Ref. Ares(2019)4138266 - 30/06/2019

DS-06-2017: Cybersecurity PPP: Cryptography

PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D4.1 – First Report on Architecture of Secure Ledger Systems

Due date of deliverable: 30 June 2019
Actual submission date: 30 June 2019

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

Lead contractor: Guardtime AS
Duration: 36 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	

D4.1

First Report on Architecture of Secure Ledger Systems

Editors
Michele Ciampi (UEDIN)
Aggelos Kiayias (UEDIN)

Contributors
Nikos Karagiannidis (IOHK)
Björn Tackmann (IBM)
Ahto Truu (Guardtime)

Reviewers
Panos Louridas (GRNET)
Ivo Kubjas (SCCEIV)
Sven Heiberg (SCCEIV)

30 June 2019
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.

The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

©Copyright by the PRIViLEDGE Consortium

Executive Summary

In this document we propose the architecture of protocols for secure ledger systems to provide a high level structure that can be used, without ambiguity, to describe the protocol architecture of use cases and toolkits. 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. In the deliverable we provide 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. With this document we provide a global picture of the progress of the research done to date in designing a detailed architecture for the Cardano update system (use case 4) and for the toolkit for anonymous authentication. The next steps in terms of research will concern the problems left open in this deliverable, that are mainly related to the achievement of a better understanding on how to concretely realize (in terms of cryptographic primitives) the protocols proposed at a high level in this document.

Contents

1 Introduction **1**

2 Overview of the architecture of secure distributed ledger systems **3**

2.1 Generic architecture . . . 3
2.1.1 Consensus . . . 3
2.1.2 Block policy and format . . . 4
2.1.3 Transaction processing . . . 5
2.1.4 Ledger store . . . 6
2.2 Hyperledger Fabric . . . 6
2.2.1 Introduction . . . 6
2.2.2 Consensus . . . 7
2.2.3 Block format and policy . . . 7
2.2.4 Transactions . . . 7
2.2.5 Ledger store . . . 8
2.3 Cardano . . . 8
2.3.1 Introduction . . . 8
2.3.2 Ledger In Cardano . . . 8
2.3.3 Consensus: Ouroboros Family of Proof-of-Stake Protocols . . . 9
Ouroboros Classic . . . 9
Ouroboros Praos . . . 10
2.3.4 Cardano (As-Is) Update System . . . 11

3 Use case 4: Cardano stake-based ledger **14**

3.1 A Logical Architecture for an Update Mechanism in Distributed Ledgers . . . 15
3.2 The Lifecycle of a Software Update . . . 16
3.2.1 Ideation . . . 17
3.2.2 Implementation . . . 19
3.2.3 Approval . . . 21
3.2.4 Activation . . . 23

4 Toolkits **26**

4.1 Toolkits for anonymous authentication and flexible consensus in Hyperledger Fabric . . . 26
4.1.1 Anonymous authentication in Hyperledger Fabric . . . 26
4.1.2 Flexible consensus in Hyperledger Fabric . . . 29
4.2 Toolkit for post-quantum secure protocols in distributed ledgers . . . 30
4.2.1 KSI time-stamping . . . 30
4.2.2 BLT signature scheme . . . 32

5 Conclusions **33**

Chapter 1

Introduction

This deliverable covers the architecture of protocols for secure ledger systems, such as authentication within the infrastructure, solutions for updating the ledger protocols, or results on consensus protocols. 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.

The work-document is divided in three parts. In Chapter 2 we propose an overview of the architecture of secure ledger systems, and describe how Hyperledger Fabric and Cardano implement this architecture. The description of these ledger systems focuses on the components that are more relevant for the use case and toolkits that we have considered above.

In Chapter 3 we propose a logical architecture of an update mechanism for proof-of-stake ledgers. 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.

In Chapter 4 we discuss two toolkits that will extend the Hyperledger Fabric open-source blockchain platform and a toolkit for post-quantum secure protocols in distributed ledgers. The toolkits related to Hyperledger Fabric aim to extend the ledger system with a more sophisticated authentication mechanism that enables protecting the identities of the parties in dynamic setting where new parties can join and leave the system at any time accordingly to the policies decided by special nodes of the blockchain called membership service providers. The toolkit for flexible consensus in Hyperledger Fabric aims to provide an architecture that allows separating the consensus mechanism from the 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 last toolkit concerns the security of

ledger systems in a post-quantum scenario and proposes solutions based on quantum-secure signatures and hash functions.

Chapter 2

Overview of the architecture of secure distributed ledger systems

2.1 Generic architecture

Distributed ledger systems 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. The goal of this section is to describe a *generic* architecture of a distributed ledger system. The subsequent sections describe how two concrete blockchain platforms, *Hyperledger Fabric* and *Cardano*, can be viewed as instances of this generic model.

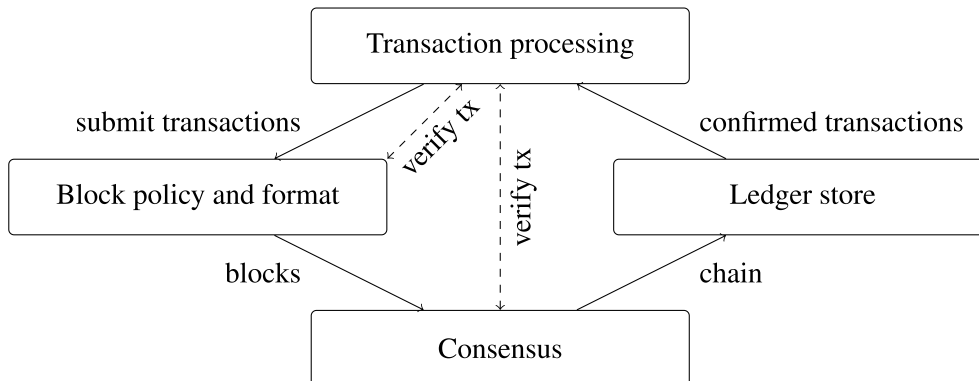


Figure 2.1: Generic architecture of a secure distributed ledger system. The dashed arrow indicates that the connection may not appear in all types of systems.

A first outline of the generic architecture we describe here is provided in Figure 2.1. The goal of the remainder of this section is not to describe the internals of the boxes shown there. Deliverable 3.1 describes these internals for existing distributed ledger platforms in detail. Work packages WP2 and WP3, in particular Deliverables 2.2 to 2.4 as well as Deliverables 3.2 and 3.3, target improving the respective components.

2.1.1 Consensus

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. A systematic analysis of different types of consensus mechanisms was provided in Deliverable 3.1.

Mechanisms. 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.

Early blockchain systems such as Bitcoin and Ethereum are based on a consensus mechanism known as *proof of work*, where the influence of a party is related to its computing power, and thereby indirectly the amount of energy wasted in the consensus-related computation. These algorithms are *permissionless* in the sense that everyone can participate in the consensus by simply installing and running the blockchain software. Many newer blockchain systems favor an approach called "proof of stake" where the influence is related to the amount of blockchain-inherent tokens that it owns. Platforms differ in how exactly the stake is used to determine the influence, but generally these protocols are significantly more efficient and less power-consuming than proof of work. In terms of *permissioning*, these protocols can be seen as sitting between the other two types: no external registration is required, but one has to receive some tokens before being able to participate in consensus.

Properties. The different types of protocols described above provide different properties to the higher-level layers. For instance, BFT protocols usually provide *finality* in the sense that any block that is output by the consensus mechanism is immediately guaranteed to remain unchanged. Also no block can be included in the chain without the agreement of honest parties. While these properties are desirable, achieving them requires that at least $2n/3$ out of the n network participants are honest and actively participating in the protocol.

Optimistic protocols that are usually used in systems based on proof of work or proof of stake often choose a different trade-off. They require only that (strictly) more than half of the resources (computing power or stake) are controlled by honest parties. On the flip side, they achieve weaker guarantees: Blocks that are included in the chain could be revoked if a longer concurrent chain appears (the probability for this decreases for blocks deeper in the chain) and the *chain quality* property requires that at least a certain fraction of blocks are contributed by honest parties, but fully adversarially determined blocks are also permitted.

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, especially in permissionless systems, call out the transaction processing component for verification of the transactions contained in a newly received blocks. While in a permissioned setting, writing to the ledger can be restricted to eligible parties that can be held accountable, a permissionless ledger in principle allows arbitrary parties to write, and verifying the consistency of transactions helps thwarting deniol-of-service attacks.

2.1.2 Block policy and format

The next level component deals with the structure of the blocks that are treated as black box by the consensus. In particular, the component decides which transactions shall be included. It also creates the *chain* structure, meaning a total order, of all blocks.

Mechanisms. The actual mechanisms implemented in the ledger policy and block format component differs between different types of blockchain systems. In permissioned systems, the included in a block can simply be chosen by order of their arrival, since all transactions are generated by legitimate users of the system. As the consensus layer guarantees finality, an arriving block can also be immediately parsed and the contained transactions considered final. In permissionless blockchain systems, network nodes are incentivized through rewards which they receive (in part) through transaction fees. Therefore, the ledger policy and format component will generally choose the transactions to be included in a given block based on the included transaction fees. The component may also check transactions for validity before including them in a block.[1] This step is not strictly necessary, as invalid transactions can still be discarded by the higher-level protocol components; however, it saves storage space by discarding invalid transactions and reduces the attack surface for denial of service attacks.

Most blockchain protocols implement the chaining of blocks by including a hash of each block in the respective subsequent block. Especially in permissioned systems, however, other methods such as a simple block number are equally valid.

Interface. The interface toward the lower-level consensus component consists of outputting generated blocks that are meant to be agreed upon in consensus.

The interface toward the higher-level component receives as input transactions that are meant to be included in a future block. Especially in permissionless systems it is desirable to check the validity of transactions before including them in a block. In this case, a callback interface to verify transactions with the transaction component can also be required.

2.1.3 Transaction processing

The transaction level obtains the transaction from client applications (often referred to as *wallets*), processes them and updates the *world state*. The transaction format and semantics differs significantly between different blockchain systems, depending on the main application area.

Mechanisms. The transaction level manages the state of the blockchain that is modified by transactions. In the case of Bitcoin, the state is the so-called *UTXO pool* that contains all available tokens that can be spent. A transaction removes some tokens from the pool (the *inputs* of the transactions) and adds new tokens generated by the transaction (the *outputs* of the transaction). Permissionless blockchain systems that support smart contracts, like Ethereum, do support a similar functionality for token transactions. Additionally, the world state contains one namespace for each contract, and the contract can store arbitrary values in its namespace. Executing a transaction then consists of performing the token transactions and executing the contract. Typically transactions are authorized by signing them with private keys corresponding to the public keys used as account identifiers. How and when such signatures are verified depends on particular ledger, but most commonly this is done before the transactions are executed.

In permissioned blockchain systems, tokens are often not used for the invocation of individual transactions, but are often still supported to implement payment functionality. Most permissioned systems also support some form of smart contracts, but the types of implementation vary and include Ethereum-like post-consensus execution in Quorum, pre-consensus execution in Hyperledger Fabric, or execution within a centralized so-called notary system in Corda.

