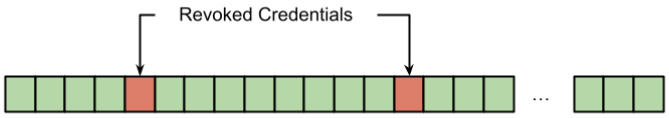


Figure 2.2: Status List 2021

The Status List managed by a given issuer can be simply viewed as a bitstring where each index corresponds to a VC issued by the issuer. A binary value of 1 indicates that the VC is revoked/suspended and a binary value of 0 indicates that it is not. In cases where the majority of the VCs are not revoked/suspended e.g., in the case of university degrees awarded by a university, the bitstrings would mostly be composed of 0s and this allows them to be highly compressible. The minimum size of a Status List is 16KB [13]. This allows it to accommodate entries for 131,072 VCs.

Figure 2.3: Extending the Verifiable Credential Standard to incorporate Status List 2021

2.3 Gaia-X: A Federated Secure Data Infrastructure

Gaia-X is a Europe-wide initiative to create a federated and secure data infrastructure to enable Europe achieve digital sovereignty. The initiative aims to give control back to the user by giving them sovereignty over their data. In this sense, it is self-evident to see the synergies between the goals of Gaia-X and SSI. In terms of tangible outcomes, the Gaia-X working group has authored several specifications which are then translated into code by the Gaia-X community. Additionally, the initiative also aims to develop the framework and infrastructure to audit systems and award them labels on the basis of their compliance to Gaia-X regulations and standards.

2.4 Interplanetary File System (IPFS)

The Interplanetary File System [14] or IPFS is a peer-to-peer file system. The three salient features of IPFS are that (1) it is distributed and therefore, has no single point of failure, (2) has a high throughput, (3) has a content-addressed and self-certifying namespace. The IPFS architecture takes inspiration from the architectures of classical peer-to-peer content sharing

systems like Distributed Hash Tables (DHTs), Git, BitTorrent, and Self-Certified Filesystems (SFS).

2.4.1 IPFS Design Principles and Architecture

The following points summarize some design features of IPFS:

1. Similar to BitTorrent, IPFS has a quasi-tit-for-tat data transfer strategy that rewards nodes that are seeders (data providers) and punishes the leechers (data downloaders).

2. It tracks the availability of file shards and prioritizes those shards that are rare. This removes bottle-necks from the data sharing network and makes these rare shards more freely available across the network.

3. For future-proofing the system, it uses self-describing values for cryptographic hash functions. This is done by expressing them in a multi-hash format which includes a header specifying the hash function that is used and the digest length in bytes.

4. It breaks files into chunks and each of these chunks is assigned a unique identifier which is called its Content Identifier (CID). The CID for a chunk is generated using its cryptographic hash. This has several positive knock-on effects: (1) the data is content addressable i.e., it is fetched on the basis of its content and not its location, (2) the data is tamper-resistant - if it is tampered with or corrupted, it would have a different hash, (3) deduplication - any two chunks that have identical values will be addressed with the same hash. IPFS organizes these chunks in the form of Merkel Directed Acyclic Graphs.

5. Kademlia, the DHT used in IPFS, helps a node on the network find a peer that stores that it is looking for. It can be visualized as a mapping between CIDs and IP addresses of the nodes storing them. All the data stored in IPFS is stored in the initiating node's local storage. This is followed by the process of pinning in which the node advertises that it has a given CID which it can provide to the network. This allows other nodes on the network to retrieve the CID from this node.

6. Once data is pinned on IPFS it can only be deleted locally. If a CID has been replicated to other nodes on the IPFS network, it will (potentially) persist on IPFS forever.

2.4.2 Achieving Mutability using Interplanetary Name System (IPNS)

Immutability is built into IPFS by design. Changing the contents of a file will consequently change its hash and therefore the CID which is used as its address. But there are several use cases where mutability is desirable to be able to update the contents of an object without changing its pointer. IPNS [15] is a simple extension of the IPFS framework that allows us to achieve mutability.

An IPNS record is associated with a public-private key pair. It contains the IPNS name which is generated by hashing the public key and is of the form `/ipns/<cid-from-public-key>`.

In addition to the IPNS name, the record contains the immutable IPFS path that it points to and a signature. This allows IPNS to be a mutable pointer to an immutable IPFS CID as shown in figure 2.4.

Figure 2.4: Achieving Mutability using Interplanetary Name System (IPNS)

IPNS names are self-certifying. The IPNS record contains the public key which maps to the IPNS name and a signature that verifies that the IPNS record was signed by the owner of the public-private key pair. This makes it easy to verify that the IPNS record was not tampered with by a malicious actor online.

2.5 The Tezos Blockchain

Tezos is an open-source blockchain that supports peer-to-peer transactions and smart contracts. Tezos went live in 2018. Its creators have their roots in the finance industry and it operates using a native cryptocurrency called Tez. The Tezos blockchain achieves consensus using the Liquid Proof-of-Stake (LPos) algorithm. Smart contracts on Tezos are written in a domain-specific language called Michelson. Michelson is a Turing-complete stack-based language and offers data structures like lists, sets, maps as well as cryptographic primitives that are extensively used in distributed ledger technologies. Other programming languages such as SmartPy, Ligo, and Lorentz can also be used to write these smart contracts, but they all eventually compile down to Michelson [16].

A blockchain's governance typically involves two categories of decisions - (1) the protocols and frameworks which materialize through the code, (2) the blockchain's incentive mechanisms or its economics. Other blockchains like Ethereum have a more centralized approach to governance with decisions about the blockchain's protocol updates and future roadmap being decided by a select set of developers and stakeholders. Tezos operates with a more decentralized approach through a process called on-chain governance. In the true style of an open-source project, the developer community is involved in system updates for the Tezos blockchain.

Before Ethereum switched to Proof-of-Stake (PoS) [17] in 2022, Tezos was one of the mainstream blockchain networks using PoS-based consensus. The main advantage PoS has over the traditional Proof-of-Work (PoW) algorithm is that it is more energy-efficient and less compute-intensive. PoS chooses validators who validate and help achieve consensus on a block being added to the blockchain. Validators are chosen on the basis of their stake in the blockchain's native cryptocurrency. The philosophy here is that the cost of malicious intent for the validator to make an intended malicious error exceeds the block reward. Delegated Proof of Stake (DPoS) is a refined version of PoS in which the participants on the blockchain vote to elect delegates who act as validators on their behalf. Delegation is a mechanism by which multiple stakeholders pool their stake and use this staking poll in the protocol's

validator selection process. So when a delegate is awarded the next block for validation, the block is indirectly awarded to all the stakeholders in the staking poll and the block reward is distributed among all the delegators in proportion to their stake in the staking poll. Delegators are required to lock their tokens with a staking service in order to participate.

Tezos uses a slightly modified version of DPoS called Liquid Proof-of-Stake (LPoS). It is convention to refer to the delegate or validator as a 'baker' in LPoS. To bake blocks, a baker needs a minimum of 6,000 Tez. Delegators pick bakers; however, unlike in DPos, the delegators' stake is not locked in place, rather they can quickly align themselves with a baker that has similar voting preferences as theirs.

3 Related Work

3.1 Approximate Membership Query (AMQ) Data Structures

Using a set data structure to answer membership queries is a commonly encountered task in Computer Science. In the absence of space constraints, we can simply store all the elements in the form of a dictionary and easily answer these queries. But in contexts where space is a constraint or the total size of the elements is simply unknown, Approximate Membership Query (AMQ) Data Structures find many applications. The oldest and the most popular data structure in this context are Bloom Filters, which we discuss at length in the next subsection.

3.1.1 Bloom Filters

First introduced in Burton H. Bloom's seminal paper [18] and named after the author, Bloom Filters are the most popular Approximate Membership Query Data Structure even after fifty years since their invention in 1970. In the context of Bloom Filters, the 'approximation' is mainly due to false positives. For a look-up query, a false positive takes place when the Bloom Filter answers true for an element that is actually not stored in it. The design of Bloom Filters ensures that the opposite error of false negatives can never occur.

Bloom Filters primarily support two functions - insert and look-up. An empty Bloom Filter is a vector v of m bits with all its bits set to 0 along with a set of k independent hash functions $h_1, h_2, ..., h_k$ with range $1, ..., m$.

In the insertion operation, for a given value $a \in A$, all bits in positions $h_1(a), h_2(a), ..., h_k(a)$ are set to 1. This is illustrated in figure 3.1.

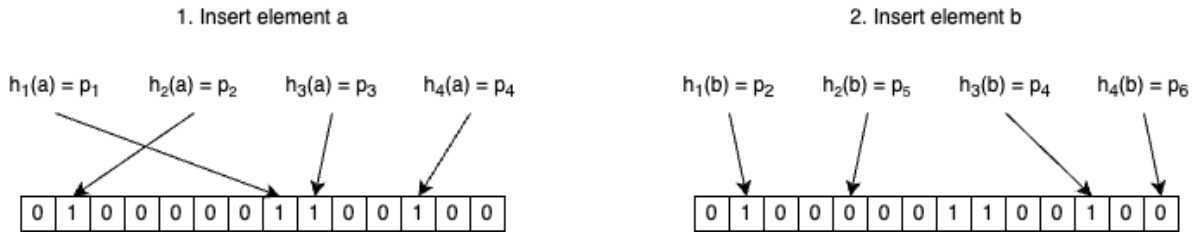


Figure 3.1: Inserting into a Bloom Filter with 4 hash functions.

In the look-up operation, for a given value b, we check for the bits in positions $h_1(b), h_2(b), ..., h_k(b)$. As indicated in figure 3.2, if any of these bits is set to zero, b is not contained in the Bloom

Filter. However, all of these bits being set to zero may not definitely indicate that the given element is contained in the filter. This is illustrated in figure 3.2.

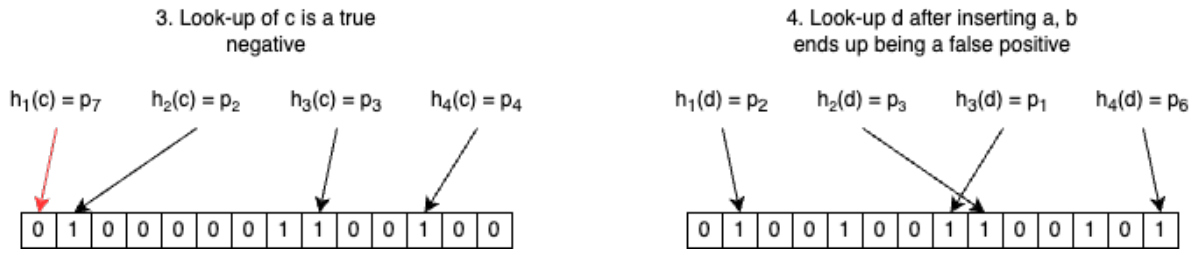


Figure 3.2: Lookup from a Bloom Filter with 4 hash functions.

The false positive rate is the expense that Bloom Filters pay for being a more space optimised set membership data structure than traditional dictionaries. The number of hash functions k, the size of the Bloom Filter m, and the number of keys n being inserted into it determine what its false positive rate would be. Table 3.1 summarises the false positive rates for different combinations of m/n and k [19].

m/n	***k=1***	***k=2***	***k=3***	***k=4***	***k=5***	***k=6***	***k=7***	***k=8***
2	0.393	0.400						
3	0.283	0.237	0.253					
4	0.221	0.155	0.147	0.160				
5	0.181	0.109	0.092	0.092	0.101			
6	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846

Table 3.1: Bloom Filter False positive rates for different combinations of m/n and k.

Bloom filters are ubiquitous on the internet. They are a great fit for use cases that require 0% false negative rates but can tolerate a small amount of false positives. Routing-table lookup, online traffic measurement, peer-to-peer systems, cooperative caching, firewall design, intrusion detection, bioinformatics, stream computing, and distributed storage systems [20] [21] all use Bloom Filters extensively.

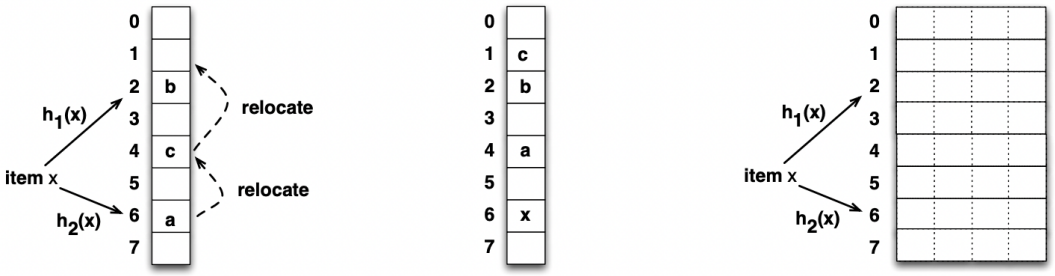
As a more concrete example, imagine a scenario where we are running a blogging website like Medium or Quora and wish to recommend articles to users. Here it would be desirable to not recommend articles to a user that they have already read. At the same time, it would be a huge overhead to store a set of articles that every user has read and then check if a given

article exists in this set before recommending it to the user. This is the perfect use case for Bloom filters because if an article does not exist in this set, it has definitely not been shown to the user and doesn't need to be filtered out and on the other hand, if an article exists in the set either the user has seen the article already (true positive) or the user hasn't (false positive) and in both cases recommending them a different article is a suitable outcome.

3.1.2 Cuckoo Filters

Proposed by *Fan et al.* in their 2014 paper *Cuckoo Filter: Practically Better Than Bloom* [22], Cuckoo Filters are introduced as an improved AMQ Data Structure and as an alternative to the already very popular Bloom Filter. Cuckoo Filters achieve good space efficiency with a small percentage of false positives. For use cases where the given system needs to store many items with moderately low false positive rates, Cuckoo Filters are more compact than Bloom Filters. Another distinguishing feature is that Cuckoo Filters support delete operations unlike Bloom Filters.

Cuckoo Filters belong to a category of filters that use *Fingerprints*. Fingerprints are bit strings that are obtained by applying a hash function on the original key. Fingerprint clashes lead to false positives in Cuckoo Filters similar to Bloom Filters. In this sense, the fingerprint size determines the false positive rates in Cuckoo Filters. For small false positive rate targets, we need longer fingerprints. Longer fingerprints increase the size of the Cuckoo Filter storing them. So the false positive rate, generally has a direct proportionality with the size of the filter.



(a) before inserting item x (b) after item x inserted (c) A cuckoo filter, two hash per item and functions and four entries per bucket

Figure 3.3: Cuckoo Hashing in practice. Cuckoo Filters are based on the concept of Cuckoo Hashing.

Cuckoo Filters trace their roots to the concept of Cuckoo Hashing. A basic cuckoo hash table is an array of buckets. Each item can be placed in one out of two buckets based on the

values of the hash functions h_1 and h_2. Figure 3.3(a) shows that item x hashes to positions 2 and 6 in the Cuckoo hash table. If either of these buckets was empty, the item would simply be placed in that bucket. If neither of the buckets is empty, as is the case here, an element-eviction process takes place. This evicted element is then relocated to its alternate location. This can lead to a cascade of evictions and relocations which eventually terminates after an item is placed into a bucket without relocating another time. In this example, the insertion of item x triggers the eviction of item a which in turn triggers the eviction of item c until all the items $a, b, andc$ have their own dedicated buckets in the hash table. There can always be a scenario in which this series of evictions and relocations exceeds the maximum number of permitted displacements. In this case, the hash table is considered too full to make an insertion. In most practical implementations, Cuckoo Hashing uses buckets that can store multiple elements with each bucket having a size b.

