DS-06-2017: Cybersecurity PPP: Cryptography

PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D4.3 – Final Report on Architecture

Due date of deliverable: 31 December 2020
Actual submission date: 31 December 2020

Grant agreement number: 780477
Start date of project: 1 January 2018
Revision 1.0

Lead contractor: IBM
Duration: 42 months

Project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020	
Dissemination Level	
PU = Public, fully open	X
CO = Confidential, restricted under conditions set out in the Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC	

D4.3

Final Report on Architecture

Editor
Marko Vukolić (IBM)

Contributors
Berry Schoenmakers (TUE)
Toon Segers (TUE)
Sergei Kuštšenko (SCCEIV)
Sven Heiberg (SCCEIV)
Nikos Karagiannidis (IOResearch)
Damian Nadales (IOResearch)
Reviewers
Nikos Karagiannidis (IOHK)

31 December 2020
Revision 1.0

The work described in this document has been conducted within the project PRIViLEDGE, started in January 2018. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477.

Executive Summary

This document presents the final high-level architecture of PRIViLEDGE use-cases and toolkits. The scope of the document is restricted to refining the architecture designs presented in earlier WP4 deliverables, notably D4.1 and D4.2. Use-cases and toolkits whose architecture had not been significantly changed compared to the earlier D4.x deliverables, are therefore omitted from this deliverable.

In summary, D4.3. presents the final architecture of two PRIViLEDGE use cases: verifiable online voting (Chapter 2) and software updates for Cardano's stake based-ledger (Chapter 3). Moreover, it presents the final architecture and the goals of PRIViLEDGE's horizontally cross-cutting toolkits that provide functionality that could be used in different situations and across different blockchains and distributed ledger networks. These include those that were modified after D4.1 and D4.2, notably: the architecture of the flexible consensus initially aiming Hyperledger Fabric (Chapter 4) and the toolkit for secure two-party and multi-party computations on ledgers (Chapter 5).

Contents

Chapter 1

Introduction

This deliverable covers the architecture of toolkits and prototypes for secure ledger infrastructure as well as of privacy-preserving ledger applications, such as authentication within the infrastructure, solutions for updating the ledger protocols, consensus protocols, or online voting. Indeed, one of the focus topic of this document is on proposing an architecture of secure ledger systems. Ledger systems consist of several different components, such as a consensus mechanism that determines the order of transactions, a ledger (policy and format) mechanism that decides which transactions are included in the ledger, or the transaction protocol itself, which specifies which messages comprise valid transactions and how they affect the *world state*, i.e., the state of the distributed database implemented by the ledger. We describe a generic architecture of a ledger system and show how a candidate instantiation for the use case for updating the ledger protocols impacts the proposed architecture. We propose a similar description regarding the toolkits for flexible consensus and authentication in Hyperledger Fabric, and we also consider the security of ledger systems in post-quantum scenarios.

This deliverable describes the architecture for PRIViLEDGE use cases and toolkits which were substantially updated since Deliverables D4.1 ("First Report on Architecture of Secure Ledger Systems") and D4.2 ("Report on Architecture for Privacy Preserving Applications on Ledgers"). The document is divided in four parts.

In Chapter 2 (UC1: Verifiable Online Voting with Ledgers), we present the architecture for the PRIViLEDGE use case which involves using ledgers to improve privacy guarantees in online voting. The proposed solution makes it possible to prove to the independent auditor in a voter privacy preserving manner that all accepted votes were stored, sent to the tabulation according to the election rules, and decrypted/tabulated correctly. The architecture of this use case was initially presented in Deliverable D4.2 — this deliverable presents the final version.

In Chapter 3 (UC4: Decentralized Software Updates for Cardano Stake-Based Ledger) we give the logical architecture of an update mechanism for proof-of-stake ledgers, with the focus on the Cardano blockchain. An update system allows the parties that run the ledger system to converge toward an improved version of the ledger (e.g., a more secure or faster ledger) and to facilitate the transition from the old to the new ledger. Traditionally, such software updates have been handled in an ad-hoc, centralized manner. Somebody, often a trusted authority, or the original author of the software, provides a new version of the software, and users download and install it from that authority's website. However, this approach is clearly not decentralized, and hence jeopardizes the decentralized nature of the whole system: In a decentralized software update mechanism, proposed updates can be submitted by anyone (just like anyone can potentially create a transaction in blockchain). The decision of which update proposal will be applied and which won't, is taken collectively by the community and not centrally. Therefore we propose a standard architecture for an update mechanism for proof-of-stake ledger, and we show how this architecture could be implemented for Cardano. We recall that even if the aim of this part of the document is to describe an update mechanism for the Cardano ledger, the description proposed here could be used as a blueprint for a generic proof-of-stake ledger system. The architecture of this use case was initially presented in Deliverable D4.1 — this deliverable presents the final version.

As the architectures of PRIViLEDGE use cases UC2: Distributed Ledger for Health Insurance, and UC3:

University Diploma Record Ledger have not changed significantly compared to designs presented in Deliverable D4.2, these use cases are not the topic of this document.

Moving from PRIViLEDGE use cases to toolkits, in Chapter 4 we present the updated architecture of flexible consensus toolkit for Hyperledger Fabric, where the initial architecture was presented in Deliverable D4.1. Notably, our revised architecture includes support for flexible number of leaders (i.e., multiple leaders) in leader-based Byzantine fault-tolerant consensus protocols. This chapter also discusses the parallel networking and processing capabilities that such a flexible toolkit needs to support. Finally, for completeness, we present the updates in the Hyperledger Fabric code base which pertain to consensus.

Finally, Chapter 5 provides technologies that combine blockchain with 2-party and Multi-Party Computation (2PC/MPC). As such it is a final version of the architecture of this toolkit presented in Deliverable D4.2. In a nutshell, the novel features of the toolkit consist of the following. The starting point is the existence of various libraries for secure two/multi-party computation in the traditional setting, where players are connected through point-to-point connections using TCP/IP. This setting is problematic when players would like to run a computation publicly, so that everyone can observe who did what and when. Moreover this traditional setting requires coordination among players that are supposed to be online all the time during the computation knowing each other IP addresses, which may be hard to realize when the number of players is large and heterogeneous and potentially leaks IP addresses to other players. The toolkit aims at adding the support of distributed ledger technology in order to mitigate the above problems, therefore allowing the use of traditional secure two/multi-party computation libraries offering public verifiability of the played messages and allowing players to participate to the computation whenever convenient for them (i.e., without requiring players to be all simultaneously online communicating with known IP addresses).

The remaining PRIViLEDGE toolkits, not included in this document, retain their architectures as presented in Deliverable D4.1 (Toolkit for post-quantum secure protocols in distributed ledgers) and Deliverable D4.2 (Toolkit for Privacy-Preserving Data Storage on Ledgers and Toolkit for Zero-Knowledge Proofs in Ledgers).

Chapter 2

UC1: Verifiable Online Voting with Ledgers

2.1 Introduction

This chapter presents Tiviledge, a system whose objective is to provide secure, usable and transparent online voting by using a protocol that makes it possible to prove to the independent auditor in a voter privacy preserving manner that all accepted votes were stored, sent to the tabulation according to the election rules, and decrypted/tabulated correctly.

Tiviledge uses the notion of the data-audit to ensure that the published voting result corresponds to encrypted preferences sent by eligible voters to the digital ballot box and the bulletin board. We mitigate the need to prove correctness of the software and its operation by demonstrating that according to the public protocol, the correct election outcome was calculated based on the given public inputs. Moreover, we emphasize the importance of long-term voter privacy over the long-term integrity of the election result.

The goal of this chapter is to provide an overview of the technical architecture of the Tiviledge, as implemented. In Figure 2.1 we can see the Tiviledge system based on the HyperLedger Fabric. The components depicted in Figure 2.1 jointly implement the Tiviledge cryptographic protocol which is based on verifiable homomorphic aggregation of rerandomized encrypted votes created with a commitment-consistent encryption (CCE) scheme. The protocol allows to publish rerandomized commitments to the public ledger, providing third-party auditability and receipt-freeness.

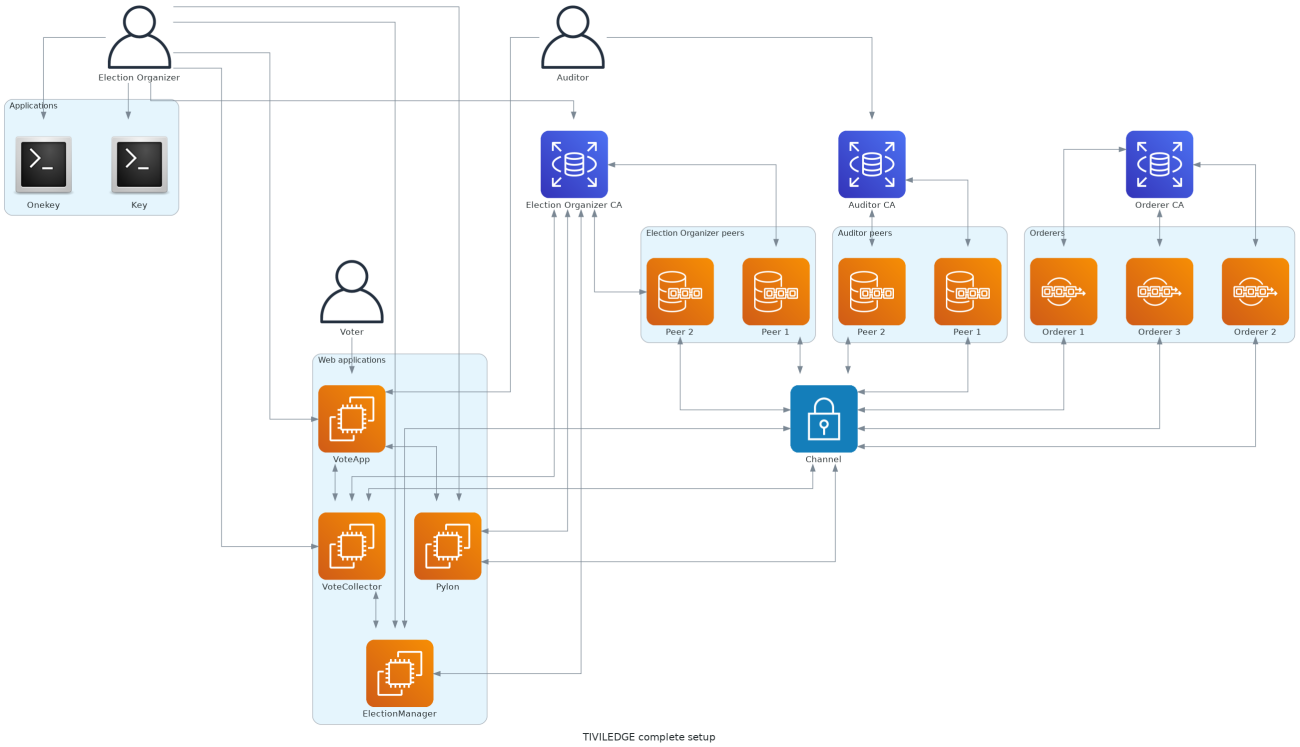

Figure 2.1: Tiviledge implemented with HyperLedger Fabric.

The CCE scheme gives a possibility to create end-to-end verifiable elections. In order to achieve that, both the Voter and Election Organizer publish some parts of election processes to the bulletin board.

Voter public part on bulletin board consists of:

– voter certificate;

– commitments for each candidate;

– commitments signature;

– proofs that commitments are encryption of 0 or 1;

– proof that commitments are sum of encrypted values which sum up to 1.

Election Organizer public part on bulletin board consists of [Avi19]:

– election configuration;

– election results;

– published commitments aggregation;

– results openings;

– voter and revocation lists.

Bulletin board gives a possibility to anyone make different verifications [Avi19; Hei+]:

– **Individual verification** means that voter can verify that vote was:

  - **Cast as intended**: ballot represents a vote for the candidate whom he or she intended to give the vote.
  - **Recorded as cast**: ballot is recorded as he or she cast it.

- **Universal verification** means that anyone can verify that:
  - Every vote was cast by an eligible voter.
  - Tally process is done correctly by verifying the results.

The commitments are rerandomized by the Election Organizer before they are published to the bulletin board. Rerandomization means that we change commitments ciphertext without changing the underlying plaintext ballot. Reason for rerandomization is to achieve receipt-freeness in this scheme. This removes the possibility of vote-buying or coercion because after these commitments are rerandomized voter can not show to coercer or someone else how he or she voted.

2.2 Implementation

This section describes external dependencies for the Tiviledge prototype. Additionally it provides overview of Tiviledge services, applications and libraries.

2.2.1 Technology stack

Go

Tiviledge backend services and in particular the chaincode running in the HyperLedger Fabric infrastructure have been written in the Go programming language.

Go language is used mainly in server-side applications because its ideology is created around packages and services. It gives a possibility to split one big application into multiple microservices because of simple concurrency. Tiviledge uses both Go and its standard library extensively.

Angular

Tiviledge web-application for voters and election manager have been written in TypeScript, using the well-known Angular framework for single-page application development.

MIRACL Core Cryptographic Library

Tiviledge uses MIRACL Core Cryptographic Library for CCE encryption implementation, rerandomization and private key generation. BLS12461 and NIST521 curves are used.

This library is an extended and re-released version of Apache Milagro Cryptographic Library (AMCL) which supports elliptic curve and pairing-friendly curve cryptography as well RSA, AES symmetric encryption and hash functions [MIR]. It has implementations in different languages including the ones that are necessary for this project - Go and JavaScript.

Docker

Docker is a tool which gives a possibility to create, deploy and run applications by using containers. These containers allow to package an application with all necessary sub-components (libraries, tools, dependencies files). In some way, Docker is like a virtual machine but it does not create a new operating system, it uses a host Linux kernel, which gives higher performance than virtual machine [ope].

HyperLedger Fabric

Tiviledge bulletin board is built on top of the HyperLedger Fabric – an open-source permissioned distributed ledger technology platform, designed to provide trust between different organizations. Because the platform is permissioned it means, that all participants inside the network know each other and agree on who, how and what can do inside the ledger [Hypa].

Figure 2.2 shows the general structure of HyperLedger Fabric:

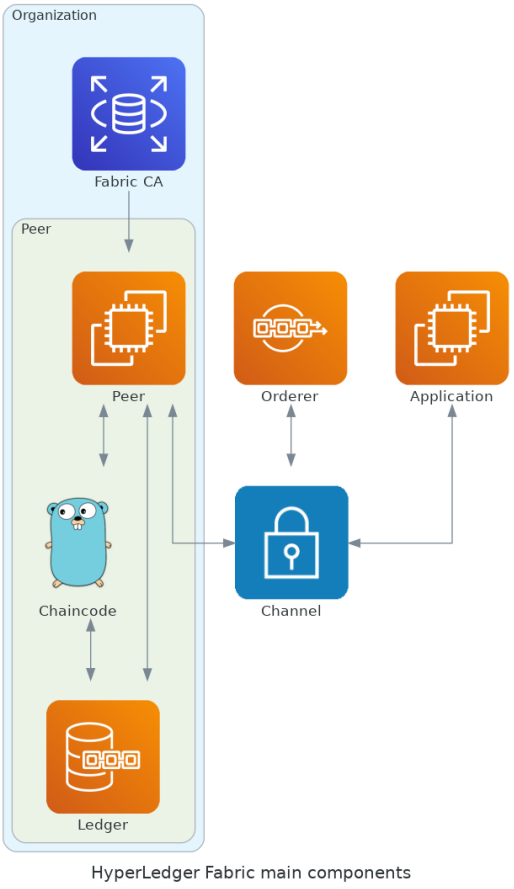

Figure 2.2: HyperLedger Fabric main components.

where [Avi19; Hypa]:

- **Organization** is a member of the blockchain network. An organization in Hyperledger Fabric forms a trust domain (i.e., peers within an organization trust each other).
- **Application** is a client that uses Fabric SDK to submit transactions to peers and transaction proposals to orderers, also it queries ledger state.
- **Peer** is a node that hosts ledgers and chaincode. The number of peers inside the organization is not limited only to one, it can be as many as the organization wants. Running the organization with only one peer is not a good choice, because if this peer not functioning correctly, then the whole organization is not functional.
- **Ledger** consists of two parts:
  - **World state** is a database that holds the current state of set of ledger values.

  - **Blockchain** is a transaction log that contains all records and changes that resulted in the world state. Blockchain gives immutability to history, means that once something was changed in the ledger, which resulted in the current world state, it is impossible to somehow remove or delete log about this modification.

- **Chaincode** or smart contract is a specific code invoked by a client application which manages access and modifications to a set of key-value pairs in the world state.
- **Orderer** is a node that creates transaction blocks for ledger modification and maintains the list of organizations that are allowed to create channels.
- **Channel** is a private ”subnet” for communication between different organizations, peers, orderers and applications. Each channel holds own ledger and all channel members can see all transactions that are made in this channel.
- **Fabric CA** is an optional Certificate Authority for HyperLedger Fabric. Which is responsible for identity creation with different roles like peer, orderer, client etc.

Fabric uses new architecture for transactions that is called execute-order-validate [Ril]:

- **Execute:** Transaction is executed on peer using chaincode, which returns *Read* and *Write* sets, where *Read* set consists of key-value pairs before the modification and *Write* set consists of key-value pairs after the modification.
- **Order:** When enough peers (the amount of peers that must execute the chaincode is defined in the endorsement policy on chaincode instantiation) agree on execution, meaning they have similar results (*Read* and *Write* sets) transaction is being ordered.
- **Validate:** Each peer validates transaction before applying it to the ledger.

This architecture has different benefits, but the most importantly it eliminates any non-determinism because all logic (chaincode execution) is done before ordering. This means that general purpose programming languages can be used for chaincode development.

More detailed transaction flow can be seen in Figure 2.3.

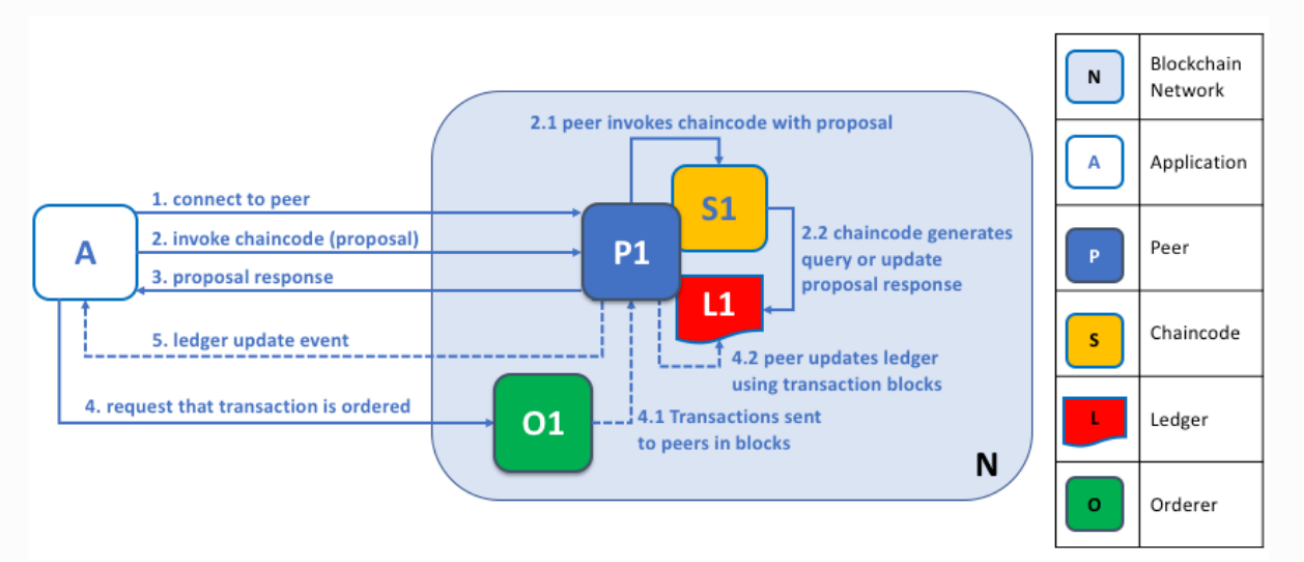

Figure 2.3: Transaction process diagram [Hypb].

2.2.2 System Architecture

The Tiviledge is split into different parts where each part is responsible for specific actions. Figure 2.4 provides general structure with actors.