Interface. The interface to the lower-level block policy and format layer consists of providing as output transactions that are to be included in a future block, and receiving as input transactions from the ledger store that have been extracted from a block decided in consensus. Additionally, the component may support a callback interface for verifying the validity of transactions. The interface to higher-level protocols and application depends on the transaction type that is implemented and is therefore dependent on the mechanism implemented.

[1] This is necessary since the chain can contain adversarially contained blocks.

2.1.4 Ledger store

The final component depicted in Figure 2.1 is the *ledger store* component that keeps the complete blockchain for reference and for distributing it to parties that are joining the network or recovering from a crash or outage. The ledger store may also deliver information to the transaction processing component so it can update its state after new blocks have arrived via the network.

Mechanisms. The main mechanism of this component is a database that stores all blocks it receives from consensus. The component also implements network protocols for distributing blocks to other nodes; this is often achieved through a gossip mechanism.

Interface. Via the interface toward the consensus layer, the ledger store receive new blocks to be stored in the database. Toward the transaction processing component, it provides as output transactions that are contained in the newly arrived block.

2.2 Hyperledger Fabric

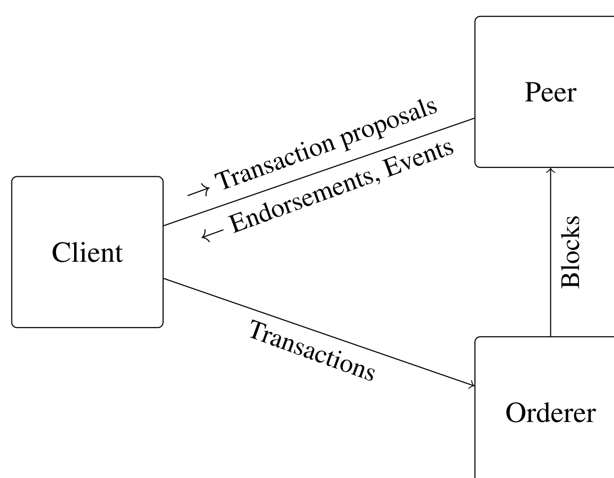


Figure 2.2: Hyperledger Fabric distinguishes different roles. *Clients* invoke transactions; *peers* execute smart contracts and store their state; *orderers* run a consensus protocol to determine the order of transactions.

2.2.1 Introduction

Hyperledger Fabric[2] 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 2.2, 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.

[2] https://www.hyperledger.org/projects/fabric/

2.2.2 Consensus

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.

As of Fabric v1.0, two ordering service implementations are 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[3], a distributed streaming platform that follows the publisher/subscriber paradigm and offers crash-fault tolerant (CFT) operation based on the Apache ZooKeeper[4] 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 recent release Fabric 1.4.1 added support for the Raft consensus protocol based on etcd[5], 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.)

Neither solo nor kafka or raft ordering 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 Vukolič [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 [ABB+18]. An implementation of a BFT ordering service that is planned[6] to be included in the main distribution of Fabric is currently under development by IBM Research – Zurich. The mechanism will be based on the PBFT protocol [CL02], but targeting improved transaction throughput.

2.2.3 Block format and policy

The policy of including transactions in a block is simple in a permissioned systems such as Fabric, where transactions are known to be submitted by legitimate users of the system. Users that exhibit malicious behaviour can be detected and have their permissions revoked. Transactions can simply be chosen based on a FIFO strategy; if two transactions conflict, only the first one is considered valid and the second one is ignored. Each block starts with a header that contains a hash of the previous block as well as a hash of the payload of the current block. The payload consists of a list of transactions.

2.2.4 Transactions

Transaction processing in Fabric differs radically from other blockchain systems. Smart contracts, dubbed *chaincode* in Fabric, can be implemented in arbitrary programming languages. More technically, they are deployed as Docker containers that provide a remote procedure call (RPC) interface through which transactions can be triggered. In turn, the chaincode container has access to an RPC interface offered by the peer implementation through which it can access state. As chaincodes can contain code that is not deterministic (e.g., computation can use randomness and even access parts of the host system), the transactions cannot be executed after ordering, as the state of different nodes could diverge.

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

To resolve the above issue, Fabric uses an *endorsement* process in which the execution of a contract takes place prior to the ordering. In more detail, a *client* invokes a transaction by sending a *transaction proposal*, which specifies the code to be executed as well as the parameters, to the *endorsing peers* (or simply *endorsers*) for that chaincode. The peers then execute the chaincode and record the effects of the execution, the read-write pattern to the state namespace and the events triggered by the chaincode. These effects are then encoded in the *endorsement* that is signed by the peer and returned to the client. After collecting sufficiently many endorsements, the client combines them with the proposal into the complete *transaction* and sends it to the ordering service, which includes it in one of the future blocks. The peers then receive the blocks from the ordering service, check whether sufficiently many endorsements are included in the transaction (each chaincode can have a specific *endorsement policy* that specifies when a transaction is considered valid), and whether all submitted endorsements are consistent, and in case of success update the current state of the chaincode.

As multiple invocations of the same chaincode may be concurrent, Fabric uses multi-version concurrency control to serialize transactions whenever possible and reject conflicting transactions.

2.2.5 Ledger store

The blocks received from consensus (or via gossip from other peers) are stored on each peer in a LevelDB key/value store which provides sufficient functionality and performance.

2.3 Cardano

2.3.1 Introduction

Cardano is a blockchain platform implemented by IOHK and underlying the Ada cryptocurrency. Cardano is running on top of the Ouroboros proof-of-stake protocol. Due to the active development of both the protocol and the whole platform, Cardano would benefit from a safe and secure decentralized software update system. In other words, a system that will ensure that: a) any honest stakeholder will always be able to submit an update proposal to be voted by the stakeholders community, b) it will prevent an update proposal that is not approved by the honest majority to be applied and c) it will provide guarantees for the authenticity of the downloaded software.

2.3.2 Ledger In Cardano

Transaction Generation. A transaction is a special data structure, which represents the act of value transfer between nodes. Thus, when the user Alice sends money to the user Bob, the new transaction is created. Let us call this transaction Tx_1, the node under Alices wallet N_1, and the node under Bobs wallet N_2.

Thus, the node N_1 does the following steps:

1. Creates transaction Tx_1 and signs it with its private key.
2. Sends it to all known nodes (i.e., neighbors).
3. Saves it in its local data.

Each of N_1's neighbors sends Tx_1 transaction to its neighbors and this process is repeated until the transaction is propagated to all nodes. In addition, some *slot leader*[7] will store this transaction in some block in the ledger. The time required for a transaction to be actually added to a block depends heavily on the network load. The more transactions flow in the network the more time will be required.

[7] A slot leader in a proof-of-stake consensus protocol is the equivalent of a miner in a proof-of-work protocol. It is the entity who has the legitimate right to issue a new block by packing new transactions into it with a specific order.

Each transaction contains a list of inputs and a list of outputs; outputs of the transaction Tx_0 can be used as inputs of another transaction Tx_1 and the outputs of this to another transaction forming a chain of value transfer via transactions. Inputs and outputs carry information about money flow: inputs inform where the money came from, and outputs inform where the money go to. The number of inputs and outputs in a transaction can be different.

In more detail, an input of a transaction consists of:

- An ID of a transaction Tx_N, whose output is used for this input. A transaction ID is a BLAKE2b-256 hash of the transaction.
- An index specifying which output of transaction Tx_N is referenced.

An output of a transaction consists of:

- An address of the node N we want to send a value to. An address is a BLAKE2b-224 hash of the hash of the public key of the N node.
- Amount of money we want to send. This value is 64-bit unsigned integer.

Transaction Verification and State Transitions An output that is not yet an input of another transaction is called an *unspent transaction output (UTXO)*. Only unspent outputs can be used as inputs for other transactions, to prevent double-spending.

Each transaction in Cardano is accompanied by a proof (also called a witness) that this transaction is valid. Even if the output is an unspent one, we have to prove that we have a right to spend it. Since a Tx_N transaction can have many inputs, the witness for it consists of the witnesses of all Tx_N's inputs, and only if all the inputs are valid, Tx_N is valid too. If a particular transaction is invalid, it will be rejected by the network. There are two types of witnesses: one based on a public key (valid input must be signed with a private key corresponding to this public key) and one based on a script.

Witnesses are stored in the blockchain and anybody can see, inspect and independently verify them. But after some time a node may delete old proofs in order to save space. The technique of storing transactions separately from their proofs is called *segregated witness*. Under this scheme, transactions and proofs are stored in two separate places in a block, and can be processed independently.

Therefore, every node in the network not only accepts transactions, but also verifies them. To this end, every node has to keep track of unspent outputs. This allows to validate that inputs in a published transaction are indeed the unspent outputs. Actually, all unspent outputs (UTXO), are stored in a special key-value database called Global State.

The collection of all UTXO is known as the UTXO set and forms *the state* of the Cardano blockchain. The UTXO set grows as new UTXO is created and shrinks when UTXO is consumed. Every transaction represents a change (*a state transition*) in the UTXO set.

2.3.3 Consensus: Ouroboros Family of Proof-of-Stake Protocols

In this section we sketch the proof-of-stake protocols Ouroboros [KRDO17] and Ouroboros Praos [DGKR17]. The former is currently underlying the deployed Cardano blockchain, while the latter is foreseen to ultimately replace it. To avoid confusion, in the following we refer to the Ouroboros protocol [KRDO17] as Ouroboros Classic.

Ouroboros Classic

Description. The protocol operates (and was analyzed) in a synchronous model that splits the time continuum into a sequence of consecutive, non-overlapping intervals called *slots* (in Cardano, each slot is 20 seconds long).

It is assumed that all parties are equipped with synchronized clocks that tell them the index of the current slot, and all messages are delivered within the next slot.

In each slot, each of the parties can determine whether it qualifies as a so-called *slot leader* for this slot.The event of a particular party becoming a slot leader occurs with a probability proportional to the stake controlled by that party, is independent for two different slots, and is determined by a public, deterministic computation from the stake distribution and so-called *epoch randomness* (we will discuss shortly where this randomness comes from) in such a way that for each slot, exactly one leader is elected.

If a party is elected to act as a slot leader for the current slot, it is allowed to create, sign, and broadcast a block (containing transactions that move stake among stakeholders). Parties participating in the protocol are collecting such valid blocks and always update their current state to reflect the longest chain they have seen so far that did not deviate from their previous state by too many blocks. As in the case of Bitcoin [Nak08], if there are multiple candidate chains, the slot leader will always choose the longest one to attach a new block.

Multiple slots are collected into *epochs*, each of which contains $R \in \mathbb{N}$ slots (in Cardano, $R = 21600$). Each epoch is indexed by an index $j \in \mathbb{N}$. During an epoch j, the stake distribution that is used for slot leader election corresponds to the distribution recorded in the ledger up to a particular slot of epoch $j - 1$, chosen in a way that guarantees that by the end of epoch $j - 1$, there is consensus on the chain up to this slot. Additionally, the *epoch randomness* for epoch j is derived during the epoch $j - 1$ via a *guaranteed-output delivery coin tossing* protocol SCRAPE [CD17] that is executed by the stakeholders.

