



DS-06-2017: Cybersecurity PPP: Cryptography


PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D4.4 – Report on Tools for Secure Ledger Systems

Due date of deliverable: 30 June 2021
Actual submission date: 29 June 2021

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

Lead contractor: Guardtime
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	

D4.4

Report on Tools for Secure Ledger Systems

Editor

Marko Vukolić (IBM)

Contributors

Kaoutar Elkhyaoui (IBM)
Nikos Karagiannidis (IOResearch)
Damian Nadales (IOResearch)
Ahto Truu (GT)

Reviewers

Nikos Voutsinas (GUNET)
Thomas Zacharias (UEDIN)

29 June 2021

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

This document presents the final report on toolkits, use cases and prototype solutions for secure ledger systems developed in PRIViLEDGE. It extends the architecture of these toolkits and use cases, presented in an earlier WP4 deliverable, D4.1. Toolkits and use cases whose architecture had not been significantly changed compared to the earlier D4.1 deliverable have been omitted.

In summary, D4.4. presents the final report on two PRIViLEDGE toolkits developed by IBM: a toolkit for anonymous authentication (Chapter 2) and a toolkit for flexible consensus in Hyperledger Fabric (Chapter 3), a toolkit for post-quantum secure protocols for distributed ledgers developed by Guardtime (Chapter 4). Moreover, it presents the report on decentralized software updates for Cardano stake-based ledger use case by IOResearch (Chapter 5).

Contents

1	Introduction	1
2	Toolkit for Anonymous Authentication	2
2.1	Anonymous Credentials for Hyperledger Fabric	2
2.2	Transaction Signature	2
2.3	Selective Disclosure of Attributes and Applications to Access Control	3
2.4	IDemix Extensions [BCET19]	3
2.4.1	Towards Multi-MSP Anonymous Credentials	3
2.4.2	Credential Revocation	3
2.4.3	Transaction Audit	3
2.4.4	Performance Evaluation	3
2.4.5	Availability	4
3	Toolkit for Flexible Consensus in Hyperledger Fabric	5
3.1	Introduction to Consensus in Hyperledger Fabric	5
3.2	Flexible Number of Leaders in a Consensus Protocol	8
3.3	Architecture of Request Duplication Prevention with Multiple Leaders	8
3.4	Availability: Mir-BFT as an Open-Source Project — a Hyperledger Lab	9
4	Toolkit for Post-Quantum Secure Protocols for Distributed Ledgers	10
4.1	KSI Time-Stamping	10
4.2	KSI Toolkit	11
4.3	BLT Signature Scheme	11
4.4	BLT Toolkit	11
4.5	Availability	12
5	Use Case: Decentralized Software Updates for Cardano Stake-Based Ledger	13
5.1	The Lifecycle of a Decentralized Software Update	13
5.2	Update Governance	17
5.2.1	Voting for Software Updates	17
5.2.2	Delegation to Experts ¹	19
5.3	The Activation of Changes	23
5.3.1	Overview	23
5.3.2	Secure Activation Protocols	24
5.3.3	Activation Phase Design	25
5.4	Threshold Analysis	30
5.5	Availability	34

¹Note that for the scope of our prototype implementation we have assumed delegation as an out-of-band solution. This is basically for two reasons: a) to reduce risks in the prototype implementation (especially of the Cardano integration part) and b) to defer introducing delegation to experts until a proper game-theoretic analysis of the expert’s incentives has been completed.

Chapter 1

Introduction

This deliverable covers the report on protocols, toolkits and use cases for secure ledger systems developed within PRIViLEDGE. 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 addition, ledgers are secured using cryptographic toolkits, such as an anonymous authentication toolkit and hash-based time-stamping and signature toolkits. This deliverable builds on the architecture toolkits and use cases for secure ledgers presented in an earlier WP4 deliverable, D4.1.

The work-document is divided in four parts.

In Chapter 2, we report on toolkit for anonymous authentication, developed by IBM, focusing on Anonymous Credentials in Hyperledger Fabric open-source permissioned blockchain. In particular, we focus on the way Hyperledger Fabric (HLF) uses IDemix to allow users to submit transactions anonymously. IDemix is an implementation of anonymous credentials that assumes that each permissioned blockchain network has a single Membership Service Provider (MSP).