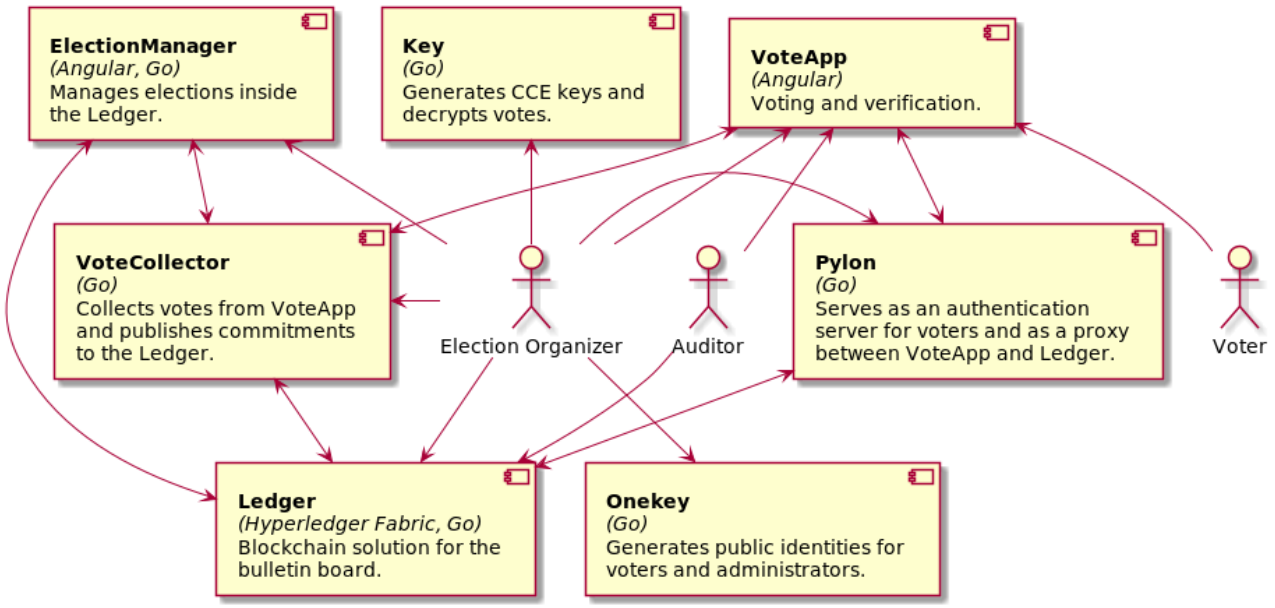

Figure 2.4: System general structure.

Next subsections will give more information about different parts of the system, they are divided into web applications, remote services, offline applications and common packages. By web applications we mean applications run on users local machine (computer, mobile, tablet) and browser. Remote services mean that logic is run on a remote server and available over an API. A very specific kind of remote service is chaincode which runs in the HyperLedger Fabric framework.

Offline applications are command-line applications used by election organizers to generate election key material (both for encryption and eligibility verification). These applications signify the necessity for proper key- and credential management in order to implement secure online voting, however the focal point of the Tiviledge is not in the key-management as we feel that e.g., threshold key management on smart-cards without a trusted dealer would be sufficient to fulfill the goals for Tiviledge.

Overview of interactions between different components is given on sequence diagrams.

Finally, common packages encapsulate data structures and algorithms used by multiple Tiviledge components.

Web applications

Voter application Voter application or **VoteApp** gives a possibility to cast vote and verify election processes, where main possibilities for verification are:

- election configuration verification;
- voter certificate verification;
- voter signature verification;
- verification of Zero-Knowledge proofs of published commitments;

– election results verification.

VoteApp is a single-page application which gives the possibility for a voter to see how the vote casting and verification logic is executed. VoteApp interacts with rest of the infrastructure through a vote collector service VoteCollector and a proxy-service called Pylon.

Election manager application Election manager (**ElectionManager**) is an application which is responsible for election management. ElectionManager consists of two applications:

– client - single-page application for Election Organizer interaction;

– server - which is a proxy between ElectionManager client application and Ledger.

The main responsibilities of ElectionManager service are:

– administrators addition, deletion;

– election addition, deletion and on demand ending;

– voter list publication;

– revocation list publication, deletion;

– votes aggregation initialization on the VoteCollector service and aggregation result retrieval;

– published commitments aggregation and resulted aggregation publication;

– tally results publication.

ElectionManager server REST API endpoints are:

Table 2.1: ElectionManager server endpoints.

Endpoint	**Description**	**Method**	**Input**	**Positive result**
/api/token	Used for a token creation takes as an input command with arguments, which returns a tokenID to be signed by the client for further signature verification.	POST	Command with arguments.	Token ID that needs to be signed by the client.
/api/login/admin	Used for election administrator authentication	POST	Administrator ID with signed token.	Administrator certificate is sent to the client.
/api/admins	Queries list of administrators.	GET	Maximum number of administrators to receive (limit).	List of administrators.
/api/admins/filter	Queries list of administrators and filtering them by ID.	GET	Maximum number of administrators to receive (limit) and filter.	Filtered list of administrators.
/api/admins/revoke	Revokes administrator active certificate.	POST	Signed action from an active administrator.	Success message.

Continued on next page.

Table2.1– *Continued from previous page.*

Endpoint	**Description**	**Method**	**Input**	**Positive result**
/api/admins/register	Registers new administrator or updates existing administrator active certificate.	POST	Action with new admin certificate signed by an active administrator.	Success message.
/api/election/add	Adds new election to the Ledger.	POST	Signed election.	Success message.
/api/election/delete	Removes election from Ledger.	POST	Signed action command by the election administrator.	Success message.
/api/election/delete-data	Removes targeted election data from the ledger (voter/revocation lists, eligible/ineligible/revoked voters, published commitments).	POST	Election ID and targeted data.	Success message.
/api/elections	Queries all elections from Ledger.	GET		List of elections inside the Ledger.
/api/election	Queries particular election from Ledger.	GET	Election ID.	Returns election.
/api/election/end	Deliberatly ends election.	POST	Signed election with changed end time.	Success message.
/api/election/exist	Checks if election exists.	GET	Election ID.	Status response.
/api/results	Gets tally results for a particular election from Ledger.	GET	Election ID.	Election tally result.
/api/results/publish	Publishes tally results to the Ledger.	POST	Signed tally results.	Success message.
/api/voter-list/add	Adds voter list to the Ledger.	POST	Signed voter list.	Success message.
/api/voter-lists	Queries voter lists for a particular election.	GET	Election ID.	List of voter lists.
/api/revocation-list/add	Adds revocation list to the Ledger.	POST	Signed revocation list.	Success message.
/api/revocation-list/delete	Deletes revocation list from the Ledger.	POST	Signed action by the administrator.	Success message.
/api/revocation-lists	Queries revocation lists for a particular election.	GET	Election ID.	List of revocation lists.
/api/commitments/ aggregation/init	Sends an initialization command to the Ledger.	POST	Signed initialization command.	Success message.
/api/commitments/ aggregation/status	Queries commitments aggregation status.	GET	Election ID.	Returns commitments aggregation status.
/api/commitments/ aggregation/delete	Deletes commitments aggregation status from Ledger.	POST	Signed action command.	Success message.

Services

Vote collector service Vote collector service or **VoteCollector** serves as a REST API. It holds the following responsibilities:

- communication between VoteApp, ElectionManager and Ledger network;
- vote casting (ballot formation with VoteApp);
- vote parts publication to a bulletin board inside the Ledger;
- votes aggregation;
- votes local storage.

After votes are cast they are saved locally in VoteCollector internal datastore (currently a file-based storage).

Table 2.2: VoteCollector endpoints.

Endpoint	Description	Method	Input	Positive result
/api/cast-message	Initial message in voting process.	POST	Receives cast message with encrypted vote and initial commitments.	Challenge response with rerandomization proofs.
/api/final-message	Final message in voting process.	POST	Receives signed voter commitment with different Zero-Knowledge proofs.	Success message.
/api/token	Used for a token creation. Takes as an input command with arguments, which returns a tokenID to be signed by the client for further signature verification.	POST	Command with arguments.	Token ID that needs to be signed by the client.
/api/votes/aggregation/init	Used as an initialization command for votes aggregation.	POST	Administrator ID with signed token.	Administrator certificate is sent to the client.
/api/votes/aggregations	Queries finished vote aggregation results.	POST	Election ID with signed token to the related command.	List of finished vote aggregation results.

Pylon Pylon service serves as a REST API proxy. It holds following responsibilities:

- voter authentication;
- bulletin board data representation, which includes:
  - elections;
  - published commitments (vote public parts);
  - revoked voters;
  - published commitments aggregation result;
  - tally results.

Table 2.3: Pylon endpoints.

Endpoint	Description	Method	Input	Positive result
/api/token	Used for a token creation. Takes as an input command with arguments, which returns a tokenID to be signed by the client for further signature verification.	POST	Command with arguments.	Token ID that needs to be signed by the client.
/api/login/voter	Used for voter authentication.	POST	Voter ID with signed token.	Returns voter certificate with available elections to a particular voter.
/api/elections	Queries all elections.	GET	Election ID.	List of all elections inside the Ledger.
/api/public-commitments	Queries public commitments	GET	Election ID, limit and a bookmark. Limit sets maximum number of commitments to retrieve and bookmark can be used to perform paginated queries.	Paginated query of public commitments.
/api/public-commitments/count	Queries public commitments count inside the Ledger.	GET	Election ID.	Number of public commitments.
/api/public-commitments/filter	Queries public commitments and filtering them by voter ID.	GET	Election ID, limit, filter and bookmark.	Paginated query of public commitments.
/api/commitments/aggregation/status	Queries public commitments aggregation status.	GET	Election ID.	Public commitments aggregation status.
/api/revoked-voter	Performs verification if voter is revoked or not.	GET	Election ID, voter ID.	Voter revocation status.
/api/revoked-voters	Queries revoked voters.	GET	Election ID, limit, filter and bookmark.	Paginated query of revoked voters.
/api/election-result	Queries election tally result from Ledger.	GET	Election ID.	Election tally result.

Blockchain network Blockchain network (**Ledger**) serves as a bulletin board where public records are being published. The publicly available data consists of:

- administrators list;
- election configuration;
- voter list;
- revocation list;
- published commitments (vote public part);
- election results.

Network topology can be seen in Figure 2.5.

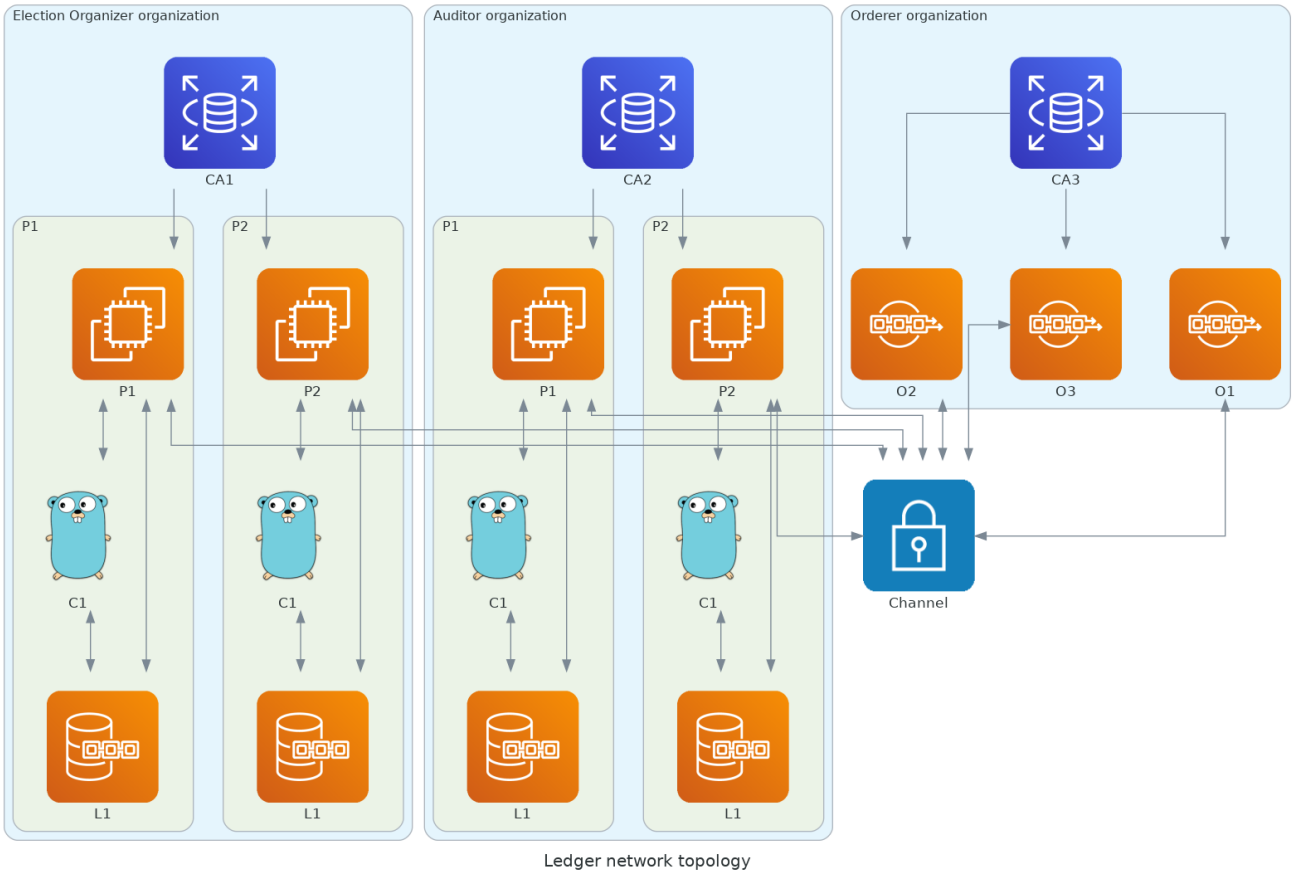

Figure 2.5: Blockchain network topology.

Figure 2.5 shows that both organizations have almost the same structure. This means that they have the same chaincode (business logic) C1 and ledger state L1. The difference comes from the fact that each organization controls it's own Fabric Certificate Authority which is shown as CA1 and CA2 for identity creation. Election Organizer uses CA1 to create an identity for peers and services, to give possibility for them to interact with Ledger. Auditor organization uses CA2 to create identities for its peers.

Both organizations have two peers (shown as P1 and P2 in Figure 2.5) for additional redundancy, which means that if one of two peers goes down inside an organization, the organization will still be functional.

Additionally, the network has a *MAJORITY* (greater than half) rule. This means that in our case both organizations should accept this action to add or modify the ledger, but this rule can be changed. For example, it can be configured in such a way that only Election Organizer or Auditor organization should accept these actions or set that all organizations should accept it.

Orderer organization can be controlled by a third party, but in our case it is controlled by Election Organizer. Orderers (O1, O2, O3) that are shown in Figure 2.5 are running in Raft [1] protocol which gives the possibility for making the transaction even if one of the orderers goes down.

The chaincode (C1) that is executed on peers (P1, P2) contains different functions.

Chaincode instantiation:

– `Init(publicAdminList)` – is executed on chaincode first instantiation and on chaincode upgrade. In case of initial instantiation it takes as an argument public administrators list which will be stored on the

[1] `https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html`

bulletin board for services if argument is not provided chaincode will not be instantiated.

Election administrators management:

- `test()` – is used as a test query on services in order to check if they can query ledger state;
- `registerElectionAdmin(manageAdminRequest)` – registers a new administrator or updates certificate of an existing administrator;
- `getElectionAdmins()` – queries list of administrators;
- `getElectionAdminsChecksum()` – queries administrator list checksum;
- `deleteElectionAdmin(manageAdminRequest)` – marks administrator certificate as inactive.

Election management:

- `newElection(election)` – adds new election to the ledger;
- `getElection(electionID)` – queries election with specific ID;
- `getAllElections()` – queries all elections from ledger;
- `getActiveElections()` – queries only active elections;
- `endElection(election)` – end election deliberately;
- `deleteElection(action)` – deletes election configuration.

Voter lists management:

- `addVoterList(electionID, voterList)` – adds new voter list to the ledger;
- `getSavedVoterListsChecksum(electionID)` – queries stored voter lists checksum;
- `getSavedVoterLists(electionID, limit, bookmark)` – queries stored voter lists;
- `getEligibleVoters(electionID, limit, bookmark)` – queries eligible voters;
- `getIneligibleVoters(electionID, limit, bookmark)` – queries eligible voters;
- `verifyEligibility(electionID, voter)` – checks if voter is eligible;
- `deleteVoterLists(electionID)` – deletes all voter lists;
- `deleteEligibleVoters(electionID)` – deletes all eligible voters;
- `deleteIneligibleVoters(electionID)` – deletes all ineligible voters.

Revocation lists management:

- `addRevocationList(electionID, revocationList)` – adds new revocation list;
- `getRevocationListsChecksum(electionID)` – queries stored revocation lists checksum;
- `getSavedRevocationLists(electionID, limit, bookmark)` – queries stored revocation lists;
- `getRevokedVoters(electionID, limit, bookmark)` – queries revoked voters list;

- `getRevokedVotersWithFilter(electionID, limit, filter, bookmark)` – queries revoked voters and filters them with matching voter ID and revocation reason;
- `deleteRevocationList(action)` – deletes a specific revocation list;
- `deleteRevocationLists(electionID)` – deletes all revocation lists;
- `deleteRevokedVoters(electionID)` – deletes all revoked voters.

Published commitments management:

- `addPublicCommitment(electionID, publicCommitment)` – adds voter public commitment;
- `getPublicCommitmentsCount(electionID)` – queries public commitments count;
- `getPublicCommitmentsFilterByVoterID(electionID, limit, filter bookmark)` – queries public voter published commitments by filtering them with matching voter ID;
- `deletePublicCommitments(electionID)` – deletes public commitments.

Published commitments aggregation management:

- `updatePublicCommitmentsAggregation(aggregation)` – publishes update for published commitments aggregation status;
- `getAggregationStatus(electionID)` – queries public commitments aggregation status;
- `deleteAggregationStatus(electionID)` – deletes public commitments aggregation status;
- `deleteAggregationStatusAsAction(action)` – deletes public commitments aggregation status as an administrator.

Offline applications

Key application A Key application is responsible for CCE keys generation and votes aggregation/decryption from VoteCollector.

For CCE key pair generation it takes as an argument election ID, where election ID defines in what datastore generated keys will be saved.

To get results for election Key application can be used in two ways:

1. Aggregate and decrypt votes, where Key application takes as an argument election configuration, CCE key pair, votes, voter list and revocation list as an option;
2. Decrypt provided aggregation, where Key application takes as an argument election configuration, CCE key pair and votes aggregation.

In case Key application used for votes aggregation, the aggregation process follows as:

1. Reconstruct voter lists to see at what time which voter was marked as eligible or ineligible;
2. Find latest vote for each voter and validate it;
3. Filter votes against voter list, where Key application checks that at the time of vote storage voter was eligible;
4. Apply revocation list (optional).

The last phase is the same with both ways, where Key application performs decryption and openings extraction from aggregation for each candidate and the result is saved to datastore.

Onekey application Onekey application is used for creating pseudonymized identites for voters and administrators which are used in the Tiviledge system. It generates a public identity with an ECDSA private key and certificate for a real person.

The private key is generated by a process called *key stretching*, where **secret** (password) is being hashed through a specific key derivation function. In our case we use **PBKDF2**[2] (Password-Based Key Derivation Function 2), which takes as an input:

- public identity ID (used as a salt);
- secret;
- key length;
- iterations count.

and results in a key that can be used in an ECDSA scheme.

The secret itself is a random string in format like: "XKJH-AWAA-SKJT" which is later given to the voter/administrator in order to authenticate into the system.

The process of identities generation follows as:

1. Generate root CA which results in a root private key and a self-signed certificate.
2. For each voter/administrator:
   (a) generate random ID (`rID`);
   (b) generate secret (password) (`secret`);
   (c) create private key `pbkdf2(secret, rID);`
   (d) extract public key from private key;
   (e) create certificate for voter/administrator with extracted public key;
   (f) sign certificate with root CA.

Later certificate is published to the Ledger and redistributed from there to different services for verification purposes and **secret** with voter/administrator ID is given to a real person.

System interactions

This subsection will tell more about how different components interact with each other, but we describe only the most important actions related to this use case like:

- election administrators addition;
- administrators authentication;
- election addition;
- voter list addition;
- voter authentication;
- voting;
- votes aggregation;

[2] `https://www.ietf.org/rfc/rfc2898.txt`

– published commitments aggregation;

– tally results publication;

– public tally results verification.

Interactions will be described as an sequence diagrams with a high-level description.

Election administrators addition On chaincode instantiation Election Organizer publishes public administrators list to the Ledger, to distribute this list to services. Figure 2.6 describes the accepting flow of the election administrators addition.

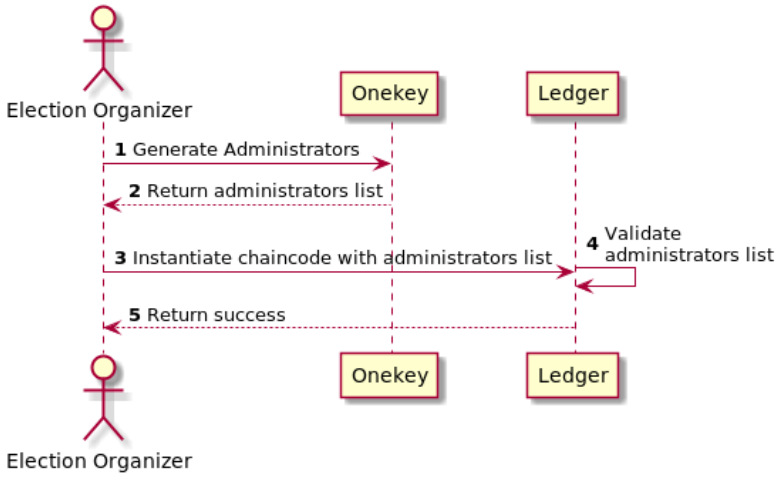

Figure 2.6: Sequence diagram of election administrators list publication.