Security. Ouroboros Classic was shown by Kiayias et al. [KRDO17] to achieve the common prefix, chain growth, and chain quality properties, proposed as security desiderata for blockchain protocols by Garay et al. [GKL15]. It is well-known that these properties imply both persistence and liveness of the resulting ledger. These properties are achieved by Ouroboros under the following assumptions:

1. synchronous communication (as outlined above);
2. majority of the stake is always controlled by honest parties (i.e., parties following the protocol);
3. parties cannot be corrupted immediately, there is a corruption delay in place;
4. the stake shift[8] per epoch is limited.

Ouroboros Praos

Description. The Ouroboros Praos protocol [DGKR17], despite sharing the basic structure with Ouroboros Classic, differs from it in several significant points; we now outline these differences.

The slot leaders are elected differently in Praos: Namely, each party for each slot evaluates a verifiable random function (VRF, [DY05]) using the secret key associated with their stake, and providing as inputs to the VRF both the slot index and the epoch randomness. If the VRF output is below a certain threshold that depends on the party's stake, then the party is an eligible slot leader for that slot, with the same consequences as in Ouroboros Classic. Each leader then includes into the block it creates also the VRF output and a proof of its validity to certify its eligibility to act as a slot leader. The probability of becoming a slot leader is again roughly proportional to the amount of stake the party controls, however now it is independent for each slot and each party, as it is evaluated locally by each stakeholder for itself. This local nature of the leader election implies that there will inevitably be some slots with no, or several, slot leaders.

In each epoch j, the stake distribution that is used in Praos for slot leader election corresponds to the distribution recorded in the ledger up to the last block of epoch $j - 2$. Additionally, the *epoch randomness* for epoch j is derived as a hash of additional VRF-values that were included into blocks from the first two thirds of epoch $j - 1$ for this purpose by the respective slot leaders.

[8] Stake shift is a change in the stake distribution among the participants of the protocol.

Finally, the protocol uses *key-evolving signatures* for block signing, and in each slot the honest parties are mandated to update their private key, making the system resilient to even adaptive corruptions.

Security. Ouroboros Praos was shown by David et al. [DGKR17] to achieve both persistence and liveness under weaker assumptions than Ouroboros Classic, namely:

1. semi-synchronous communication, where messages can be adversarially delayed for up to Δ slots (where Δ affects the security bounds but is unknown to the protocol);
2. majority of the stake is always controlled by honest parties (i.e., parties following the protocol);
3. the stake shift per epoch is limited.

In particular, Ouroboros Praos is secure in face of fully adaptive corruptions without any corruption delay.

2.3.4 Cardano (As-Is) Update System

The current update system of Cardano is quite restricted due to the fact that research work on blockchain updating is necessary for implementing a decentralized[9] update mechanism. Therefore, for lack of research results in this area the current implementation, although it provides a basic updating mechanism, it operates with significant constraints. In the following paragraphs we provide a short overview of the current implementation and pinpoint various restrictions.

Governance. Only a specific set of users can issue an Update Proposal (UP). These are the direct delegates of a specific set of private keys (the so-called *genesis keys*). In other words, the genesis keys can delegate their right to submit a UP to some other key and then only a key has the right to submit a UP.

An UP is a separate object that it is created, relayed to the network and after successful verification, it is stored in a memory pool. An UP contains among others, a block version, a set of protocol parameters (e.g., max block size, slot duration etc.), a hash ID that is the result of applying a hash function to the actual update code for the update (and thus uniquely identify this piece of code), the UP issuer's public key and the issuer's signature of this UP. A vote for an UP is another object that can be transmitted on the network and after verification it is temporarily stored in a memory pool. A vote consists of the voter ID (stake public key), the ID of the UP, the ballot and and the voter's signature.

Note that the current update system works only in a federated setting, allowing only a specific set of keys (the delegates of the Genesis keys) to have the right to propose and vote for updates. In order to lift this restriction solid research results on the governance of decentralized UPs are due that will provide the necessary guarantees that the update mechanism will work with no problems.

States of an Update Proposal.

- *Submitted*: A user submits an Update Proposal to the network. This is validated and kept in a memory pool. At this state, stakeholders can submit their votes for this UP. UPs and votes reside in the memory pool.
- *Active*: When an UP from the previous state is included in a block (update payload within a block body) and thus gets committed into the blockchain, then, it becomes active. Once a UP becomes active, then a poll starts for a specific duration, in order to either confirm it or reject it.

[9]Decentralized here means that is accessible by any stakeholder, as well as that the decisions are based on stake majority and not only on a restricted set of privileged users.

- *Confirmed or Rejected*: When an update proposal gathers votes above a confirmation threshold,[10] we say that it becomes confirmed. For each UP we store the earliest slot (i.e., block number) that has been confirmed. If an update proposal does not manage to get confirmed within its voting period, it becomes rejected. Confirmed/rejected proposals may become active again, and even end up rejected/confirmed, if a blockchain rollback occurs and poll ends with an opposite decision on the alternative branch. That is why for each update proposal that gets confirmed we need to wait for a number of slots, in order for the confirmation to become stable.

- *Confirmed permanently*: If a proposal has been approved in some block and there are at least $2k$ blocks after this one (where k is a security parameter), the update proposal becomes confirmed permanently. Permanently Confirmed state reflects the fact that the confirmed state cannot be changed anymore, because we have the guarantee that at most 2k blocks must be rolled back.

- *Discarded*: If a proposal has been rejected in some block and there are at least $2k$ blocks after this one, the update proposal becomes discarded. Discarded state reflects the fact that the rejected state cannot be changed anymore. If a proposal is discarded then it does not affect consensus rules anymore and it is safe to evict it out of consideration/storages.

- *Endorsed* After the confirmation of an update proposal becomes stable, then the endorsement phase begins. An endorsement is a signaling mechanism that signifies upgrade readiness that takes place only for confirmed UPs. Endorsement is possible only for confirmed UPs that have been stabilized. When a user endorses a UP, essentially the user states readiness to adopt this UP. We count the endorsements for each UP that has been confirmed at least $2k$ slots before the current slot (i.e., confirmed UPs that have passed their stabilization period), and check if they have exceeded a certain adoption threshold. If a UP gets enough endorsements at the current slot, it is kept in a list of to-be adopted update proposals.

- *Activated* Activation (also called adoption) of a UP is triggered only by a change of the epoch. When a new epoch comes, then we look among UPs that have achieved to pass the endorsement threshold at least $2k$ slots back, and from these, we choose the one with the highest protocol version (each UP corresponds to a specific protocol version).This will be the UP adopted (i.e., activated). Therefore, UP activations can only take place at the beginning of a new epoch and at no other time. All other UPs —even if they have been endorsed above the threshold also— that correspond to smaller versions than the newly adopted one are discarded.

Deployment The current mechanism supports adoption, without the need of a software upgrade, only for a limited set of protocol parameters, such as the block size limit, the duration of a slot, the transaction size limit, etc. This is possible, because a fixed set of protocol parameters has been bound to the protocol version number. So, when a version becomes adopted (see discussion above), then all nodes validate new blocks with the set of protocol parameters bound with the adopted version, without the need to do any software updates. All other consensus rules changes must be handled through a soft or a hard fork type of change.

Conflicts and Dependencies No special care for conflicts and dependencies between update proposals has been taken. Updates are merely based on a basic versioning scheme.

Update Voting Delegation No special delegation mechanism is in place for update proposal voting. For delegating update proposal votes a separate delegation mechanism than the standard delegation for block issuing must be used.

[10] The confirmation threshold is a protocol parameter. For example, it could have the same value as that of the honest (total) stake majority, which is typically 50%.

Update Policies No distinction between hot-fixes, change requests is made. In addition, there is no way declaring update priority/criticality. In general, all changes are considered of being of ”the same type” and the only differentiation is declared through the versioning scheme, which signifies major and minor updates.

Update Metadata No consideration for the definition of update proposal metadata has been made in the current implementation.

Update Code linkage UPs include a hash of the software update to be downloaded from the network and installed. The downloaded software is signed by IOHK (i.e., a central authority) and thus considered trusted. Of course, this departs from an ideal decentralized setting, where no central authorities (that can be compromised) exist.

All in all, the current update mechanism serves very well the first incarnation of Cardano, which is known as the *Byron release*. It is a release, where the blockchain system does not operate on a fully decentralized mode, but essentially, the consensus protocol runs by a set of trusted nodes (i.e., a federated setting). This is of course, only the first step in the Cardano evolution roadmap, since a gradual transition to decentralization has been planned (code-named *Shelley release*). We have seen that from a decentralization perspective the as-is update systems bears significant limitations. The most obvious one is that the decisions for the update proposals are not taken but the majority of stake, but rather by a limited set of trusted stakeholders. Moreover, we have seen that the current update system implements a very simple “update logic” model, where software updates are limited to a predefined set of protocol parameters. The lack of specialized update metadata, the disability to distinguish different “types” of updates, as well as the appropriate resolution of update conflicts and in general, the fulfillment of update constraints (e.g., update dependencies), render it insufficient for a decentralized proof-of-stake blockchain system.

Chapter 3

Use case 4: Cardano stake-based ledger

Cardano is an actively developed system, and as such will clearly need updates and improvements when new requirements arise, bugs are discovered, standards or protocols change, or new technologies become relevant.

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. Thus, the road-map of the system is decided jointly. Moreover, there is no central code repository, nor there is some main software maintainer, who decides on the code. All versions of the code are distributed and only local copies exist, in the same manner that the ledger of transactions is distributed in blockchain. The decentralized software update system must reach a *consensus*, as to what version is the current one (main branch) and which code branch will be merged into main. Finally, the decentralized software update system guarantees the authenticity of the downloaded software, without the need to have some central authority to sign the code, in order to be trusted.

At present, there is no standard way to propose updates, reach consensus on such proposals and then—once consensus has been reached—distribute updates securely to all participants. There is no way to guard against malicious proposals on the one hand, while on the other hand enabling critical security updates or bug-fixes to be distributed in swiftly.

Ideally, each participant (with sufficient stake) of the Cardano ledger should be allowed to make update proposals and have a vote on them, then deploy a corresponding update efficiently. This raises several natural requirements:

– The mechanisms for handling proposals, voting and deployment of updates, should to as large extent as possible leverage the blockchain itself, with its "built-in" consensus mechanism.

– Concurrent, potentially contradictory proposals need to be handled, guaranteeing that they will not be accepted and deployed at the same time.

– Community splits over controversial updates should be prevented.

– Participants who have been inactive when some update was deployed need to be prevented from getting locked out of the system.

– Any proposal needs to be securely linked to the actual binaries that will be distributed if the proposal is approved.