Algorithm 1 Cuckoo Filter: Insert(x)

$f = fingerprint(x)$;
$i_1 = hash(x)$;
$i_1 = i_1 \oplus hash(f)$;
if $bucket[i_1]$ or $bucket[i_2]$ has an empty entry **then**
    add f to that bucket;
    return done;
end if
i = randomly pick i_1 or i_2;
for n = 0; n < MaxNumKicks; n++ **do**
    randomly select an entry e from $bucket[i]$;
    swap f and the fingerprint stored in entry e;
    $i = i \oplus hash(f)$;
    if $bucket[i]$ has an empty entry **then**
        add f to $bucket[i]$;
        return done;
    end if
end for
// Hashtable is considered full;
return failure;

As mentioned above, Cuckoo Filters use fingerprints to store elements in them. One of the primary reasons is space-optimisation - hashing elements to constant-sized fingerprints reduces the hash-table size. It is self-evident from the eviction and relocation algorithm described above that it is essential that even after storing the element's fingerprint in a given bucket, we to need to compute the element's alternate position. A rudimentary approach here could be to store the alternate position along with the fingerprint in the hash-table. A more space-optimised approach is to use partial-key hashing. Partial-key hashing is a technique to obtain an element's alternate location based on its fingerprint. The hashes that map the item

x to two indices in the hash table are calculated as follows:

$$h_1(x) = hash(x)$$
$$h_2(x) = h_1(x) \oplus hash(x's fingerprint).$$

Based on the above equations, h_2 can be computed using h_1 and the fingerprint. Interestingly, h_1 can also be computed using h_2 and the fingerprint by simply XORing -

$$h_1(x) \oplus hash(x's fingerprint)$$

In this way, we can obtain the alternate bucket for an element using its current bucket i and the fingerprint stored in this bucket using the following equation:

$$j = i \oplus hash(fingerprint)$$

In this way Partial-key hashing ensures that the insertion operation uses only the information stored in the hash table and there is no need to maintain an auxiliary data structure to keep track of the original item or its alternate location. We use this Partial-key hashing technique to populate the hash table for Cuckoo Hashing as described in algorithm 1.

The lookup operation is quite straightforward. We simply compute the fingerprint of x and the two buckets that the element x can be placed in using the hashing described above. We then check if the fingerprint of the element is found in any of these buckets. If it is, the hash table contains the element otherwise it doesn't. Collisions in the hashing that computes the fingerprint can lead to false positives in this process.

Algorithm 2 Cuckoo Filter: Lookup(x)

$f = fingerprint(x)$;
$i_1 = hash(x)$;
$i_2 = i_1 \oplus hash(f)$;
if $bucket[i_1]$ or $bucket[i_2]$ has f **then**
    return true;
end if
return false;

Cuckoo Filters outperform Bloom Filters on some key parameters:

1. **Space Consumption:** As shown in figure 3.1.2, for low false positive rates $\varepsilon < 3\%$, Cuckoo Filters, especially there Semi-Sorted variant consume lesser bits per item than Bloom Filters.

2. **Number of Memory Accesses:** In a Bloom Filter with k hash functions, a positive-outcome query (resulting in the outcome that the element exists in the filter) requires k bits to be read from the bit array. For example, for $\varepsilon = 1\%$, Bloom Filters need $k = 7$. Cuckoo Filters, on the other hand, require reading a fixed number of buckets, resulting in (at most) two cache line misses.

3. **Operational Improvements:** Cuckoo Filters support (1) delete operations, (2) value association: unlike Bloom Filters, Cuckoo Filters can support an external lookup table that associates each element's fingerprint with an additional value. It should be noted that this lookup is prone to the same false positives that the regular lookup operation is prone to due to fingerprint collisions, and have a (3) load threshold: Cuckoo Filters give a clear indication of when to stop (whenever an insert operation exceeds the *maxNumKicks* to ensure the required false positive rates. Bloom Filters, to the contrary, can keep inserting elements at the expense of an increasing false positive rate.

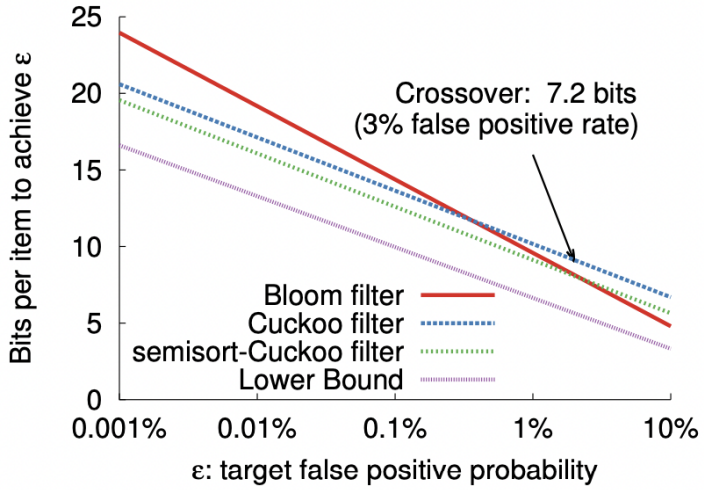


Figure 3.4: Bits per item vs. False positive rate ε

3.1.3 XOR Filters

XOR Filters are a more recent addition to the Approximate Membership Query class of data structures. They build on the concept of Fingerprints that were introduced in the section on Cuckoo Filters and aim to make improvements in terms of speed of construction, speed of lookups, and the filter's space consumption per element [23].

Table 3.2 summarises the notation used in this section. For an element x we generate a k-bit fingerprint through a hash function. We work with the assumption that all fingerprints are equally likely such that the probability P(fingerprint(x) = c) = $1/2k$ for any x and c. All the items we wish to store in the filter are in set S and the set U is the superset of all possible elements in the corresponding universe. The fingerprints are stored in an array of buckets B which has a capacity c which is marginally larger than the cardinality of the set S (e.g., $c = 1.23x|s|$). h_0, h_1, h_2 are three randomly chosen

U	universe of all possible elements (e.g., all strings)
S	a set of elements from universe U (also called "keys")
S	cardinality of the set S
B	array of k-bit values
c = B	size (or capacity) of the array B, we set c = 1.23 · |S | + 32
fingerprint	random hash function mapping elements of U to k-bit values (integers in [0, 2k))
h_0, h_1, h_2	hash functions from U to integers in [0, c/3), [c/3, 2c/3), [2c/3,c) respectively
x xor y	bitwise exclusive-or between two values
B[i]	the k-bit values at index i (indexes start at zero)
\varepsilon	false-positive probability

Table 3.2: XOR Filter Notation

hash functions that map the elements of S to consecutive and disjoint integer ranges - $(h_0 : S \to \{0, ..., c/3 - 1\}$, $h_1 : S \to \{c/3, ..., 2c/3 - 1\}$, $h_2 : S \to \{2c/3, ..., c - 1\})$ e.g., for $c = 15$, these target ranges would be $\{0, ...4\}, \{5, ...9\}, \{10, ..., 14\}$. The values in B are set in a way that the following criterion is satisfied:

$$B[h_0(x)] \oplus B[h_1(x)] \oplus B[h_2(x)] = fingerprint(x), \quad \forall x \in S$$

As with Bloom and Cuckoo filters, membership testing in XOR Filters is also a straightforward operation. As illustrated in algorithm 3, we simply calculate the hash values $h_0(x)$, $h_1(x)$, $h_2(x)$ and the expected fingerprint from the values stored in the array B. If the expected fingerprint matches $fingerprint(x)$, we know that x is contained in the filter.

Algorithm 3 XOR Filter: Lookup(x)

Require: $x \in U$
return $fingerprint(x) = B[h_0(x)] \oplus B[h_1(x)] \oplus B[h_2(x)]$

Consistent, with the results we noted in the section on Cuckoo Filters, the benchmarking done in figure 3.5 reveals that for lower false positive rates both Cuckoo and XOR filters have smaller sizes than Bloom Filters. For very low false-positive probabilities ($5.6 * 10^{-6}$), Cuckoo Filters use less space than Xor Filters. But for more realistic values of false positive rates, XOR Filters are more space-efficient than Bloom and Cuckoo Filters, as illustrated in figure 3.5.

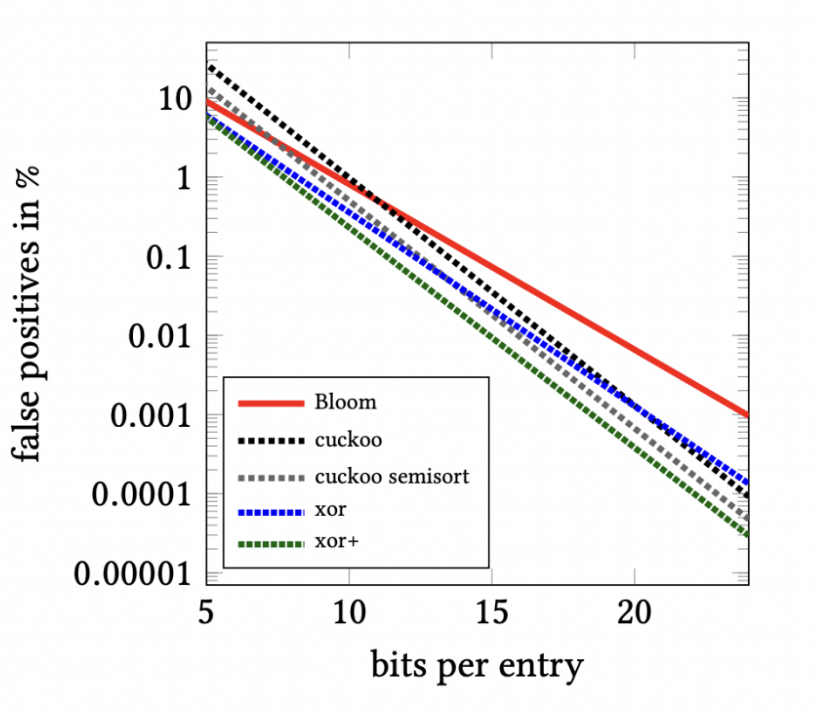


Figure 3.5: False positive rate vs. Bits per item ε

3.2 CRLite: TLS Certificate Revocation at Scale

3.2.1 Transport Layer Security (TLS): A Quick Primer

The TLS (Transport Layer Security) protocol and its concept of certificate chains is foundational to security on the internet. It relies on certificates issued and cryptographically signed by Certificate Authorities (CAs) and more coveted CAs called Root CAs that sign certificates for other CAs. Revocation of these issued certificates whether they are issued to CAs or to websites, it is imperative to achieve completeness in the TLS standard. The private keys of the CA issuing certificates may get compromised at some point rendering any certificates claiming to be signed by them as invalid - revoking this CA's original certificate is a quick way to solve this problem. Similarly, any entity receiving a certificate once must continue to be held accountable and the strongest lever to achieve this is certificate revocation on account of impropriety. CRLite (or Certificate Revocation List Lite) is a solution to solve the TLS Certificate Revocation problem at scale [24].

3.2.2 Solutions Proposed before CRLite

Since revocation is a core concept in TLS certificate issuance, there have been multiple attempts in this area before CRLite. A common denominator in all these solutions was their inability to handle scale. Two popular attempts were Google's CRLSets and Mozilla's OneCRL. In

January 2017, there were more than 12.7M revoked certificates that had been issued by major CAs on the internet [24]. The underlying data formats of CRLSet required 110 bits per revocation and OneCRL required 1928 bits revocation. Storing these 12.7M revocations would make these sets be as large as 166 MB and 2.9 GB respectively. The sizes of these sets make them virtually unusable for a global-scale internet.

3.2.3 Algorithmic Overview

CRLite uses Bloom Filters, that we introduced in section 3.1.1, as building blocks. Taking inspiration from *Chazelle et al.'s* work on Bloomier Filters [25], CRLite arranges several Bloom Filters into a sequence of filters with progressively smaller sets of false positives and calls them Filter Cascades.

To understand the algorithm more concretely, let's assume we have a set R that we wish to store in the cascade and another set S that we wish to not be included. $S \bigcap R = \varnothing$ and $S \bigcup R = U$. It should be noted if that an element $u \in U$ satisfies $u \in R$ then it also satisfies $u \notin S$ and vice-versa.

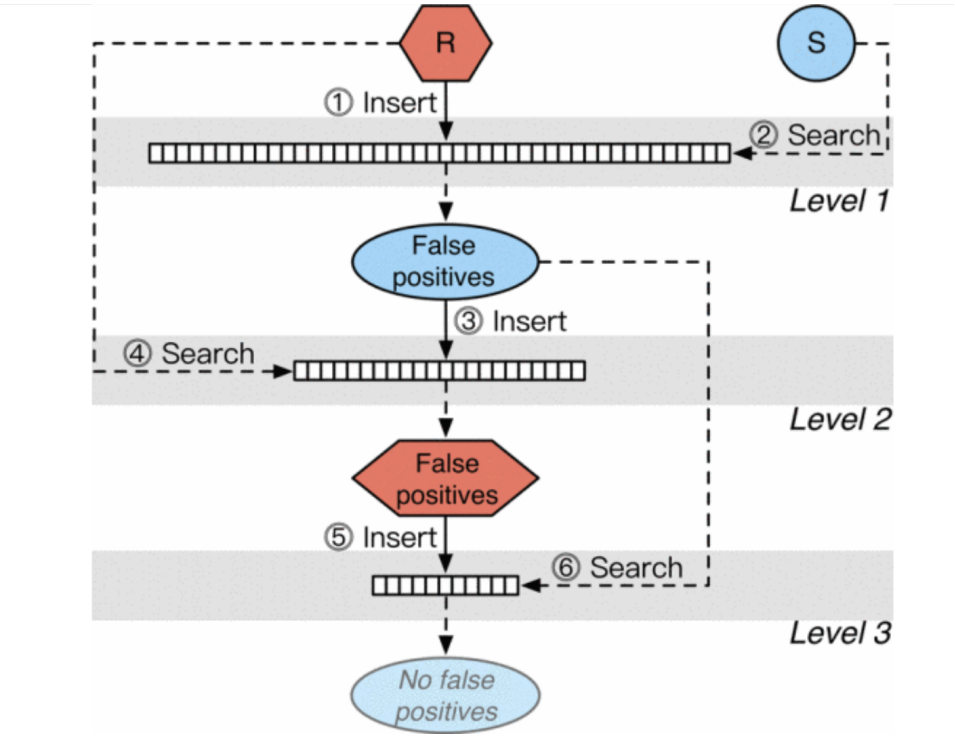


Figure 3.6: CRLite's Insertion Algorithm [24]

In the insertion step, we begin with a single empty Bloom Filter, BF_1 and simply insert all the elements of R to it. For every element $u_1 \in U$ with $contains(BF_1, u_1) = false$, we can surely say that $u_1 \notin R$. However, for every element $u_2 \in U$ with $contains(BF_1, u_2) = true$, we can't surely say that $u_2 \in R$. Some of these elements like u_2 with $u_2 \in S, contains(BF_1, u_2) = true$

form the false positive set, V_1 of BF_1.

We use all the elements in V_1 to create the next filter in the cascade, BF_2. BF_2 is created to contain all the elements which were contained in BF_1 but should not have been there because these elements were not in R. For every element $u_3 \in U$ with $contains(BF_2, u_3) = false$, we can surely say that $u_3 \notin V_1$. By definition of u_3 was not among the false-positives of BF_1 and we can conclude that $u_3 \in R$. Now BF_2 may end up containing some elements of R. These elements u_4 with $u_4 \in R, contains(BF_2, u_4) = true$ form the false positive set, V_2 of BF_2.

Continuing in this fashion, BF_3 would be formed by inserting elements of V_2. To find false positives at this level, we don't need to search through the whole set S but only through elements of V_1 that were not meant to be in BF_3. In figure 3.6, there are no such elements found in BF_3 implying that there are no false positives at this level and that BF_3 is not an *Approximate* Membership Query Data Structure but a *Deterministic* Membership Query Data Structure i.e., for any u_5 and $contains(BF_3, u_5) = true$, $u_5 \in BF_3$. At this point the algorithm terminates and the sequence of cascading filters $[BF_1, BF_2, BF_3]$ ends up being a *Deterministic* Membership Query Data Structure as well.