Administrators list addition phase may fail in step 4 if the administrators list has an invalid structure or miss some values.

Administrators authentication The administrators authentication can be seen in Figure 2.7.

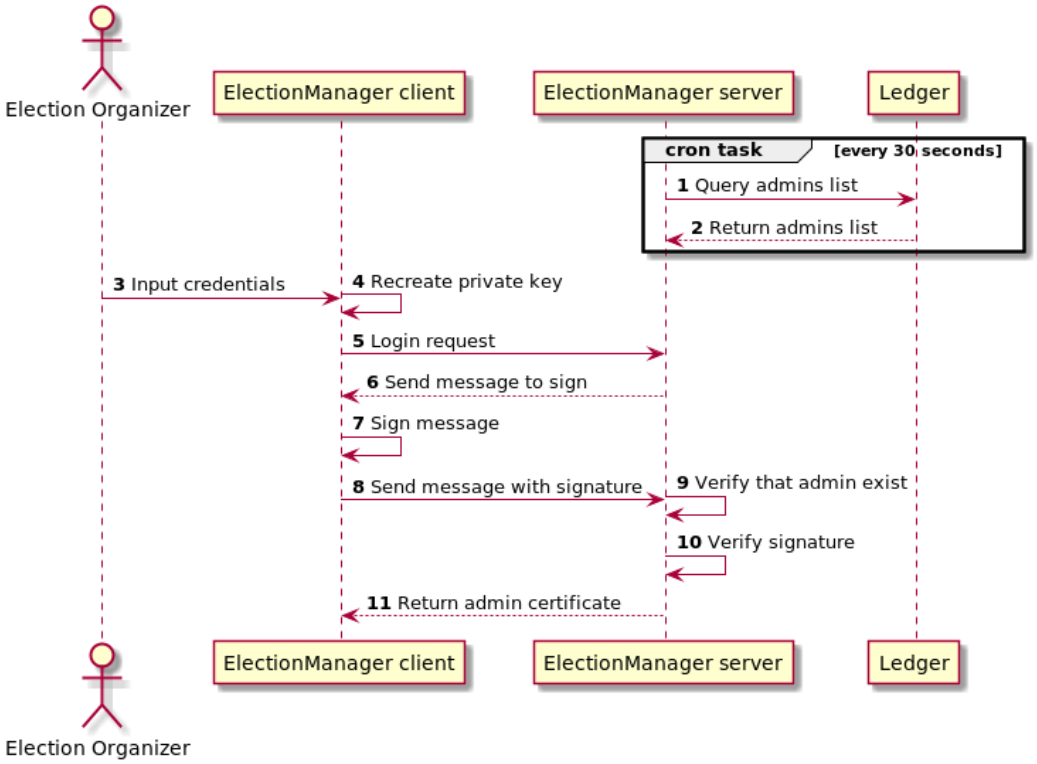

Figure 2.7: Sequence diagram of administrator authentication.

Authentication phase may fail on these steps:

– on step 9 server finds out that administrator with such ID does not exist;

– on step 10 signature verification fails.

After successful authentication administrator can proceed with election data management.

Election addition Election addition is made by the election administrator via ElectionManager and the process can be seen in Figure 2.8.

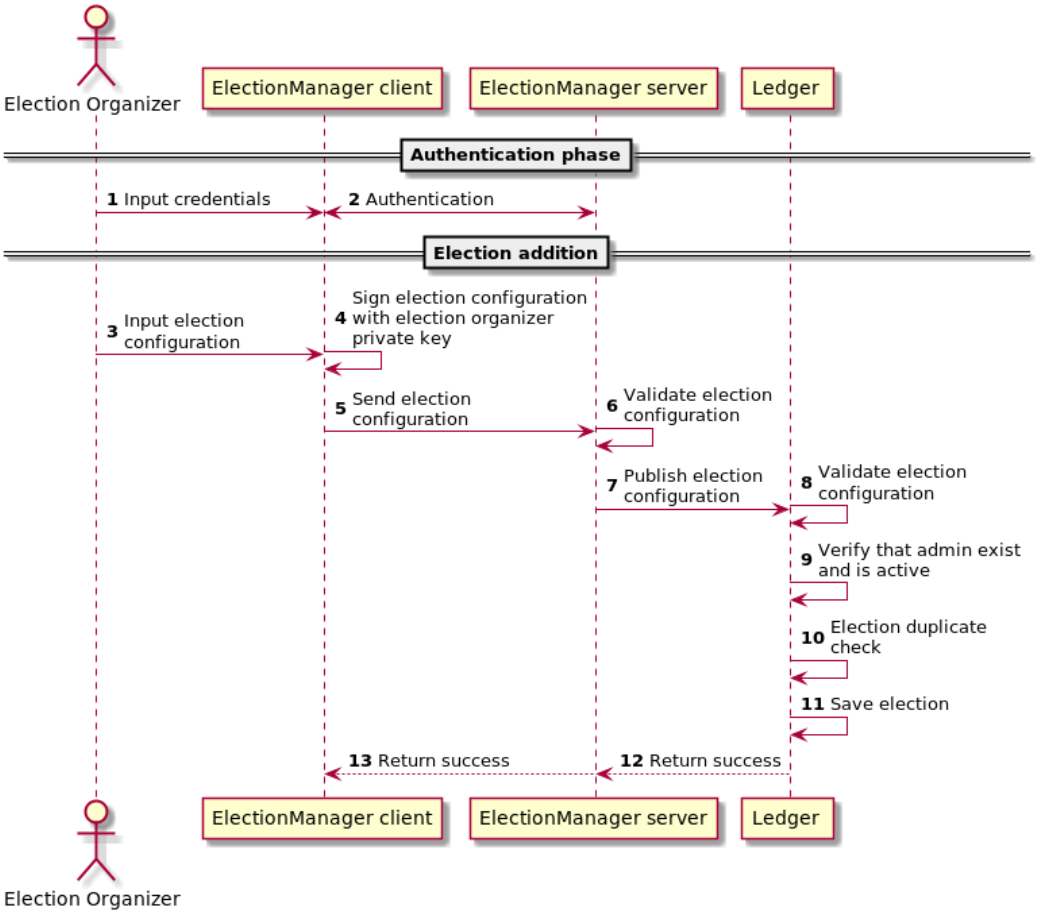

Figure 2.8: Sequence diagram of election addition to the Ledger.

Election addition can fail on different steps:

- on step 2 authentication fails and administrator is not logged in;
- on step 6 signed election validation failed;
- on step 8 signed election validation failed;
- on step 9 Ledger finds out that admin does not exist or is revoked;
- on step 10 Ledger finds out that election already exists or was previously published.

Voter list addition Voter list addition can be seen in Figure 2.8. Here we need to note that voter list can be published to the system only by the same person who published election configuration to the Ledger.

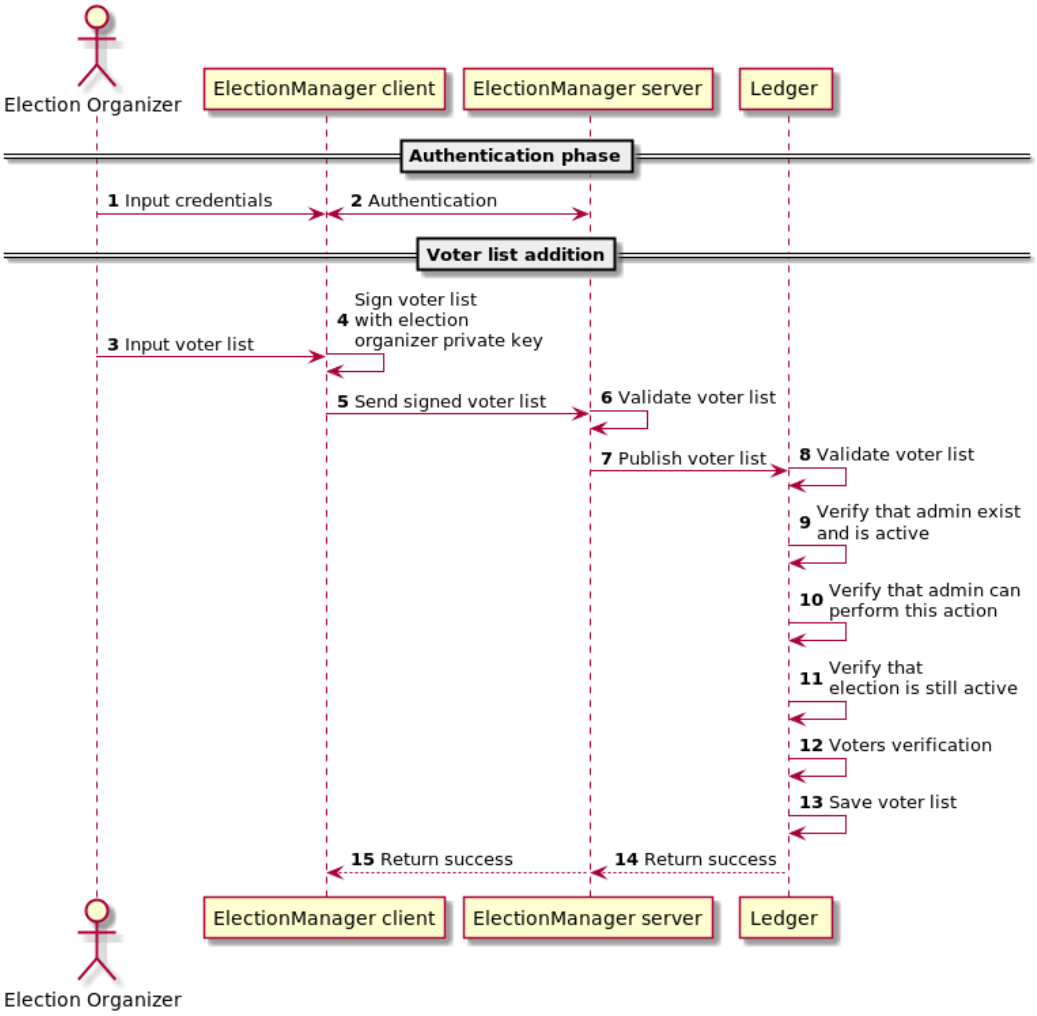

Figure 2.9: Sequence diagram of voter list addition to the Ledger.

Voter list addition can fail on next steps:

- on step 2 authentication fails and administrator is not logged in;
- on step 6 voter list validation fails;
- on step 8 voter list validation fails;
- on step 9 Ledger finds out that admin does not exist or is revoked;
- on step 10 Ledger finds you that administrator is not eligible to perform this action;
- on step 12 Ledger finds out that voter was not issued from the root CA defined in the election configuration or voter is already eligible/ineligible.

Voting Figure 2.10 describes the accepting flow of the voting process.

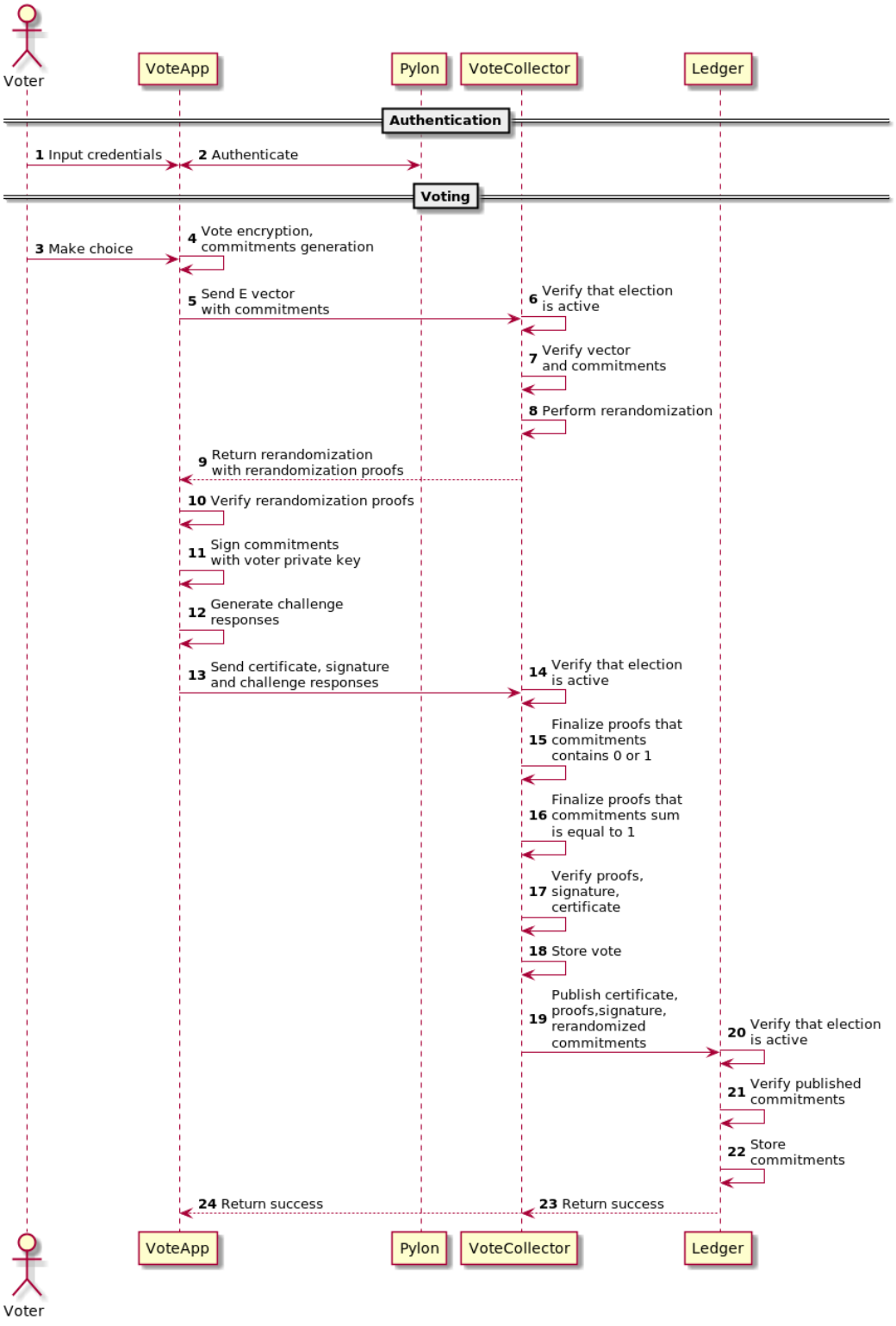

Figure 2.10: Sequence diagram of voting process.

Voting phase contains a lot of verifications between requests which means it can be stopped by VoteApp, Pylon, VoteCollector or Ledger. Here is the list of checks and possible points of failure:

– on step 2 authentication fails and voter does not receive an election;

- on step 6 VoteCollector finds out that commitments are not in the expected format or contains something different which triggers VoteCollector to return an error;
- on step 9 verification of rerandomization proof fails and VoteApp stops vote casting process. This failure means that VoteCollector made such rerandomization that it changed the content of initially provided commitments;
- on step 15 verification fails which resolves in casting process failure. Verifications that are done:
  - check that received signature verifies with received commitments;
  - check that voter certificate was issued from election configuration root certificate;
  - check that each commitment contains encryption of 0 or 1;
  - check that commitments sum is up to 1.
- on step 16 if VoteCollector can not save vote to datastore it will stop casting process with failure;
- on step 17 if commitments addition to the Ledger fails, the casting process is interrupted and vote from datastore on step 16 is deleted as well.

Votes aggregation
Figure 2.11 describes how votes aggregation is initialized and executed inside the VoteCollector.

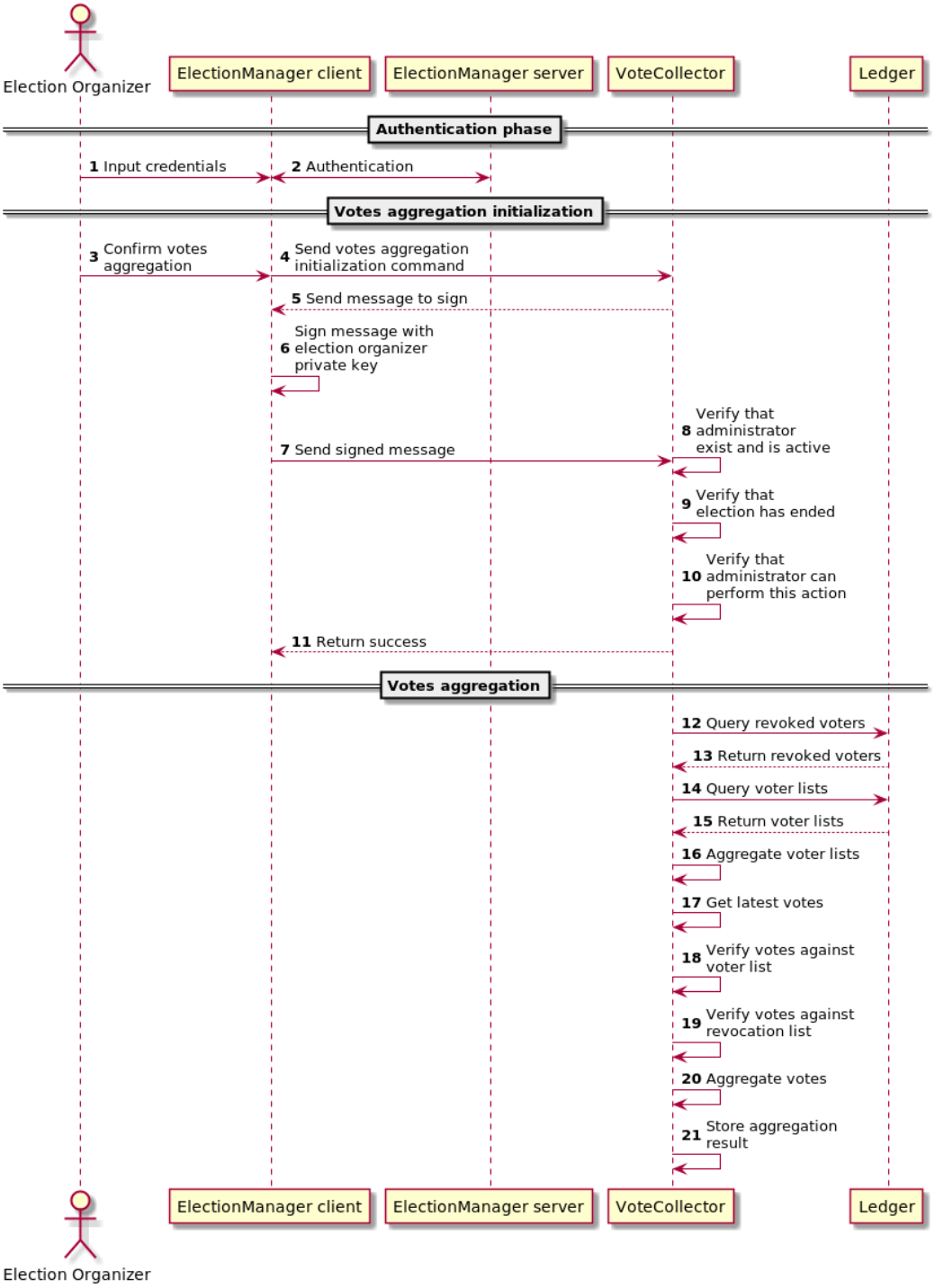

Figure 2.11: Sequence diagram of votes aggregation.

Possible points of failure:

– on step 2 authentication fails and election administrator is not logged in;

– on step 8 Ledger finds out that admin does not exist or is revoked;

– on step 9 Ledger finds out that election has not yet ended;

– on step 10 Ledger finds you that administrator is not eligible to perform this action.

In case on steps 17 to 20 something goes wrong with vote aggregation, this particular vote is marked as an invalid and voter is added to the revocation list.

When aggregation result is stored inside the VoteCollector, it can be queried via ElectionManager by the administrator.

Commitments aggregation Figure 2.12 shows how commitments aggregation is performed on the ElectionManager.