In Chapter 3, we report on a toolkit for flexible consensus in Hyperledger Fabric, also developed by IBM. The initial architecture of these two toolkits was presented in Deliverable D4.1. In this deliverable, we pay particular attention to support for flexible number of leaders (i.e., multiple leaders) in leader-based Byzantine fault-tolerant consensus protocols. This chapter also discusses the parallel networking and processing capabilities that such a flexible toolkit needs to support. Finally, for completeness, we present the updates related to open-sourcing activities, as the basis for the Toolkit for Flexible Consensus in Hyperledger Fabric, became a standalone Hyperledger Lab project, called Mir-BFT.

In Chapter 4, we present the toolkit for post-quantum secure protocols for distributed ledgers developed by Guardtime. This toolkit provides client with the components to enable access to two security mechanisms, the KSI time-stamping and the BLT digital signatures, expected to provide post-quantum security (resilience to attacks that could be mounted using quantum computers). Both mechanisms are based on hash functions which are widely believed to be resistant to quantum attacks.

Finally, in Chapter 5, we come back to use case UC4: decentralized software updates for Cardano stake-based ledger and give the logical architecture of an update mechanism for proof-of-stake ledgers, with the focus on the Cardano blockchain. We present all the remaining technical details of our update mechanism that have not been already reported in the previous deliverables, namely D4.1 “Report on Architecture of Secure Ledger Systems” [PRI19] and D4.3 “Final Report on Architecture” [PRI20]. In particular, we start with an overview of the distinct phases of our update mechanism, which correspond to the lifecycle of a decentralized software update, then we provide a detail design of our update governance scheme, then we present the technical details of the activation phase, which is the critical phase where the changes actually take effect on the chain, and conclude with a threshold analysis that demonstrates the important trade-off between safety and liveness.

Chapter 2

Toolkit for Anonymous Authentication

Protecting the privacy of a blockchain transaction consists of hiding not only its content but also its origin. In permissionless blockchains, anonymity is ensured through ephemeral signing keys. This however, is not an option for permissioned networks. Luckily, solutions based on anonymous credentials can be effectively leveraged to allow blockchain participants to sign transactions using their long-term identities while ensuring their anonymity. The only information such a signature leaks is that the origin of the transaction is a member of the network.

2.1 Anonymous Credentials for Hyperledger Fabric

Hyperledger Fabric (HLF) uses IDemix to allow users to submit transactions anonymously. IDemix is an implementation of anonymous credentials that assumes that each blockchain network has a single Membership Service Provider (MSP). When a user enrolls into the network, they engage in an interactive zero-knowledge proof (ZKP) to obtain a credential from the MSP. The ZKP helps the user to prove that they know the secret key underlying their public key. If the MSP is convinced, then it generates a signature that binds the user's secret key to their attributes.

In HLF, a user is associated with three attributes: the *Organization* that they belong to denoted *org*, their *Role* within the organization denoted *role*, and finally, their unique *Enrollment ID*, *eid* for short. Before issuing a user credential, the MSP is required to check the authenticity of the user's attributes. If the user is who they claim to be, then MSP returns an IDemix credential. This corresponds to a variant of Boneh-Boyen signature [CKS09, BB04] that yields itself easily to zero-knowledge proofs of knowledge of signatures.

2.2 Transaction Signature

An HLF transaction contains two dedicated fields for transaction authorization: *Creator*, which encodes the identity of the transaction submitter, and *Signature*, which represents the authorization of the transaction.

To sign an HLF transaction anonymously, a user first generates a *Pseudonym* *nym*, which is a Pedersen commitment of the user's secret key *sk* with randomness *r*, and places it in the *Creator* field. After *Pseudonym* generation, the user produces a Schnorr-based ZKP of knowledge of tuple $t = (sk, org, role, eid, r)$, where *is* and a Boneh-Boyen signature σ such that the following holds:

- $nym = g^{sk}h^r$;
- $verify(t', \sigma, pk_{msp}) = 1$ where $t' = (sk, org, role, eid)$ and pk_{msp} is the MSP's public key.