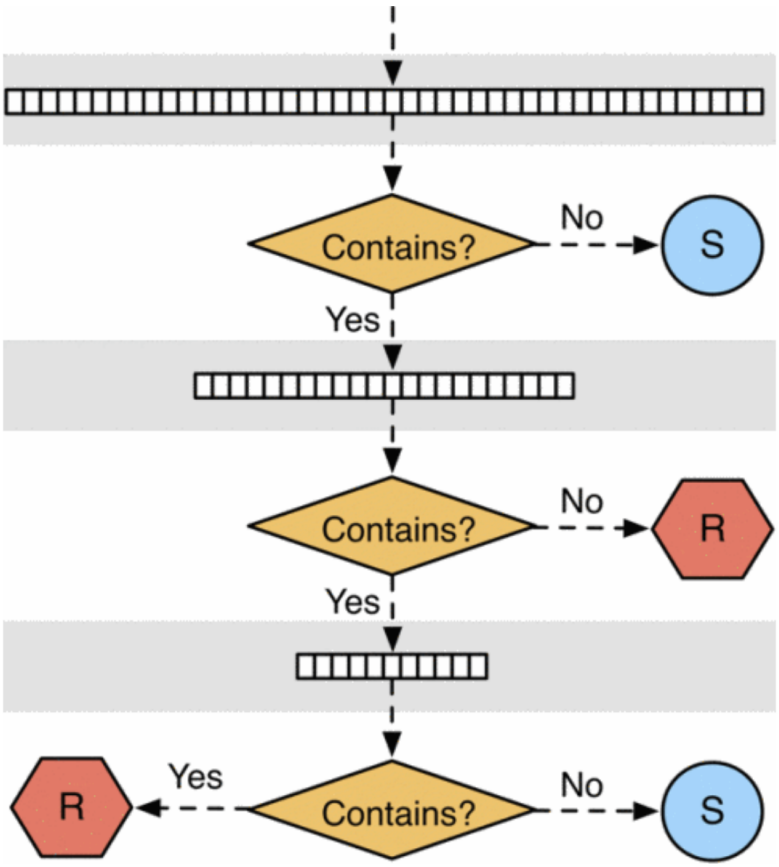


Figure 3.7: CRLite's Lookup Algorithm

Unlike the insertion algorithm, a lookup client does not need to be aware of the universe of values $U = R \bigcup S$ that could have been stored in the cascading filter. Lookup queries take a top-down approach similar to the insertion algorithm. It is worth remembering that Bloom Filters are only able to definitively answer what they do *not* contain and any claims they make about what they do contain can be prone to false positives. This is true for all the filters in the cascade except for the last filter which, by definition, is a *Deterministic* Membership Query Data Structure.

The lookup algorithm terminates as soon as we encounter a filter where the lookup value $u \notin BF_i$, where $i >= 1$. This can be interpreted as follows:

- if i is odd, $u \notin R$
- if i is even, $u \in R$

As an example, if we consider this for the first filter BF_1, computing $contains(BF_1, u) = false$ indicates that R does not contain u which is consistent with 'if i is odd, $u \notin R$' stated above.

If all the filters contain the lookup value u, then the result is computed based on the total number of levels l in the cascading filter.

- if l is odd, $u \in R$
- if l is even, $u \notin R$

As an example, if we again consider this for the first filter BF_1, computing $contains(BF_1, u) = true$ indicates that R does contains u and computing $contains(BF_1, u) = false$ indicates that R does not contain u which is consistent with the algorithm described above.

3.3 GX Credentials

GX Credentials is an open-source project developed primarily by the Software Engineering for Business Information Systems (SEBIS) Chair at the Technical University of Munich. It is a partial implementation of the Verifiable Credentials standard which enables the issuance of VCs to companies and their employees. The deployer of the application operates as a trust anchor and enables identity management among a dataspace or consortium. One of the core principles of GX Credentials is that signing keys must not be stored or managed by anyone besides their owners. This requires a stable interface with wallet apps that Company Admins can operate using their smartphones (or using browser plugins/extensions) to sign Verifiable Credentials which GX Credentials issues on their behalf.

There are primarily three user personas for this application:

1. Operator & Trust Anchor (TA): Operates the GX Credential application and functions as a trusted entity within this ecosystem.
2. Company: These are legal entities that register themselves with the TA, receive Verifiable Credentials, and then go on to issue Verifiable Credentials to their employees.

3. Employee: These are individuals working at these certified companies who wish to prove their association with the company and can do so using Verifiable Credentials.

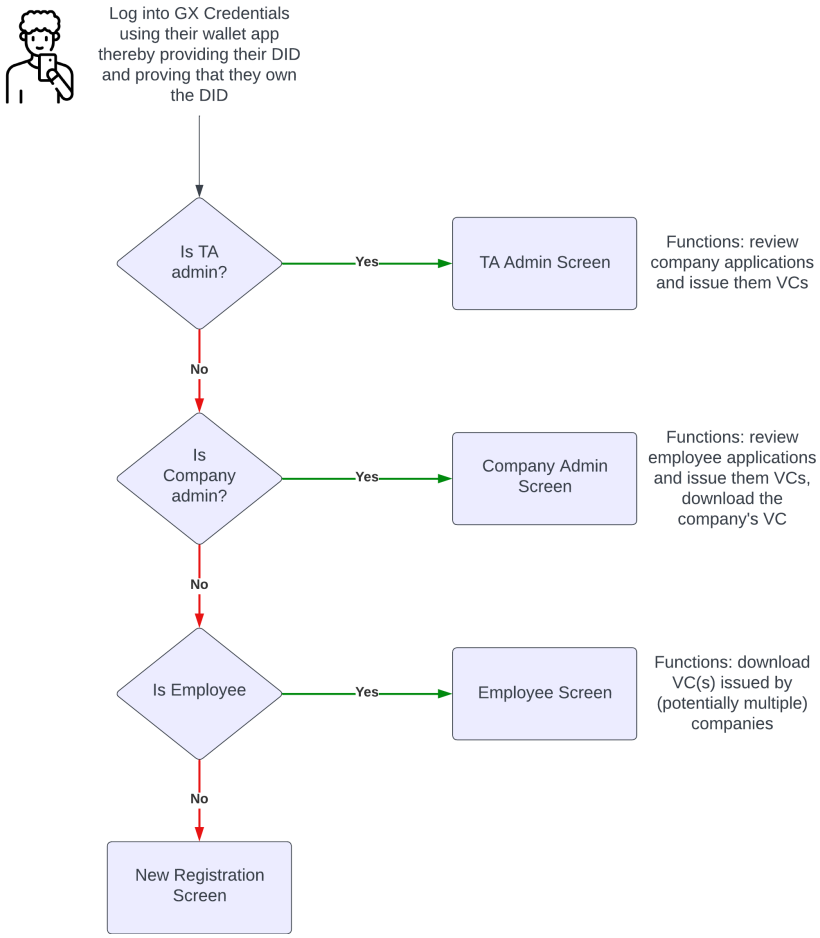


Figure 3.8: Login flow for GX Credentials

While logging into the application, a user may be in one of the following states:

- The user may be a Trust Anchor (TA) admin.
- The user may be a Company admin.
- The user may be an Employee of one or more companies.
- The user may not be registered with GX Credentials.

The application determines which of these states the user is in on the basis of their DID (and by extension through the public-private key pair that they own). Only a TA admin

whose public key is stored in the Tezos blockchain can claim using their wallet that they are the owner of that public key. By presenting a cryptographic challenge to the user which can confirm that they own the public key stored in Tezos, the application determines if the user is a TA admin or not. A similar process is followed for the Company Admins and Employees. If a user's DID is simply not stored in the application, the user would go through the 'New Registration Flow' and register themselves as either a Company or an Employee.

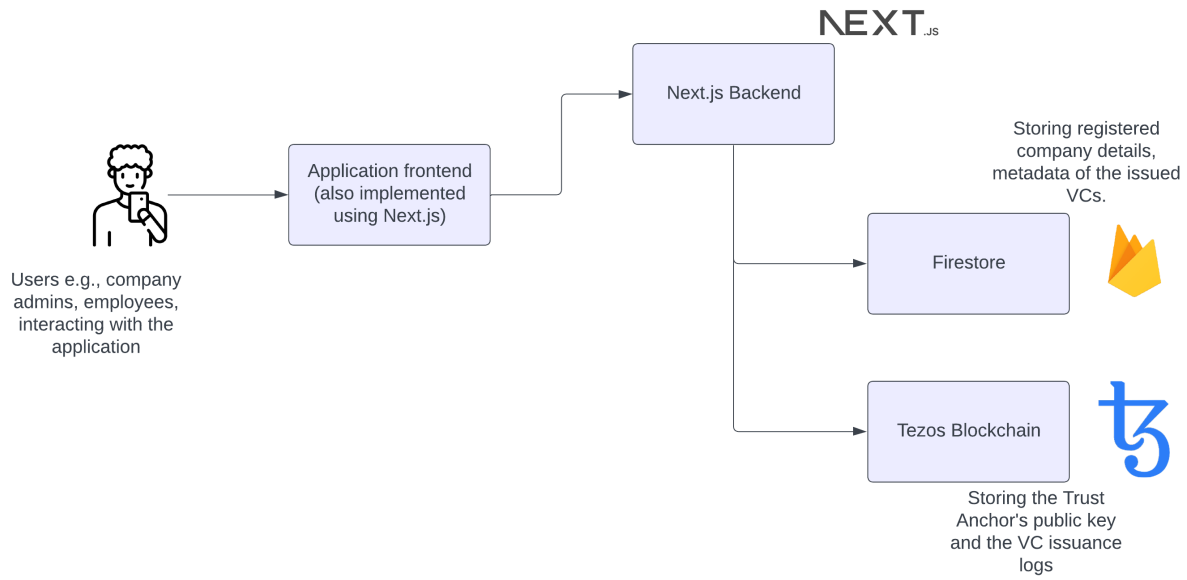


Figure 3.9: Component Diagram for GX Credentials

As illustrated in figure 3.9, the application is developed using Next.js and uses Google Firestore and the Tezos blockchain for storing the application data. GX Credentials uses Public Key Infrastructure (PKI) extensively. The Trust Anchor owns a private-public key pair of which the public key is stored on the Tezos blockchain. Additionally, Tezos is also used to store the issuance log of the VCs that have been issued by GX Credentials. The Firestore database stores the data regarding the companies that have registered with the Trust Anchor and allows GX Credentials to offer convenience features such as downloading a VC.

The data model in figure 3.10 shows the five collections that store the application's data in Firestore. The following points describe the purpose of each of these collections and the use cases that they serve:

- AddressRoles: stores a mapping between DIDs and their roles which is an enum describing the type of user they are (company or employee) and what their approval status is.
- CompanyApplications: stores applications from approved and pending companies.

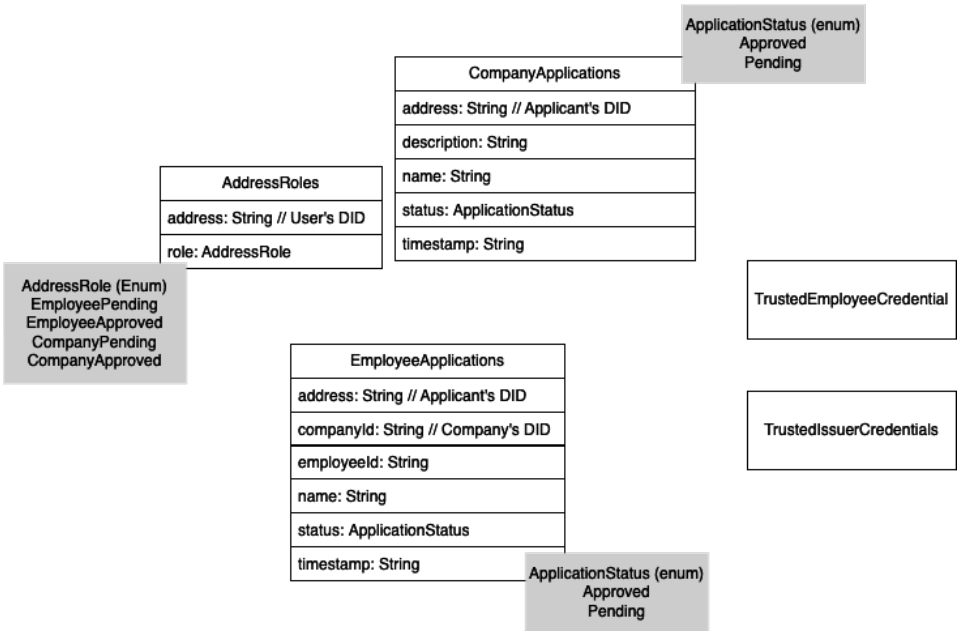


Figure 3.10: Data Model for GX Credentials.

- EmployeeApplications: stores applications from approved and pending employees.
- TrustedEmployeeCredentials: stores credentials issued to approved employees.
- TrustedIssuerCredentials: stores credentials issued to approved companies.

While setting up GX Credentials, the Trust Anchor admin must first setup their wallet with a private-public key pair, store the public key on the Tezos smart contract, and deploy the application after configuring the details of the smart contract. Any subsequent logins will allow the application to successfully identify the user as a TA admin. This flow is illustrated in figure 3.11.

When any user other than the TA admin logs in with their DID, the application first checks if they are already registered. If they aren't, they are shown the new registration flow. Company admins can register themselves as companies by providing background information and applying for a Company VC. These applications are stored in the Firestore database and are reviewed by a TA admin whenever they log in next. This flow is illustrated in figure 3.12.

Finally, the TA admin reviews the list of company applications. For the ones that it chooses to approve, the application generates raw (unsigned) VCs which are then signed by the TA admin using the public key in their wallet. These VCs are stored in the Firestore database for the companies to then download them whenever they log in next. This flow is illustrated in figure 3.13.