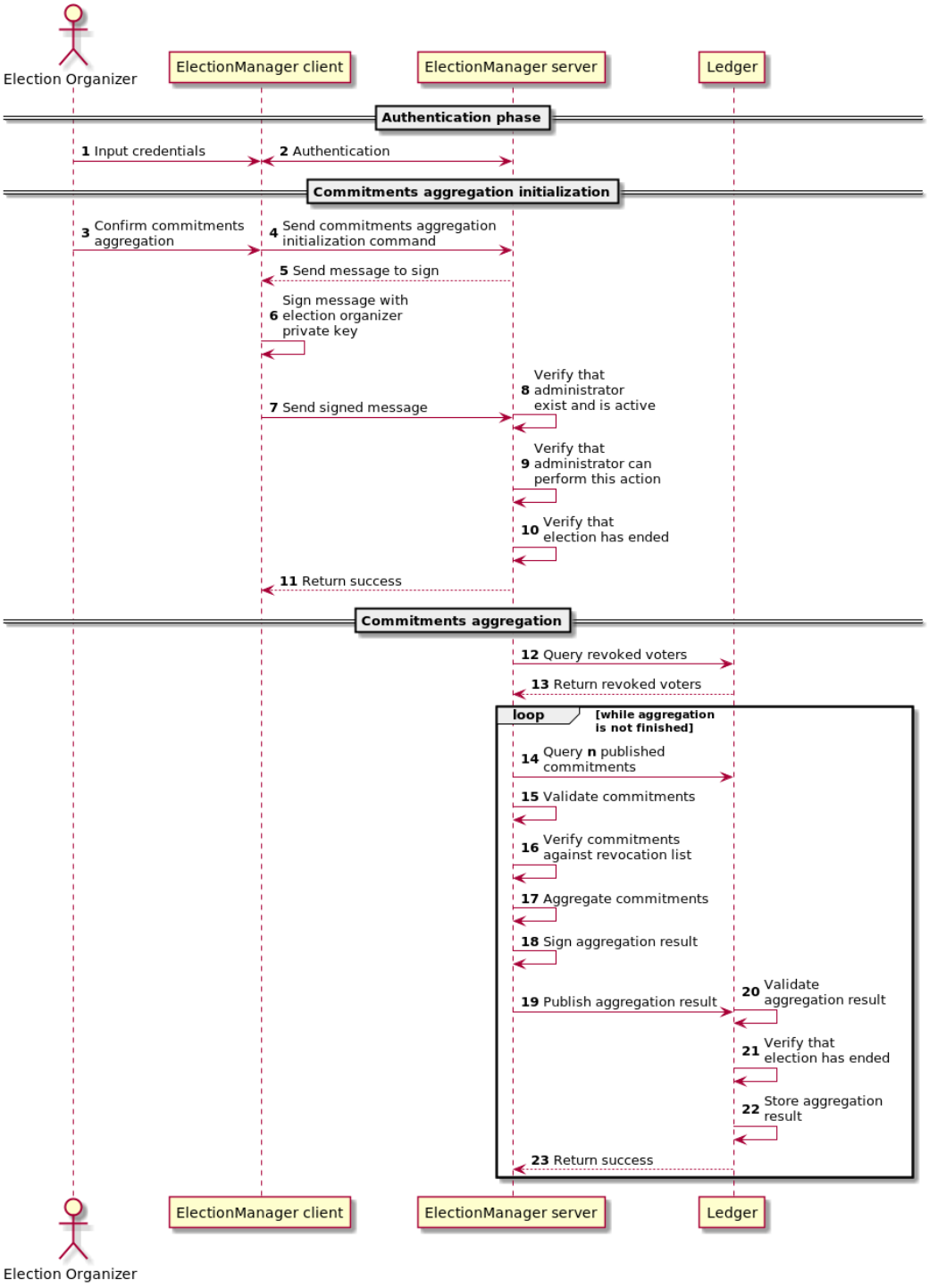

Figure 2.12: Sequence diagram of commitments aggregation.

On step 14 *n* is a number of commitments that are queried from the Ledger, which comes from the Election Organizer command on step 4.

Commitments aggregation can fail on these next steps:

– on step 2 authentication fails and administrator is not logged in;

– on step 8 ElectionManager finds out that administrator does not exist or is revoked;

– on step 9 ElectionManager finds out that administrator is not eligible to perform this action;

– on step 10 ElectionManager finds out that election has not yet ended;

– on step 20 validation of aggregation result fails;

– on step 21 Ledger finds you that election has not yet ended.

Tally results publication Firstly votes are being decrypted by Key application (see Section 2.2.2) and later are published via ElectionManager to the Ledger.

Figure 2.13 provides description of how tally results are published to the Ledger.

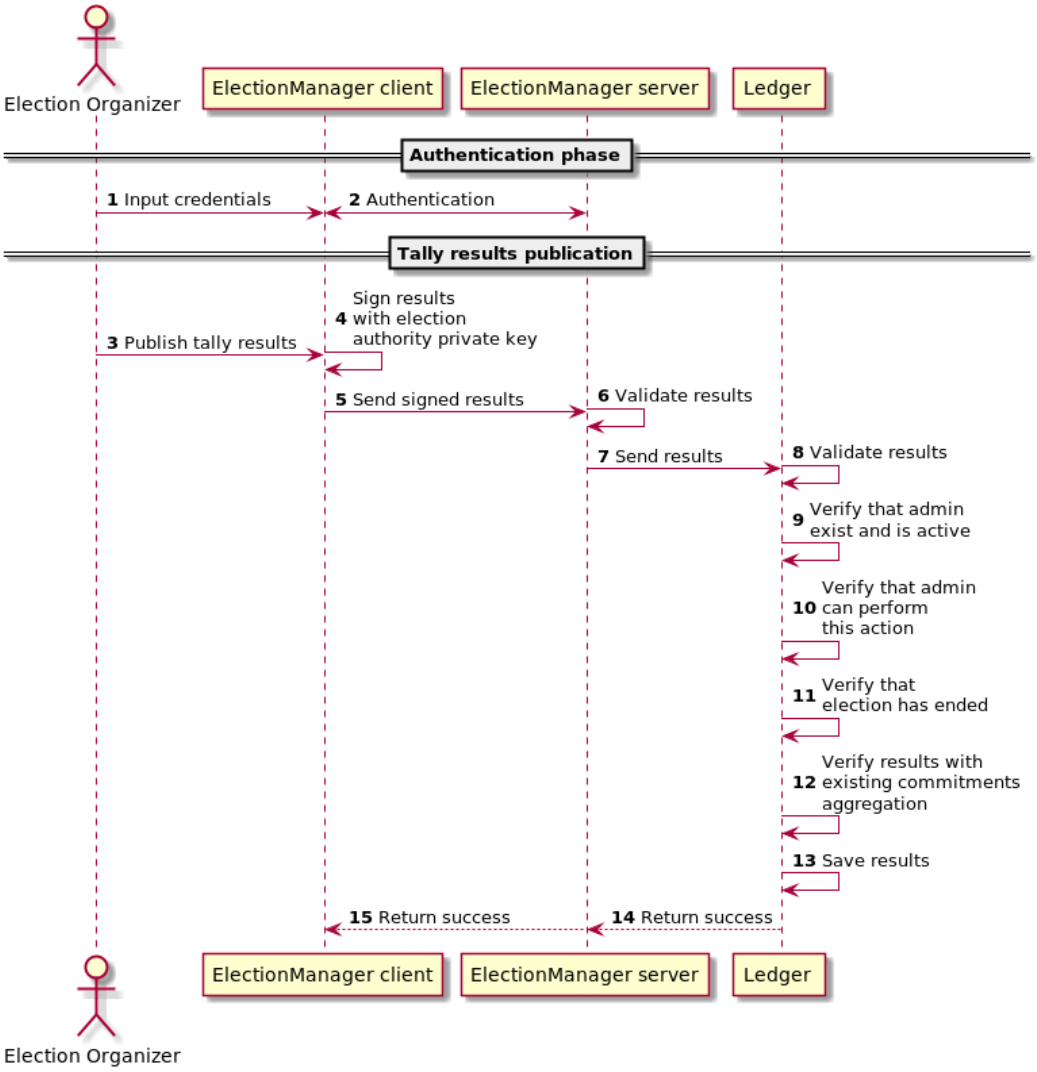

Figure 2.13: Sequence diagram of tally results publication to the Ledger.

Tally results publication can fail on these steps:

– on step 2 authentication fails and administrator is not logged in;

– on step 6 results validation fails;

- on step 8 results validation fails;
- on step 9 Ledger finds you that administrator is not active or is revoked;
- on step 10 Ledger finds you that administrator is not eligible to perform this action;
- on step 11 Ledger finds out that election has not yet ended;
- on step 12 Ledger finds out that commitments aggregation has not yet published or aggregation does not verify with tally results.

Public tally results verification Tally results verification can be made by any observer via VoteApp, where Figure 2.14 gives more information about the verification process flow.

By observer we mean absolutely anyone (voters, auditors, election organizers etc.) who can access the VoteApp.

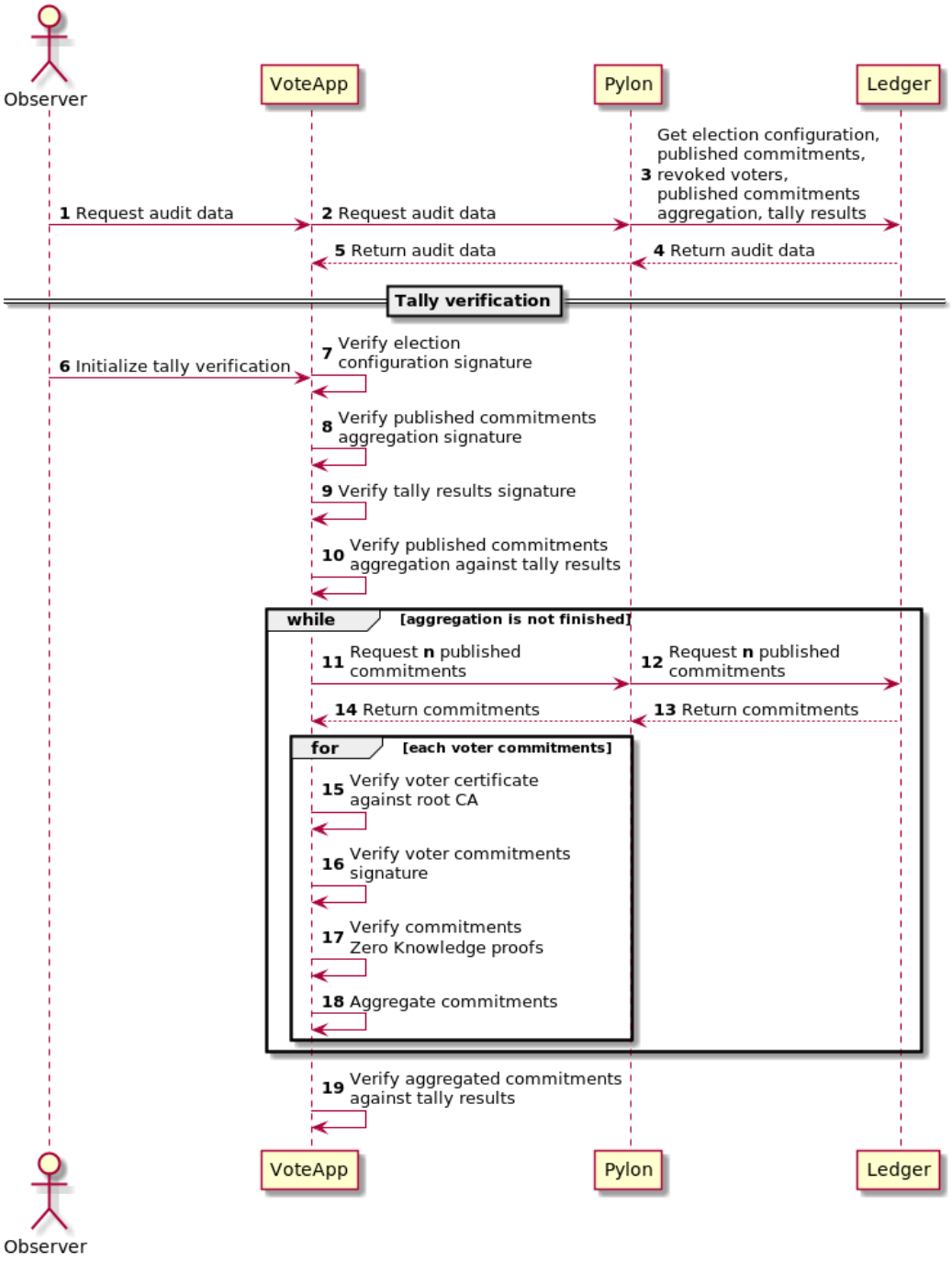

Figure 2.14: Sequence diagram of tally results verification in VoteApp.

Possible points of failure:

– on steps 7, 8, 9 signature verification fails;

– on step 10 verification of provided aggregation against tally results fail;

– on steps 15, 16, 17 verification fails, which results in whole verification process to fail;

– on step 19 verification fails where there are multiple possibilities of why this happened:

1. Votes were tallied incorrectly.
2. Published commitments aggregation were aggregated incorreclty.
3. Wrong openings/tally were/was provided.

Step 10 and step 19 verification are performed on different data. On step 10 VoteApp verifies aggregation provided by the Election Organizer where on step 19 VoteApp verifies tally result with recreated commitments aggregation. Step 19 ensures Observer that no votes were lost nor added to count without an evidence.

Common Packages

Multiple packages are used by services and chaincode in order to unify the code.
Here is the list of these shared packages written in Go language:

– `tiviledge.io/common/list` – contains data structures for voter and revocation lists as well different functions for their validation and serialization.

– `tiviledge.io/common/types` – contains data structures for voters, administrators, identities (real person his/her public identity), session manager. In addition has multiple functions and structures related to signing.

– `tiviledge.io/common/bulletinboard` – contains data structures for storage inside the Ledger but additionally used in different services for data serialization.

– `tiviledge.io/common/processor` – responsible for votes aggregation. Contains logic for different aggregation stages.

– `tiviledge.io/common/election` – contains data structures and functions for serialization/validation related to election configuration and election results.

– `tiviledge.io/common/cce` – is main logic of votes encryption and decryption. Creates different proofs with possibility to verify them.

– `tiviledge.io/common/voting` – is a wrapper around `tiviledge.io/commin/cce` package. It forms different messages that are used in the voting protocol.

– `tiviledge.io/common/rnd` – contains logic for random string and BIG number generation.

Chapter 3

UC4: Decentralized Software Updates for Cardano Stake-Based Ledger

3.1 Introduction

This use case deals with the problem of the secure decentralization of software updates for blockchain systems. In particular, our focus is the stake-based blockchain system Cardano [20]. In this use case, we have defined a secure software update mechanism for Cardano where all the key decision points in the lifecycle of a software update are decentralized through a voting process. So it is the community that decides on the evolution of the blockchain system. To this end, we are implementing a research prototype that materializes our ideas and gives us the opportunity to test them in an actual system.

In this chapter, we present the architecture of the said prototype. This a more mature and elaborate version of the architecture that had been presented in deliverable D4.1 "Report on Architecture of Secure Ledger Systems". In particular, we present our architecture at three different levels of detail:

- **The global decentralized governance level.** This is the view that corresponds to the overall Cardano decentralized governance landscape and shows how the software update system fits in the *big picture*.
- **The intra-node level**. This is the view that corresponds to the Cardano node architecture and show how the update system is integrated within the Cardano node.
- **The update system level**. This is the most detailed architectural level and corresponds to the design of the update system per se.

In the following sections, we describe the said three levels of architecture and discuss core design issues in the software updates prototype.

3.2 The Global Decentralized Governance Level

There are two important questions related to governance: a) Who decides? and b) Who pays? Indeed, this is true for any organization, or any system and it is also true for blockchain systems. Decentralized governance on the other hand, essentially means that the answer to these questions is: "the community". Cardano aspires to become a truly community-managed blockchain. A self-sustaining blockchain system, where stakeholders can influence the future development of the network and fund new development through a treasury system. However, decentralized governance is a long journey. At the end of this journey -code-named the Voltaire era- Cardano's future will be in the hands of the community!

What does it take to achieve decentralized governance? Can there even be a "centrally governed decentralized system"? Isn't this an oxymoron? There are many blockchain systems that claim that have achieved decentralized

governance, but have they really? Imagine a blockchain system where the community collectively decides what changes should be funded. However, when the change is implemented, then there is a trusted central authority who decides if this implementation is appropriate to be deployed and when the changes will take effect. Can we call this decentralized governance? It is important to understand that governance does not have to do with just one decision. There are a plethora of core decisions to be made in the whole lifecycle of a change; from the initial idea conception to the very end, where changes take effect into the system. Decentralized governance requires to decentralize all decision points, or else it remains an utopia.

In this section, we will provide an overview of the overall Cardano decentralized governance landscape. We will identify the main components in this landscape and describe how these can be integrated into a global decentralized governance architecture. We start by describing the main players in the governance game. We define what are the core decisions that have to be made and identify who is responsible to take each decision. Then we move on to describe what are the main architectural components that provide the means for these decisions to be made collectively. To this end, we identify three components: a) the CIP process, b) the Treasury system and c) the Software updates system. The logical flow of a change through these three systems will essentially materialize decentralized governance in Cardano.

The main players These are the main players in Cardano decentralized governance:

- *The Cardano Community*. This is the community in the broader sense. This means that there is no technical constraint in order to be part of the Cardano community; for example the ownership of stake. Anyone could participate. The community consists of Ada owners, Cardano enthusiasts, Developers, stake pool operators, etc.
- *The Stakeholders*. These are all the parties that own Ada.
- *The Stake Pool Operators (SPOs)*. SPOs are entities whose job is to participate in the Proof-of-Stake consensus protocol, with the stake that has been *delegated* to them by the stakeholders. By doing this, SPOs earn rewards. They cannot use the delegated stake to make transactions and thus transfer the money that they do not own.
- *The Experts*. Experts are a still to-be-defined entity whose purpose is to provide technical expertise to stakeholders, in order to help them decide on governance issues, like the voting of a software update, the funding of a proposal etc. Experts are rewarded for the services they provide.

The main components In figure 3.1, we identify three major components in the Cardano decentralized governance landscape.

- *The CIP Process*. *CIP* stands for *Cardano Improvement Proposal* and it is a document. This document is the main vehicle to formally describe and justify any *idea* for improving the Cardano system. It is similar in concept to BIPs (Bitcoin Improvement Proposals) and EIPs (Ethereum Improvement Proposals). The Cardano Improvement process is intent on enabling a public discussion place for "Cardano Improvement Proposals" as Process, Standard or Informational proposals, in a source-controlled, Foundation-managed, GitHub repository.The aim is to have an open-sourced collection of CIPs available and proposed for the community, enabling different flavors of implementation and a varied ecosystem, maybe at first mostly curated by IOHK/Emurgo/Cardano Foundation via a group of CIP Editors, but eventually community-maintained. Getting an idea formalized in the repository provides the foundation for solid conversation and public inquiry. From an architectural perspective, please note that the CIP Process is an *off-chain* process.
- *The Treasury System*. The Treasury system is the means by which proposals and ideas can be funded in a collective, community-based manner. Funds in this global, community-owned pot, are raised via taxation,

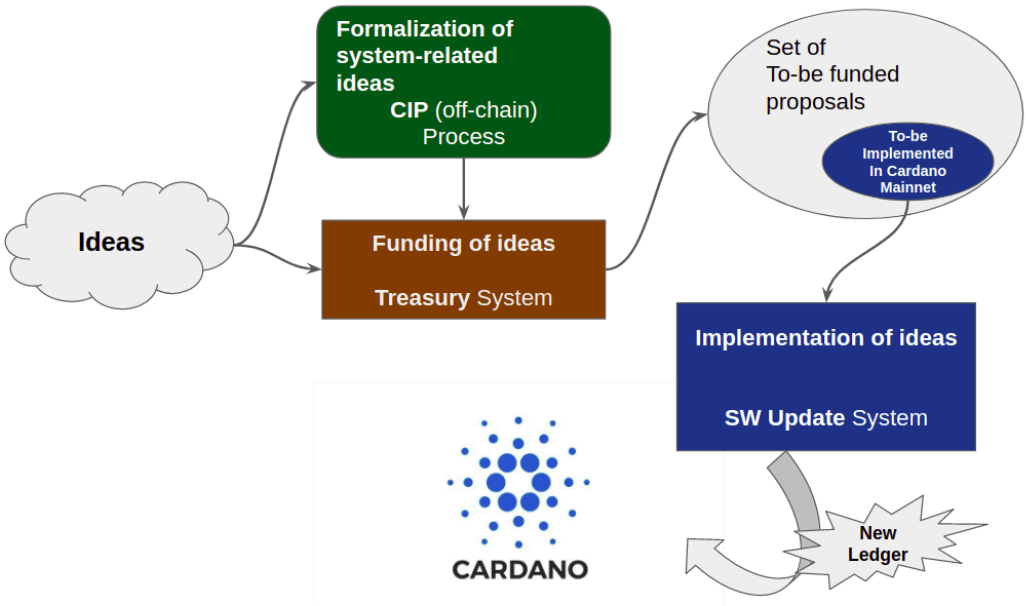

Figure 3.1: The Cardano decentralized governance main components.

donations or other methods and are offered for the funding of new projects through the process of *funding ballots*. It is a fully-fledged decentralized governance system that utilizes advanced voting protocols, liquid democracy models and advanced cryptography to ensure the secure and decentralized operation of the Treasury. A good description of the research-related issues regarding a Treasury system can be found in [ZOB18]. The Cardano Treasury system can fund *any* idea that improves the Cardano ecosystem and not necessarily only system-related ideas; for example to launch a new marketing campaign, or create educational material for the Cardano community and so on. In fact, the system-related ideas, i.e., the ones that will eventually become a *software update* proposal, are only a subset of the total set of the to-be funded proposals. This is depicted in figure 3.1.

– *The Update system*. Finally, the Update system is the means for implementing and applying to the Cardano blockchain all system-related ideas. It is responsible for taking a software update through out all the steps in its lifecycle, from ideation to implementation and finally the activation of changes on the blockchain and do that in a secure but also *decentralized* manner. Of course, the Cardano update system is the main goal of this PRIViLEDGE use case.