There are several ways to approach the problem of constructing an update system meeting the above requirements. In the following subsections, first we focus on a logical architecture that depicts the various components

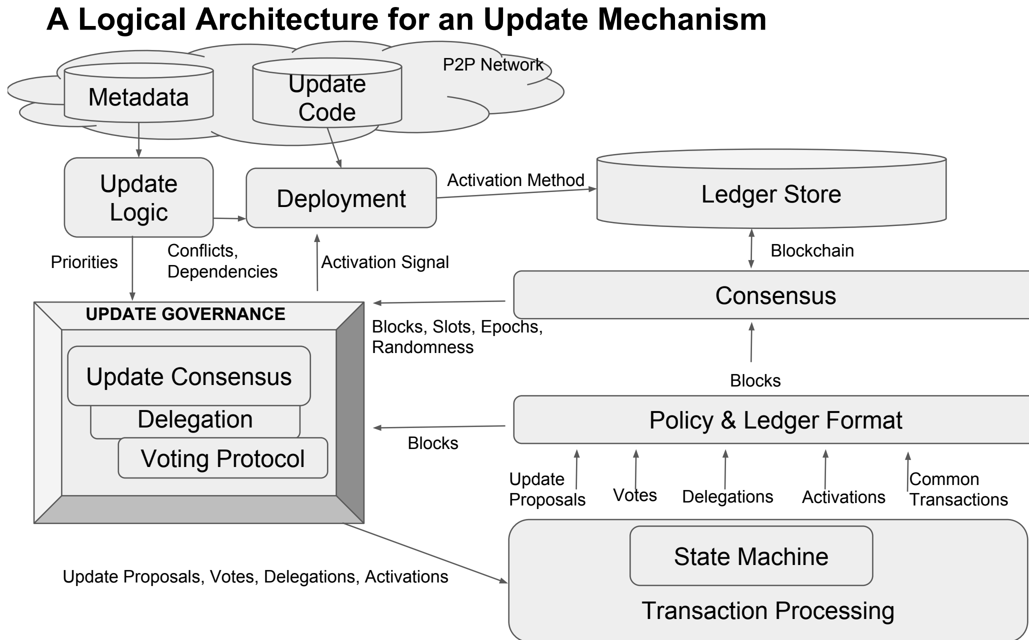


Figure 3.1: A logical architecture for an update mechanism.

of a ledger system and their interactions for this use-case and then, we describe the phases in the lifecycle of a decentralized software update.

3.1 A Logical Architecture for an Update Mechanism in Distributed Ledgers

In Figure 3.1, we depict a logical architecture that incorporates a software update mechanism in a distributed ledger. In this figure, we can see that the set of events flowing into the ledger system has been enhanced. Apart from the traditional transactions, which deal with the transfer of value between stakeholders and trigger changes to the global state, now we can see other events such as *Update Proposals*, *Votes*, or even *Vote Delegations* and *Activation events* being generated from *Update Governance* component and entering the ledger system.

Our primary goal is to store all these new events that pertain to the update mechanism, within the ledger itself, as an immutable record of historical events. To this end, these new events are transformed to special types of transactions and as such, have to go through the rigorous transaction validation process, performed by each node of the network. This is depicted at the Transaction Processing layer, where all the logic for what makes a legitimate transaction is implemented.

Ordinary transactions change the global state of value distribution but in the case of smart contracts can also change the state of smart contract persistent data by triggering the execution of smart contract code. Update events do not change the global state in the same sense; however, one can imaging an "Update Proposal state", representing the number of Update Proposal submitted by a user. Similarly, one could define a "Voting state" representing the votes gathered by each proposal.

Transactions are included into blocks by the Policy and Ledger Format layer and these newly generated blocks are passed on to the next layer. The Consensus layer is where we achieve consensus on what block is legitimate and which blockchain branch is the legitimate one and a new block can be attached to it safely. Regardless of the "type" of the consensus protocol (either proof-of-work, or proof-of-stake) the consensus rules that guide the block validation logic must be enhanced to incorporate the new events (i.e., transaction types)

that are to be stored in the ledger. As we depict in the figure, the final destination of all types of events is the distributed ledger, which serves as the *single version of the truth* for the updating history of the system.

In particular, for the Cardano case one has to consider the possible enhancements necessary for the Ouroboros family of protocols [KRDO17]. Apart from the block validation affecting the consensus rules, one has to take into account the possible impact from the protocols from the *Update Governance* component depicted in Fig. 3.1.

Update Governance is the component in the architecture that enables a decentralized updating mechanism. It allows all users to freely vote on update proposals and then considers the stake distribution for forming the final result. This is achieved via a secure *voting protocol* that must be tightly integrated with Ouroboros. Moreover, for these users that might be owners of stake but lack the skills and expertise to vote for or against a software update proposal, a *voting delegation protocol* must be in place. This will enable the delegation of the voting right to some other user (regardless of his/her stake), a so-called *expert*, that will assume the voting on behalf of the delegating user. The ultimate goal of the voting and delegation protocols is all the participating users to reach at an *update consensus*. In other words, to reach a common agreement on the update proposal priorities and on the evolution roadmap of the ledger system. Voted update proposals must be eventually deployed to the system and become *adopted updates*, which are activation events that are stored in the ledger, as well.

The *Update Logic* component handles the rules that will guarantee a seamless and versatile activation of software updates into the ledger system. More specifically, this component implements the necessary "logic" for conflict resolution and secures the updating mechanism from updates that might lead the system to an unstable state. In addition, it supports the correct enforcement of update dependencies and prevents update patches to be installed, if required previous patches are missing. Finally, it implements the notion of *Update Policies*, which differentiate speed of deployment and method of deployment based on: a) the type of change (bug-fix or change request), b) the part of the system that is affected by the change (consensus rules impact, or only software impact) and c) the urgency of the change (severity level). The method of deployment determines the actual mechanism that will be used in order to activate an update in the ledger system. It is executed by the *Deployment* component depicted in 3.1. For protocol changes one must consider the most appropriate method for such a deployment and choose among a set of well-known practices such as hard-forks and soft-forks but also from more niche techniques, such as velvet-forks [ZSJ+18] and the sidechains mechanism [BCD+14].

For all these to work, the Update Logic component must be based on an appropriate set of *Update Metadata*. These metadata must accompany each submitted Update Proposal and sufficiently describe the change, its expected benefit, its type, its urgency for deployment, its possible conflicts with other updates and many more. In addition to the metadata, every update proposal must be accompanied by the update patch that comprises the actual code to be installed. Both of these two, the metadata and the code, cannot be stored in the ledger, due to their sizing requirements. In order to avoid to store these data into any type of centrally owned servers (e.g., into the cloud). One should consider the use of a P2P file storage or database solution, with content addressable storage capabilities. Moreover, the proposed solution must guarantee the authenticity of the downloaded software with respect to the original update proposal.

3.2 The Lifecycle of a Software Update

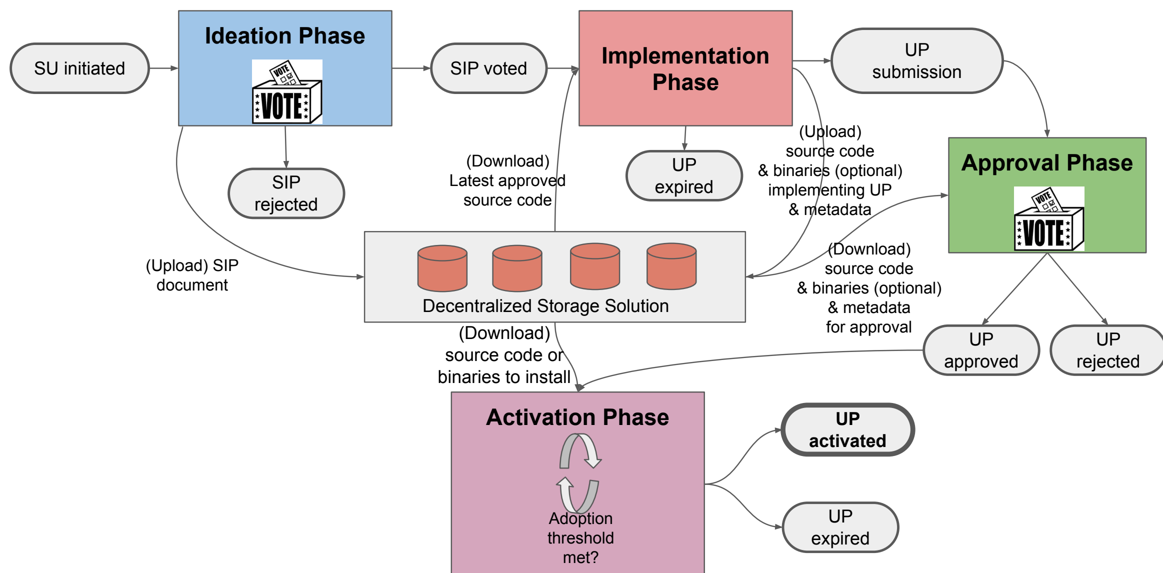
A *software update (SU)* is the unit of change for the blockchain software. It must have a clear goal of what it tries to achieve and why it would be beneficial, if applied to the system. Moreover, it should have a clear scope. In Figure 3.2, we depict the full lifecycle of a software update following a decentralized approach. In this lifecycle, we identify four distinct phases: a) the *ideation phase*, b) the *implementation phase*, c) the *approval phase* and d) the *activation phase*. In this section, we briefly outline each phase and at the subsequent subsections, we provide all necessary details for realizing each phase in a decentralized setting.

The SU starts from the ideation phase which is the conceptualization step in the process. It is where an SU is born. During this phase, a justification must be made for the SU and this has to be formally agreed by the community (or the code owner). This justification takes the form of an improvement proposal document (appears as *SIP* in the figure and will be defined shortly). Once the SU's justification has been approved, then we enter

the implementation phase. It is where the actual development of the SU takes place. The result of this phase is a bundle (Update Proposal, UP) consisting of source code (implementing the SU), metadata and optionally binaries produced for one, or more, specific platforms. This is submitted for approval and thus the approval phase follows. Once the UP (i.e., the source code implementing the SU) has been approved (by the community, or the code owner), the community is called for upgrading. The actual upgrading takes place in the activation phase, which is there to guard against chain-splits by synchronizing the activation of the changes.

Interestingly, the phases in the lifecycle of a SU are essentially independent from the approach (centralized or decentralized) that we follow. They constitute intuitive steps in a software lifecycle process that starts from the initial idea conception and ends at the actual activation of the change on the client software. Based on this observation, one can examine each phase and compare the traditional centralized approach, used to implement it, to its decentralized alternative. Moreover, not all phases need to be decentralized in a real world scenario. One has to measure the trade-off between decentralization benefits versus practicality and decide what phases will be decentralized. Our decomposition of the lifecycle of a SU in distinct phases helps towards this direction.

Figure 3.2: The lifecycle of a software update (a decentralized approach)



3.2.1 Ideation

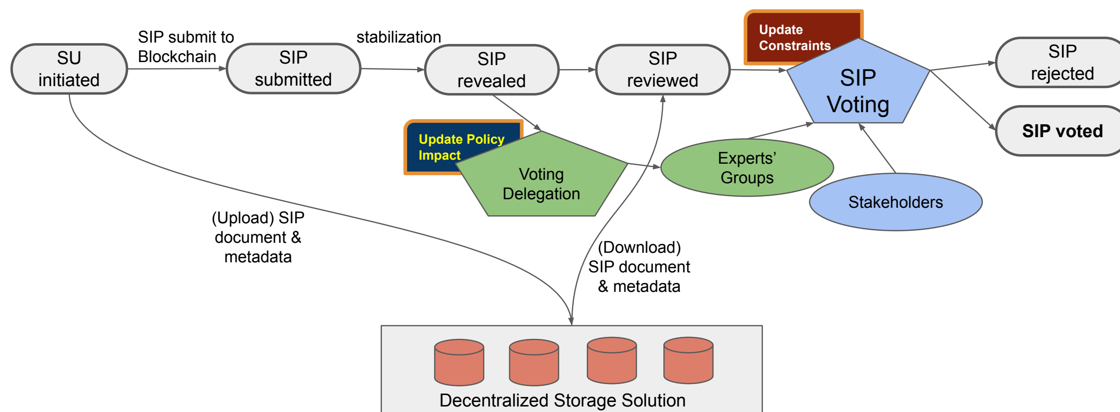
A SU starts as an idea. Someone captures the idea of implementing a change that will serve a specific purpose (fix a bug, implement a new feature, provide some change in the consensus protocol, perform some optimization etc.). The primary goal of this phase is to capture the idea behind a SU, then record the justification and scope of the SU in some appropriate documentation and finally come to a decision on the priority that will be given to this SU.