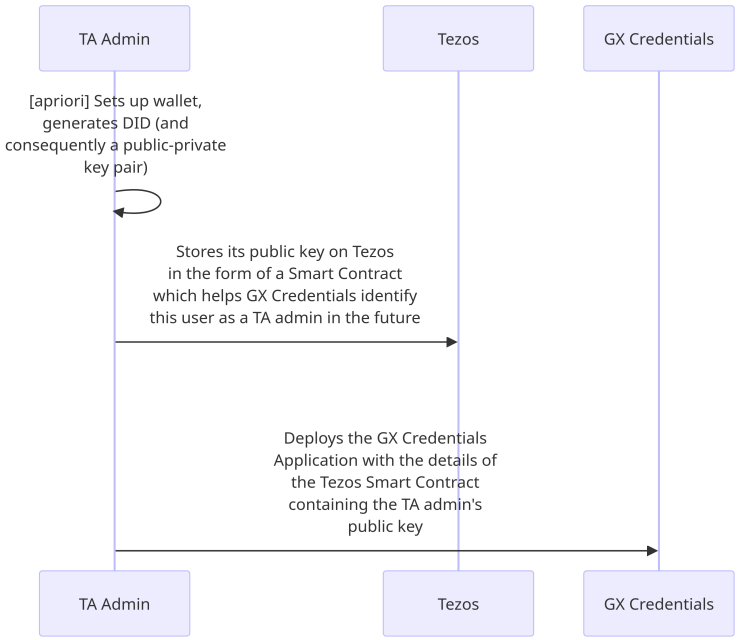


Figure 3.11: TA admin registration flow

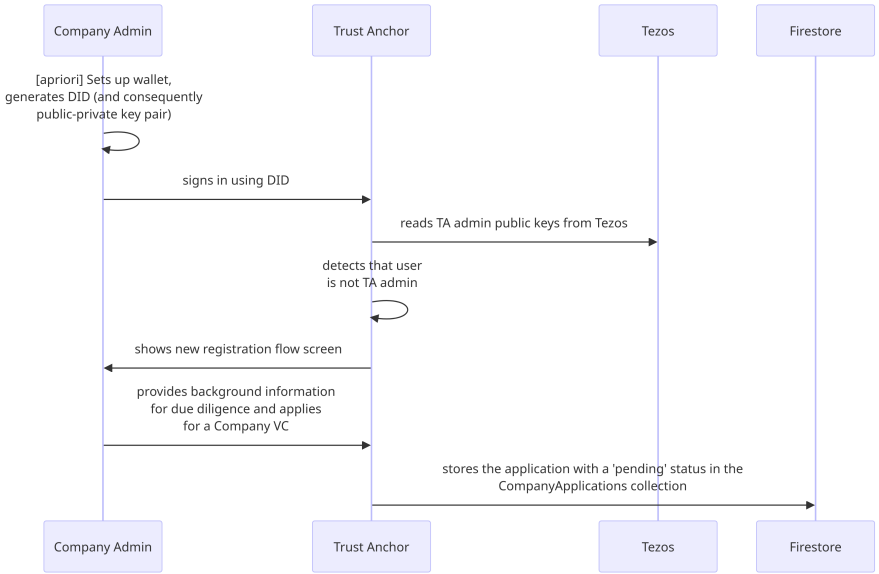


Figure 3.12: Company registration flow

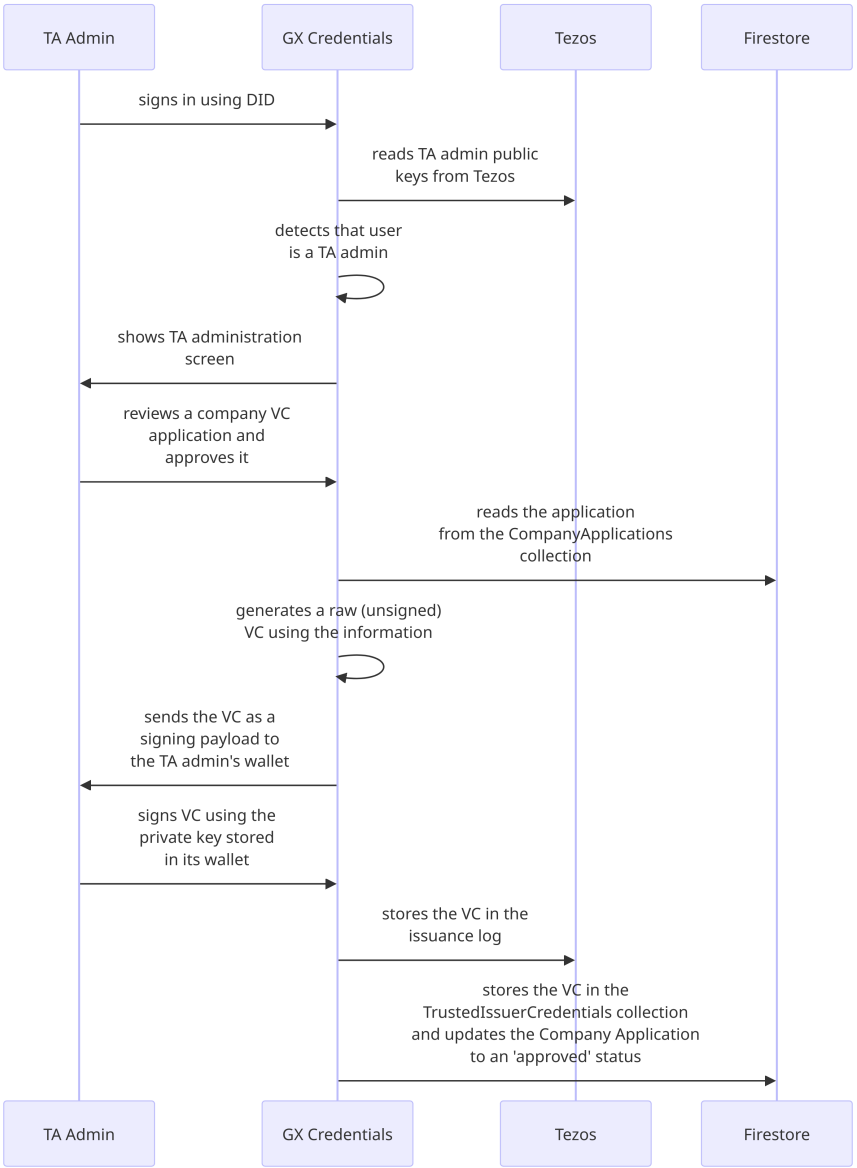


Figure 3.13: TA admin approving a company's request for a VC

4 Analysis

4.1 Robustness of the Current Revocation Setup

4.1.1 Design Goals

There are three main design concerns while engineering revocation/suspension mechanisms such as Status Lists:

1. **Privacy:** There are two parts to the privacy argument in the context of VCs and revocations/suspensions - (1) no parties other than the issuer, the holder of the VC, and the verifying party to whom the VC has been presented should be able to deduce that the holder's VC has been revoked/suspended, (2) the issuer should not be able to deduce the verifying party that the holder wants to assert some claims to e.g. the government should not find out how often a citizen purchases alcohol simply because the cashier checks the citizen's government-issued VC to verify their age.

2. **Scalability:** The revocation/suspension mechanism should be able to accommodate planet-scale numbers e.g. in 2020, there were 2M+ university graduates. Persisting this data for (say) 100 years requires 200M+ VCs to be issued and the revocation/suspension data for these users to be managed.

3. **Minimal propagation delay:** The revocation/suspension of a VC should propagate with minimal delay to minimize the misuse of an effectively invalid VC.

4. **Security:** Lastly, the revocation setup should be secure and uphold the security guarantees that are inherent in the Verifiable Credentials standard.

4.1.2 Why are the Usual Suspects Suboptimal?

Typically in software engineering, we would use data structures like Hash Sets or Unordered Sets for storing elements of arbitrary types. Such a data structure can then be used to answer simple set membership queries like 'is element X contained in the data structure?'. There are two reasons why these data structures don't work in this context - (1) the data structure stores the element itself and therefore, allows a verifier to not only check if a VC is contained in the data structure, but to also see all the other VCs in it - this poses a privacy risk to all the VCs stored in the data structure that are not being verified by the verifier, (2) since the data structure stores the VC itself, it is not space-optimized for the use case of simply finding if the VC is present in it or not. A simple solution to both these problems is to not store the whole VC in the data structure but to apply a one-way hash function to all the VCs we wish

to store and then store only the hashes. If we choose a secure hash function like SHA-256, the size of each hash value is 32 bytes. Assume that we have a relatively small set of 1024 VCs, of which 102 (10%) have been revoked. Storing these SHA-256 hashes for these 102 VCs will consume 3264 bytes. If we were to store the same data in a Status List 2021, it would consume 128 bytes (one bit per VC i.e., 1024 / 8), which is 25 times smaller. A similar pattern would hold true if we assume a larger VC set with the same revocation rate (10%).

4.1.3 Shortcomings in the Status Quo

Status Lists prove insufficient on three out of the four design concerns listed in the previous section.

1. **Privacy:** While attempting to verify a VC, the verifier looks up the `credentialStatus.statusListCredential` field in the VC. This holds an address which can be resolved to find the entire StatusList maintained for this VC (and thousands of others) by the issuer. Once the verifier has this StatusList, it can look up the value at the index specified by the `credentialStatus.statusListIndex` field in the VC. If the value == 1, the VC is revoked. In this process, the issuer knows that the given verifier wishes to know the revocation/suspension status of one of the thousands of VCs in the given StatusList and sufficient **herd privacy** ensures that the issuer can not pinpoint to the specific VC (and therefore, real world entity) that is interacting with the given verifier. However, a malicious issuer can easily break this herd privacy by creating a StatusList for each VC it issues. This way, when a verifier requests for a specific StatusList, the issuer knows exactly which VC is attempting to verify with the given verifier. With this loophole, in the example above, the government could not only track down the individuals who are attempting to purchase alcohol but also which store (i.e., verifier) they like to purchase their alcohol from.

2. **Scalability:** The scale of the updates to a StatusList is easily manageable when it is being stored in and requested from a centralized authority such as a web server managed by the issuer. But when the Status Lists are being published and managed in a decentralized manner e.g., in a distributed ledger, the scale of these updates becomes very germane. In 2021, there were 250M licensed drivers in the US. Assuming that 1 out of every 10,000 drivers has his/her license temporarily suspended once a year for 3 months, we have 25,000 drivers whose licenses were suspended in the whole year. Therefore, on a given day 70 drivers licenses were suspended and the suspension of 70 drivers licenses was reversed. This leads us to 150 changes to the American Drivers' License StatusList 2021 every day. Updating the Status List 150 times daily on a centrally hosted web server is not a challenge at all but making 150 blockchain transactions can be extremely costly due to high gas fees.

3. **Minimum Propagation Delay:** At the outset it is worth mentioning that the Status List 2021 standard does not explicitly make any recommendations about how to ensure minimum propagation delay. In order to address the scalability challenges mentioned

in the point above, we can batch the updates to the Status List and perform all of them together. This would curtail the gas fee overheads caused by frequent updates to the Status List. However, these delayed updates would cause a high propagation delay and may allow a malicious actor to use a VC even after it has been revoked. As a real-world example, an individual whose passport/visa may have been suspended due to legal proceedings may still be able to use their passport/visa because the batch job that updates to the passport revocation list will execute only after a critical mass of updates has accumulated.

4.2 Absence of an End-to-End Architecture

The Status List 2021 standard deliberately makes no recommendations about how the Status List data structure fits into the bigger picture of issuing and managing Verifiable Credentials. This is in harmony with the Verifiable Credential standard which also omits implementation details such as how an issuer must architect its VC issuing system. This is perhaps best demonstrated in the definition of the Verifiable Data Registry which the standard defines as "a system that mediates the creation and verification of identifiers, keys, and other relevant data, such as verifiable credential schemas, revocation registries, issuer public keys, and so on, which might be required to use verifiable credentials" without necessitating any strict guidelines about where these public keys, revocation registries, etc. should be stored. Abstracting away implementation details while authoring these standards is a battle-tested way of preventing them from becoming too opinionated.

In the scope of this thesis, however, we take a more opinionated approach. In addition to proposing a data structure that addresses the design goals delineated in the previous section, we also propose an end-to-end Verifiable Credential issuance architecture along with a revocation mechanism that supports its VC revocation and suspension use cases. Collaborating closely with the team at the Software Engineering for Business Information Systems (SEBIS) Chair, we use their existing work on the GX Credential application and propose modifications and appendages to that system for achieving an end-to-end architecture. In this section we discuss the present-day architectural shortcomings in GX Credentials to achieve this.

4.2.1 Information Siloes Must Exist

Presently, the only backend server in the GX Credential architecture is run and managed by the Trust Anchor. While the Trust Anchor, by definition, is a trusted entity in this ecosystem, there are some inherent flaws with this approach. These are described in the following points:

1. The issuers should have complete sovereignty over the VCs they issue and be able to store them on infrastructure they own.

2. As a single point of failure for storing VCs, the Trust Anchor becomes vulnerable to attacks from malicious actors, making the overall system more vulnerable.

3. Storing these VCs has cost implications for the Trust Anchor both for storage, data replication, and security infrastructure. These costs should ideally be born by the VC issuers.

4. In the present setup, there is a fatal flaw - the Trust Anchor publishes all the ids of all the issued VCs in an auditable log called the 'issuance log'. This issuance log is stored on the Tezos blockchain and is therefore, readily available for anyone on the internet to read. This is a privacy violation. Any portion of a VC (prior to a verification interaction) must be accessible only to their holders, issuers, and in case of GX Credentials, (temporarily) to a trusted party such as the Trust Anchor.

GX Credentials has successfully ensured that signing keys are not stored or managed by anyone besides their owners. In a similar way, information siloes need to be defined and managed for all other data in the ecosystem such as the VCs issued via a given Trust Anchor deployment. A guiding principle for this should be that while the Trust Anchor is required for new VCs to be issued conveniently, it should not be essential to the system for VCs to be verified (including revoked/suspended ones), or for issuers to get access to all the VCs that they have issued to their employees, or the VCs that they have suspended/revoked.

4.2.2 No Verifier Implementation

GX Credentials is currently viewed as a server managed and run by the Trust Anchor. In reality, it's an open-source library that can be downloaded and run by any Trust Anchor. After enabling revocation/suspension of VCs, it is a natural next step for GX Credentials to offer a library that can perform VC verification. This library can then be used by any verifier (e.g., an alcohol store owner, a passport control officer, hospital staff, etc.) to verify the authenticity of a given VC.

Verifying a VC in this setup would be a two-step process - (1) Check if the VC's signature matches its contents to ensure that it has not been tampered with, (2) Check a revocation source of truth to find out if the VC was not revoked by its issuer. This process would have to be recursively executed for all the VCs in the Verifiable Presentation.

Figure 4.1: Issuance Log for a deployment of GX Credentials. This log is stored in the storage of a Smart Contract hosted on the Tezos Blockchain. Both the VCs here are valid (see the `active` field).

4.2.3 Limited Support for Revoking/Suspending VCs

The current mechanism to revoke VCs is via the `revokeIssuance` endpoint in the smart contract GX Credentials deploys on Tezos. This means that the smart contract's storage maintains public log of which VCs have been revoked/suspended which is a violation of the privacy design goal that is outlined in the previous section.

5 Component Design

The thesis aims to propose an alternative mechanism for revoking Verifiable Credentials. In this section, we introduce our design proposal by dividing the underlying problem space into two questions and answering them in the subsections below. The first question is 'Which data structure should the revoked VCs be stored in?' and the second question is 'Where should the data structure be computed and stored?'

5.1 An Alternative Revocation Mechanism

This subsection is concerned with the data structure that we use for storing the revoked VCs. As described in section 4.1.1, we have four design goals for this data structure - (1) It should preserve the privacy of the VC holder, (2) It should be able to handle planet-scale volumes, (3) It should involve a minimum propagation delay to ensure that there is a minimum delay in the revocation decision being enforced, (4) It should be secure.

We acknowledge that compared to alternatives like unordered sets containing VCs, sets of VC hashes, etc., the Status List 2021 offers the simplest and most space-optimised revocation data structure as of this writing. With the combination of the design discussion in this subsection and the next one, we aim to offer a data structure that improves upon Status List 2021 on the design goals we describe in section 4.1.1.

5.1.1 Drawing Inspiration from CRLite

CRLite is a scalable system for pushing TLS certificate revocations to web browsers. The framework uses Bloom Filters, which are inherently approximate membership query data structures and arranges them into a sequence of Bloom Filters and the overall data structure then becomes a deterministic membership query data structure.

AMQ data structures are a space-optimised way to answer membership queries but the lack of determinism (due to false positives) make them less ideal for our use case. The data structure containing the revocation list for Verifiable Credentials must also be space-optimized so that updates to it can be made without large storage overheads and it can be easily transported over computer networks. For this reason, the idea to use a sequence of these filters to achieve determinism shows promise for a VC revocation list as well. An additional benefit is the privacy-related advantage stemming from the fact that AMQ data structures and consequently, sequences of those AMQ data structures, don't need to store the actual VC.

We go a step further than CRLite and construct the VC revocation list in a way that it can support different types of AMQ data structures like Cuckoo Filters, XOR Filters, etc. This

allows the possibility to extend this setup to new AMQ data structures as and when they are invented. The only restriction is that they must conform to the simple API that we describe in the following subsection.

5.1.2 The AMQ API

As mentioned in the previous subsection, our implementation allows the revocation list to use different AMQ data structures under the hood as long as they support the following simple API:

```
void insert(E element)
boolean maybeContains(E element)
```

Listing 5.1: The API an AMQ data structure must support

5.1.3 Cascading AMQs: Storing VCs

The central idea behind storing a set of VCs in a cascading AMQ is that we progressively decrease the false positive set from one level to another. As illustrated in figure 5.1, We always start by storing revoked VCs in the first level. We then check which of the VCs in the valid set (VCs which were issued but not revoked) are contained in level-1 due to false positives in the AMQ. These VCs form the false positive set for this level. The false positive set for level-1 is stored in level-2. In this sense level-2 stores the valid VCs which were incorrectly included in level-1. Level-2's false positives are the revoked VCs which were incorrectly included in it. These are identified by checking against the set of revoked VCs. Level-3 is used to stored these false positives. The cascade of these AMQ data structures continues to get built till we encounter a level that has no false positives. The AMQ data structure at this level is actually a deterministic membership query data structure because it has no false positives, by definition.