The basic flow depicted in figure 3.1, shows that system-related ideas going through a formalization process, in the form of a CIP document. Non system-related ideas do not need this step. In any case, all ideas will end up in the Treasury system requesting for funding. Finally, the subset of the proposals that have an approved funding and correspond to software updates, will go through the SW Update system to enable the implementation of the initial idea and the activation of the changes on the blockchain system.

We have seen that decentralized governance is all about community-driven decision making. Moreover, the components that we have described provide the means by which decisions can be made collectively; but what are the core decisions? In table 3.1, we list the main decisions of the decentralized governance circle. Also in this table, we propose which player should make the corresponding decision (”Who” column) and via which component this decision process will take place (”Means” column).

In the lifecycle of a software update, we identify three core decisions that are recorded in table 3.1. The first decision has to do with the approval of the *design* of the update proposal. This design is formalized and submitted for review and approval via a technical document called *System Improvement Proposal (SIP)*. The next important decision has to do with the approval of the implementation of the update proposal. Finally, for all

Decision	Who	Means
What proposal should be funded?	Stakeholders	Treasury system
What Cardano (system-related) improvement proposal (CIP) should move forward?	Community	CIP off-chain Process
What Design (SIP) should be approved for implementation?	Experts	SW Updates System
What Implementation of a design should be approved for deployment	Experts	SW Updates System
When should we activate the deployed changes (synchronization)?	Stakepool Operators	SW Updates System

Table 3.1: Main decision of the decentralized governance circle.

approved implementations there is a final decision of when to activate the corresponding changes. This is critical because, if the parties in the Cardano network activate without appropriate *synchronization*, then there is a risk for a chain split and for activating a new consensus protocol in which the security assumptions do not hold.

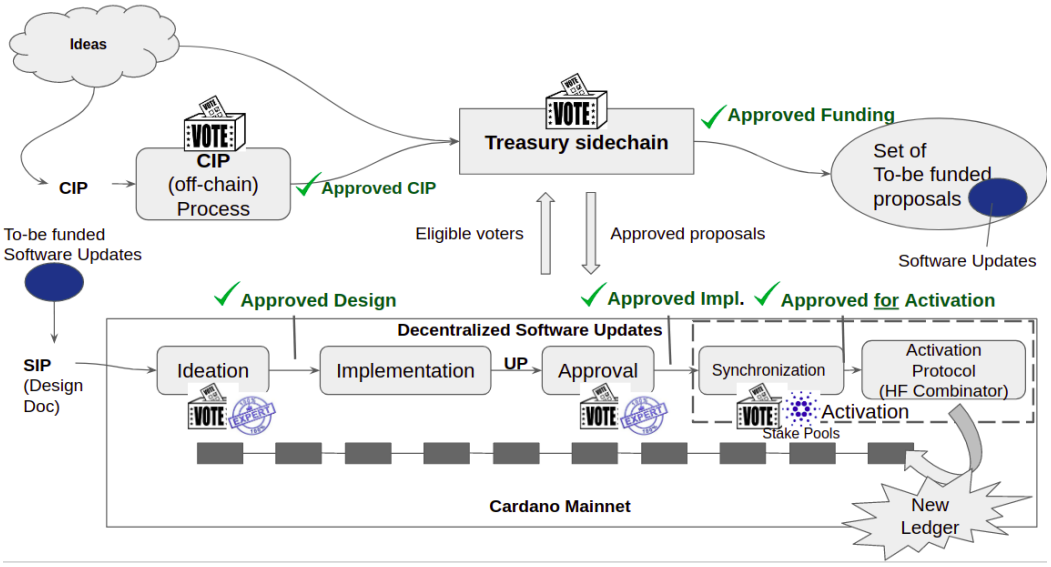

Figure 3.2: The Cardano decentralized governance global architecture.

In figure 3.2, we can see the Cardano decentralized global architecture. Observe that the aforementioned three main components of the decentralized governance landscape are depicted. In particular, the software updates system appears integrated with the Cardano main-chain and is depicted as a flow of the core phases in the lifecycle of a software update, namely: Ideation, Implementation, Approval and Activation. In contrast, the Treasury system runs as a sidechain [Bac+14]. This is a conscious architectural decision, which we explain next.

The update system enables changes on the main-chain, so it is very important that it is secure, because if the chain gets compromised or the protocol fails people can lose their money. So the chain upon which the update system runs must be *at least* as secure as the main Cardano chain. This requirement calls for a tight integration between the update system and the Cardano node software, as we will show in section 3.3. For example, the triggering for the activation of a change that comes from the update system is integrated within the ledger layer of the Cardano node (see section 3.3).

If we had followed a sidechain approach for the update system, then we would have to utilize more advanced cryptography [GKZ18], in order to provide strong evidence that indeed the changes should be activated on the

main-chain. Moreover, the requirement for a robust, simple-to-implement, fast-and-scalable and also transparent and auditable update mechanism has lead us to the decision to implement the update mechanism on the Cardano main-chain. In contrast, the Cardano Treasury system has built advanced voting protocols and utilizes advance cryptography to ensure the privacy of the voters (something that is not required for the update system due to the transparency requirement and also because votes for updates must be open, in order for the experts to be accountable for their vote). By implementing the Treasury system as a sidechain, we eliminate the need to implement complex cryptographic algorithms on the Cardano chain and gives the ability to the Treasury team to experiment with different voting protocols and governance models, without the need to trigger hard forks on the Cardano chain.

In figure 3.2, we can also see the main decision points in the governance cycle depicted as check-marks. We see and initial idea to take the form of a CIP and go through the off-chain CIP process. Then, once it is approved it is submitted to the Treasury system for requesting funding. The Treasury sidechain gets from the main-chain the list of eligible voters and a ballot is executed. If the funding gets approved, then the sidechain must interact with the main-chain in order to send the ballot result (in a secure undisputed way[1]) and release the funds. Next, the approved proposal must be submitted on the main-chain as a SIP and the update protocol starts. During the Ideation phase a voting round will run, in order to approve the submitted SIP. In this voting process, experts will vote through a delegation process in place. Approved SIPs proceed in the implementation phase, the end-result of which is to submit the implementation for approval. This is the Approval phase of the update protocol where another voting round with the experts takes place, in order to review and approve the submitted implementation. The implementations that are approved move on to the Endorsement phase where parties (Stake Pool Operators) download and upgrade their software and signal their readiness for activating the changes. Once, the required *adoption threshold* of endorsements has been reached, the update mechanism gives a green light to the activation protocol to run. This protocol will take care of transitioning to the new consensus protocol in a secure manner. This is implemented by a component called *hard fork combinator*, which is shown in the figure and will be also discussed in the next section.

3.3 The Intra-Node Level

The *Cardano node* is the software that must be run by anyone who wants to be part of the the Cardano blockchain network. In particular, it is the software necessary in order to participate in the Ouroboros consensus protocol [Kia+17]. In this section, we will take a closer look into the Cardano node internals and see where the update system fits within the Cardano node architecture.

In figure 3.3, we provide a very high level view of the Cardano node architecture. At its core, the Cardano node consists of three main components:

- the consensus layer,
- the ledger layer and
- the network layer.

The consensus layer is the heart of the node. It is where the Ouroboros consensus protocol [Kia+17] is implemented. In this layer, the notion of the *blockchain* is materialized. The consensus layer is responsible for deciding on and maintaining the *chain of blocks* that will be the *single version of the truth*. Based on this truth, the ledger layer will build the notion of a *transaction ledger*. The primary job of the consensus layer is to "listen" for blocks from the network, or to create new blocks with transactions drawn from the *mempool* and maintain the blockchain by faithfully following the directives of the consensus protocol. It implements all the logic how to handle forks and apply the *chain selection rules*, so that there will always be a single chain of blocks exposed

[1]The exact protocol for the interaction of the Treasury sidechain with the Cardano main-chain is out of the scope of this document and thus omitted.

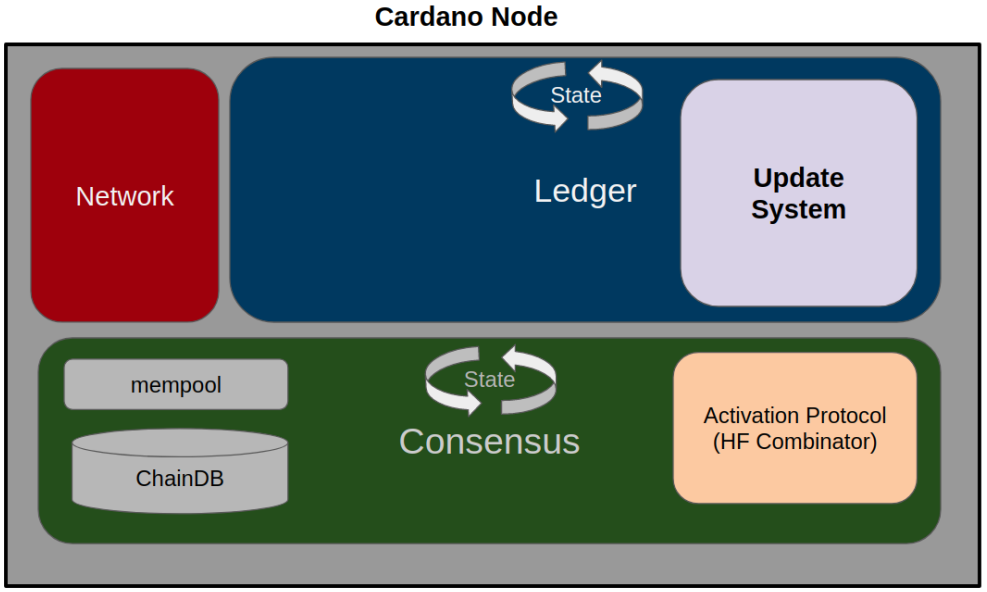

Figure 3.3: The Cardano node architecture.

to the other components. The actual blockchain data are stored in a database called ChainDB. Also, we must note that this layer maintains its own specialized state, which updates with every operation taking place at the blockchain level. Finally, it is important to stress that the consensus layer knows *nothing* about the *contents* of the blocks. This is the responsibility of the ledger layer.

The ledger layer knows how to validate the contents of a block (including the block header). For this reason, the ledger layer is the ultimate authority within the node to decide if a block header, or a block is valid. Upon rejection of a block the consensus layer immediately discards the invalid block and blacklists the peer who transmitted it. Therefore, the ledger layer is where all the *validation rules* are implemented. Moreover, since the ledger layer can interpret the contents of the blocks, i.e., the transactions, it is the place where the transaction logic is implemented. So the ledger layer knows what is a transaction that moves value from one party to another and maintains the *ledger state* appropriately to correspond to the actual transfer of value.

Finally, the network layer is a specialized layer that tries to fulfill the network needs of the consensus layer. Naturally, the node has extensive needs of network communication, since it is a system that communicates directly with both upstream and downstream peers (i.e., other nodes), but also with clients like the wallet or the cli. Upstream peers are the nodes from which the node receives data (transactions or blocks) and downstream peers are the nodes to which a node sends data. Although in practice, in some cases, the distinction between the network and consensus layer is not so clear, at the level of detail we are discussing, it is sufficient to think the network layer as the main network services provider of the consensus layer.

At this point, it is important to point out one significant design principle that has been followed in the Cardano node architecture, namely the *isolation principle*. Essentially, this means that the consensus layer knows nothing about transactions and their validity rules and similarly the ledger layer knows nothing about the actual blockchain with all its forks and rollbacks taking place at the consensus protocol level. The consensus layer exposes to the ledger layer just a *single* history of transactions, nicely grouped into blocks. Among the many features of this isolation principle a very important one is that one can thoroughly test each layer separately. Moreover, one can move to a new version of ledger rules without affecting the other layers.

In figure 3.3, we can see the update system as a component *within* the ledger layer. Indeed, just like the ledger layer is the ultimate authority for validating the contents of a block, i.e., transactions, similarly the update system is the sole responsible for interpreting the special content of a transaction that is called the *update payload*. In our software updates solution, update events are transmitted via the payload (i.e., metadata) of individual

transactions. This is a design choice that enables: a) the open participation in the update protocol by anyone (i.e., stake owner) who can submit a common transaction and b) the enforcement of a *fee* with every update event transmitted, which is a protection against denial of service attacks. Therefore, the ledger layer knows nothing about the update payload and needs the update system sub-component to validate this payload and act upon it by updating appropriately the ledger state.

The last component of figure 3.3 that we need to describe is the *hard fork combinator* (HFC). This is a specialized component residing in the consensus layer. Its sole purpose is to execute securely the transition from the current consensus protocol to the upgraded consensus protocol triggered by a software update. The HF combinator implements a secure activation protocol such as the ones that we have proposed recently [Cia+20]. Please note that the HF combinator does not know when to initiate the transition, i.e., when the security conditions hold for such an endeavor; it only knows how to execute the transition. The trigger for initiating the activation must come from the component that implements the *update logic* and knows when it is the right time to activate; this is of course the update system within the ledger layer.

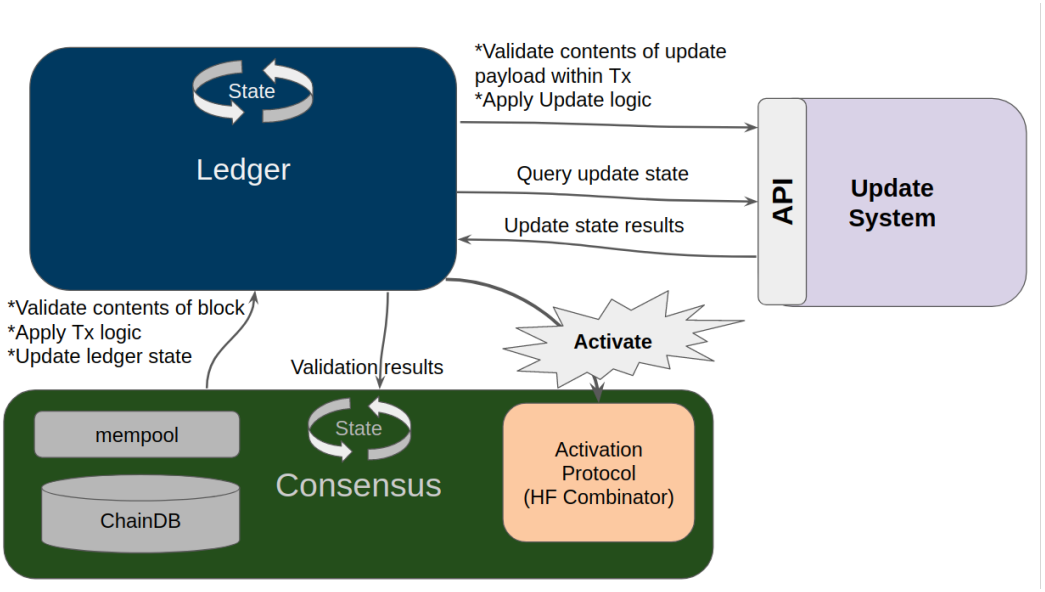

Figure 3.4: The integration of the update system in the Cardano node.

Finally, in figure 3.4, we depict how the integration of the update system within the node architecture is actually implemented. Starting at the bottom of the picture, we can see the consensus layer handing over to the ledger layer the contents of a block (i.e., transactions) to be validated and wait to get back the validation results. The ledger layer is responsible for validating each transaction except for one particular case: the update payload. Therefore, in the figure, we can see the ledger layer interacting with the update system through a well-defined *API* that the latter exposes to the outside world. Basically, through this API, the ledger layer can hand over to the update system a specific update payload. The update system validates this update-specific payload (e.g., a proposal submission, a vote for a proposal, an endorsement of a proposal etc.) and applies the appropriate *update logic*. Furthermore, with each and every update event the update system maintains its own update state and offers a state query interface through the exposed API. The ledger constantly queries the update state and in turn updates the ledger state. Finally, the ledger layer ”asks” the update system when it is the right time to activate a change and only when the update system gives the green light, then the ledger layer triggers the HF combinator to activate the changes.

3.4 The Update System Level

In this section we describe the 3rd level of detail of our proposed architecture that of the update system per se. For the convenience of the reader, we start with a short overview of the decentralized software update lifecycle, the phases of which, the update system essentially implements.

3.4.1 An Overview of the Decentralized Software Update Lifecycle

In order to understand the logic that the update system implements, we need to closely follow a software update through out its whole life-cycle; from the very first phase, where it is born as an idea, to the last phase, where the changes are activated on the blockchain. In deliverable *D4.1 Report on Architecture of Secure Ledger Systems*, we have provided a detailed description of all the steps in the said lifecycle. In this section, we only provide a brief overview (see figure 3.5), in order to set the context for the reader's convenience.

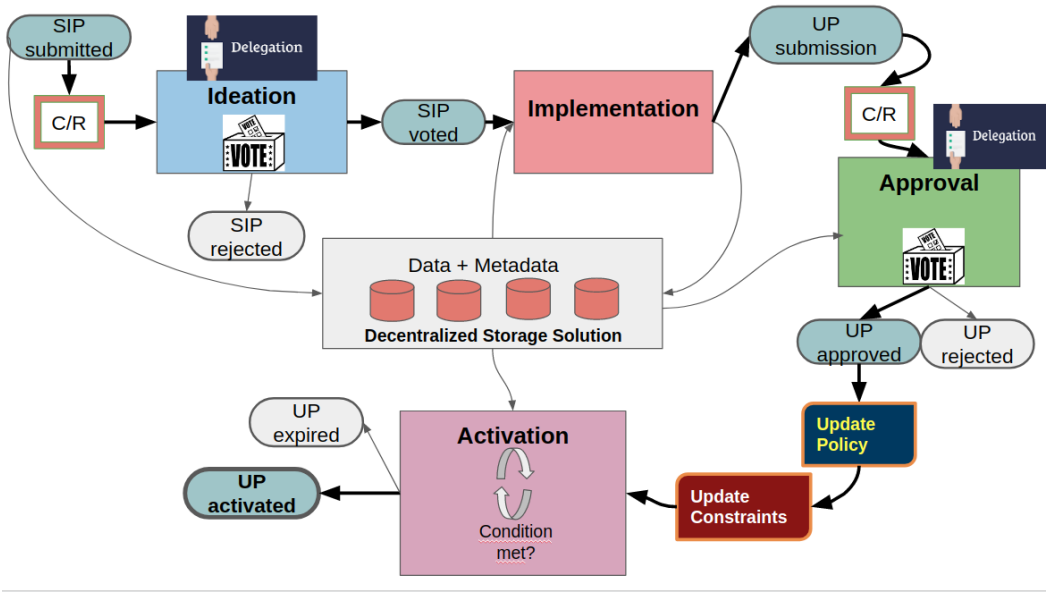

Figure 3.5: The Decentralized Software Update Lifecycle.

A software update starts its life as a *System Improvement Proposal (SIP)*, which is submitted to the blockchain. This submission takes place with a fee-based transaction, which carries the proposal in its metadata (*transaction payload*). A SIP is a structured document describing an update proposal. This submission goes through a typical *commit-reveal scheme*, in order to ensure the rightful authorship of the proposal. Once the SIP is revealed, a voting period starts for this SIP; the software update has entered the *Ideation phase*. The duration of the voting period is metadata-driven, i.e., defined in the SIP metadata[2].

The purpose of the Ideation phase is to help the community decide which SIP will move forward to the next phase. Votes are also fee-based transactions with a special payload. Anyone who owns enough stake to make a transaction can potentially vote. Votes count proportionally to the owned stake. Votes are accepted only within the voting period and multiple proposals can compete in parallel. The outcome (verdict) of the voting process for a specific proposal might be: a) *Accepted*, when stake in favor is above a threshold, b) *Rejected*, when stake against is above a threshold, c) *No Quorum*, when stake abstaining is above a threshold; this outcome leads to revoting, d) *No Majority*, when non of the previous three results occur, which also leads to revoting and e) *Expired*, when after the maximum number of revoting periods has been reached and still the proposal was neither been accepted or rejected. Finally, note that for the voting, we allow delegation of a stakeholder's voting right to an *expert*. For the scope of our prototype implementation we have assumed delegation as an out-of-band solution

[2]There is an upper limit for the voting period duration in the implementation, to prevent DoS attacks.

and provided direct voting power to stakeholders. This is basically for two reasons: a) to reduce risks in the prototype implementation (especially of the Cardano integration part) and b) to defer introducing delegation to experts until a proper game-theoretic analysis of the expert's incentives has been completed.

The *Implementation* phase is an off-chain process, where an approved SIP is implemented. It ends with the submission of the implementation, which we formally call *Update Proposal (UP)*[3], to the blockchain. Once this UP is revealed, the we have entered the *Approval* phase. This is the phase where the community is called to approve a submitted implementation. The voting process and the technical details are similar to the Ideation phase. Once the UP is approved it enters the *Activation* phase depicted in figure 3.6.

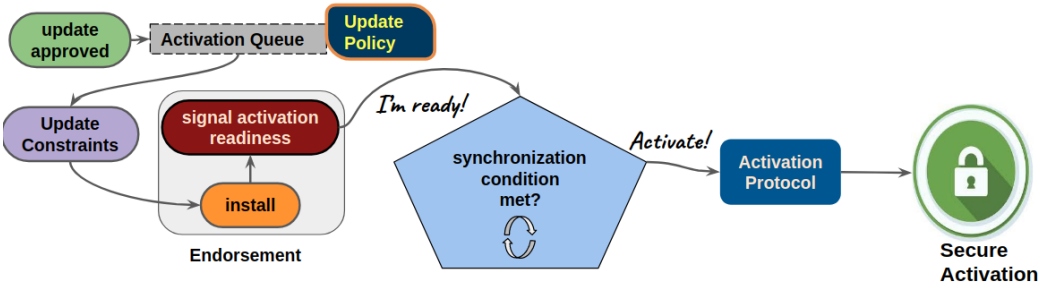

Figure 3.6: The Activation Phase.