A transaction is effectively signed by including its content in the challenge of the above ZKP. Consequently, a transaction is authenticated by checking that the ZKP is valid and that the transaction is part of the ZKP challenge.

2.3 Selective Disclosure of Attributes and Applications to Access Control

Not only does IDemix facilitate anonymous authentication, but it also helps enforce access control rules. Take for example the following access control policy "only *admins* can submit config transactions". This can be easily enforced using IDemix by having *admins* disclose the value of attribute *role* when signing config transactions.

2.4 IDemix Extensions [BCET19]

We extend IDemix to ensure anonymity in the presence of multiple MSPs. We also enhance it with revocation and auditing capabilities.

2.4.1 Towards Multi-MSP Anonymous Credentials

IDemix only ensures perfect anonymity in single-MSP setting, otherwise, IDemix leaks the public key of the MSP. To address this issue, Camenisch et al. [CDD17] introduced a solution for delegatable credentials that enables users to sign transactions without revealing the public key of their MSP. The solution relies on a root authority that delegates issuance capabilities following a chain of delegation: the root authority delegates issuance capabilities to intermediate MSPs, which in turn delegate issuance to other intermediate MSPs; the last level of the chain corresponds to users. The proposed scheme combines Groth signatures [Gro15] and Schnorr signatures, such that the former are used to delegate credential issuance capabilities, whereas the latter are used to sign messages. A user signs a message by showing that there is a chain of delegation from their MSP to the root authority. The signature, as such, does not leak the public key of the user's MSP, instead only the public key of the root authority is revealed.

2.4.2 Credential Revocation

To enable efficient and privacy-preserving revocation we couple epoch-based whitelisting with signatures in a way that yields efficient proofs of non-revocation. Namely, we divide the timeline into epochs that define the validity periods of the credentials. For each epoch, a non-revoked participant requests an epoch handle (a signature) from a non-revocation authority. The epoch handle binds the public key of the user to the epoch.

Now, when a user submits a transaction, they provide a proof of non-revocation that shows in zero-knowledge that the user holds an epoch handle for the current epoch. Credentials that are valid for a certain epoch are automatically revoked the moment the epoch expires. An epoch expires either naturally (epoch elapses) or manually (authorized parties advance the epoch by putting a special message on the ledger).

Epochs are defined in terms of ledger height. This ensures that transactions of revoked users will be rejected by all peers in HLF networks.

2.4.3 Transaction Audit

Assuming that each HLF network is assigned a single auditor, we are able to inspect the origin of transactions. First, the auditor is assigned an Elgamal public key that all network participants know. IDemix signature, consequently, is enhanced with an Elgamal ciphertext and a ZKP that shows the ciphertext encrypts the public key of the transaction submitter under the public key of the auditor. Given its Elgamal secret key, the auditor can trace the origin of all the transactions recorded in HLF ledger.

2.4.4 Performance Evaluation

We have run comprehensive benchmarks for our extensions. All benchmarks were run on `onc2-standard-60GCE` VM running Ubuntu 18.04 (60vCPUs, Intel Cascade Lake 3.1 GHz, 240 GB RAM). We used the go version of

the Apache Milagro Cryptographic Library (AMCL) [Sco] with a 254-bit Barreto-Naehrig curve [BN05]. The benchmarks involve no pre-computations and all benchmarked operations were run 100 times.

In HLF, having 2-level credential delegation and 2 attributes (*Role*, *EnrollementID*) covers most use-cases. Attribute *Organization* is represented by the identifier of the MSPs at the last level of the delegation chain (i.e. users from *Org1* receive credentials from *MSP1*).

The results of the benchmarks show that signature generation takes 192 *ms* whereas the corresponding verification takes 198 *ms*. The size of the signature, on the other hand, is around 1.6 *KB*.

For more details on the implementation and its performances, the reader can refer to [BCET19].