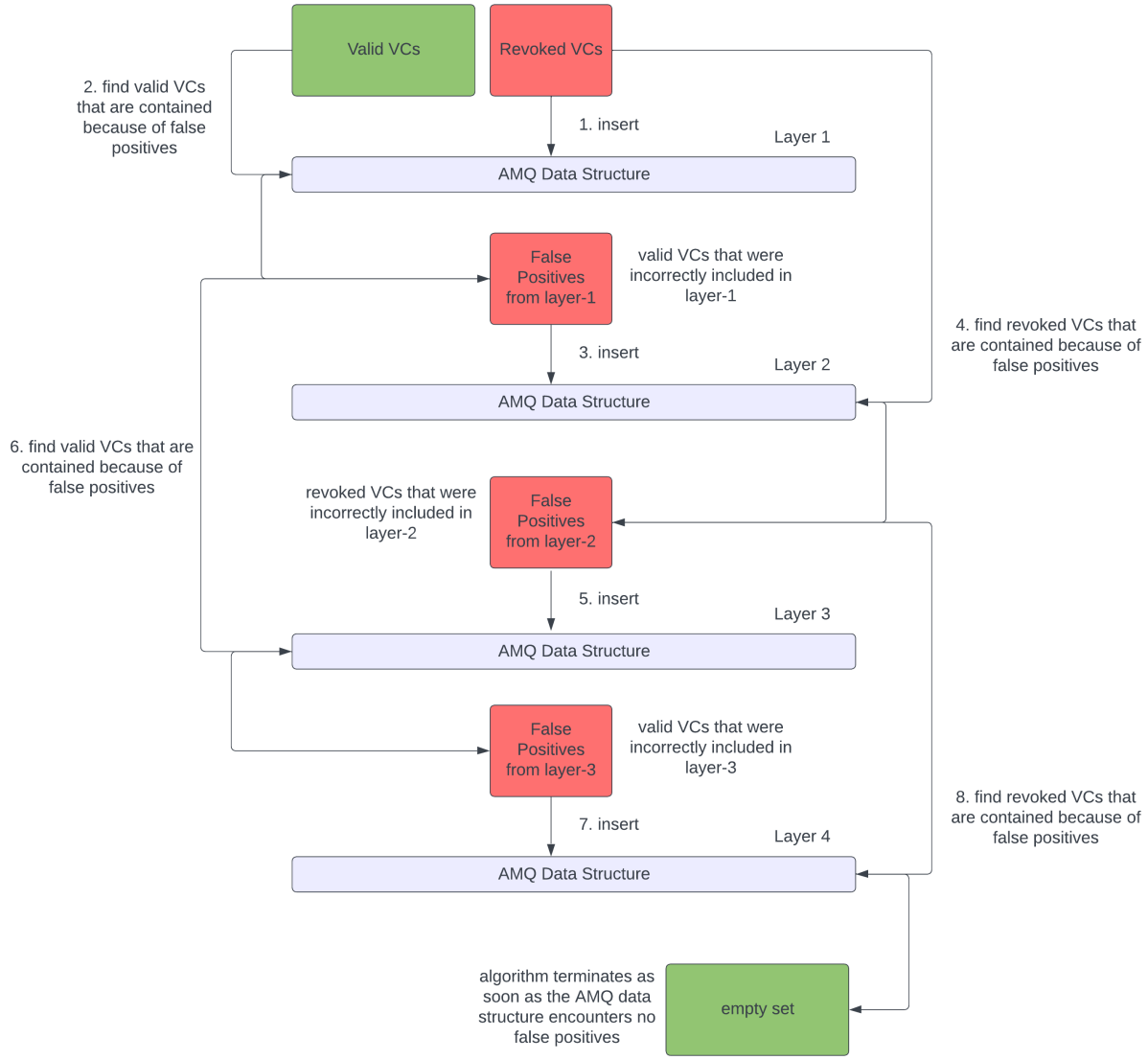


Figure 5.1: Inserting revoked VCs into a Cascading sequence of AMQ Data Structures.

5.1.4 Cascading AMQs: Checking the Presence of a VC

To understand the lookup algorithm, it helps to think of the *intentions* of each layer in the casacading AMQ data structure. The first level intends to store revoked VCs. Each individual AMQ data structure has no false negatives. The implication here is that if level-1 returns false for a given VC's membership query, the VC is not in the revoked set and is therefore in the valid set. Similarly, level-2 intends to store valid VCs so if its membership query returns false for a given VC, we know that it is in the revoked set. If the membership query returns true at any of the levels (except for the last level), we can not be sure if the VC is contained in that level or if its because of a false positive. To be sure, we check the VCs presence in the next level and interpret it according to what the given level intends to store. The AMQ

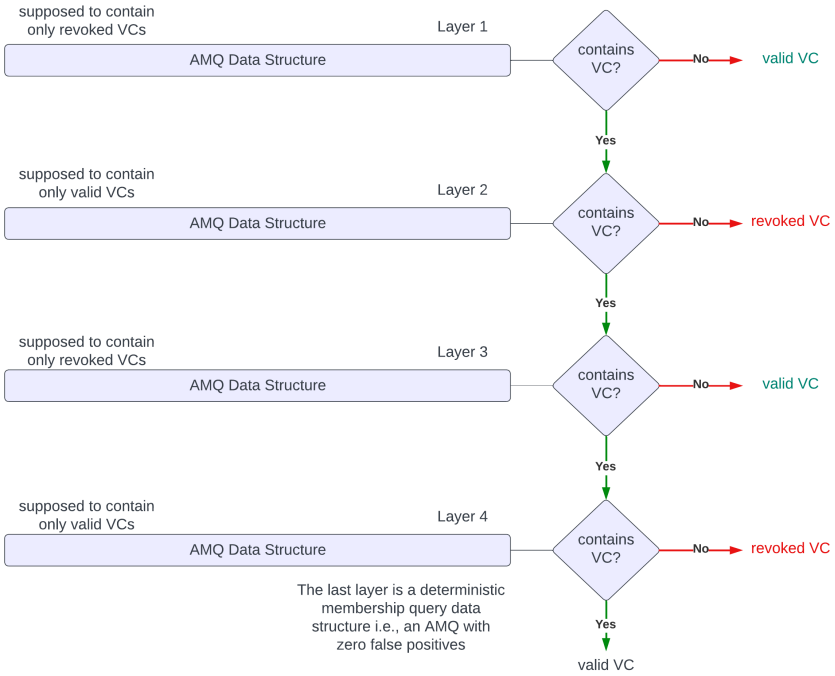


Figure 5.2: Checking if a VC is contained in the given cascading sequence of AMQ Data Structures.

in the final level is deterministic by definition i.e., the membership query returning a true for a VC implies that the VC is contained in the AMQ. If we traverse the complete cascade encountering a true result for membership queries at every level, the total number of AMQs in the cascade allow us to interpret the membership query result of the last AMQ.

5.2 Architectural Extensions of GX Credentials

As described in section 5.2, we have three stated goals with the architectural extensions of GX Credentials. These are as follows:

1. **Information Siloes:** Ensuring that systems within GX Credentials have access to information on a need-to-know basis.
2. **Revocation:** Enabling revocation/suspension of Verifiable Credentials.
3. **Verification:** Checking the authenticity and validity of Verifiable Credentials.

5.2.1 Creating Information Siloes

In table 5.1, we summarise the different categories of data in GX Credentials and which systems store them. Unsurprisingly, in the current design, all the different information types

are managed by the Trust Anchor Admins and stored on infrastructure run by the Trust Anchor. In this section, we propose changes to the distribution of this data and in some cases, removing certain information types altogether.

Information Type	System storing the information GX Credentials	Actors Managing the System
Trust Anchor Public Key	Tezos Blockchain	Trust Anchor Admins
Issued Employee Credentials	Firestore Database	Trust Anchor Admins
Issued Company Credentials	Firestore Database	Trust Anchor Admins
Company Applications	Firestore Database	Trust Anchor Admins
Employee Applications	Firestore Database	Trust Anchor Admins
Issuance Logs (for Companies and Employees)	Tezos Blockchain	Trust Anchor Admins

Table 5.1: A summary of the different information types in GX Credentials and which systems store them.

Information Type	Status in proposed architecture
Trust Anchor Public Key	unchanged
Issued Employee Credentials	not stored by TA
Issued Company Credentials	unchanged
Company Applications	unchanged
Employee Applications	not stored by TA
Issuance Logs (for Companies and Employees)	deprecated

Table 5.2: Information types in GX Credentials and their status in the newly proposed architecture.

As illustrated in table 5.2, the Trust Anchor's public key, the Company Applications it receives, and the Company Credentials it issues can continue to persist in their original locations. The public key is stored on Tezos so that it is accessible to any verifier wishing to verify the authenticity of a company credential issued by the Trust Anchor. Additionally, the Trust Anchor also uses this public key to verify that a given user is a Trust Anchor admin. In

a similar way, since the Trust Anchor issues the company credentials, it is natural for it to store the applications it receives and the credentials it issued for bookkeeping purposes.

The issuance logs mainly serve the purpose of tracking VCs that have been revoked/suspended by the Trust Anchor or the Companies issuing them. We have already established that this is far from ideal due to the privacy violations inherent in this setup. The issuance logs will no longer be stored, especially on publicly readable infrastructure like the Tezos blockchain. In section 5.2.2, we propose an alternative way to support revocations/suspensions.

The remaining information types - Employee Credentials issued by Companies, and the applications those Companies receive also need not be stored by the Trust Anchor. The VCs in this context are issued by the Companies and they must have sovereignty over the applications for VCs and the VCs themselves.

5.2.2 Introducing the Company Membership Service (CMS)

In order to enable data sovereignty for the Companies, we introduce a new library in the GX Credentials code base which will be deployed and run by the Companies themselves. This component is called the Company Membership Service (CMS). The CMS is a microservice responsible for storing, querying, and managing the lifecycle of the VCs issued by a Company and their metadata.

The Company Registration Flow described in figure 3.12 stays largely unchanged. The minor changes made to the flow are illustrated in figures 5.3 and 5.4. The Company admins must set up an instance of the CMS before applying for a VC, provide its URL during the VC application process, which would then be checked by the Trust Anchor before issuing the Company's VC.

In the current setup, the flow for issuing VCs to Employees is identical to the flow for issuing VCs to Companies with the only difference being that the former are signed by the Company admins and the latter by TA Admins.

As stated above, the Company registration and VC issuance flow stays largely unchanged. The user interactions in the Employee registration flow also remain unchanged, however, the new architecture proposes several backend changes. Figures 5.5 and 5.6 illustrate all these changes. It should be noted that in the proposed architecture, all the interactions with the Firestore database are eliminated in the Employee registration flow and the VCs are stored and served only by the CMS.

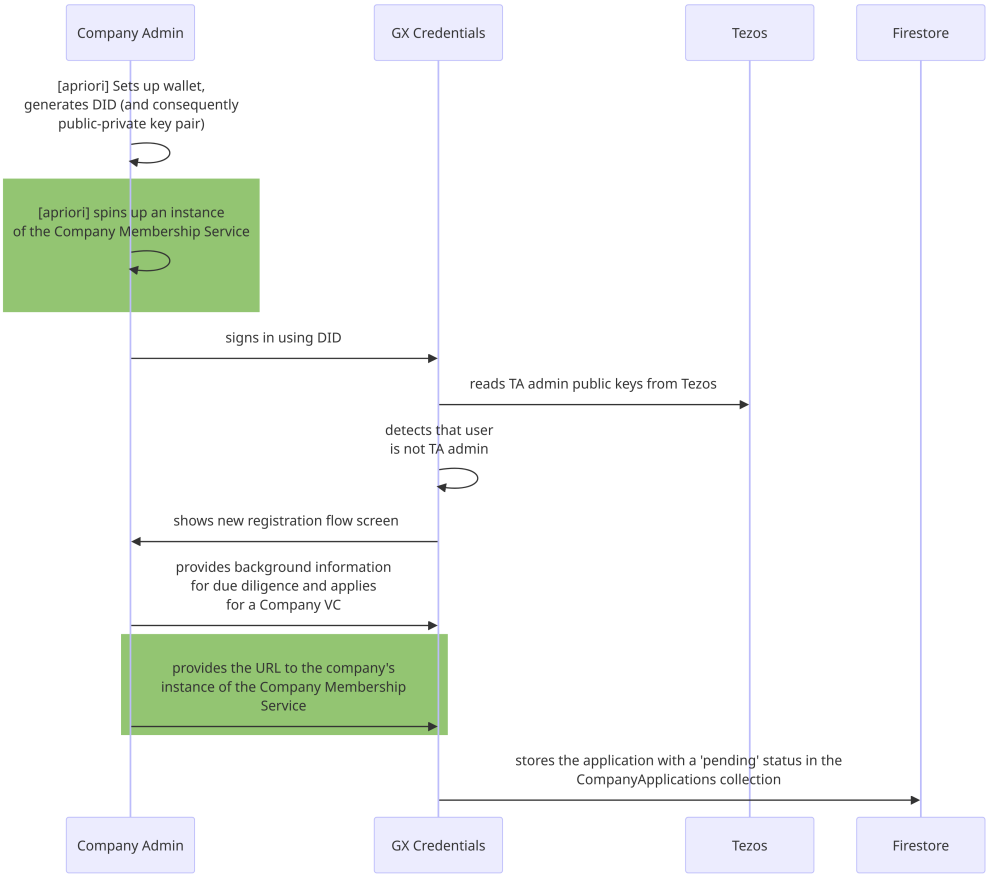


Figure 5.3: Changes to the Company registration flow: The Company is required to setup an instance of the CMS and register its URL with the Trust Anchor.

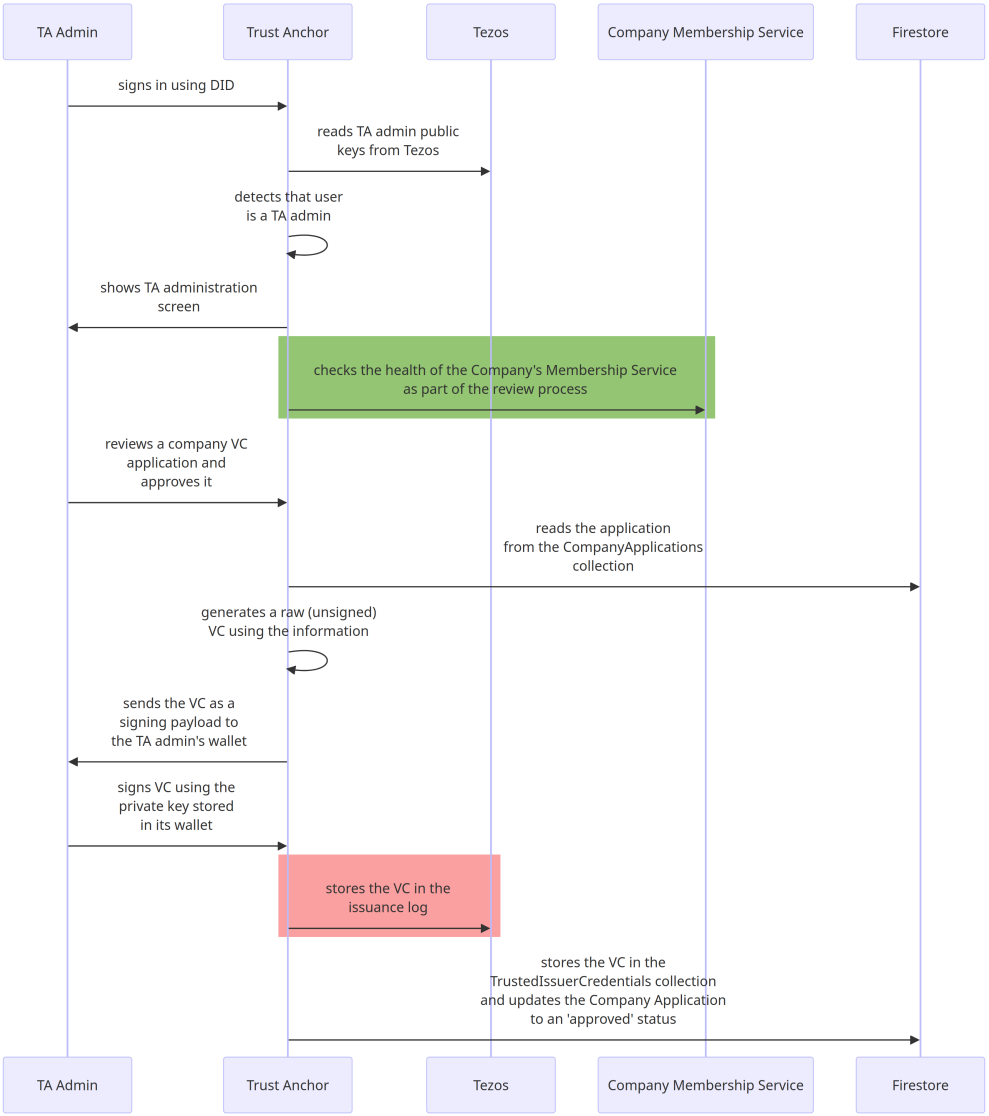


Figure 5.4: Changes to the Company registration flow: The Trust Anchor needs to check the health of the CMS before approving the VC application. The Trust Anchor no longer stores the VC in the issuance log.