An approved update proposal enters the activation phase and is placed in the *activation queue*. At this point, the *update constraints* of the proposal will be evaluated. A proposal satisfies the update constraints when:

– is approved,

– meets its dependencies,

– does not conflict with the current version,

– has the highest priority among competing proposals

If a proposal satisfies its update constraints it enters the *Endorsement Period*. This is the period where the *block issuers* download and install the update and *declare* upgrade readiness. Note that only a *single* proposal can be endorsed at a time. The endorsement period lasts N number of *epochs*, which is a metadata-defined parameter, called the *safety lag*; the safety lag corresponds to the sufficient deployment time window required for the specific update proposal. Once the endorsements reach a specific stake threshold, called the *adoption threshold*, the activation gives the green light to the *activation protocol* to run. The activation protocol ensures the secure activation, i.e., the secure transfer from the old ledger to the upgraded ledger, based on our formal definition of activation security and corresponding security proofs [Cia+20]. In a nutshell, *secure activation* means, the secure transition from the old ledger (L1) to the new ledger (L2) in a way where:

– L2 enjoys liveness

– L2 enjoys consistency

– L2 has L1 as a prefix

Finally, note that the all the metadata-defined information about an update proposal, such as the deployment window length, the proposal's priority, the version dependencies etc., form the proposal's *update policy* to be followed by the update system. This policy is accepted and confirmed by the community through the previous voting process during the Approval phase.

[3]We will use the terms *UP* and *Implementation* interchangeably

3.4.2 The Update System Software Architecture

The update system has the `Cardano.Ledger.Update` module as its entry point. The ledger layer of Cardano uses this module when processing the update payload. In addition, it has to register slot ticks in this module as well, since the update logic is influenced by the passage of time (for instance, version changes occur at epoch boundaries).

Functions of the update module operate on the update state, which is polymorphic on the Ideation and Approval phases payload -SIP and Implementation (or UP) respectively- and includes the state of all the update phases:

```
data State sip impl =
  State
  { ideationSt :: !(Ideation.State sip)
  , approvalSt :: !(Approval.State impl)
  , activationSt :: !(Activation.State sip impl)
  }
```

The update state can be manipulated by means of three functions, which are polymorphic on the ideation and implementation payload, and on the environment:

- `initialState`
- `tick`
- `apply`

The `initialState` function returns the initial state of the update system, using the given protocol as starting protocol (or genesis protocol). This function requires that the `sip` type is a proposal and that the `impl` type is an implementation of this (SIP) proposal. Later on, we will explain the `Proposal` and `Implementation` typeclasses[4].

```
initialState
  :: (Proposal sip, Implementation sip impl)
  => Protocol impl
  -- ^ Initial protocol. This determines the current version.
  -> State sip impl
```

The `tick` function registers the passage of time. Its first parameter, `env`, is an environment that:

- is assumed to contain slot number (`TracksSlotTime`), which is used to register the passage of time;
- has a state distribution for:
  - SIP voters
  - Implementation voters
  - Proposal endorsers

  This stake distribution is used to tally the votes;
- has an adversarial stake ratio, which is used to compute the different voting and activation thresholds. This ratio is a theoretical value.

[4] A typeclass in the functional programming language Haskell is a group of data types that implement a specific interface to the outside world, consisting mainly of functions but also constituent data types that represents a specific behavior. For example, The `Eq` typeclass provides an interface for testing for equality. Any type where it makes sense to test for equality between two values of that type should be a member of the Eq class

Given an environment and a state, the tick function registers the slot change and updates this state. A slot tick changes the given state when:

- a tally slot is reached, meaning that votes need to be counted, which might change the proposals status (e.g. approved, rejected, scheduled), and the phase in which a proposal is in. For instance, an approved SIP will move to the approval phase, which means that the SIP will be known to the approval state.
- an epoch changes, which might cause a scheduled update proposal to become active, and therefore become the current version of the blockchain protocol.

```
tick
  :: ( TracksSlotTime env
     , HasStakeDistribution env (VoterId sip)
     , HasStakeDistribution env (VoterId impl)
     , HasStakeDistribution env (EndorserId (Protocol impl))
     , HasAdversarialStakeRatio env

     , Proposal sip
     , Implementation sip impl
     )
  => env -> State sip impl -> State sip impl
```

The `apply` function applies a certain payload to the given state. It also requires an environment that:

- contains a slot number, which is used to determine at which slot the payload was applied. This is required to:
  - implement the commit-reveal scheme of proposals, since a reveal must be submitted at least $2k$ slots after its corresponding commit (where k is the maximum number of blocks that the chain can rollback).
  - register a vote, which must occur in a slot in which the voting period for proposal being voted is open.
- has a voting period cap, which specify the maximum number of voting periods, this is used to determine whether a proposal still has a voting period.

```
apply
  :: ( TracksSlotTime env
     , HasVotingPeriodsCap env
     , Proposal sip
     , Implementation sip impl
     )
  => env
  -> Payload sip impl
  -> State sip impl
  -> Either (Error sip impl) (State sip impl)
```

The internals of the update state are not exposed. This contributes to making the code more maintainable, since changes in the internal representation of the update state and its sub-components do not affect the clients of the module. The update module includes functions for performing *queries on the update state*. For instance:

- is a proposal (stably) submitted?
- is a proposal approved, rejected, or expired?

– is a proposal being endorsed?

– what is the current protocol version?

The update system is polymorphic on the proposal type. This allow us not only to achieve a great level of decoupling w.r.t. the other components of the ledger, but it also help us in running the tests faster, since we can mock expensive operations like computing hashes, signing data, and verifying signed data.

Additionally, by making the proposals type abstract, we achieve a simple design of the update system since we include only the details that are essential to the protocol. For instance a concrete proposal might contain information like URL's and proposal description, which are irrelevant to the protocol. Including such details in the update module is not a good design since it includes additional superfluous details, and forces us to make a decision on their concrete representation. To make things worse, when generating test data to be used in property based tests, we would need to generate this irrelevant data making the test slower and more complex.

Instances of the `Proposal` (type-)class must define:

– a submission and revelation associated (data) types (see `Submission proposal` and `Revelation proposal` in figure 3.7);

– a function to extract a commit for the revelation (`revelationCommit`);

– a function to extract the proposal from the revelation (`proposal`);

– a function to obtain the voting period duration (`votingPeriodDuration`);

– a vote and voter types (`Vote proposal` and `Voter proposal`);

– a function for getting the voter id of a vote (`voter`);

– a function for obtaining from a vote the candidate proposal for which a vote is casted;

– a function for obtaining the voter's confidence;

Furthermore, the proposal and its associated types must satisfy the following constraints:

– we should be able to compute a commit on revelation values (`Revelation proposal`);

– the proposal submission and votes must be signed (`Signed(Submission proposal)` and `Signed (Vote proposal)`);

– we should be able to compute an id for proposals and voters (`Identifiable (proposal)` and `Identifiable (Voter proposal)`).

The type of commits and ids are abstract, in a concrete implementation these would be hashes, but in the tests we can choose simpler types, like unsigned short integers, that can be quickly and easily computed. Since at testing time we control all the data, we can easily generate unique commits and identifiers for it.

A proposal can in turn be an `Implementation`. An instance of the implementation class must define:

– a predecessor proposal type (`sip`);

– a function to get the proposal that the implementation implements (`preProposalId`);

– a function to query the implementation's type (`implementationType`). An implementation type can be a cancellation, a protocol update, or an application update;

– a protocol associated type (`Protocol`). The update system must be able to activate the implementation's protocol. The `Activable (Protocol impl)` constraint makes sure the protocol type associated to the implementation type has all the information the update protocol requires. For instance: the protocol must have a version number that forms a total order. The update mechanism requires this since it relies on the ordering of protocol versions.

– an application associated type (`Application`). The `Implementation` class only requires that the applications have an id, since they are stored in a set of updated applications.

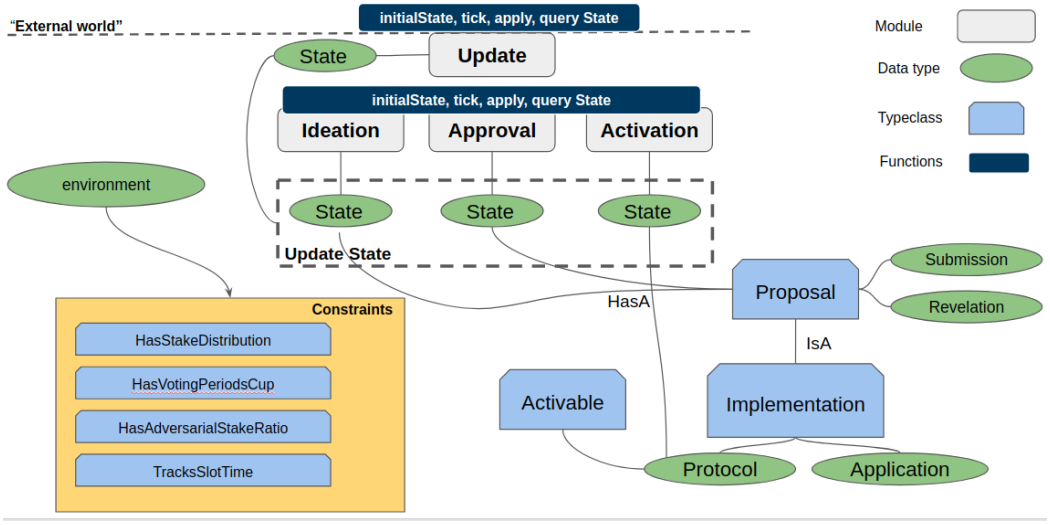

Figure 3.7: The update system software architecture.

Figure 3.7, shows a very simplified view of the inner module structure of the update system. The update module exposes a core set of functions (API) to the "external world" that implement the "update logic". These are the `initialState`, `tick` and `apply` functions discussed above and also a set of functions for querying the update state. Essentially each invocation of one of these three functions updates the internal (update) state. This state can be accessed via the query interface.

The update module dispatches the update payload to any of the ideation, approval, and activation modules, depending on the payload type. These modules correspond to the respective phases in the lifecycle of a software update depicted in figure 3.5 and expose the same set of functions as the update module, specialized to the specific payload. The approval module depends on the ideation module to provide information about which SIP's have been approved and can proceed to the approval phase. The activation module relies on the approval module to obtain information about which implementations can proceed to the activation phase. All these module have their own private state, which can be accessed only through their interfaces to it. The update state includes these three states.

All states require that the data stored corresponds to instances of the `Proposal` typeclass. The `Implementation` typeclass, depicted in the figure, is a further elaboration of the `Proposal` typeclass specific to implementation proposals (a.k.a. UPs). As you can see, an implementation proposal needs to implement either a `Protocol` change, or an `Appplication` change. With the latter, we mean a software update that does not impact the consensus protocol. Moreover, as discussed above, the protocol changes in particular, must be also instances of the `Activable` typeclass. This enables certain attributes to the software update that have to do with the synchronization (endorsement period) prior to the activation of the change, which is imperative for protocol updates, in order to avoid chain splits.

Finally, all the exposed functions from the modules depicted, require an *environment* input parameter. This can correspond to any environment data type, as long it honors a specific set of *constraints*, expressed through

a group of typeclasses, depicted in the figure. In particular,the ideation and approval `State` modules make use of the `HasStakeDistribution`, `HasVotingPeriodsCap`, `HasAdversarialStakeRatio`, and `TracksSlotTime` type classes for registering and tally votes. Since the activation state only deals with endorsements, it only uses the assumption that the environments passed to it tracks the slot time.

Chapter 4

Toolkit for Flexible Consensus in Hyperledger Fabric

This section covers architectural extensions to the Toolkit for flexible consensus in Hyperledger Fabric, described initially in Deliverable D4.1. The toolkit on *Flexible consensus in Hyperledger Fabric* builds on research performed within Work Package 3. The description of the specific protocols will be provided in D3.3. For completeness, we include the summary of information provided in D4.1 relevant to this toolkit.

4.1 Introduction to Consensus in Hyperledger Fabric

Generic ledger architecture

Distributed ledger systems, including Hyperledger Fabric, consist of several different components, such as a consensus mechanism that determines the order of transactions, a distributed ledger (policy and format) mechanism that decides which transactions are included in the distributed ledger, or the transaction protocol itself, which specifies which messages comprise valid transactions and how they affect the *world state*, i.e. the state of the distributed database implemented by the distributed ledger. This generic architecture is depicted in Figure 4.1.

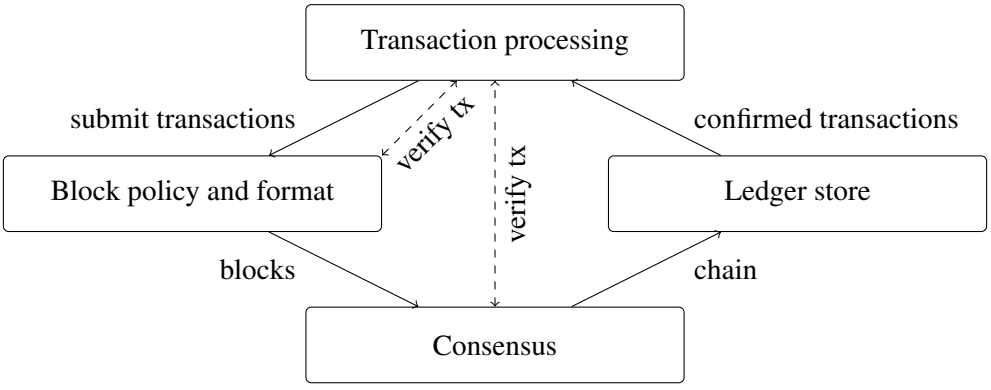

Figure 4.1: Generic architecture of a secure distributed ledger system. The dashed arrow indicates that the connection may not appear in all types of systems.

The foundational component of every distributed ledger system is the *consensus* mechanism. The main role of consensus is to determine the next block of transactions that is to be written to a distributed ledger. The consensus component itself treats the blocks as a black box; the only goal is to determine which candidate block will be appended to the chain.

There are different trust assumptions and mechanisms that consensus protocols for distributed ledgers can be built on. The traditional (i.e., pre-blockchain) line of work on consensus considers Byzantine fault tolerant (BFT) protocols, where the group of participants is fixed and known in advance, and nodes can communicate authentically. This type of protocol is used in the *permissioned* setting where the ledger is distributed among a fixed set of organizations. In such a setting, parties are usually authenticated by digitally signing their votes in the consensus protocol. Hyperledger Fabric belongs to this class of permissioned blockchains.

Consensus interface The interface offered by the consensus component to higher-level protocols receives as input blocks that are composed from the local view of a party and are meant to be agreed upon by the consensus participants. The output of the consensus components consists of blocks that have been agreed upon by the consensus and are ready for processing by the higher-level components.

The consensus protocol may keep state (such as the set of current protocol participants) and may furthermore call out the transaction processing component for verification of the transactions contained in a newly received blocks. In a permissioned setting, writing to the ledger can be restricted to eligible parties that can be held accountable.

Architecture Implementation in Hyperledger Fabric.

Hyperledger Fabric[1] is a permissioned blockchain platform that targets enterprise applications, developed under the umbrella of the Linux Foundation. Fabric is among the most actively developed enterprise blockchain platforms and focuses on flexibility: it supports different consensus mechanisms, and it supports smart contracts (dubbed *chaincode*) that is written in different widely used programming languages such as Go, Java, or JavaScript. Since Fabric v1.0, the network allows to specify for each node participating in the network its dedicated roles as depicted in Figure 4.2, i.e. whether it acts as a node participating in consensus (so-called *orderers*), or as a node execution specific chaincode (so-called *peers*), or as a *client* that merely invokes transactions or listens to events.

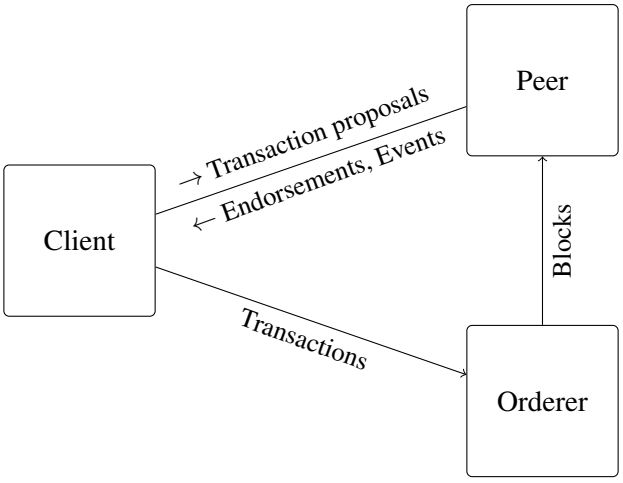

Figure 4.2: Hyperledger Fabric distinguishes different roles. *Clients* invoke transactions; *peers* execute smart contracts and store their state; *orderers* run a consensus protocol to determine the order of transactions.

Hyperledger Fabric has a modular architecture in which the *ordering service* can be implemented based on different types of consensus protocols. The ordering service receives transactions from the clients, orders them and puts them into blocks, and then distributed those blocks to all peers in the network. The ordering service,

[1] `https://www.hyperledger.org/projects/fabric/`

therefore, offers two types of APIs that we will describe in the following, one towards the clients, and one towards the peers.

Consensus in Hyperledger Fabric One of the main design principles of Fabric is its *modular* consensus architecture. The goal of consensus in Fabric is specified as ordering the transactions, without validating the contents of the transaction. The component is therefore mostly referred to as *ordering service*. As a permissioned blockchain platform, Fabric strives to support different (small or large) deployments for diverse use cases. Each use case comes with specific trust assumptions that parties are willing to make for the ordering service. As Fabric strives to support the majority of use cases, a modular or *pluggable* consensus architecture is needed to fulfill that goal.

With Fabric v1.0, two ordering service implementations were provided in the main distribution: *solo* and *kafka*. The *solo* version implements a trivial type of consensus: a single node is used to process all blocks. This implementation is not meant for production purposes and is mostly used for development. The *kafka* implementation is based on Apache Kafka[2], a distributed streaming platform that follows the publisher/subscriber paradigm and offers crash-fault tolerant (CFT) operation based on the Apache ZooKeeper[3] CFT consensus system. In terms of trust, the kafka implementation is also centralized; a single malicious consensus node can attack the integrity of the system. Kafka improves the reliability of the system over solo ordering, but does not change the security of the system with in presence of malicious nodes. The subsequent release Fabric 1.4.1 added support for the Raft consensus protocol based on etcd[4], another CFT consensus system. Just like kafka, it does not improve the resiliency against misbehaving nodes. (The system tolerates malicious clients transmitting transactions and malicious peer nodes executing smart contracts up to a level specified in the contract policies.). From Fabric v2.0, solo and kafka are deprecated in Fabric, and only raft is recommended as Fabric ordering service.

However raft does not provide the resilience necessary in use cases where there is no single party that can be trusted to properly perform the ordering. An experimental Byzantine-fault tolerant ordering service based on the BFT-SMaRt implementation has been described by Sousa, Bessani, and Vukolić [SBV18]. The paper showed that while the performance on the system depends on the number of nodes and the size of transactions, the BFT ordering service, when deployed on 10 nodes, can handle tens of thousands of transactions per second and is unlikely to be the bottleneck in a Fabric deployment, as it exceeds the number of transactions a node can verify [And+18]. An implementation of a BFT ordering service that is planned[5] to be included in the main distribution of Fabric is currently under development by IBM Research – Zurich, and the protocol, called Mir-BFT, will be described in Deliverable D3.3. The mechanism of Mir-BFT are based on the PBFT protocol [CL02], but targeting improved transaction throughput. We detail the main architecture principles of Mir-BFT in Sections 4.2 and Section 4.3.

Implementation of the ordering service. The main consensus protocol is executed on the ordering service nodes. The toolkit will implement the consensus protocol, including the necessary communication between different ordering service nodes. For integration with the Fabric infrastructure, an interface dubbed `ConsenterSupport` is provided to the consensus algorithm. In a nutshell, the "shell" of an ordering service node is independent of the consensus mechanism that it runs, and it provides a callback interface to the consensus mechanism. This way, the implementation of the consensus mechanism can be independent of other parts of the system, such as the policy methods for generating blocks, or the network protocols used to disseminate completed blocks to the entire network.

The `ConsenterSupport` interface in particular specifies a so-called *block cutter* that implements the policy service that determines which transactions will be included in the next block. (This *policy* is thereby separated from the consensus *mechanism*.) The interface also allows to access a shared configuration that contains

[2] `https://kafka.apache.org/`
[3] `https://zookeeper.apache.org/`
[4] `https://coreos.com/etcd/`
[5] `https://jira.hyperledger.org/browse/FAB-33`

parameters controlling the behavior of the consensus algorithms; these are specific to each consensus method but read and provided by the main ordering service software.