2.4.5 Availability

This toolkit is available from <https://github.com/IBM/dac-lib> under MIT licence. Please note that its maturity is currently at the level of research prototype.

Chapter 3

Toolkit for Flexible Consensus in Hyperledger Fabric

This section covers architectural extensions to the Toolkit for Flexible Consensus in Hyperledger Fabric, described initially in Deliverable D4.1 and extended in Deliverable D4.3. The toolkit on *Flexible consensus in Hyperledger Fabric* builds on research performed within Work Package 3. The description of the specific protocols is provided in Deliverable D3.3, which is submitted concurrently with this deliverable. For completeness, we include the summary of information provided in D4.1 and D4.3 relevant to this toolkit. We extend this by reporting on open-sourcing efforts of this toolkit, which is open sourced as Mir-BFT Hyperledger Lab.

3.1 Introduction to Consensus in Hyperledger Fabric

Generic ledger architecture

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

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

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

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

The consensus protocol may keep state (such as the set of current protocol participants) and may furthermore call out the transaction processing component for verification of the transactions contained in a newly received

block. In a permissioned setting, writing to the ledger can be restricted to eligible parties that can be held accountable.

Architecture Implementation in Hyperledger Fabric.

Hyperledger Fabric¹ 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, 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.

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 distributes 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.

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

With Fabric v1.0, two ordering service implementations were provided in the main distribution: *solo* and *kafka*. The *solo* version implements a trivial type of consensus: a single node is used to process all blocks. This implementation is not meant for production purposes and is mostly used for development. The *kafka* implementation is based on Apache Kafka², a distributed streaming platform that follows the publisher/subscriber paradigm and offers crash-fault tolerant (CFT) operation based on the Apache ZooKeeper³ 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 in presence of malicious nodes. The subsequent release Fabric 1.4.1 added support for the Raft consensus protocol based on etcd⁴, another CFT consensus system. Just like *kafka*, it does not improve the resiliency against misbehaving nodes. (The system tolerates malicious clients transmitting transactions and malicious peer nodes executing smart contracts up to a level specified in the contract policies.). From Fabric v2.0, *solo* and *kafka* are deprecated in Fabric, and only *raft* is recommended as Fabric ordering service.

However, *raft* does not provide the resilience necessary in use cases where there is no single party that can be trusted to properly perform the ordering. An experimental Byzantine-fault tolerant ordering service based on the BFT-SMaRt implementation has been described by Sousa, Bessani, and Vukolić [?]. The paper showed that while the performance of 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 [?]. An implementation of a BFT ordering service that is planned⁵ to be included in the main distribution of Fabric is currently under development by IBM Research – Zurich, and the protocol, called Mir-BFT, will be

¹<https://www.hyperledger.org/projects/fabric/>

²<https://kafka.apache.org/>

³<https://zookeeper.apache.org/>

⁴<https://coreos.com/etcd/>

⁵<https://jira.hyperledger.org/browse/FAB-33>

described in Deliverable D3.3. The mechanism of Mir-BFT is based on the PBFT protocol [?], but targeting improved transaction throughput. We detail the main architecture principles of Mir-BFT in Sections 3.2 and Section 3.3.

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

The `ConsenterSupport` interface in particular specifies a so-called *block cutter* that implements the policy service that determines which transactions will be included in the next block. (This *policy* is thereby separated from the consensus *mechanism*.) The interface also allows to access a shared configuration that contains parameters controlling the behavior of the consensus algorithms; these are specific to each consensus method but read and provided by the main ordering service software.

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

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

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

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

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

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

3.2 Flexible Number of Leaders in a Consensus Protocol

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

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

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

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

3.3 Architecture of Request Duplication Prevention with Multiple Leaders

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

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

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

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

3.4 Availability: Mir-BFT as an Open-Source Project — a Hyperledger Lab

As of April 2021, Mir-BFT is a fully open source project under the auspices of Hyperledger. It is open sourced as a Hyperledger Lab, which denotes projects that are incubating towards full-fledged Hyperledger projects.