Traditionally, in the centralized approach, a SU is proposed by some central authority (original author, group of authors, package maintainer etc.), who essentially records the need for a specific SU and then decides when (or, in which version) this could be released. In many cases, (e.g., Bitcoin [Nak08], Ethereum [B$^+$14]) the relevant SU justification document (called BIP, or EIP respectively) is submitted to the community, in order to be discussed. Even when this "social alignment" step is included in this phase, the ultimate decision (which might take place at a later phase in the lifecycle), for the proposed SU, is taken by the central authority. Therefore, the road-map for the system evolution is effectively decided centrally. Moreover, this social consensus approach is informal (i.e., not part of a protocol, or output of an algorithm) and is not recorded on-chain as an immutable

historical event.

The ideation phase in the decentralized approach is depicted in Figure 3.3.

Figure 3.3: The ideation phase.



In the decentralized setting, a SU starts its life as an idea for improvement of the blockchain system, which is recorded in a human readable simple text document, called the *SIP (Software* [1] *Improvement Document)*. The SU life starts by submitting the corresponding SIP to the blockchain by means of a special fee-supported transaction. Any stakeholder can potentially submit a SIP and thus propose a SU.

A SIP includes basic information about a SU, such as the title, a description, the author(s) etc. Its sole purpose is to justify the necessity of the proposed software update and try to raise awareness and support from the community of users. A SIP must also include all necessary information that will enable the SU validation against previous SUs (e.g., update dependencies or update conflict issues), or against any prerequisites required, in order to be applied.

A SIP is initially uploaded to some external (to the blockchain system) *decentralized storage solution* and a hash id is generated, in order to uniquely identify it. This is an abstraction to denote a not centrally-owned storage area, which is content-addressable, i.e., we can access stored content by using the hash of this content as an id. A change in the content will produce a different hash and therefore this will correspond to a different id and thus to different stored content. This hash id is committed to the blockchain in a two-step approach, following a hash-based commitment scheme, in order to preserve the rightful authorship of the SIP.

Once the SIP is revealed a voting period for the specific proposal is initiated. Any stakeholder is eligible to vote for a SIP and the voting power will be proportional to his/her stake. Votes are specialized fee-supported transactions, which are committed to the blockchain.

Note that since a SIP is a document justifying the purpose and benefit of the proposed software update, it should not require in general sufficient technical expertise, in order for a stakeholder to review it and decide on his/her vote. However, in the case that the evaluation of a SIP requires greater technical knowledge, then a voting delegation mechanism exists. This means that a stakeholder can delegate his/her voting rights to an appropriate group of experts but also preserving the right to override the delegate's vote, if he/she wishes.

[1] "Software" and "System" are two terms that could be considered equivalent for the scope of this document and we intend to use them interchangeably. For example, a SIP could also stand for a System Improvement Proposal.

The delegation mechanism will also be used in order to implement the concept of an *update policy*, which enables different activation speeds for a SU depending on its type (e.g., a bug-fix versus a change request, a SU that has a consensus protocol impact versus a no-impact one, etc.). For all these special *delegation groups* will be considered. A SIP after the voting period can either be voted or rejected. Note that in the decentralized approach the ideation phase could very well be implemented by a treasury system (e.g., similar to the one proposed by Bingsheng et al. [ZOB18]). A treasury is a decentralized and secure system aimed at the maintenance of a blockchain system that allows the submission of proposals (i.e., candidate projects) for the improvement of the system. These proposals go through a voting process, in order to select the surviving ones. More importantly, the system is supported by a funding mechanism, where funds raised are stored in the treasury. These funds are used for funding the approved projects. Implementing the ideation phase with a treasury system, would enable additionally the appropriate management of the funding of each SU.

3.2.2 Implementation

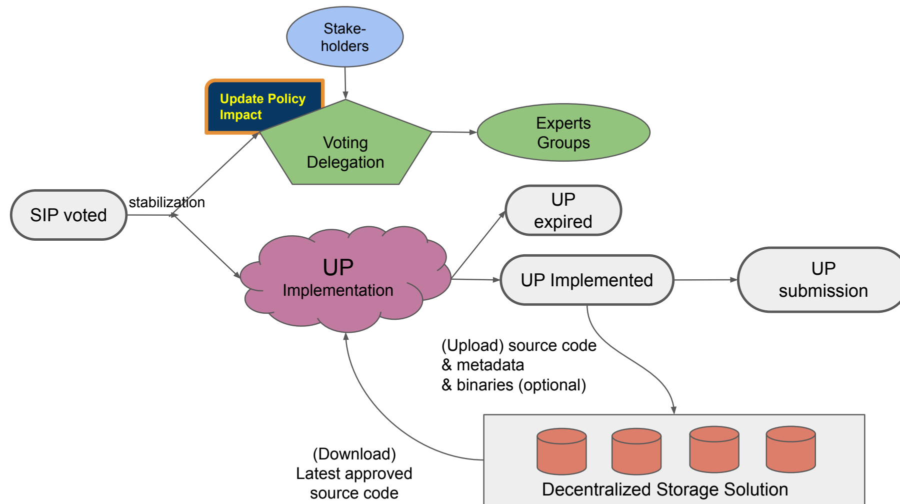
The voting of a SIP is the green light signal for entering the implementation phase. This is the period where the actual implementation of a SIP takes place. So one could very roughly imagine this phase as a box, where a SIP comes in as input and source code implementing the SIP comes out as output.

The scope of this phase is twofold: a) to develop the source code changes that implement a specific voted SIP and b) to execute a second voting delegation round, in order to identify the experts that will approve the new source code. At the end of this phase, the developer creates a bundle comprising the new source code, the accompanied metadata and optionally produced binaries, which we call an *update proposal (UP)*. The newly created UP must be submitted for approval, in order to move forward.

In the centralized setting, it is typical (in the context of an open source software development model), when a developer wants to implement a change, first to download from a central code repository the version of the source code that will be the base for the implementation and then, when the implementation is finished, to upload it to the same central code repository and submit a pull-request. The latter is essentially a call for approval for the submitted code. The central authority responsible for the maintenance of the code-base, must review the submitted code and decide, if it will be accepted, or not. Therefore, in the centralized approach the implementation phase ends with the submission of a pull-request.

The decentralized alternative for the implementation phase is identical to its centralized counterpart as far as the development of the new code is concerned. However, in the decentralized setting, there exist these major differences: a) there is not a centrally-owned code repository (since there is not a central authority responsible for the maintenance of the code), b) a delegation process is executed, in parallel to the implementation, as a preparation step for the (decentralized) approval phase that will follow and c) the conceptual equivalent to the submission of a pull-request must be realized.

Figure 3.4: The implementation phase.



In Figure 3.4, we depict the decentralized implementation phase. Similar to the centralized case, the implementation of a change must be based on some existing code, which we call the base source code and the developer must download locally, in order to initiate the implementation. However, in the decentralized setting there is not a centrally-owned code repository. All the approved versions of the code are committed into the blockchain (i.e., only the hash of the update code is stored on-chain). Therefore, we assume that the developer finds the appropriate (usually the latest) approved base source code in the blockchain and downloads it locally, using the link to the developer-owned code repository provided in the UP metadata. We abstract this code repository in Figure 3.4 with the depicted decentralized storage solution. This conceptually can be any storage area that is not centrally-owned; from something very common, like a developer-owned Github repository, to something more elaborate as a content-addressable decentralized file system.

It is true that the review of source code is a task that requires extensive technical skills and experience. Therefore, it is not a task that can be assumed by the broad base of the stakeholders community. A voting delegation mechanism at this point must be in place, to enable the delegation of the strenuous code-approval task to some group of experts. In a similar logic with the delegation process, within the ideation phase, discussed above, the delegation process could be leveraged to implement different update policies per type of software update.

As we have seen, the voting approval of a SIP signals the beginning of the implementation phase for this SIP. The SIP has an estimated implementation elapsed time that was included in the SIP metadata, submitted along with the SIP at the ideation phase. This time period, increased by a contingency parameter, will be the available time window for a SIP to be implemented. Upon the conclusion of the implementation, a bundled (source code and metadata) UP is created. The UP must be uploaded to some (developer-owned) code repository and a content-based hash id must be produced that will uniquely identify the UP. This hash id will be submitted to the blockchain as a signal for a request to approval. This is accomplished with a specialized fee-supported transaction, which represents the decentralized equivalent to a pull-request. SIPs that fail to be implemented within the required time framework, will result to expired UPs and the SIP must be resubmitted to the ideation phase, as a new proposal. The UP submission transaction signals the entering into the approval phase.

3.2.3 Approval

The submission of an UP to the blockchain, as we have seen, is the semantic equivalent to a pull-request, in the decentralized approach. It is a call for approval. Indeed, the main goal of the approval phase is to approve the proposed new code; but what exactly is the approver called to approve for?

The submitted UP, which as we have seen, is a bundle consisting of source code, metadata and optionally produced binaries, must satisfy certain properties, in order to be approved:

- *Correctness and accuracy.* The UP implements correctly (i.e., without bugs) and accurately (i.e., with no divergences) the changes described in the corresponding voted SIP.
- *Continuity.* Nothing else has changed beyond the scope of the corresponding SIP and everything that worked in the base version for this UP, it continues to work, as it did (as long as it was not in the scope of the SIP to be changed).
- *Authenticity and safety.* The submitted new code is free of any malware and it is safe to be downloaded and installed by the community; and by downloading it, one downloads the original authentic code that has been submitted in the first place.
- *Fulfillment of update constraints.* We call the dependencies of an UP to other UPs, the potential conflicts of an UP with other UPs and in general all the prerequisites of an UP, in order to be successfully deployed, *update constraints*. The fulfillment, or not, of all the update constraints for an UP, determines the feasibility of this UP.

From the centralized approach perspective the above properties of the new code that the approver has to verify and approve are not uncommon. In fact, one could argue that these are the standard quality controls in any software development model. The first property has to do with testing; testing that verifies that the changes described in the SIP have been implemented correctly and accurately. In the centralized approach this means that the main maintainer of the code has to validate that the new code successfully passes specific test cases, either by reviewing test results, of executed test cases, or by running tests on his/her own. Regardless, of the testing methodology or type of test employed (unit test, property-based test, system integration test, stress test etc.), this is the basic tool that helps the central authority to decide on the correctness and accuracy of the new code.