The interface provides further calls that allow the consensus mechanism to deliver the next block that was decided by the consensus mechanism, In particular, the call `WriteBlock(...)` writes the new block to the disk of the ordering service node; from there it will be distributed to all peers in the network. (This mechanism does not depend on the consensus implementation used.)

Implementation of the client interface. Clients submit transactions (that contain endorsements by peers) to the ordering service. As the exact method of submission may depend on the consensus method in place, such as whether the transactions have to be sent to a single or to multiple nodes, Fabric specifies an interface through which the interaction with the ordering service takes place, which can be used by clients and must be implemented by the consensus mechanism.

The main command of this interface is `Order(...)`, which takes as parameter the *envelope*, a data structure that contains all the transaction data. Additional parameters ensure that the client sends the transaction to the channel in the expected configuration (as the channel configuration can change, such as when organizations join or leave the network). In short, a channel in Hyperledger Fabric is a ledger shard which is replicated across all peers in the given channel. Peers in turn belong to organizations, which define trust domains (all peers belonging to a single organization trust each other).

This interface is implemented on the client side; calling `Order(...)` will then compile the actual network-level message that is sent to the ordering service nodes, using a protocol that may be specific to the consensus implementation.

Other methods specified in the interface `Configure(...)`, which allows to send a special blockchain-reconfiguration message to the ordering service, which allows to, e.g., add a further organization to the blockchain or change other parameters such as the target wait time for each block. The calls `WaitReady` and `Errored` allow the client to observe the state of the ordering service, and `Start` and `Halt` allow to initiate or terminate the network connection between the client to the ordering service.

Implementation of a consensus mechanism. The exact internal software design of the component depends on the work on consensus protocols in WP3, some of which are close to being finalized and will be included in D3.3. In the following, we described the main architectural principles of Mir-BFT (see also [SDV19]) which allows for the flexible and dynamically adaptive number of leaders in a BFT protocol, building on PBFT [CL02]

4.2 Flexible number of leaders in a consensus protocol

The defining feature of the flexible consensus in Hyperledger Fabric will be to use *robust* consensus protocols with *multiple, parallel* leaders. The flexibility here is with respect to number of leaders - classical consensus protocols such as PBFT [CL02], are inflexible in their using single leader at the time.

Recently, many blockchain proposals aimed at addressing scalability issues in BFT has been allowing multiple nodes to act as parallel leaders and to propose batches independently and concurrently, either in a coordinated, deterministic fashion [CNG18; MBS13], or using (inherently more symmetric) randomized protocols [Mil+16]. With multiple leaders, the CPU and bandwidth load related to proposing batches are distributed more evenly. However, the issue with this approach is that parallel leaders are prone to wasting resources by proposing the same duplicate requests in parallel. Request duplication is difficult to avoid as a BFT protocol needs to enable more than one leader to include a particular request in its batch in order to address potential *request censoring attacks* by Byzantine leaders which would violate liveness of the BFT protocol. In corner cases, with up to n leaders, request duplication attacks may induce an n-fold duplication of every single request and bring the effective throughput to its knees, practically voiding the benefits of using multiple leaders.

For this reason, the key architectural novelty in our toolkit is to allow a set of leaders to propose request batches independently (i.e., parallel leaders), in a way that precludes *request duplication*. This is depicted in

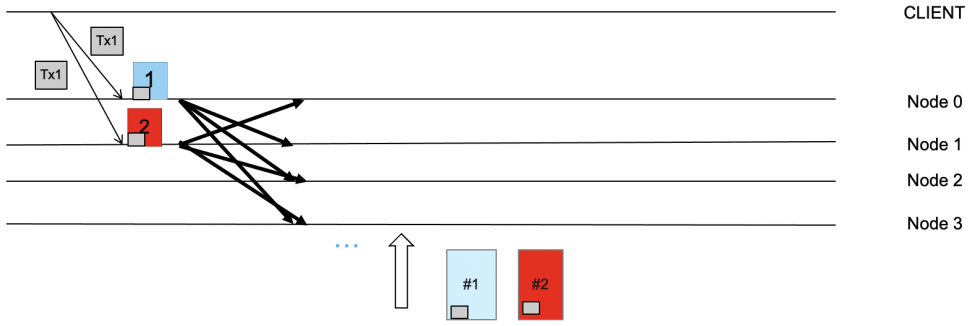

Figure 4.3: Transaction duplication with multiple leaders in leader-based consensus protocols.

Figure 4.3.

To achieve this, as the main architectural novelty, our toolkit partitions the request hash space across replicas, preventing request duplication, while rotating this partitioned assignment across protocol configurations/epochs, addressing the request censoring attack. This is explained in more details in the following.

4.3 Architecture of request duplication prevention with multiple leaders

Moving from single-leader consensus to multi-leader consensus poses the challenge of request duplication. A simplistic approach to multiple leaders would be to allow any leader to add any request into a batch/block ([MBS13; CNG18]), either in the common case, or in the case of client request retransmission. Such a simplistic approach, combined with a client sending a request to exactly one node, allows good throughput with no duplication only in the best case, i.e., with no Byzantine clients/leaders and with no asynchrony.

However, this approach does not perform well outside the best case, in particular with clients sending identical request to multiple nodes. A client may do so simply because it is Byzantine and performs the *request duplication* attack. However, even a correct client needs to send its request to at least $f + 1$ nodes, where f is the threshold of faulty nodes (i.e., to $\Theta(n)$ nodes, when $n = 3f + 1$), in the worst case in *any* Byzantine Fault Tolerant (BFT) consensus protocol, in order to avoid Byzantine nodes (leaders) selectively ignoring the request (*request censoring* attack).

Therefore, a simplistic approach to parallel request processing with multiple leaders [MBS13; CNG18] faces attacks that can reduce throughput by factor of $\Theta(n)$, nullifying the effects of using multiple leaders.

Buckets and Request Partitioning. To cope with request censoring attacks, our design partitions the request hash space into buckets of equal size (number of buckets is a system parameter) and assigns each bucket to exactly one leader, allowing a leader to only propose requests from its assigned (*active*) buckets (preventing request duplication). For load balancing, we distribute buckets evenly (within the limits of integer arithmetics) to all leaders in each phase of consensus protocol. To prevent request censoring, our design makes sure that every bucket will be assigned to a correct leader infinitely often. We achieve this by periodically redistributing the bucket assignment (bucket rotation). This will be explained in more details in D3.3 (preliminary version of the full protocol is available in [SDV19]).

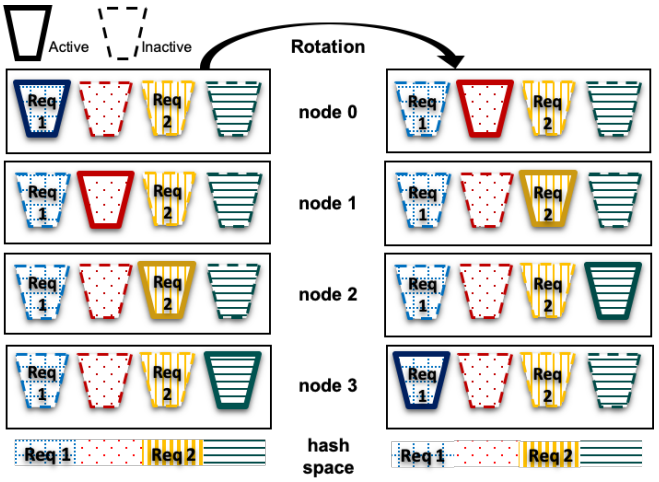

Figure 4.4: Request mapping with $n = 4$ nodes (all nodes are leaders): Solid lines represent the active buckets. Req. 1 is mapped to the first bucket, first active in node 1. Req. 2 is mapped to the third bucket, first active in node 3. Rotation redistributes bucket assignment across leaders.

Chapter 5

Toolkit for Ledger-Oriented Secure Two/Multi-Party Computation

The toolkit presented in this document can be used to enhance secure two/multi-party computation protocols exploiting features of distributed ledgers. The starting point is the existence of various libraries for secure two/multi-party computation in the traditional setting, where players are connected through point-to-point connections using TCP/IP. This setting is not ideal when players would like to run a computation publicly, so that everyone can observe who did what and when. Moreover the traditional setting requires coordination among players that are possibly required to be online all the time during the computation knowing each other IP addresses. This might be in particular non-trivial to realize when the number of players is large and heterogeneous. Last but not least, revealing directly the IP address to others might in some case be considered already a leak.

The toolkit aims at adding the support of distributed ledger technology in order to mitigate at least in part the above problems, therefore allowing the use of traditional secure two/multi-party computation libraries offering at least partially a public verifiability of the played messages and possibly allowing players to participate to the computation whenever convenient for them (i.e., without requiring players to be all simultaneously online communicating with known IP addresses).

The design of this toolkit consists of two components. The first component is designed by UNISA and is a module that offers an interface to safely read and write on a ledger. This module adds value to current libraries used to read/write on ledgers. First of all, it abstracts the use of a ledger allowing to securely run on a generic ledger a communication that was originally designed to run on point-to-point channels. With additional measures, such as proper message authentication and tagging, the original point-to-point communication can be made publicly verifiable. This could enable the proper management of protocol abortion, for example. Furthermore, the module could prevent security issues as a result of fork.

The module will support at least one popular forking ledger and will offer features to maintain the security of the computation in presence of forks without penalizing excessively the performance. Indeed the design of this module will exploit research results achieved in WP2/WP3 and presented in [Bot+19]. In that paper, the authors (two of which are affiliated with UNISA) prove that, in the presence of forks, quick players (i.e., players that do not wait for confirmation before posting new transactions) run the risk that the result or the privacy of the computation will be polluted by an adversary.

The above abstract interface that allows the safe use of multiple ledgers will be exploited by a software layer that by working as a bridge between the two/multi-party computation software and the ledger interface will properly and safely redirect the communication to the ledger.

This first module strongly focuses on modularity so that multiple two/multi-party computation protocols can benefit from these features and multiple ledgers can be integrated without requiring major changes.

The second module is designed by TUE and consists of an extension of MPyC [Sch18], a library for honest-but-curious secure multi-party computation in order to add verifiability of correct computation through zk-SNARKs, and is discussed in the rest of this chapter.

5.1 Introduction

This chapter describes the architecture of new a ledger-oriented MPC toolkit based on the MPyC framework [Sch18]. The main two components are a *verifiable MPC* protocol and a threshold cryptosystem with extended functionalities, referred to as *extended threshold cryptosystem*.

Combining these two components with a bulletin board or smart contract-capable blockchain makes the whole larger than the sum of its parts. We propose a scheme that combines these components into a *privacy-preserving distributed computer*: a distributed computer where clients can task a set of workers to compute a state update function on batched public and private inputs. State updates are publicly verifiable.

Roadmap. Section 5.2 provides an overview of the architecture and introduces notation. Section 5.3 explains at high-level how the scheme can be applied to two use-cases. Section 5.4 introduces the verifiable MPC functionality. Section 5.5 introduces the extended threshold cryptosystem. Section 5.4.1 introduces how to instantiate the verifiable MPC protocol. Section 5.5.1 introduces how to implement verifiable MPC using a new technique from [AC20] referred to as compressed Σ-protocols.

5.2 Overview

Suppose a setting with a bulletin board and a set of agents, referred to as *clients*, that are permitted to post data to the bulletin board. Clients wish to apply a public function to data referenced on the bulletin board by delegating the computation to a selected set of computation agents. Furthermore, values on the bulletin board can be plaintext or ciphertext. Ciphertext values are encryptions associated with keys held by clients. Public values may include commitments to inputs associated with commitment keys and private inputs held by clients. Verified, published outputs of computations initiated by clients are viewed as state updates of a distributed computer. We refer to this setting as *privacy-preserving distributed computer*.

The computer is instantiated using a bulletin board, an *extended* threshold cryptosystem and a *verifiable MPC* protocol. The bulletin board hosts (optionally, time-stamped) inputs, outputs of state update functions, definitions of permitted state updates and proofs of correct state updates. The verifiable MPC protocol provides the function to securely and verifiably compute the state update. The extended threshold cryptosystems facilitates secret inputs and outputs to these computations.

Let $\mathcal{B}$ denote the bulletin board, let $\mathcal{C}$ denote the set of clients and let $\mathcal{P}$ the set of computation agents, referred to as MPC parties.

Bulletin Board. $\mathcal{B}$ abstractly encompasses two operations: 1) A post operation involving users who make post requests, and a subsystem of item collectors (ICs) that receive and store the submitted elements. 2) A publish operation, where the IC subsystem publishes the stored elements on an audit board (AB) from where any party can read. Depending on the implementation, the IC and the AB could be distributed or centralized.[1]

State. Given a plaintext a, client $\mathcal{C}_u$ can post plaintext a, encryption $E(a)$ and/or commitment $C(a)$ of plaintext a. Denote the published state of $\mathcal{B}$ at time τ by S_τ, i.e. S_τ is a collection of plaintexts, ciphertexts and/or commitments at time τ.

State Updates using Verifiable MPC. At time τ, any client $\mathcal{C}_u \in \mathcal{C}$ is permitted to update the global state by applying a function f on a subset of state elements $A \subseteq S_{\tau-1}$. For simplicity, we assume only one state update is done at time τ, but multiple updates are permitted without loss of generality. At time τ, client $\mathcal{C}_u$ selects a set of m MPC parties $\mathcal{P}^{(\tau)}$. Note that clients can dynamically select MPC parties to compute each state update. To simplify notation, we write $\mathcal{P}^{(\tau)} = \{\mathcal{P}_1, ..., \mathcal{P}_m\}$, dropping $^{(\tau)}$ from the notation for individual parties.

[1] Abstraction taken from [Kia+18].

To perform the state update on inputs $A \subseteq S_{\tau-1}$, $\mathcal{C}_u$ defines a state update (batch job) $\sigma_\tau := (\mathcal{C}_u, f, A, \{\mathcal{P}_1, ..., \mathcal{P}_m\})$, posts σ_τ to $\mathcal{B}$, and delegates the computation of $f(A)$ to the specified MPC parties. The parties compute the result securely, as long as at most t out of m parties are corrupt, and post on time τ the (encrypted) result, b_τ, and a *publicly verifiable* proof of correctness, π_{σ_τ}, to $\mathcal{B}$. $\mathcal{B}$ publishes the result to the global state S_τ, optionally only after verification of the result.

Secret Inputs and Outputs using an Extended Threshold Cryptosystem. State updates can take any of the following inputs:

- Cleartexts included in the global state of $\mathcal{B}$;
- Ciphertexts included in the global state;
- Cleartexts held or (randomly) generated by a client, not in the global state;
- Ciphertexts held by a client, not in the global state.

Assume we have access to a so-called *extended threshold cryptosystem* that provides the protocols to securely convert ciphertexts to secret shares, ciphertexts to new ciphertexts associated with different public keys, or secret shares to ciphertexts. (We refer to Section 5.5 for the description of this cryptosystem.)

For example, suppose client $\mathcal{C}_u$, holding private key x_1 corresponding to public key h_1, wishes to perform state update $\sigma_\tau := (\mathcal{C}_u, f, A, \{\mathcal{P}_1, ..., \mathcal{P}_m\})$, where A contains ciphertext $E_{h_1}(a)$. To securely convert this ciphertext to secret shares, $\mathcal{C}_u$ sends secret shares $[\![x_1]\!]$ to the MPC parties $\mathcal{P}_1, ..., \mathcal{P}_m$. Then the MPC parties apply the extended threshold cryptosystem to securely compute shares $[\![a]\!]$ from $E_{h_1}(a)$.

The drawback of this example is that dishonest MPC parties may reconstruct private key x_1, which then gives them the ability to decrypt other ciphertexts associated to (h_1, x_1). To address this problem, assuming that $\mathcal{C}_u$ holds a second, ephemeral (single use) keypair (h_2, x_2), $\mathcal{C}_u$ could reencrypt $E_{h_1}(a)$ to $E_{h_2}(a)$, distribute the shares of ephemeral key x_2 to the MPC parties, and post $E_{h_2}(a)$ together with a zero-knowledge proof that $E_{h_2}(a)$ corresponds to $E_{h_1}(a)$.

Alternatively, $\mathcal{C}_u$ may open the random nonce u used to form the encryption $E_{h_1}(a)$ and distribute secret shares $[\![u]\!]$ to the MPC parties. This approach also works if only a commitment to a is available.

Another aspect is that a client $\mathcal{C}_u$ may cheat when distributing shares of an input a. Publicly Verifiable Secret Sharing (PVSS) may be used to stop this type of cheating. Note that PVSS also covers encryption of the shares of a for the MPC parties, and there is no need for the client and MPC parties to be online at the same time.

Furthermore, the threshold cryptosystem supports the encryption of the output of a secure computation, $[\![b]\!] \leftarrow f([\![A]\!])$, under a new public key, h_3, allowing the MPC parties to compute $E_{h_3}(b)$, which can then be posted to the bulletin board if needed.

Public Verifiability. The bulletin board can publish output and proof b, π_{σ_τ} corresponding to computation σ_τ directly to the global state, or only after verification of a cryptographic proof π_{σ_τ} corresponding to σ_τ. In case of a smart contract-capable blockchain, a contract can replace bulletin board $\mathcal{B}$ and provide the functionalities to publish the updates and verify proofs. Using a smart contract permits *honest-verifier zero-knowledge proofs* to be verified once, and trusted permanently by the public.

Secret Sharing and MPC. For our MPC framework, we use the BGW protocol [BGW88] with Shamir secret shares. Let $[\![a]\!]$ denote a Shamir secret sharing for any finite field element $a \in \mathcal{F}$. For a given computation σ_τ, let m denote the number of MPC parties and let t denote the maximum number of parties that can be adversarial, where $t < m/2$.

5.3 Use-cases

Health Insurance. PRIViLEDGE use case 2 (UC2) provides the means for a "pay for performance" insurance model, where the payments that providers receive depend on achieving certain measurable outcomes. For this to be feasible, it is necessary that all health care providers participating in a so-called "accountable care organization" (ACO) share the same view of patient records, that patient records are kept private, that the integrity of medical histories is not compromised, that care providers are accountable for records they pass among themselves, and that reported metrics are consistent. This toolkit is a component to meet these requirements [Lou20].

Using the definitions of the privacy-preserving distributed computer, insurers are *clients* and care providers (members of an ACO) are both *client* and *computation agent*. Care providers (in their role as client) are permitted to post cryptographic commitments of events (corresponding to updates to patient records). Each event, a, is a tuple consisting of patient ID, event type and zero or more attributes.

For a performance report at time τ, insurer $\mathcal{C}_u$ defines the aggregate values (outputs) agreed to in the care provisioning contract for the ACO by a function f on input A. A is a union of m sets $A_1, A_2, \ldots, A_m$ corresponding to the inputs of m care providers. At time τ, the insurer $\mathcal{C}_u$, defines the batch job $\sigma_\tau := (\mathcal{C}_u, f, A, \{\mathcal{P}_1, ..., \mathcal{P}_m\})$, where $\{\mathcal{P}_1, ..., \mathcal{P}_m\}$ corresponds to the care providers acting as computation agents. In this example, clients only store commitments C_i to A_i for $i \in \{1, \ldots, m\}$ on the bulletin board.

$\mathcal{C}_u$ then delegates to the members of the ACO a verifiable MPC protocol to compute $f(A)$ with a zero-knowledge proof of correct computation of $f(A) = b$ referring to the commitments on the bulletin board. One of the care providers is tasked with posting the proof to the bulletin board.

Billionaires' Problem. The Billionaires' Problem extends Yao's Millionaires' Problem [Yao82; Yao86], by requiring that any external party can verify who is richer without actively taking part in the secure computation. It is defined as follows: Given everybody's committed tax return statements on a blockchain, produce a list of the top 400 billionaires world wide including a proof of correctness, without leaking any information of the people who do not appear on the list. The net-worth for the 400 richest people may be revealed, but that's not necessary. Solving the Billionaires' problem requires the scheme to certify inputs and outputs of an MPC protocol [Seg20].

5.4 Verifiable MPC

Verifiable MPC extends the idea of actively secure MPC protocols. Recall that in actively secure MPC protocols, one honest MPC party is required to detect malicious behavior by other MPC parties. Verifiable MPC instantiated with non-interactive zero-knowledge proofs allows anyone, particularly someone external to the secure computation, to check the correctness of the output, while preserving the privacy properties of the MPC protocol.

A verifiable MPC protocol requires a cryptographic setup, which can be trusted or untrusted and support specific or general circuits. The verifiable MPC computation takes as input a cryptographic commitment or public key encryption of the input values. The output consists of the result of the MPC computation plus a cryptographic proof of correctness of this result.

Verifiable MPC in conjunction with a bulletin board permits several new functionalities:

- Storing commitments immutably on a bulletin board provides a function to permanently and publicly verify the inputs to a computation;
- Storing public inputs on a bulletin board provides a function to refer computations to these inputs;
- Storing tuples defining the function, identities of MPC parties, output parties and (references to) inputs to a specific computational job permit authenticated participation in the computation.

Verifiable MPC in conjunction with a smart contract-capable blockchain permits additional functionalities:

- Smart contracts verifying the (non-interactive) proof of correctness and posting the verification output bit on-chain provides a function that enables the public to efficiently trust the computation by only verifying this bit;
- Smart contracts authenticating the client provides an extra (optional) level of access control of who is permitted to compute state updates;
- Smart contracts holding deposits of MPC parties that can be slashed if adversarial behavior is detected.