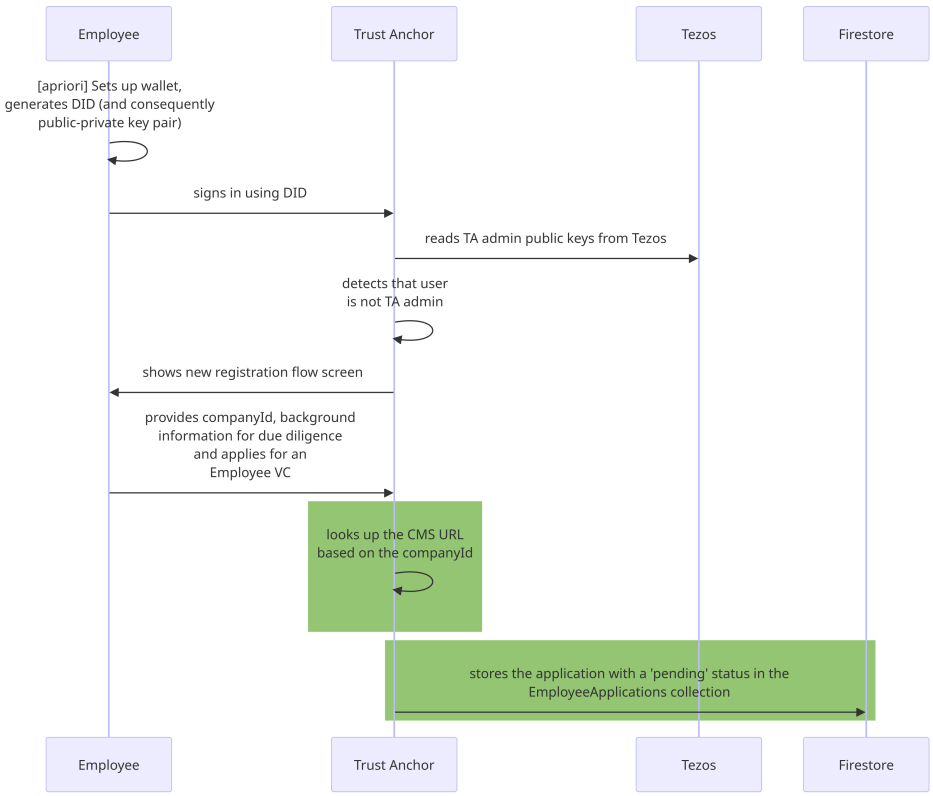


Figure 5.5: Changes to the Employee registration flow: The Trust Anchor does not store the Employee applications. This data is now managed by the CMS.

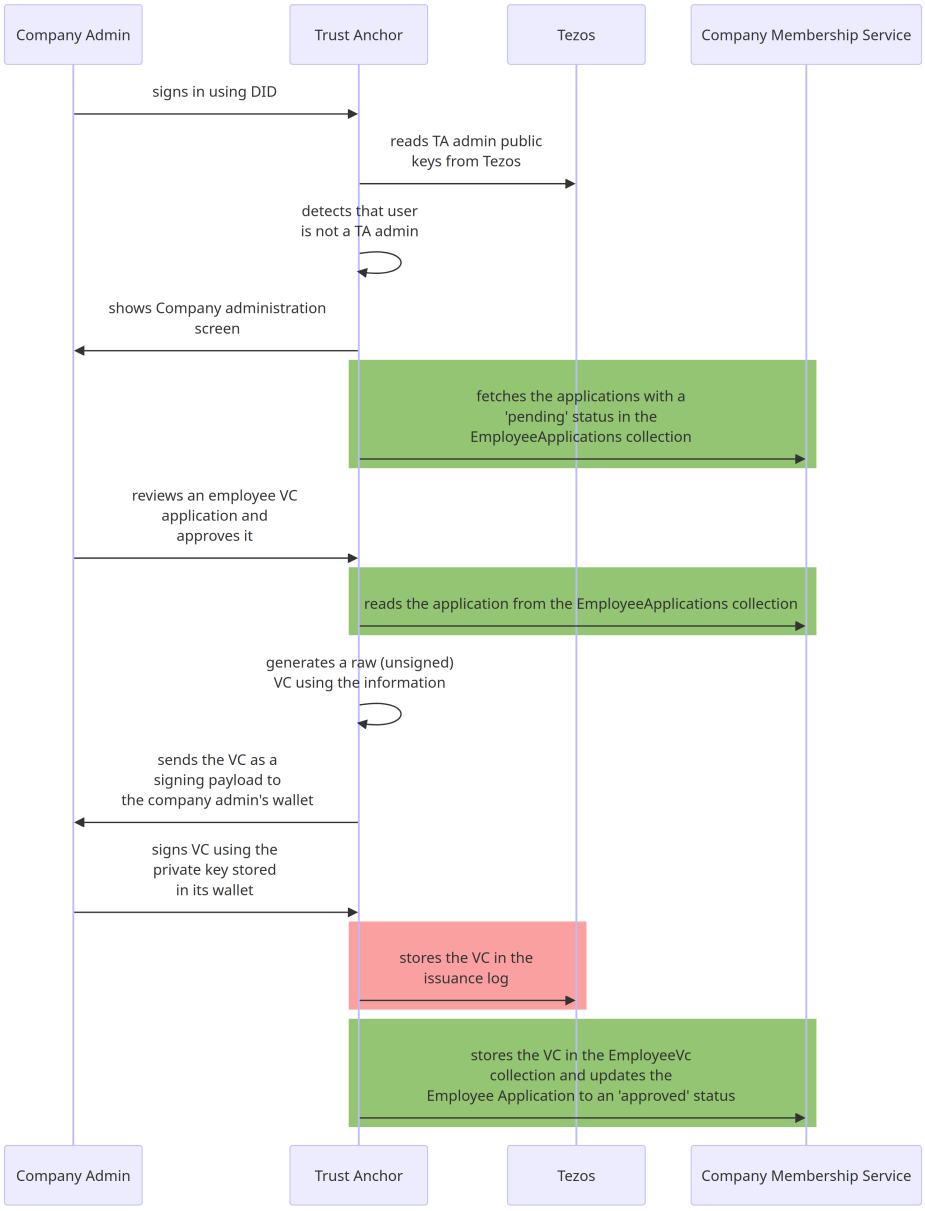


Figure 5.6: Changes to the Employee registration flow: The Trust Anchor does not store the Employee VCs. This data is now managed by the CMS. The Trust Anchor no longer stores the VC in the issuance log.

5.2.3 The CMS API

After introducing the Company Membership Service above, we now define the CMS API and explain how it operates securely. A Company Admin running an instance of CMS must ensure that it serves a set of mandatory endpoints. All these endpoints are implemented as part of the GX Credentials CMS library. Therefore, an admin gets them out of the box if they use the library to implement their CMS. Using this library as a scaffold, Company Admins are free to implement any other endpoints that they deem necessary for their use cases.

Endpoint	Short Description
/listPendingApplications	returns all the employee applications that have status == 'PENDING'. Called by Trust Anchor to display the Company Administration screen.
/createApplication	adds an Employee's request for a VC as an application and sets its status to 'PENDING'
/readApplication	fetch the employee application for a given applicationId.
/storeVc	store a signed VC issued to an employee.
/revokeVc	revoke an issued VC.
/suspendVc	suspend an issued VC until a specified suspension expiry time.
/undoRevokeVc	un-revoke a previously revoked VC.
/undoSuspendVc	un-suspend a previously suspended VC.
/getVc	fetch the VC for a given VcId.

Table 5.3: A summary of the mandatory endpoints implemented by the Company Membership Service.

All these endpoints are accessible to anyone on the internet and therefore are vulnerable to attacks from malicious actors. In order to secure these endpoints, we need an authentication mechanism. As part of this authentication mechanism, the CMS needs to answer one question - "is the given request being made by one of the Company Admins?". VCs are inherently meant as a way for people and systems to assert claims about themselves. Since GX Credentials already establishes a seamless workflow for issuing VCs, we can use VC to answer the above question regarding the request's origin.

For every operation, the Trust Anchor creates an unsigned VC that only contains the name of the operation and a variable number of arguments pertaining to that operation. For example, for the `/listPendingApplications` endpoint, there are no arguments while the `/revokeVc` endpoint receives the id of the VC we wish to revoke as an argument. Once this operation-specific VC has been signed by a Company Admin, it is unimpeachable proof that the instructions encapsulated in the VC was issued by an authorised actor. The VC can then

be used by the Trust Anchor to perform a trusted interaction with the CMS on behalf of the Company Admin.

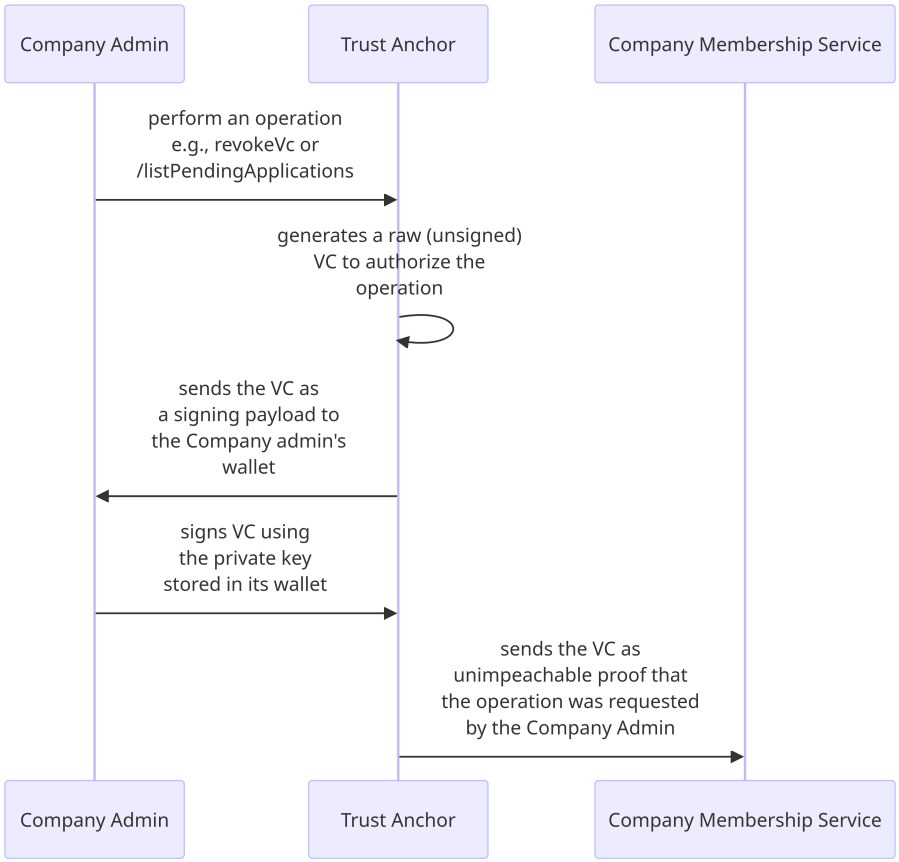


Figure 5.7: Sequence Diagram for an operation such as /revokeVc or /listPendingApplications being triggered by the Company Admin.

5.2.4 VC Revocation

For the sake of brevity, we are not elaborating on the functioning of each of the API endpoints introduced above. However, in this section, we discuss the `/revokeVc` endpoint at length.

Similar to the other operations, the `/revokeVc` endpoint also uses an operation-specific VC as an unimpeachable instruction from the Company Admin that instructs the CMS to revoke the specified VC.

```
{
  "credentialSubject": {
    "gx:issuerCompanyName": "Srajit Inc.",
    "gx:operation": "REVOCATION",
```

```
      "gx:employeeVcId": "b1226336-8e7d-11ee-b9d1-0242ac120002",
      "id": "did:pkh:tz:tz1MRoRc5DgRmXHiyfxtGm3EB8mqeLpBfrUA",
    },
    "type": [
      "VerifiableCredential",
      "Operation Credential"
    ],
    "issuanceDate": "2023-10-07T16:41:56.044Z",
    "id": "urn:uuid:20e34a3d-fec5-46d7-949f-7ebf3947642d",
    "proof": {
      "proofValue": "edsigtajEzCi1DApxSue15AC5uw...sUxkyn9mtjyo648",
      ...
      "created": "2023-10-07T16:41:56.082Z",
     }
}
```

Listing 5.2: Sample VC for the Revocation operation. It specifies the id of the VC to be revoked through the gx:employeeVcId field

We may assume that the details of the following discussion are applicable to both revocations and suspensions, although we only explicitly talk about revocations here.

All the revoked (or suspended) VCs for a Company at a given point in time will be stored in a data structure called the revocation list. This revocation list needs to be publicly accessible to a verifier wanting to check if a given VC issued by the Company has been revoked (or suspended). Storing this revocation list on a centralised server managed by the Company or the Trust Anchor is fraught with all the dangers of centralized infrastructure and single points of failure. So we definitely wish to store in a decentralised way. Depending on the issuer's use case, the revocation/suspension list may need to be updated very frequently. The issuer could be a small country's passport office which rarely ever revokes or suspends a citizen's passport or it could be the California DMV which may revoke/suspend (and revert these revocations/suspensions) several thousand driving licenses in a week. Making these updates as they occur to a blockchain like Tezos would entail frequent overheads in gas fees. On the other hand, making these updates in batches to curtail the gas fees would lead to a high propagation delay causing lags in enforcing VC revocations (and suspensions). For these reasons we choose to store the revocation list on the Interplanetary File System (IPFS) and address it using a self-certifying Interplanetary Name System (IPNS) name.

The content stored on IPFS i.e., the revocation list, will be updated whenever a new VC is revoked, suspended, or the revocations/suspensions are reverted. Each time this happens, the IPFS Content Identifier (CID), which is derived from the content it points to, will also change. The IPNS name will stay static and its contents will be mutated point to the new IPFS CID. The contents of an IPNS record can only be updated by the Company admins. This is because the update operation requires the private key that was initially used to allocate

the IPNS record. Additionally, the IPNS record contains a signature which allows anyone to verify that the record was signed by the private key holder i.e., the Company Admins in this context.

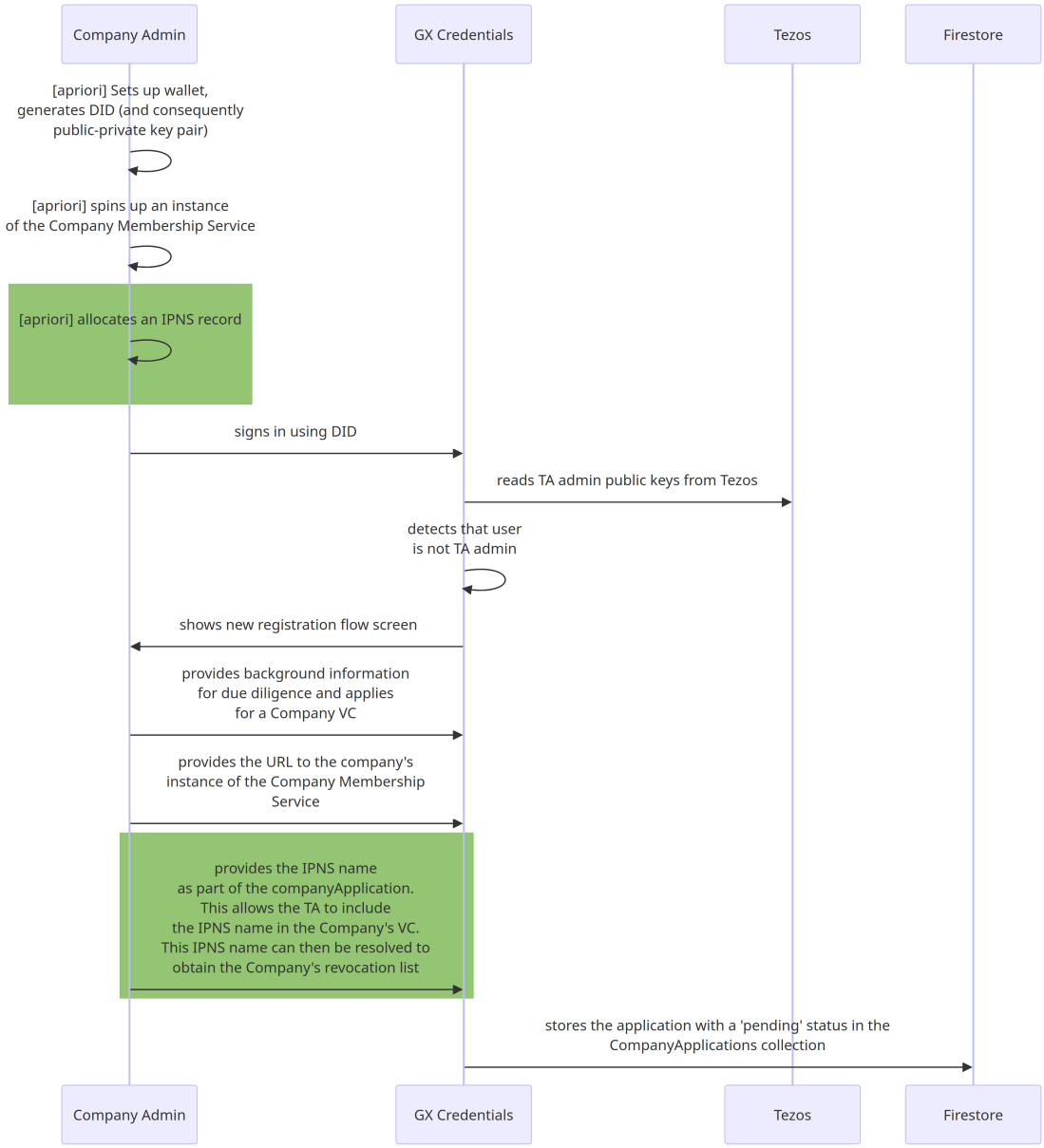


Figure 5.8: Allocating storage on IPFS and IPNS for the revocation list and including it in the Company VC.