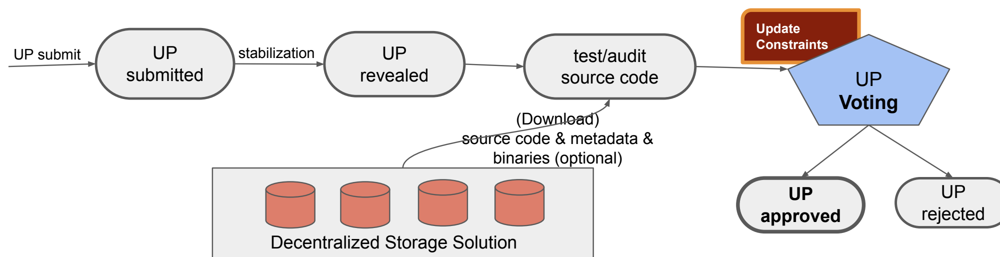
The second property for approving the new code has to do with not breaking something that used to work in the past. In software testing parlance, this is known as regression testing. Again, in the centralized approach, it is the main maintainer's responsibility to verify the successful results of regression tests run against the new code.

The third property has to do with the security of the new code and the authenticity of the downloaded software. The former calls for the security auditing of the new code. The latter, in the centralized case, is easy. Since, there is a trusted central authority (i.e., the main code maintainer), the only thing that is required, is for this authority to produce new binaries based on the approved source code, sign them and also the source code with his/her private key and distribute the signed code to the community. Then, the users only have to verify that their downloaded source code, or binaries, has been signed by the trusted party and if yes, then to safely proceed to the installation.

Finally, the last property that has to be validated by the approver pertains to the fulfillment of the update constraints. All the prerequisites of an UP must be evaluated and also the potential conflicts triggered by the deployment of an UP must be considered. For example, an UP might be based on a version of the software that has been depreciated, or rejected; or, similarly, it might be based on a version that has not yet been approved. Moreover, it might require the existence of third party libraries that it is not possible to incorporate into the software (e.g., they require licenses, or are not trusted). Then, we have the potential conflicts problem. What if the deployment of an UP cancels a previously approved UP, without this cancellation to be clearly stated in the scope of the corresponding SIP? All these are issues that typically a code maintainer takes into consideration, in order to reach at a decision for a new piece of code.

Once more, the essential part that differentiates the decentralized from the centralized approach is the lack of the central authority. All the properties that have to be validated basically remain the same but in this case the approval must be a collective decision.

Figure 3.5: The approval phase.



The approval phase in the decentralized approach is depicted in Figure 3.5. As we have seen, an UP is a bundle consisting of source code, update metadata and optionally binaries produced from the source code, aiming at a specific platform (e.g., Windows, Linux, MacOS etc.). The update metadata have to include basic information about the update, its justification, they have to clearly state all update constraints and finally declare the type of the change (e.g., bug-fix, or change request, soft/hard fork etc.) and priority, in order to enable the appropriate *update policy*. The UP bundle is uploaded to some developer-owned code repository and a unique hash ID, from hashing the content of the UP is produced. This UP hash ID is submitted to the blockchain, along with a link to the code repository.

Similar to the ideation phase (where the corresponding SIP was submitted), the submission of a UP is a special fee-supported transaction that can be submitted by any stakeholder. The UP is committed to the blockchain following again a hash based commitment scheme, in order to preserve the rightful authorship of the UP.

Once the UP is revealed the delegated experts (remember the delegation that took place during the implementation phase) for this UP, will essentially assume the role of the main code maintainer that we described in the centralized setting. In other words, they have to download the source code, metadata and possible binaries and validate the aforementioned properties. The tools (e.g., testing) that the experts have available for doing the validation are no different than the tools used by the main maintainer in the centralized approach. Moreover, if binaries for a specific platform have been uploaded by the UP submitter, then the delegated experts must go through the process of reproducing a binary from the source code and verifying that it matches (based on a hash code comparison) the one submitted. If not, then the submitted binary must be rejected and this will cause a rejection of the UP as a whole. So, there must be some extra caution when binaries are submitted along with source code, since the metadata need to include sufficient information for the approver to be able to reproduce the same binaries per platform.

Therefore the revealing of a UP, initiates a voting period for the specific proposal, in which the delegated experts must validate *all* the UP properties posed and approve, or reject it, with their vote. Any stakeholder is eligible to vote for an UP and the voting power will be proportional to his/her stake. If a stakeholder wishes to

cast a vote, although he/she has already delegated this right to an expert, then this vote will override the delegate's vote. Votes are specialized fee-supported transactions, which are committed to the blockchain.

One final note is that, as we have described, the decentralized approval phase that we propose, entails transaction fees. This means that the approval phase is not so flexible from a practical perspective, as to be used iteratively (although technically this is possible). In other words, to reject an UP, then fix some bugs and upload a new version for review etc. An UP rejection means that a resubmission must take place, with all the overhead that this entails (transaction fees, storage costs, a new voting must take place, etc.). This is a deliberate design choice that guards the system against DoS attacks. From a practical perspective though, it means that the submitted UPs must be robust and thoroughly tested versions of the code, in order to avoid the resubmission overhead. We do not want to pollute the immutable blockchain history with intermediate trial-and-error UP events.

3.2.4 Activation

The final phase in the lifecycle of a software update, depicted in Figure 3.2, is the activation phase. This is a preparatory phase before the changes actually take effect. It is the phase, where we let the nodes do all the manual steps necessary, in order to upgrade to an approved UP and at the end, send a signal to their peers that they are ready for the changes to be activated. Thus, the activation phase is clearly a signaling period. Its primary purpose is for the nodes to signal upgrade readiness, before the actual changes take effect (i.e., activate).

Why do we need such a signaling period in the first place? Why is not the approval phase enough to trigger the activation of the changes? The problem lies in that there are manual steps involved for upgrading to the new software, such as downloading and building the software from source code, which entail delays that are difficult to foresee and standardize. This results into the need for a synchronization mechanism between the nodes that upgrade concurrently. The lack of such a synchronization between the nodes, prior to activation, might cause a chain split. This synchronization mechanism exactly is the activation phase and for this, it is considered very important.

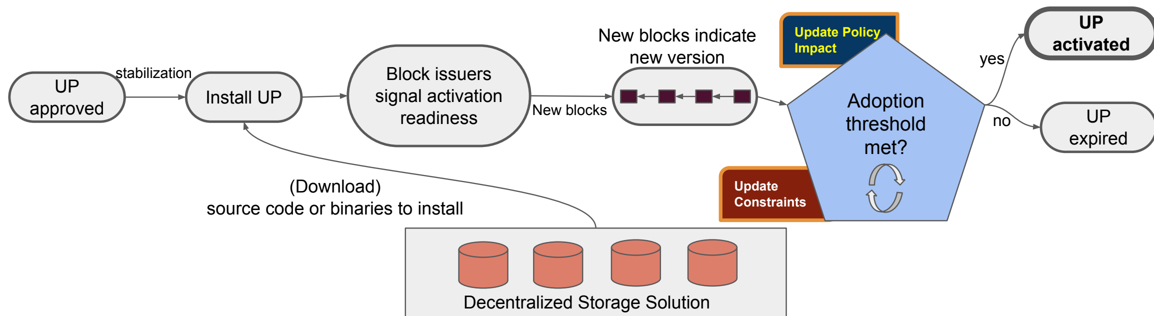
Clearly, the activation phase is not aimed as a re-approval phase for the UP. It is there to allow a smooth incorporation of the software update into the network. Therefore it becomes relevant only for those UPs that impact the consensus and can risk a chain split. For UPs that don't impact the consensus (e.g., a code refactoring, or some short of optimization, or even a change in the consensus protocol rules, which is a velvet fork [ZSJ+18]) there is essentially no need for an activation phase and the change can activate, as soon as the software upgrade takes place.

Traditionally, when a software update needs to be activated and it is known that it is likely to cause a chain split, a specific target date, or better, a target block number is set, so that all the nodes to get synchronized. Indeed, this is a practice followed by Ethereum [B+14]. All major releases have been announced enough time before the activation, which takes place when a specific block number arrives (i.e., the corresponding block is mined). All nodes must have upgraded by then, otherwise they will be left behind. In Bitcoin [Nak08], there also exists a signaling mechanism[2]. In this case, the activation takes place, only if a specific percentage of blocks (95%) within a retargeting period of 2016 blocks, signal readiness for the upgrade.

Once the UP approval result has been buried under a sufficient number of blocks (i.e., the stabilization period passes), then the activation period is initiated. In Figure 3.6, we depict the activation period in the decentralized setting.

[2] see BIP-9 at https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki

Figure 3.6: The activation phase.



The first step in the activation phase is the installation of the software update. Typically, as soon as the UP approval is stabilized in the blockchain, the GUI of the client software (e.g., the wallet) prompts the user to download and install the update, using the link that accompanies the UP. If in the UP bundle there exist an approved binary, then the user can download and install this, otherwise the user must download the approved source code. In the latter case, there exist an extra step of producing the binary code from the source code. In any case, it is important to note that the new software is just installed but not activated. It will remain in a latent state until the actual activation takes place.

For the nodes participating in the consensus protocol the installation of a software update means that they are ready to activate, but wait to synchronize with their other peers. To this end, they initiate signaling. This means that every new block issued will be stamped with the new version of the software, signifying their readiness for the new update.

When the first block with the new version appears, we enter the adoption period for the specific UP. During the adoption period the following conditions have to be met, in order for the activation to take place: a) The number of blocks with a signal must exceed a specific threshold, b) the update constraints for the specific UP must be fulfilled and c) the adoption time period must not be exceeded, otherwise the UP will become expired.

The blocks generated in a proof-of stake protocol are proportional to the stake and therefore, we can assume that the signaling mechanism is also proportional to the stake. We can also assume that the honest stake majority, will follow the protocol and eventually will upgrade and thus signal this event with their generated blocks. This means that the minimum expected percent of signals (i.e., the activation threshold) cannot be other that the minimum percent of honest stake majority required by the proof-of-stake consensus protocol. Of course, as we have noted above, for changes that don't impact the protocol, the activation threshold could be zero; meaning that even if only one node upgrades, then the changes can be immediately activated.

Moreover, the adoption time period is not fixed for all UPs. It varies based on the type of the change, which is something recorded in the UP metadata. One size does not fit all, and this is indeed true for the adoption time period of UPs. For example, major updates that require a lot of manual steps, or significant build time, or even hardware upgrade, should be adopted in a sufficient period of time, while small updates should be activated more swiftly.

Finally, before the actual activation of a change, the validation of all the update constraints must take place once more. This is true, although we make the assumption that from the approval phase all relevant update

constraints' issues (like conflicts, or dependencies) have been considered. The fact that the adoption period might require significant time for a UP, whose update constraints were fulfilled at the approval phase, while concurrently there are other UPs that become activated, means that the conditions might have changed and the update constraints must be reevaluated to make sure that no problems will arise upon activation of a UP.

Chapter 4

Toolkits

4.1 Toolkits for anonymous authentication and flexible consensus in Hyperledger Fabric

This section covers two toolkits that will extend the Hyperledger Fabric open-source blockchain platform described in Section 2.2. The first toolkit relates to anonymous authentication of parties, and the second toolkit relates to consensus mechanisms as described in Section 2.2.2.