We define a *verifiable MPC* scheme as follows: Let $\mathbb{G}$ be a cyclic group of prime order q with generator $g_0 \in \mathbb{G}$. A verifiable MPC scheme consists of the following protocols:

Setup. Given generator g_0, a circuit f with f_{in} input gates and f_{mul} multiplication gates, generate generators $g = (g_1, \ldots, g_n) \in \mathbb{G}^n$ such that for all users (clients and MPC parties) finding nontrivial linear relations between them is computationally as hard as computing discrete logarithms in $\mathbb{G}$. The value n is linear in f_{in} and f_{mul}.[2]

Prove. Given input $[\![a]\!]$, nonce $[\![\gamma]\!]$ for $\gamma \in_R \mathbb{Z}^*$, Pedersen (vector) commitment $C(a, \gamma)$, output b and circuit f, compute zero-knowledge proof π of correct evaluation of $f(a) = b$ for relation $(g, C(a, \gamma), f, b; a, \gamma)$.

Verify. Given proof π and knowledge of the statement $(g, C(a, \gamma), f, b)$, verify in zero-knowledge the correct evaluation of $f([\![a]\!]) = b$.

Ongoing research is to extend the above scheme with the following protocols:

Prove Secret Output. Given input $[\![a]\!]$, nonce $[\![\gamma]\!]$ for $\gamma \in_R \mathbb{Z}^*$, Pedersen commitment $C(a, \gamma)$, secret output $[\![b]\!]$ and circuit f, compute zero-knowledge proof π of correct evaluation of $f(a) = b$.

Prove Encrypted Input. Given public-private keypair $(h_1, [\![x_1]\!])$, encrypted input $E_{h_1}(a)$, public output b and circuit f, compute zero-knowledge proof π of correct evaluation of $f(a) = b$ for relation $(g, E_{h_1}(a), f, b; a, \gamma)$.

Prove Encrypted In-/Output. Given keypairs $(h_1, [\![x_1]\!])$ and $(h_2, [\![x_2]\!])$, encrypted input $E_{h_1}(a)$, encrypted output $E_{h_2}(b)$ and circuit f, compute zero-knowledge proof π of correct evaluation of $f(a) = b$ for relation $(g, E_{h_1}(a), f, E_{h_2}(b); a, \gamma)$.

5.4.1 Instantiating Verifiable MPC

Trinocchio. The first publicly verifiable MPC protocol [SVV16] is based on the well-known Pinocchio zk-SNARK [Par+13]. The zk-SNARK used in this scheme requires a trusted setup that is specific to the computation. Geppetri, its successor, extends Trinocchio in several ways. First, Geppetri enables efficient proofs that can refer to committed data. Second, Geppetri allows different verifiable computations on that data while partially reusing the trusted setup using adaptive zk-SNARKs. after generating the CRS, one more trapdoor-generating step is required to generate key material for the specific function to be computed. With knowledge of the latter trapdoor, an attacker could generate false proofs for that specific instance, but not for other computations on the committed data in general.

Compressed Σ-protocols. [AC20] presents a reconciliation of Bulletproofs [Bün+18] and Σ-Protocol theory. The main protocol is based on the discrete log assumption, has communication complexity that is linear in the circuit size and has a proof size that is logarithmic in the circuit size. The setup can be reused for circuits up to a given size and only requires that finding nontrivial linear relations between group elements (generators) is as hard as solving the Discrete Log Problem (DLP).[3]

[2] For simplicity, we take [AC20] as the basis for our scheme. A setup protocol for Gepettri [Vee17] would generate evaluation and verification keys, given a common reference string and commitment keys.

[3] [AC20] also presents a program based on the knowledge-of-exponent assumption with constant-sized proofs.

Our toolkit makes the [AC20]-protocols non-interactive by applying the Fiat-Shamir heuristic and then constructs a verifiable MPC protocol using the circuit satisfiability argument from [AC20].

5.5 Extended Threshold Cryptosystem

Completing the scheme introduced in Section 5.1 requires a functionality to convert, ad hoc, encrypted inputs from the bulletin board to secret sharess held by the compute parties. Further extending this with a functionality to securely proxy reencrypt, ad hoc, $E_{h_1}(a)$ to $E_{h_2}(a)$ permits dynamic client access to the encrypted state. Lastly, permitting encryption of a shared message under a new public key, allows MPC parties to post an encrypted result back to the bulletin board. Hence, a threshold cryptosystem with these extended functionalities provides necessary functionalities for the proposed computer.

We define an *extended threshold cryptosystem* as follows: Let $\mathbb{G}$ be a cyclic group with generator g of large prime order p. Given protocols for secure groups (see [IV20]), extend a classical $(t+1, m)$-threshold ElGamal cryptosystem [Ped91] with the following protocols:

Encryption of shared message. Given message $[\![M]\!]_{\mathbb{G}}$, the parties generate $[\![u]\!]$ with $u \in_R \mathbb{Z}_p$, and output ciphertext for public key h as the pair $(A, B) := (g^{[\![u]\!]}, h^{[\![u]\!]}[\![M]\!]_{\mathbb{G}})$.

Threshold decryption to shared message. Given ciphertext (A, B), compute $[\![A^x]\!]_{\mathbb{G}} = A^{[\![x]\!]}$. The parties compute and keep message $[\![M]\!]_{\mathbb{G}} = [\![B]\!]_{\mathbb{G}} / [\![A^x]\!]_{\mathbb{G}}$ in shares.

Proxy reencryption. Assume the parties hold shares for two private keys $[\![x_1]\!]$ and $[\![x_2]\!]$, then they may compute $[\![x_1/x_2]\!]$ and convert ciphertext (A, B) for public key $h_1 = g^{x_1}$ into ciphertext $(A^{[\![x_1/x_2]\!]}, B)$ for public key $h_2 = g^{x_2}$.

Proxy reencryption key. Assume the parties hold shares for two private keys $[\![x_1]\!]$ and $[\![x_2]\!]$, then they may compute and open $[\![x_1/x_2]\!]$ as a proxy reencryption key.

5.5.1 Implementation of Verifiable MPC using [AC20]

Sub-protocols

We focus on the new verifiable MPC protocol based on compressed Σ-protocols from [AC20]. (For details on Trinocchio, see [SVV16].) To construct a veriable MPC protocol using [AC20], the prover's computations of the circuit satisfiability protocol (Π_{cs} in [AC20, Section 6.2]) are performed using an MPC protocol.

The toolkit implements the protocols presented in [AC20] in the MPC setting, particularly:

- the "pivot"-protocol that yields zero-knowledge proofs for arbitrary linear statements;
- the "compressed pivot"-protocol to reduce the communication by using techniques from Bulletproofs [Bün+18];
- the utility protocol to combine many "nullity proofs" into one statement;
- the circuit satisfiability protocol, using the arithmetic secret sharing technique to linearize non-linear statements.

The verifiable MPC toolkit is build on the *secure groups scheme* by Schoenmakers and Segers (See [IV20] for an early draft and [SS20] for the code), which is based on the MPyC framework by Schoenmakers [Sch18]. The secure groups scheme significantly simplifies secure computation with finite groups used in cryptography. With secure groups, a software engineer can implement groups as an oblivious data structure by composing well known MPC protocols.

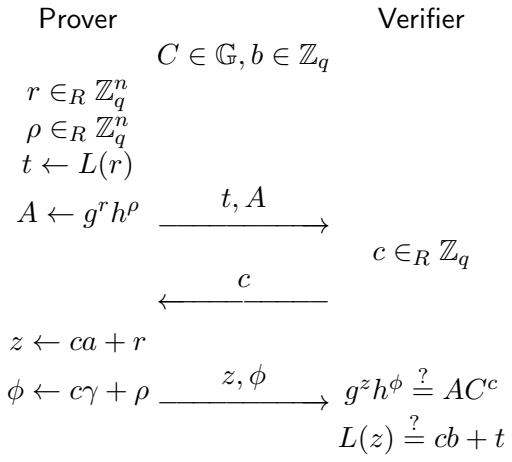

Figure 5.1: Σ-protocol Π_0 from [AC20]

Input$(g, h, C, L, b; [\![a]\!], [\![\gamma]\!])$
1: $[\![r]\!] \in_R \mathbb{Z}_q$, $[\![\rho]\!] \in_R \mathbb{Z}_q$
2: $[\![t]\!] \leftarrow L([\![r]\!])$
3: $[\![A]\!]_{\mathbb{G}} \leftarrow g^r h^\rho$
4: $t \leftarrow \mathsf{open}([\![t]\!])$ and $A \leftarrow \mathsf{open}([\![A]\!])$
5: $c \leftarrow H(t, A, g, h, C, L, b)$
6: $z \leftarrow c[\![a]\!] + [\![r]\!]$
7: $\phi \leftarrow c[\![\gamma]\!] + [\![\rho]\!]$
8: **return** $\pi = (c, z, \phi)$ ▷ s.t. $t_v := L(z) - cb, c_v := H(t, A, \ldots), c = c_v$

Protocol 1: Non-interactive argument of knowledge for opening of linear form

Making Arguments Non-interactive using Fiat-Shamir Heuristic

In our veriable MPC protocol, the prover's side of the circuit satisfiability (protocol Π_{cs} for relation R_{cs} in [AC20, Section 6.2]) is done in MPC. This is similar to earlier verifiable MPC protocols.

To implement the non-interactive Π_{cs} protocol in the MPC setting, we construct non-interactive versions for its sub-protocols: the compressed pivot Π_c and the nullity protocol $\Pi_{nullity}$. While we will not go into details for all sub-protocols here, we will provide an example for protocol Π_0, a Σ-protocol to prove validity of the opening of $L(a)$, for linear form L, input (vector) a and a commitment C of a, without revealing any additional information on a. See Figure 5.1.

Formally, given generators $g = (g_1, \ldots, g_n), h$ and a Pedersen (vector) commitment $C := C(a, \gamma)$, Π_0 is a Σ-protocol for relation $R = \{(g, h, C, L, b; a, \gamma) : C = g^a h^\gamma, b = L(a)\}$. Protocol 1 implements the prover-side of the non-interactive version of Π_0 in MPC.

After applying the Fiat-Shamir heuristic to Σ-protocol Π_0, illustrated in Figure 5.1, the proof consists of response $\{z, \phi\}$ and hash $c \leftarrow H(t, A, g, h, C, L, b)$ for cryptographic hash function H. Note that the hash includes both the announcement of the Σ-protocol, $\{t, A\}$, as well as the public part of the statement to be proven, $\{g, h, C, L, b\}$.

A public (non-designated) verifier reconstructs A_v with c by calculating

$$g^z h^\phi C^{-c}. \tag{5.1}$$

Verifier then reconstructs t_v by calculating $L(z) - cb$. It then checks $c_v \leftarrow H(t_v, A_v, g, h, C, L, b)$ and verifies if $c \stackrel{?}{=} c_v$.

Note: Recombining $t = L(r)$ leaks information about r to the MPC parties, while we want - in principle - to

leak nothing about input a in the MPC setting. However, the information being leaked is inherently leaked when proving $L(a) = b$, and therefore recombining t is harmless.

Setup in MPC

Setup for AC20. The setup can be implemented using a distributed key generation protocol: To generate a set of n new generators, parties generate $[\![r_i]\!]$ with $r_i \in_R \mathbb{Z}_p$, and compute $g_i \leftarrow g_0^{[\![r_i]\!]}$, for $i \in \{1, \ldots, n\}$, using Shamir-in-the-exponent for some public generator g_0.

Setup for Pinocchio/Trinocchio Conducting the setup for the Pinocchio (and Trinocchio) protocol, [Par+13], requires MPC-friendly elliptic curves that permit pairings. For curves to be MPC-friendly, we require the group law to be complete, i.e., it correctly computes the group law of any two points in the group without exceptional cases for specific group elements. (See for example [RCB16] for prime order Weierstass curve groups.)

5.5.2 Software

Software prototypes of the extended threshold cryptosystem and the verifiable MPC protocols are implemented using the MPyC framework [Sch18] and the secure groups scheme described in [Seg20].

Chapter 6

Conclusions

This document presented the final high-level architecture of PRIViLEDGE use-cases and toolkits. It covered two use cases and two toolkits whose architecture had evolved since Deliverables D4.1 and D4.2. These include use cases: UC1: Verifiable Online Voting with Ledgers and UC4: Decentralized Software Updates for Cardano Stake-Based Ledger, as well as toolkits for flexible consensus in Hyperledger Fabric and secure two-party and multi-party computation using ledgers.

The architecture of the remaining use cases and toolkits remains unchanged and can be found in Deliverables D4.1 and D4.2.

Bibliography

[20] *cardano.org*. https://cardano.org/. 2020.

[AC20] Thomas Attema and Ronald Cramer. "Compressed Σ-Protocol Theory and Practical Application to Plug & Play Secure Algorithmics". In: *IACR Cryptology ePrint Archive* 2020 (2020), p. 152. URL: https://eprint.iacr.org/2020/152.

[And+18] Elli Androulaki et al. "Hyperledger Fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. Ed. by Rui Oliveira, Pascal Felber, and Y. Charlie Hu. ACM, 2018, 30:1–30:15. DOI: 10.1145/3190508.3190538. URL: https://doi.org/10.1145/3190508.3190538.

[Avi19] Gennaro Avitabile. "End-To-End Verifiable Internet Voting on Permissioned Blockchains". University of Salerno, 2019.

[Bac+14] Adam Back et al. "Enabling blockchain innovations with pegged sidechains". In: (2014). http://www.opensciencereview.com/papers/123/enablingblockchain-innovations-with-pegged-sidechains.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)". In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. Ed. by Janos Simon. ACM, 1988, pp. 1–10. DOI: 10.1145/62212.62213. URL: https://doi.org/10.1145/62212.62213.

[Bot+19] Vincenzo Botta et al. "The Rush Dilemma: Attacking and Repairing Smart Contracts on Forking Blockchains". In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 891. URL: https://eprint.iacr.org/2019/891.

[Bün+18] Benedikt Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 315–334. ISBN: 978-1-5386-4353-2. DOI: 10.1109/SP.2018.00020. URL: https://doi.org/10.1109/SP.2018.00020.

[Cia+20] Michele Ciampi et al. "Updatable Blockchains". In: *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*. Ed. by Liqun Chen et al. Vol. 12309. Lecture Notes in Computer Science. Springer, 2020, pp. 590–609. DOI: 10.1007/978-3-030-59013-0_29. URL: https://doi.org/10.1007/978-3-030-59013-0%5C_29.

[CL02] Miguel Castro and Barbara Liskov. "Practical byzantine fault tolerance and proactive recovery". In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 398–461. DOI: 10.1145/571637.571640. URL: https://doi.org/10.1145/571637.571640.

[CNG18] Tyler Crain, Christopher Natoli, and Vincent Gramoli. "Evaluating the Red Belly Blockchain". In: *CoRR* abs/1812.11747 (2018). arXiv: 1812.11747. URL: http://arxiv.org/abs/1812.11747.

[GKZ18] Peter Gazi, Aggelos Kiayias, and Dionysis Zindros. *Proof-of-Stake Sidechains*. Cryptology ePrint Archive, Report 2018/1239. https://eprint.iacr.org/2018/1239. 2018.

[Hei+] Sven Heiberg et al. *Improving the verifiability of the Estonian Internet Voting scheme*. https://research.cyber.ee/~janwil/publ/ivxv-evoteid.pdf. (21.04.2020).

[Hypa] Hyperledger. *Hyperledger Fabric*. https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html\#hyperledger-fabric. (19.01.2020).

[Hypb] Hyperledger. *Peers*. https://hyperledger-fabric.readthedocs.io/en/release-2.0/peers/peers.html. (14.04.2020).

[IV20] Vincenzo Iovino and Ivan Visconti, eds. *Report on Privacy-Enhancing Cryptographic Protocols for Ledgers*. PRIViLEDGE project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020. http://priviledge-project.eu/publications/deliverables. 2020.

[Kia+17] Aggelos Kiayias et al. "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *CRYPTO 2017, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. LNCS. Springer, Heidelberg, Aug. 2017, pp. 357–388.

[Kia+18] Aggelos Kiayias et al. "On the Security Properties of e-Voting Bulletin Boards". In: *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*. Ed. by Dario Catalano and Roberto De Prisco. Vol. 11035. Lecture Notes in Computer Science. Springer, 2018, pp. 505–523. DOI: 10.1007/978-3-319-98113-0_27. URL: https://doi.org/10.1007/978-3-319-98113-0%5C_27.

[Lou20] Panos Louridas, ed. *Report on Architecture for Privacy-Preserving Applications on Ledgers*. PRIViLEDGE project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020. http://priviledge-project.eu/publications/deliverables. 2020.

[MBS13] Zarko Milosevic, Martin Biely, and André Schiper. "Bounded Delay in Byzantine-Tolerant State Machine Replication". In: *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*. IEEE Computer Society, 2013, pp. 61–70. DOI: 10.1109/SRDS.2013.15. URL: https://doi.org/10.1109/SRDS.2013.15.

[Mil+16] Andrew Miller et al. "The Honey Badger of BFT Protocols". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 31–42. DOI: 10.1145/2976749.2978399. URL: https://doi.org/10.1145/2976749.2978399.

[MIR] MIRACL. *MIRACL Core Cryptographic Library*. https://github.com/miracl/core. (19.01.2020).

[ope] opensource.com. *What is Docker?* https://opensource.com/resources/what-docker. (08.04.2020).

[Par+13] Bryan Parno et al. "Pinocchio: Nearly Practical Verifiable Computation". In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, pp. 238–252. ISBN: 978-1-4673-6166-8. DOI: 10.1109/SP.2013.47. URL: https://doi.org/10.1109/SP.2013.47.

[Ped91] T. P. Pedersen. "A Threshold Cryptosystem without a Trusted Party". In: *Advances in Cryptology—EUROCRYPT '91*. Vol. 547. LNCS. Springer, 1991, pp. 522–526.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. "Complete Addition Formulas for Prime Order Elliptic Curves". In: *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. Lecture Notes in Computer Science. Springer, 2016, pp. 403–428. ISBN: 978-3-662-49889-7. DOI: 10.1007/978-3-662-49890-3_16. URL: https://doi.org/10.1007/978-3-662-49890-3%5C_16.

[Ril] Kynan Rilee. *Understanding Hyperledger Fabric – Endorsing Transactions*. https://medium.com/kokster/hyperledger-fabric-endorsing-transactions-3c1b7251a709. (14.04.2020).

[SBV18] João Sousa, Alysson Bessani, and Marko Vukolic. "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform". In: *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 2018, pp. 51–58. ISBN: 978-1-5386-5596-2. DOI: 10.1109/DSN.2018.00018. URL: https://doi.org/10.1109/DSN.2018.00018.

[Sch18] Berry Schoenmakers. *MPyC Secure Multiparty Computation in Python*. GitHub https://github.com/lschoe/mpyc. 2018.

[SDV19] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. "Mir-BFT: High-Throughput BFT for Blockchains". In: *CoRR* abs/1906.05552 (2019). arXiv: 1906.05552. URL: http://arxiv.org/abs/1906.05552.

[Seg20] Toon Segers, ed. *Report on Design of Extended Core Protocols*. PRIViLEDGE project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020. http://priviledge-project.eu/publications/deliverables. 2020.

[SS20] Berry Schoenmakers and Toon Segers. *Secure Groups*. GitHub https://github.com/toonsegers/secure_groups. 2020.

[SVV16] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. "Trinocchio: Privacy-Preserving Outsourcing by Distributed Verifiable Computation". In: *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*. Ed. by Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider. Vol. 9696. Lecture Notes in Computer Science. Springer, 2016, pp. 346–366. ISBN: 978-3-319-39554-8. DOI: 10.1007/978-3-319-39555-5_19. URL: https://doi.org/10.1007/978-3-319-39555-5%5C_19.

[Vee17] Meilof Veeningen. "Pinocchio-Based Adaptive zk-SNARKs and Secure/Correct Adaptive Function Evaluation". In: *Progress in Cryptology - AFRICACRYPT 2017 - 9th International Conference on Cryptology in Africa, Dakar, Senegal, May 24-26, 2017, Proceedings*. Ed. by Marc Joye and Abderrahmane Nitaj. Vol. 10239. Lecture Notes in Computer Science. 2017, pp. 21–39. ISBN: 978-3-319-57338-0. DOI: 10.1007/978-3-319-57339-7_2. URL: https://doi.org/10.1007/978-3-319-57339-7%5C_2.

[Yao82] A. Yao. "Protocols for Secure Computations". In: *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*. IEEE Computer Society, 1982, pp. 160–164.

[Yao86] A. Yao. "How to Generate and Exchange Secrets". In: *Proc. 27th IEEE Symposium on Foundations of Computer Science (FOCS '86)*. IEEE Computer Society, 1986, pp. 162–167.

[ZOB18] Bingsheng Zhang, Roman Oliynykov, and Hamed Balogun. *A Treasury System for Cryptocurrencies: Enabling Better Collaborative Intelligence*. Cryptology ePrint Archive, Report 2018/435. https://eprint.iacr.org/2018/435. 2018.