Figure 5.8 illustrates that the Company Admin must allocate an IPNS record (with an empty IPFS storage since at this point, the recovation list would be empty) before participating the company registration flow. The admin must provide the IPNS name for its allocated IPNS

record in its VC application. This IPNS name would then be included in the VC issued to the Company. The Company VC storing the IPNS name is preferred over each Employee VC storing it. This is mainly to prepare for an unforeseen scenario in which the key-pair controlling the Company's IPNS record gets compromised. In this case, the Company can allocate a new record and reissue a Company VC for itself which includes the new IPNS name.

Figure 5.9 shows the interaction between the CMS and IPFS when the former receives a VC revocation request. The CMS is responsible to check the signature of the operation VC that it receives, recompute the revocation list, publish it to IPFS, and update the content path in the Company's IPNS record to point to the new CID.

5.2.5 Extending Revocation with VC Suspension

The infrastructure used for VC revocation can be completely reused for VC suspension. Suspending a VC is akin to revoking a VC for a finite amount of time. This time duration is specified using the `gx:suspensionDuration` field in the VC for the Suspension operation.

```
{
  "credentialSubject": {
    "gx:issuerCompanyName": "Srajit Inc.",
    "gx:operation": "SUSPENSION",
    "gx:suspensionDuration": "P1Y2M3DT1H30M10S",
    "gx:employeeVcId": "b1226336-8e7d-11ee-b9d1-0242ac120002",
    "id": "did:pkh:tz:tz1MRoRc5DgRmXHiyfxtGm3EB8mqeLpBfrUA",
  },
  "type": [
    "VerifiableCredential",
    "Operation Credential"
  ],
  "issuanceDate": "2023-10-07T16:41:56.044Z",
  "id": "urn:uuid:20e34a3d-fec5-46d7-949f-7ebf3947642d",
  "proof": {
    "proofValue": "edsigtajEzCi1DApxSue15AC5uw...sUxkyn9mtjyo648",
    ...
    "created": "2023-10-07T16:41:56.082Z",
   }
}
```

Listing 5.3: Sample VC for the Suspension operation. It specifies the id of the VC to be suspended and the duration for the suspension through the gx:employeeVcId and the gx:suspensionDuration fields respectively

All suspended VCs are initially added to the revocation list and published to IPFS in the same way as revoked VCs. In addition to this, the CMS runs a cron job that executes at fixed

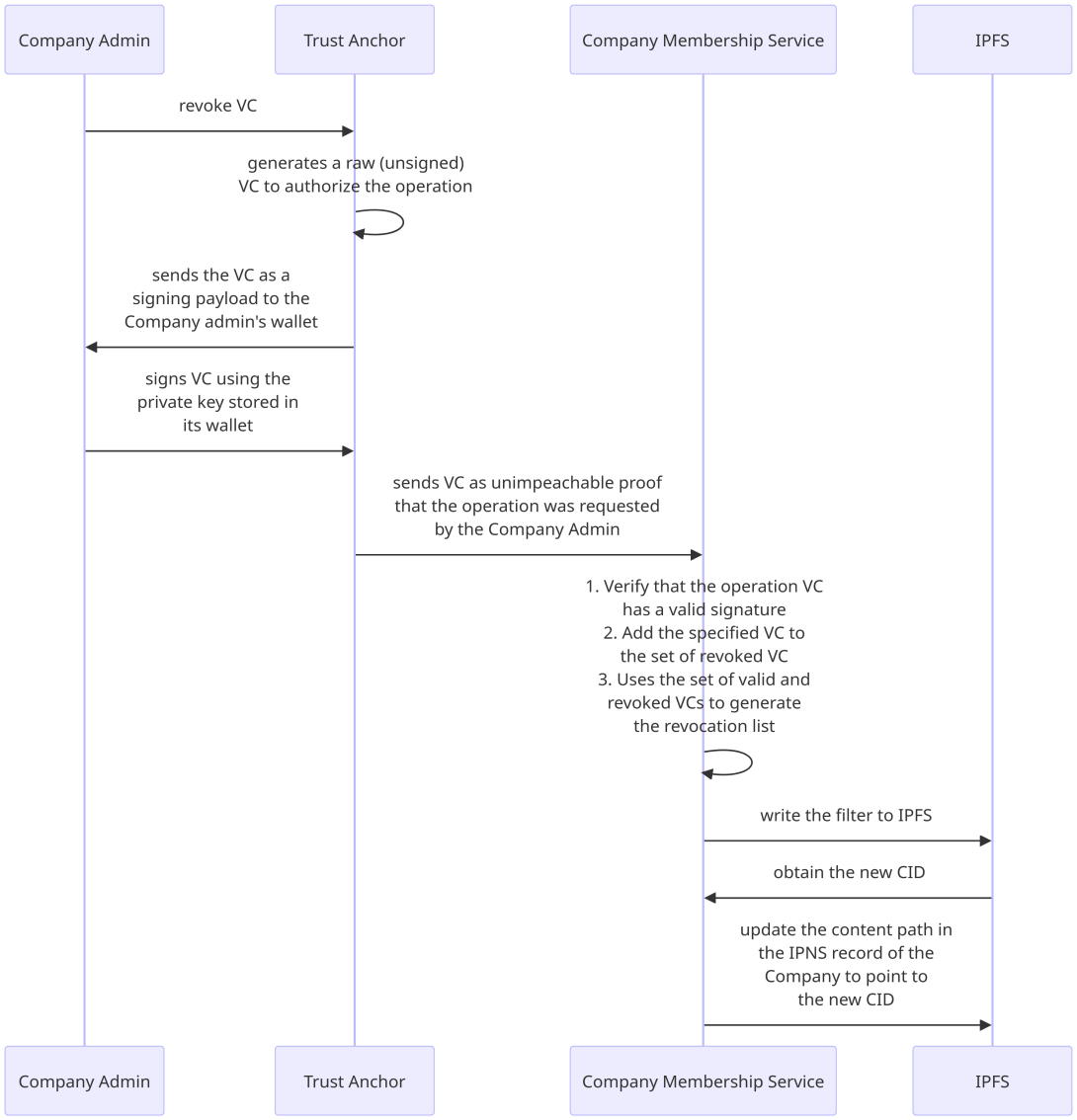


Figure 5.9: Updating the IPFS storage and the IPNS record as part of a VC revocation request.

intervals of time to check which VCs have served their suspension periods. These VCs are removed from the revoked set, the revocation list for the Company is recomputed, published to IPFS, and the content path in the Company's IPNS record is updated to include the new CID.

5.2.6 Verifying VCs

Verifying VCs is closely related to VC revocation/suspension. Currently, GX Credentials has no support for verifying VCs. The following algorithm can be used to verify the validity of a

given VC. The VC is considered valid only if all the checks in the algorithm succeed.

1. Check the validity of the Employee VC's signature using the Company's public key.
2. Check the validity of the Company VC's signature using the TA's public key.
3. Find the IPNS address for the Company's revocation list in the Company's VC.
4. Read the Company's revocation list from IPFS and check that Employee VC is not included in it.
5. Read the IPNS address for the TA's revocation list from Tezos.
6. Read the TA's revocation list from IPFS and check that Employee VC is not included in it.

6 Component Implementation

In section 5.1, we set the context for the Cascading AMQ data structure that we propose as part of this thesis to store a VC revocation list belonging to an organization that has been issuing its VCs to holders such as employees, students, citizens, etc. Furthermore, in section 5.2, we describe how this data structure fits into the broader architecture of a VC-issuing system like GX-Credentials. In this section, we discuss the implementation details of the test bench that benchmarks the Cascading AMQ data structure with incumbents such as Status List 2021 as well as the architectural extensions of GX Credentials.

6.1 Test Bench Setup for Benchmarking Cascading AMQs

6.1.1 Code Structure

The primary purpose for creating a test bench for Cascading AMQ Data Structures is performance benchmarking. We organise the code such that the underlying building block of the data structure can be switched from (say) a Bloom Filter to a new AMQ data structure that gets invented in the future. To achieve this flexibility we define the `CascadableFilter` interface and require that the different implementations we experiment with adhere to this interface.

```
class CascadableFilter {
public:
    virtual void put(int value);

    virtual bool mightContain(int value);

    virtual size_t getBitSize();

    virtual void finalise();
}
```

Listing 6.1: The Cascadable Filter Interface

We use three different AMQ data structures - Bloom Filters, Cuckoo Filters, and XOR Filters - as building blocks for the Cascading AMQ data structures. Since all these data structures are called Filters, we use the term "Cascading Filters" interchangeably with the term "Cascading AMQ data structures", in the rest of the thesis.

The benchmarking tool is implemented entirely in C++. We chose C++ because it is a low-level and efficient programming language which supports object-oriented programming. Besides this, all the data structures that we wish to compare, are available as open source libraries in C++. We use the following libraries in the test bench.

1. Bloom Filter: Arash Partow's Open Bloom Filter Library (link to source code).
2. Cuckoo Filter: An implemention by the original authors of the "Cuckoo Filter: Practically Better Than Bloom" paper (link to source code).
3. XOR Filter: FastFilter's XOR implementation (link to source code).

The logic for implementing the Cascading Filter is contained in the `CascadingFilter` class. Listing 6.2 is a code snippet from the `init` method that assembles the filters in a cascading structure, performs sanity checks to ensure that the underlying filter is correctly configured, and additionally measures performance benchmarks (not shown here) e.g., measuring the time taken for cascading filter creation, calculating the size of the cascading filter, etc.

```
class CascadingFilter {
    private:
    // Stores an ordered list of all the filters in the cascade.
    vector<CascadableFilter*> filters;

    public:
    void init(ExperimentData experimentData) {
        unordered_set<int> shouldContain =
            experimentData.getRevokedCredentials();
        unordered_set<int> shouldNotContain =
            experimentData.getValidCredentials();

        while(true) {
            // createFilter(...) creates an AMQ Data Structure of the
            // specified type e.g., Cuckoo Filter, XOR Filter,
            // Bloom Filter, etc.
            CascadableFilter* filter =
                createFilter(shouldContain.size(),
                    shouldNotContain.size());
            for (int value: shouldContain) {
                filter->put(value);
            }
            filter->finalise();

            filters.push_back(filter);
```

```
            // We perform a sanity check here.
            // findContainedCredentials(...) checks which of the
            // provided elements are contained in the provided
            // filter.
            unordered_set<int> shouldContainThatAreContained =
                findContainedCredentials(filter, shouldContain);
            if (shouldContainThatAreContained.size()
                != shouldContain.size()) {
                // The max_num_elements property in the CuckooFilter
                // being set to a very small value can cause this
                // sanity check to fail.
                fail();
            }

            unordered_set<int> shouldNotContainThatAreContained =
                findContainedCredentials(filter, shouldNotContain);

            // Once there are no false positives, the
            // algorithm terminates.
            if (shouldNotContainThatAreContained.empty()) {
                break;
            }

            shouldNotContain = shouldContain;
            shouldContain = shouldNotContainThatAreContained;
        }
    }
};
```

Listing 6.2: The Cascading Filter Implementation

6.1.2 Data Generation

For the scope of the test bench we generate and store integer values into the cascading filter. In a real world approach, we would either store VC IDs or hash entire VC values to store them in the revocation list i.e., the Cascading Filter data structure. In the data generation process, we simply create a set of integer VC IDs, put a percentage of these (e.g., 10%) in the `revokedVcSet` and the remaining in the `validVcSet`. These sets are the encapsulated in an `ExperimentData` object and passed to the `CascadingFilter` class.

6.1.3 Sanity Testing

Once we have assembled the Cascading Filter data structure and collected all the relevant performance values, we perform a final sanity check to ensure that (1) The Cascading Filter

stores all the valid VC IDs, (2) The Cascading Filter stores none of the revoked VC IDs.

```
void performSanityCheck(CascadingFilter* cascadingFilter,
    ExperimentData data) {

    for (int validCredential: data.getValidCredentials()) {
        if (cascadingFilter->isRevoked(validCredential)) {
            throw std::logic_error(Sanity check failed: filter claims
            that valid credential is revoked.);
        }
    }

    for (int revokedCredential: data.getRevokedCredentials()) {
        if (!cascadingFilter->isRevoked(revokedCredential)) {
            throw std::logic_error(Sanity check failed: filter claims
            that revoked credential is valid.);
        }
    }
}
```

Listing 6.3: Sanity Testing the Cascading Filter data structure

6.2 Architectural Extensions of GX Credentials

In section 5.2, we prescribe certain design changes to GX Credentials. These involve modifications to the existing GX Credentials codebase and implementing two new sub-repositories in it that contain the code for: (1) The Company Membership Service (CMS) which is run by Companies to store and manage the VCs that they issue, and (2) The Membership Checking Service (MCS) which implements VC verification capabilities that include the revocation mechanism proposed in this thesis. In this section, we go over the implementation details of the proposed design changes.

6.2.1 Changes to the existing GX Credentials Codebase

The existing GX Credentials codebase implements functionality for the Trust Anchor. For this reason, we use the terms "Trust Anchor" and "the existing GX Credentials codebase" interchangeably. The Trust Anchor is a Next.js app that uses a Firebase Cloud Firestore database and makes network calls to interact with the Tezos Smart Contract owned by an instance of the application. In line with the design changes proposed in 5.2, we make the following code changes in the Trust Anchor:

1. Issuance Logs: We remove the code that stores the issuance logs on the Tezos blockchain. Additionally, we also modify the smart contract itself to remove the `log_issuance` and

`revoke_issuance` endpoints.

2. Changes to the Firestore data model: We add the option for a Company to supply its CMS URL during the Company Registration Flow. The `IssuedCompanyCredentials` Firestore collection is extended to store this URL. Additionally, the `EmployeeApplications` and the `IssuedEmployeeCredentials` collections are entirely removed from the Firestore database.

3. Additional network calls: The Trust Anchor now makes network calls to the CMS in order to (1) store an employee application, (2) store an issued VC, (3) revoke/suspend an issued VC, (4) reverse a revocation/suspension, (5) fetch pending employee applications.

4. Modifying the Company VC: In addition to the claims that it already stores, the Company VC now needs to have the IPNS name for the Company. We require the Company to provide this during the Company Registration flow. Since the TA assembles and enables the VC to be signed, we modify the code for the Company VC to include this additional field.