4.1.1 Anonymous authentication in Hyperledger Fabric

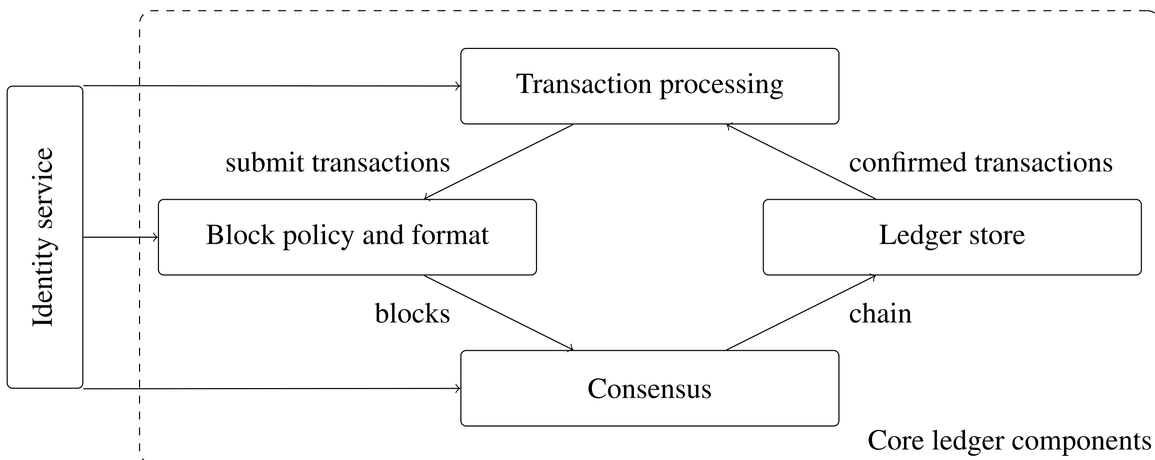


Figure 4.1: Connection between identity service and the core ledger components.

A permissioned blockchain system such as Hyperledger Fabric is dependent on an authentication infrastructure in order to differentiate between legitimate network participants and external parties. The fact that identities are used does not presume a central entity, as the identity service itself can be federated. In Fabric, each trust domain, such as one organization that participates in the platform, has its own *Membership Service Provider (MSP)* in which entities of that organization are enrolled and through which these entities are authenticated. The genesis block of the blockchain lists the MSPs for each organization. As each entity is authenticated via the MSP of its own organization, network nodes are always aware of the organization to which an entity belongs, and can make decisions based on how much they trust that organization. The role of an MSP in Fabric is depicted in Figures 4.1 and 4.2.

The current main MSP implementation in Fabric is built on X.509 certificates and is referred to as *Fabric-CA*.

Using X.509 has the advantage that legacy infrastructure can be used; however, it severely limits the privacy that can be achieved as all transactions are signed with respect to the client's certificate and endorsed with respect to those of the peers. To resolve these privacy concerns, an implementation of an MSP based on the Identity Mixer[1] anonymous credential system is currently under development.

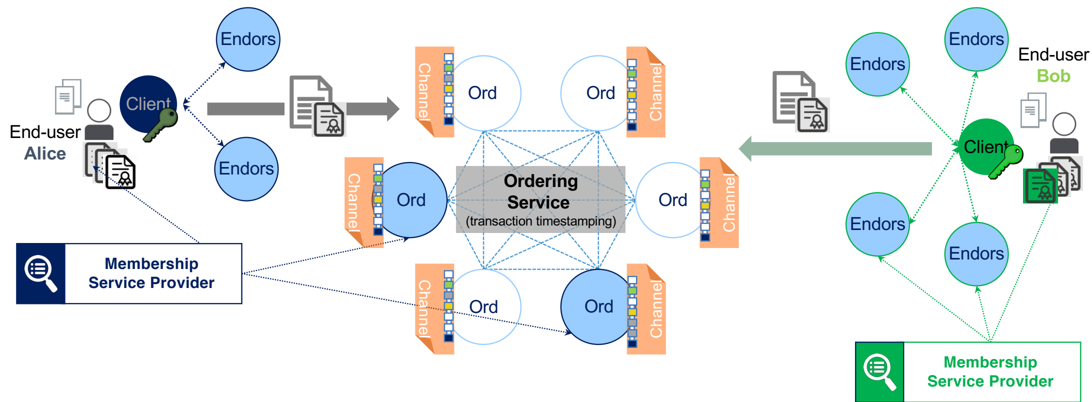


Figure 4.2: The MSP of each organization provides identities for all entities of that organization. Two organizations are depicted: Alice and Bob. Alice's MSP provides identities for Alice's endorsers (i.e. peers that execute the chaincode) and clients, and two orderer nodes under Alice's control. Bob's MSP provides identities for Bob's endorsers and clients; Bob does not control orderers.

The main implementation of an MSP in Fabric is based on X.509. Each organization uses an X.509 certificate authority (CA) to provide certificates for all its entities. The use of X.509 has the advantage that a lot of infrastructure is available already, and many large organizations already run an X.509 CA to manage some part of their infrastructure. This reduces the migration effort. Yet, X.509 certificates state all attributes of the respective entity in the clear. As network nodes must be able to verify transactions, this means that the certificate of the party authorizing the transaction must be known to those nodes. Consequently, an authentication infrastructure based on X.509 certificates does not allow clients to anonymously invoke transactions, which is, however, needed in order to protect user privacy and business data confidentiality in many use cases.

Anonymous credentials and Identity Mixer. An anonymous credential system allows a party to prove certain attributes about its identity without revealing any additional information. In a nutshell, an anonymous credential system has an Identity Provider, which acts similarly to a CA in a X.509 public-key infrastructure in that it provides certificates to the users. When presenting a certificate to a verifier, however, the user does not simply send the certificate it received from the Identity Provider. Instead, it proves with zero-knowledge that it knows the key associated to its credential and selectively opens attributes contained in the certificate. Identity Mixer is a sophisticated anonymous credential system, which was mostly developed at IBM Research and is presented in various research publications [CL01, CH02, CL04].

[1] https://www.zurich.ibm.com/identity_mixer/

Identity Mixer in Fabric. Identity Mixer has been included in Hyperledger Fabric since release v1.3.[2] This implementation, however, focuses on the basic functionality: generating anonymous credentials for users and verifying them in the context of a smart contract invocation. The current implementation misses out on several aspects needed for an enterprise-grade implementation, such as an efficient method for revoking users that leave an organization or whose secret information has been exposed.

The toolkit: Revocation of user identities. In public-key infrastructures based on X.509, revocation of identities is performed by regularly (e.g., once a day or week) publishing Certificate Revocation Lists (CRLs) that contain all certificates that have not expired but shall be considered invalid for other reasons, such as because an employee left an organization or the private key associated to a certificate was exposed. Some CAs additionally offer access via the Online Certificate Status Protocol (OCSP), which allows to interactively check whether a certain certificate is still considered valid.

An anonymous credential system cannot implement revocation in the same, straightforward manner, since the individual authentication sessions of a user shall be unlinkable. Either publishing the revocation information would break the privacy of past interactions, which is undesired in cases where the revocation is not related to exposure of the private key, or it would also not allow to detect the use of the certificate in future sessions.

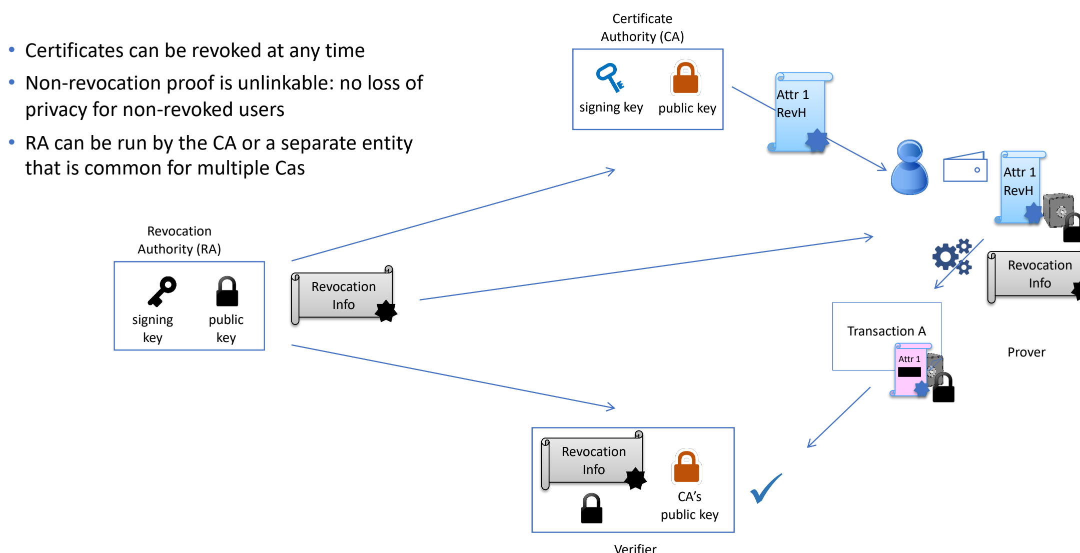


Figure 4.3: Roles in the revocation scenario. The Revocation Authority keeps track of the revoked credentials in the CRI, which is published to the CA, the provers, and the verifiers.

The system architecture of the solution we plan to implement in Fabric is depicted in Figure 4.3. The Revocation Authority (RA) keeps track of the revoked credentials in the Certificate Revocation Information (CRI). The CRI is periodically sent to the CA, and is also (partly) accessible for all provers and verifiers in the protocol. The solution is based on the following concepts:

Epoch: An epoch is a time window that is identified by an integer, the epoch number. Epoch numbers increase monotonically over time.

Revocation handle: A revocation handle is a specific (secret) attribute of a user that is included in the certificate.

[2] https://hyperledger-fabric.readthedocs.io/en/release-1.3/whatsnew.html

Revocation keys: A specific private/public key pair used by the CA to issue information related to revocation.

In Identity Mixer, the credential of a user is the signature of the Identity Provider on the attributes. Presenting a credential is performed by providing a (non-interactive) zero-knowledge proof of knowledge of this signature. This proof also contains a statement about the attributes that a user wants to present. In the method we propose, the attributes also contain the revocation handle. In every epoch, a user has to once obtain a new confirmation that his credential has not been revoked; this confirmation is the signature on the revocation handle and the epoch number, relative to the revocation keys. Presenting a credential then also includes knowledge of a confirmation for the current epoch number.

Alternatively, the CA could send the information to the users proactively after a new epoch is started. This is, however, first difficult to implement, since clients are not required to be online and cannot be contacted by the CA. Second, as (based on experience from use cases) clients tend to have long periods of inactivity, having them request the current period before issuing a transaction is expected to be cheaper than proactively creating the information for each client at the start of a new epoch, at least for most use cases.

4.1.2 Flexible consensus in Hyperledger Fabric

The purpose of the toolkit on *Flexible consensus in Hyperledger Fabric* is to support asymmetric trust assumptions. The toolkit builds on research performed within Work Package 3.

Traditional consensus algorithms in the permissioned setting such as Paxos or PBFT operate in a setting where the trust assumption is *symmetric*. In other words, a global assumption specifies which processes may fail, such as the simple and prominent *threshold quorum* assumption, in which any subset of the participants of a given maximum size may collude and act against the protocol. The standard threshold assumption, for instance, allows all subsets of up to $f < n/3$ processes to fail. Some classic works also model arbitrary, non-threshold symmetric quorum systems, but these have not actually been used in practice.