6.2.2 Implementing the Company Membership Service (CMS)

The Company Membership Service (CMS) is run by Companies to store and manage the VCs that they issue. As a purely backend component (it has no UI component), the CMS is free to use any general purpose server-side programming language having libraries and frameworks that allow (1) building a REST API for the application, (2) interacting with a database, (3) making API calls to other services, and (4) using AMQ data structures like Bloom Filters. Java offers all these features and for these reasons, we chose to implement both the Company Membership Service (CMS) and the Membership Checking Service using Java.

We use the Spring framework's `spring-boot-starter-web` library to implement the CMS API described in 5.2.3 and the `spring-boot-starter-data-jpa` library to interact with the database. Spring's JPA (Java Persistence API) implementation is agnostic to the database we use for storing the data, allowing us to switch to a more feature-rich database (e.g., PostgreSQL) at a later point, if necessary. For ease of implementation, we use the popular in-memory H2 database, however we set it to persist-mode to ensure that the data stored in it is not ephemeral and is persisted to disk.

All the REST endpoints (including the `/listPendingApplications` endpoint) implement the POST HTTP method. This is because the calls to all of them require request payloads. Web security in the CMS is achieved directly through the contents of these request payloads. The payloads contain VCs which are expected to be signed by Company Admins and then may be forwarded to the CMS via middleware systems like the Trust Anchor. These operation-specific VCs are unimpeachable proofs that the instructions contained in them are signed by the private key in an authorised Company Admin's wallet. CMS uses the DIDKit library (`com.spruceid.didkit::didkit::0.1`) for checking the signatures of the VCs in the payload. This ensures that the contents of these payloads have not been modified by a malicious actor online and that the requests do not originate from anyone other than a Company Admin.

While several endpoints (such as the `/listPendingApplications`, `/createApplication` endpoints) simply perform CRUD operations all the operations that modify the revocation list (such as the `/revokeVc`, `/suspendVc`, `/undoRevokeVc` endpoints) also need to publish this change to IPFS. These endpoints recompute the Cascading Filter to store the updated set of revoked VCs. For this purpose, we simply replicate the algorithm introduced in listing 6.2, in Java.

6.2.3 Implementing the Membership Checking Service (MCS)

The Membership Checking Service (MCS) is meant to be operated by a Verifier wanting to check if a Verifiable Presentation made to it is valid or not. VPs can involve complex forms that involve Zero Knowledge Proofs. Those capabilities are out of scope for this implementation, however the code can simply be extended to also support those. Here we build the MCS to support the verification algorithm described in section 5.2.6.

Similar to the CMS, the Membership Checking Service needs to interact with the IPFS storage where the given Company stores its revocation list. Additionally, it needs to deserialise the contents of the storage into an object of the `CascadingFilter` type in order to check if the given VC is contained in it or not. Since the CMS also uses these codepaths, we factor them out into a stateless utility library which contains (1) an IPFS Client and (2) the Cascading Filter implementation.

7 Component Evaluation

We began with the hypothesis that these data structures should outperform incumbents such as Status List 2021 on the design goals that we delineate in section 4.1.1. In this section, we perform a quantitative and qualitative analysis to confirm this.

7.1 Comparing the Use of Different AMQ Data Structures in the Cascading Filters

During the performance benchmarking, we compared the different Cascading Filters on two parameters - (1) the time taken for constructing the Cascading Filter, and (2) its size. In the test bench we vary the size of the total VC set at a 10% revocation rate. We vary the total number of VCs from 2^{13} to 2^{23}, apply the revocation rate to create a set of valid VCs and a set of revoked VCs. These valid and revoked sets are then fed to the test bench which calculates the size of the cascading filter in kilobytes and computation time in milliseconds for the three filter types that we are evaluating here - Cuckoo Filters, Bloom Filters, XOR Filters. We summarise all the results we get from these experiment runs in table 7.1. While the compute time is not appreciably different, the sizes of the revocation lists are significantly smaller when we use XOR filters for constructing them. As a comparison with the status quo, for a total VC count of $2^{13} = 8388608$, the cascading filter created using XOR filters consumed 134 kB of space as compared to a Status List 2021 implementation which would consume 1024 kB (8388608/8/1024). Therefore, the proposed implementation achieves a 7.5x improvement over the status quo on the metric of memory and space consumption.

7.2 Revisiting the Design Goals for the Proposed Revocation Mechanism

7.2.1 Privacy

In section 4.1.3, we explain the concept of herd privacy in Status List 2021 and how supplying the entire revocation list along with the `statusListIndex` restricts the issuer to only knowing that one of the VCs stored in the given revocation list is attempting to verify itself with the verifier. We also described the potential loop hole in this approach where the issuer assigns a different revocation list to each VC it issues and therefore has a one-to-one mapping between a revocation list the VC and its holder. This problem gets further aggravated in a setup where we decide to not store the revocation lists on a blockchain and rather on a peer-to-peer storage

Total VC Count	Filter Type	Revoked VC count (10% revocation rate)	Total cascading filter size (in kB)	Computation time (in ms)
8192	Cascading Cuckoo Filter	820	0.2	4
	Cascading Bloom Filter		1.49	5
	Cascading XOR Filter		0.16	4
16384	Cascading Cuckoo Filter	1639	0.39	15
	Cascading Bloom Filter		2.97	11
	Cascading XOR Filter		0.29	9
32768	Cascading Cuckoo Filter	3277	0.78	23
	Cascading Bloom Filter		5.94	23
	Cascading XOR Filter		0.55	17
65536	Cascading Cuckoo Filter	6554	1.59	39
	Cascading Bloom Filter		11.86	34
	Cascading XOR Filter		1.08	42
131072	Cascading Cuckoo Filter	13108	3.11	64
	Cascading Bloom Filter		23.73	63
	Cascading XOR Filter		2.13	62
262144	Cascading Cuckoo Filter	26215	6.21	102
	Cascading Bloom Filter		50.64	112
	Cascading XOR Filter		4.22	127
524288	Cascading Cuckoo Filter	52429	12.4	226
	Cascading Bloom Filter		94.9	294
	Cascading XOR Filter		8.41	262
1048576	Cascading Cuckoo Filter	104858	24.82	454
	Cascading Bloom Filter		189.72	658
	Cascading XOR Filter		16.81	439
2097152	Cascading Cuckoo Filter	209716	49.64	1089
	Cascading Bloom Filter		380.35	1048
	Cascading XOR Filter		33.54	1128
4194304	Cascading Cuckoo Filter	419431	99.17	1942
	Cascading Bloom Filter		759.47	2159
	Cascading XOR Filter		67.08	2328
8388608	Cascading Cuckoo Filter	838861	198.34	5390
	Cascading Bloom Filter		1518.59	4482
	Cascading XOR Filter		134.08	4319

Table 7.1: Tabulation of total cascading filter size (in kB) and computation time (in ms) for different filter types while varying the Total VC count at a constant revocation rate of 10%

system like IPFS because the cost overheads inherent in a blockchain-based setup now no longer prevent the issuer from actually storing multiple lists on the peer-to-peer storage.

From our experiments, we conclude that for a VC set that contains > 8 million credentials and (say) 10% of these are revoked, the revocation list can fit into nearly 135 kB of storage. The small size of this revocation list eliminates the need for multiple reovcation lists per issuer. The changes made to the Company (or more generally, Issuer) VC's structure and the implementation of the CMS ensure that there is exactly one IPNS name per Company VC. This IPNS name corresponds to the single revocation list stored for the given Company on IPFS. Analogously, the verification process implemented in the MCS resovles the IPFS CID from the Company VC, downloads the revocation list, and uses it for checking the given VC's presence in it. This setup explicitly prevents the existence of multiple revocation lists and ensures that no issuer can reverse engineer the VC during a verification request as there would be a single Company VC providing a single revocation list for all the VCs issued by the Company.

7.2.2 Scalability

As mentioned in section 4.1.1, we wish to build the revocation infrastructure for planet-scale use cases that involve millions of VCs being issued and hundreds of thousands among those being revoked/suspended every day. Table 7.1 illustrates how the total cascading filter size (or the total size of the revocation list) and their computation times vary for different values of total VC count. If we take the largest VC set under experiment (Total VC Count = 8388608 VCs) as an example, the size of the revocation list here is 134 kB which is nearly 7.5 times smaller than a similar revocation list created using the Status List 2021 standard (at one bit per VC, this revocation list should be 1024 kB in size). This is a significant improvement in the storage footprint of the revocation list allowing it to be frequently downloaded from IPFS by verifiers around the world.

In order to take a real-world example, we can consider the American state of California which, at 27 million licenses in 2021 [26], had the maximum number of driving licenses issued by a state in the US. At 7% drivers with suspended licenses [27], North Dakota leads license suspension rates in the country. For ease of calculation, we assume that 10% of the licenses issued in California were revoked/suspended in 2021. Using the data in table 7.1, we can observe that the total cascading filter size grows almost linearly with the total VC count. Using a linear trend line analysis, we extrapolated this data to 27 million licenses with 2.7 million revocations and noted that the revocation list size extrapolates to 432 kB. With internet upload and download speeds of a few Mbps, an issuer updating this revocation list on IPFS and a verifier downloading it should take only a few milliseconds. A similar trend analysis on the computation time reveals that it would take nearly 14 seconds for the Californian DMV (Department of Motor Vehicles) to compute its revocation list.

7.2.3 Minimum Propagation Delay

We define the propagation delay as the time difference between a Company Admin requesting a VC to be revoked/suspended and a verifier being able to see this VC as revoked. The system's design opts to include IPFS in the Verifiable Data Registry to store the revocation list. This decision ensures that we achieve the design goal of having minimum propagation delay in the system. We avoided using centralised setups such as a revocation lists hosted by each issuer on its own centralised servers to prevent the existence of single points of failure. On the other end of the horizon, using a distributed ledger would either mean high gas fee bills or batched updates which would imply delays in the updates to the revocation list. In the proposed design, the only delays that occur in the update step are the computation time and the network propagation delay. As discussed above, the computation time is the larger among these two but still contributes only a few seconds of delay.

The above conclusions are applicable for revocations, suspensions, and reversing revocations because these actions are triggered by requests originating from issuer admins. The design proposes that the reversal of suspensions is achieved using a cron job and the frequency of this cron job decides the delays in the suspensions being reversed. This frequency is controlled by the owners of the CMS i.e., the issuer admins. The admins can increase the frequency to achieve a low propagation delay allowing a better experience for the VC holders. This would come at some additional cost of compute resources.

7.2.4 Security

The system relies heavily on public key infrastructure (PKI) to operate securely and to establish trust between trustworthy distributed systems operating on untrustworthy computer networks. The two places where PKI is used extensively are: (1) the interactions between the Company Admin wallets and the CMS (including the ones that go via the Trust Anchor), (2) the interface between the CMS and IPFS/IPNS.

All the REST endpoints implemented by the CMS expect request payloads that are signed by the private key of one of the Company Admins. All these payloads are specialised operation-specific VCs with the exception of the payload for the `/storeVc` endpoint which simply contains the VC that we wish to store. While the operation-specific VCs are unlikely to be easily found by a malicious actor, the holder's VC will likely be shared with several verifiers and those verifiers can potentially reuse this VC and interact with the `/storeVc` endpoint. A simple fix here is to make this endpoint (and all the other endpoints) idempotent - for instance, if the VC is already stored in the CMS, calling the `/storeVc` endpoint is simply a no-op. If a Company Admin's wallet and therefore, private key is leaked for some reason, we simply remove its corresponding public key from the CMS, reissue a private key to the admin and store its corresponding key in CMS instead.

As described in section 2.4, the IPNS infrastructure ensures that a given IPNS record can only be updated by its owner i.e., the Company Admins who first commissioned the IPNS record. The private-public key pair that secures this interface is different from the key pairs that are used to the sign the VCs that the issuers issue or the operation-specific VCs that they

create for interacting with the CMS. The key pair for the interface with IPNS can also get compromised or the Company Admins may decide to perform key rotation as a security best practice. It is for this reason that we decided to store the IPNS name for the issuer in the issuer VC (and not in each of the individual holder VCs). This allows the admins to rotate keys, commission a new IPNS record, store its IPNS name in its issuer VC, and then make this issuer VC available to any verifier looking to obtain the issuer's revocation list, via the Verifiable Data Registry.

List of Figures

List of Tables

Bibliography

[1] *Estonia e-Identity*. URL: https://e-estonia.com/solutions/e-identity/id-card/ (visited on 12/12/2023).

[2] *India Digilocker*. URL: https://www.digilocker.gov.in/dashboard/states (visited on 12/12/2023).

[3] *Gaia-X secure and trustworthy ecosystems with Self Sovereign Identity*. URL: https://gaia-x.eu/wp-content/uploads/2022/06/SSI_White_Paper_Design_Final_EN.pdf (visited on 12/12/2023).

[4] M. e. a. Sporny. *Decentralized Identifiers (DIDs) v1.0*. 2022. URL: https://www.w3.org/TR/did-core/ (visited on 12/12/2023).

[5] F. Boudot. "Efficient Proofs that a Committed Number Lies in an Interval". In: May 2000, pp. 431–444. ISBN: 978-3-540-67517-4. DOI: 10.1007/3-540-45539-6_31.

[6] N. S. et al. *OpenID Connect Core 1.0 incorporating errata set 1*. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 12/12/2023).

[7] F. Schardong and R. Custódio. "Self-sovereign identity: a systematic review, mapping and taxonomy". In: *Sensors* 22.15 (2022), p. 5641.

[8] C. Allen. *The Path to Self-Sovereign Identity*. 2016. URL: https://www.lifewithalacrity.com/article/the-path-to-self-soverereign-identity/ (visited on 12/12/2023).

[9] C. Allen. *Self-Sovereign Identity Principles*. 2016. URL: https://github.com/ChristopherA/self-sovereign-identity/blob/master/self-sovereign-identity-principles.md (visited on 12/12/2023).

[10] C. Garman, M. Green, and I. Miers. "Decentralized anonymous credentials". In: *Cryptology ePrint Archive* (2013).

[11] M. e. a. Sporny. *Verifiable Credentials Data Model v2.0*. 2023. URL: https://www.w3.org/TR/vc-data-model-2.0/#what-is-a-verifiable-credential (visited on 12/12/2023).

[12] I. Herman. *Decentralized Identifiers (DIDs)*. 2020. URL: https://iherman.github.io/did-talks/talks/2020-Fintech/#/10 (visited on 12/12/2023).

[13] M. e. a. Sporny. *Bitstring Status List v1.0*. 2023. URL: https://www.w3.org/TR/did-core/ (visited on 12/12/2023).

[14] J. Benet. *IPFS - Content Addressed, Versioned, P2P File System*. 2014. arXiv: 1407.3561 [cs.NI].

[15] *InterPlanetary Name System (IPNS)*. URL: https://docs.ipfs.tech/concepts/ipns/ (visited on 12/12/2023).

[16] *Michelson*. 2018. URL: https://www.michelson.org/ (visited on 12/12/2023).

[17] *Proof of Stake on Tezos*. 2018. URL: https://tezos.com/proof-of-stake/ (visited on 12/12/2023).

[18] B. H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[19] P. Cao. *Bloom Filters - the math*. 1998. URL: https://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html (visited on 12/12/2023).

[20] E. Szabo-Wexler. "Approximate Membership of Sets: A Survey". In: ().

[21] Y. Lu, B. Prabhakar, and F. Bonomi. "Bloom filters: Design innovations and novel applications". In: *43rd Annual Allerton Conference*. 2005.

[22] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. "Cuckoo filter: Practically better than bloom". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014, pp. 75–88.

[23] T. M. Graf and D. Lemire. "Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters". In: *ACM J. Exp. Algorithmics* 25 (Mar. 2020). ISSN: 1084-6654. DOI: 10.1145/3376122. URL: https://doi.org/10.1145/3376122.

[24] J. Larisch, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. "CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 539–556. DOI: 10.1109/SP.2017.17.

[25] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables". In: vol. 15. Jan. 2004, pp. 30–39. DOI: 10.1145/982792.982797.

[26] *Total number of licensed drivers in the United States in 2021, by state*. URL: https://www.statista.com/statistics/198029/total-number-of-us-licensed-drivers-by-state/ (visited on 12/12/2023).

[27] *States with the Most License Suspensions*. URL: https://insurify.com/insights/states-with-the-most-license-suspensions/ (visited on 12/12/2023).