Trust, however, is inherently subjective. *De gustibus non est disputandum.* Estimating which processes will function correctly and which ones will misbehave may depend on personal taste. A myriad of local choices influences one process' trust in others, especially because there are so many forms of "malicious" behavior. Some processes might not even be aware of all others, yet a process should not depend on unknown third parties in a distributed collaboration.

The toolkit described in this section and planned as an outcome of PRIViLEDGE will implement the protocol that is developed in Work Package 3. As the research in that work package is still ongoing, the description of the protocol cannot be provided at this point in time. We proceed by describing the interfaces between Hyperledger Fabric and implementations of consensus protocols; these are the interfaces that the toolkit will implement.

Implementation in Hyperledger Fabric. As discussed in Section 2.2.2, 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, therefore, offers two types of APIs that we will describe in the following, one towards the clients, and one towards the peers.

Implementation in 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 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. As described in Section 2.2, 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). 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. As the exact internal software design of the component depends on the work on consensus protocols in WP3, which is currently in progress and is expected to be included in D3.2.

4.2 Toolkit for post-quantum secure protocols in distributed ledgers

This toolkit will provide client components to enable access to two security mechanisms expected to provide post-quantum security. Both mechanisms are based on hash functions which are widely believed to be resistant to quantum attacks, suffering only polynomial security losses (in contrast to the exponential losses of systems based on the hardness of factoring or computing discrete logaritms). Formal analysis of the security of these mechanisms in the quantum setting is an ongoing research effort, though.

The puropose of the BLT signature scheme is to provide an alternative autentication mechanism. The KSI time-stamping service in turn is a supporting component for the BLT signature scheme.

4.2.1 KSI time-stamping

The KSI blockchain provides a hash-and-publish time-stamping service backed by control publications in physical media. Once connected to the publications, the evidentiary value of the time-stamp tokens relies only on security properties of cryptographic hash functions and does not depend on asymmetric cryptographic primitives or secrecy of any keys or passwords [BKL13].

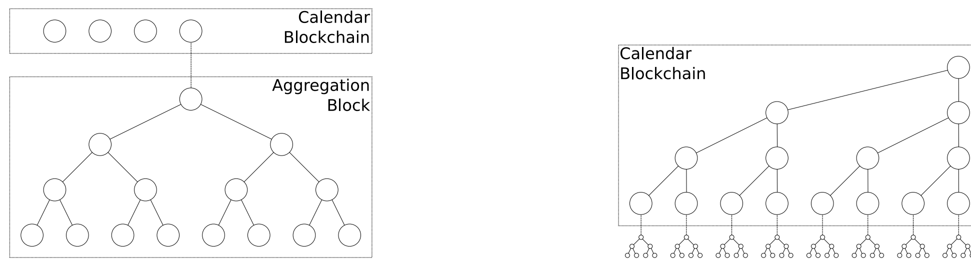


Figure 4.4: KSI architecture: aggregation (left) and linking (right).

Aggregation and linking. The service operates in fixed-length rounds. During each round, incoming client requests are aggregated into a globally distributed temporary hash tree (Fig. 4.4, left). At the end of the round, the root of the aggregation tree is added to an append-only data structure built around another hash tree, called the calendar blockchain (Fig. 4.4, right), and each client receives a two-part response consisting of

– a hash chain linking their request was to the root of the aggregation tree; and

– another hash chain linking the root of the aggregation tree to the new root of the calendar tree; this chain can be used to verify that the new state of the calendar blockchain was obtained from the previous one by appending the new aggregation tree root without changing anything else.

After that, the aggregation tree can be discarded and a new one is built for the next round.

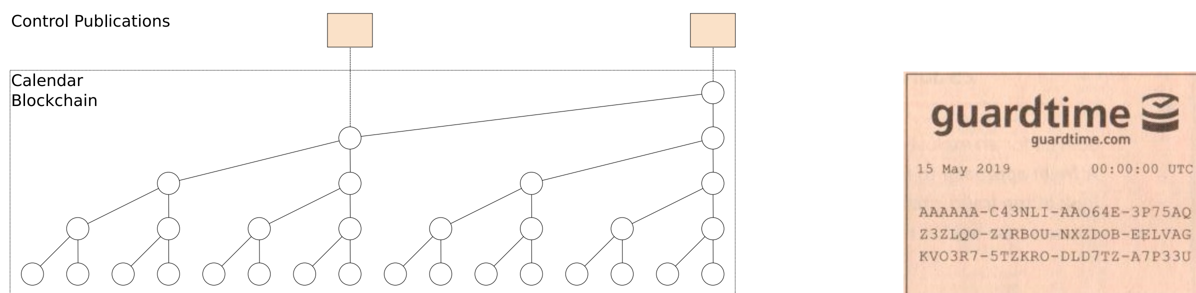


Figure 4.5: KSI publishing (left) and a publication (right).

Publishing. Periodically, the root hash value of the calendar blockchain is printed as a control publication in physical media (Fig. 4.5, left). Each such publication printed in a widely circulated newspaper (Fig. 4.5, right) acts as a trust anchor that protects the integrity of the history of the blockchain up to the round from which the publication was extracted.

Client passwords or keys may be used for service access control, but they do not affect the proof of data integrity or time. A set of symmetric keys is used to authenticate communications among the consensus nodes maintaining and updating the calendar blockchain, but their compromise also does not affect the evidentiary value of the time-stamp tokens already issued and linked to control publications.

A possible use of the KSI service in other ledgers is to time-stamp blocks of a private ledger to provide immunity against the so-called rewind-edit-replay attacks where the consortium managing the ledger agrees to retroactively edit some past transactions and modify all subsequent blocks to match the modified history. Alternatively, direct access to the KSI service from smart contract applications may be enabled for similar purposes.

Of course, the consortium maintaining the ledger could issue their own publications, but this would incur additional cost and hassle that can be avoided by delegating the task to the KSI service designed specifically for

the purpose. Additionally, most ledgers have a linear structure and a proof linking any particular transaction to a publication would have to traverse many blocks. The KSI blockchain uses a hash tree to enable proofs whose size is logarithmic in the number of aggregation rounds.

4.2.2 BLT signature scheme

BLT combines a time-stamping service and message authentication codes with time-bound one-time keys to obtain a server-assisted signature scheme whose long-term evidentiary value relies only on security of cryptographic hash functions and the assumption of secrecy of the one-time keys up to the signing time [BLT17].

Key generation. Each of the signing keys $z_1, z_2, \ldots, z_n$ is generated as an unpredictable value drawn from a sufficiently large set. Each key is pre-bound to a designated usage time by computing commitmets $x_i = h(t_i, z_i)$, where h is a hash function and t_i is the designated usage time for the key z_i. The commitments are aggregated into a hash tree T and the root hash value of the tree is published as the signer's public key p.

Signing. To sign the message m at time t_i, the signer authenticates the message with the corresponding signing key by computing $y = h(m, z_i)$ and then proves the time of usage of the key by obtaining a time-stamp a_t on the message authenticator y. The signature is then composed as $\sigma = (t_i, z_i, a_t, c_i)$, where c_i is the hash chain authenticating the membership of the pair (t_i, z_i) in the hash tree of the client's signing keys. Note that it is safe to release z_i as part of the signature, as its designated usage time has passed and thus it can't be used to generate any new signatures.

Verification. To verify the signature $\sigma = (t, z, a, c)$ on message m against the public key p, the verifier

- checks that z was commited as signing key for time t by verifying that the hash chain c links the commitment $x = h(t, z)$ to the public key p; and
- checks that m was authenticated with z at time t by verifying that the hash chain a links the authenticator $y = h(m, z)$ to the summary commitment R_t of the time-stamping aggregation round for time t.

The primary goal of the BLT component of the toolkit is to provide a replacement for asymmetric-key based digital signature schemes as transaction authentication mechanism.

Note that the security of the signature scheme critically depends on the security of the time-stamping service against back-dating attacks by the adversary. Therefore, a post-quantum secure time-stamping service is a necessary pre-condition for post-quantum security of the signature scheme.

Additionally, the ledger whose transactions are to be authenticated by the signatures cannot itself be used to prove the usage times of the keys, and therefore an external time-stamping service is needed for that purpose.

Chapter 5

Conclusions

This deliverable described the architecture of protocols for secure ledger systems to provide a high level structure that can be used, without ambiguity, to describe the protocol architecture of use cases and toolkits. Following this approach, an overview is provided for the architecture of the secure ledger systems that are more relevant to PRIViLEDGE: Hyperledger Fabric and Cardano. The final part of the document (Chapter 3 and 4) presented the preliminary outputs of the research done in the context of use case 4 and of the toolkits related to Hyperledger Fabric and to secure ledger systems in a post-quantum scenario. Overall, this document provides a global picture of the progress of the research done to date, and the process of writing the deliverable has provided many advances and a better understanding of the studied problems. This has allowed designing a detailed architecture for the Cardano update system (use case 4) and for the toolkit for anonymous authentication. The next steps in terms of research will concern the problems left open in this deliverable, that are mainly related to the achievement of a better understanding on how to concretely realize (in terms of cryptographic primitives) the protocols proposed at a high level in this document.

Bibliography

[ABB$^+$18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018.

[B$^+$14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[BCD$^+$14] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. 2014. `http://www.opensciencereview.com/papers/123/enablingblockchain-innovations-with-pegged-sidechains`.

[BKL13] Ahto Buldas, Andres Kroonmaa, and Risto Laanoja. Keyless signatures' infrastructure: How to build global distributed hash-trees. In Hanne Riis Nielson and Dieter Gollmann, editors, *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*, volume 8208 of *Lecture Notes in Computer Science*, pages 313–320. Springer, 2013.

[BLT17] Ahto Buldas, Risto Laanoja, and Ahto Truu. A server-assisted hash-based signature scheme. In Helger Lipmaa, Aikaterini Mitrokotsa, and Raimundas Matulevicius, editors, *Secure IT Systems - 22nd Nordic Conference, NordSec 2017, Tartu, Estonia, November 8-10, 2017, Proceedings*, volume 10674 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2017.

[CD17] Ignacio Cascudo and Bernardo David. SCRAPE: Scalable randomness attested by public entities. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 537–556. Springer, Heidelberg, July 2017.

[CH02] Jan Camenisch and Els Van Herreweghen. Design and implementation of the *idemix* anonymous credential system. In Vijayalakshmi Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 21–30. ACM, 2002.

[CL01] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Birgit Pfitzmann, editor, *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 93–118. Springer, 2001.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In Matthew K. Franklin, editor, *Advances in Cryptology - CRYPTO 2004, 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 56–72. Springer, 2004.

[DGKR17] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. Cryptology ePrint Archive, Report 2017/573, 2017.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 416–431. Springer, Heidelberg, January 2005.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, August 2017.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[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*, pages 51–58. IEEE Computer Society, 2018.

[ZOB18] Bingsheng Zhang, Roman Oliynykov, and Hamed Balogun. A treasury system for cryptocurrencies: Enabling better collaborative intelligence. Cryptology ePrint Archive, Report 2018/435, 2018. `https://eprint.iacr.org/2018/435`.

[ZSJ+18] Alexei Zamyatin, Nicholas Stifter, Aljosha Judmayer, Philipp Schindler, Edgar R. Weippl, and William J. Knottenbelt. A wild velvet fork appears! Inclusive blockchain protocol changes in practice - (short paper). In *Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*, pages 31–42, 2018.