Mir-BFT is licensed under Apache 2.0 license and is available at <https://github.com/hyperledger-labs/mirbft>. It is developed as a library, which will be used by Hyperledger Fabric, but could be also used by other permissioned and permissionless blockchain projects. In the following, we briefly overview the structure of the Mir-BFT open source library. Currently the production-level library is under development.

The research prototype implementation, described in this deliverable and in details, in Deliverable D3.3, is available in the “research” branch of the above mentioned repository, i.e., at <https://github.com/hyperledger-labs/mirbft/tree/research>.

The high level structure of the MirBFT library is inspired by etcd-raft. The protocol logic is implemented as a state machine that is mutated by short-lived, non-blocking operations. Operations which might block or which have any degree of computational complexity (like hashing, I/O, etc.) are delegated to the caller.

Additional components required to use the library are:

- A write-ahead-log (WAL) that the protocol uses for persisting its state. This is crucial for crash-recovery - a recovering node will use the entries in this log when re-initializing. External actions (like sending messages or interacting with the application) are always reflected in the WAL before they occur. This way a crash of a node and a subsequent recovery are completely transparent to the rest of the system, except a potential delay incurred by the recovery. The users of this library can use their own WAL implementation or use the library-provided one
- A request store for persisting application requests. This is a simple persistent key-value store where requests are indexed by their hashes. The users of this library can use their own request store implementation or use the library-provided one.
- A hashing implementation. Any hash implementation can be provided that satisfies Go’s standard library interface, e.g., sha256.
- An application that will consume the agreed-upon requests and provide state snapshots on library request. Looking at MirBFT as a state machine replication (SMR) library, this application is the “state machine” in terms of SMR. It is not to be confused with the library-internal state machine that implements the protocol.
- A networking component that implements sending and receiving messages over the network. It is the networking component’s task to establish physical connections to other nodes and abstract away addresses, ports, authentication, etc. The library relies on nodes being addressable by their numerical IDs and the messages received being authenticated.

For basic applications, the library-provided components should be sufficient. To increase performance, the user of the library may provide their own optimized implementations.

Chapter 4

Toolkit for Post-Quantum Secure Protocols for Distributed Ledgers

This toolkit provides client components to enable access to two security mechanisms, the KSI time-stamping and the BLT digital signatures, expected to provide post-quantum security. Both mechanisms are based on hash functions which are widely believed to be resistant to quantum attacks. The best known quantum attacks against hash functions still have exponential complexity, in contrast with the polynomial complexity of attacks against systems based on the hardness of integer factoring or computing discrete logarithms. Formal analysis of the security of these mechanisms in the quantum setting is an ongoing research effort, though.

The purpose of the BLT signature scheme is to provide an alternative authentication mechanism. The KSI time-stamping service is a supporting component for the BLT signature scheme, but can also be of independent value in providing long-term proofs of integrity. The latter includes post-quantum indemnification for ledger operations carried out based on authentication with quantum-vulnerable mechanisms: the post-quantum secure time-stamps can prove that the quantum-vulnerable operations were indeed completed before the advent of practical quantum computers, and thus could not have been affected by quantum attacks.

4.1 KSI Time-Stamping

KSI blockchain [BKL13] provides a hash-and-publish time-stamping service backed by control publications in physical media. Once connected to the publications, the 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.

The service operates in fixed-length rounds. During each round, incoming client requests are aggregated into a globally distributed temporary hash tree. 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, and each client receives a two-part response consisting of

- a hash chain linking their request 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.

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

Periodically, the root hash value of the calendar is printed as a control publication in physical media. Each such publication printed in a widely circulated newspaper 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 value of the time-stamp tokens already issued and linked to control publications.

4.2 KSI Toolkit

The KSI toolkit enables calls to the KSI time-stamping service from Hyperledger Fabric (HLF).¹ It is implemented as a wrapper around the open-source KSI Java SDK.²

The toolkit also includes a daemon that monitors the HLF ledger and adds KSI time-stamps to new blocks as they are appended to the ledger, and a tool for verifying the integrity of the time-stamped blocks. These could be seen as examples of how calling the KSI service from within HLF works, or used as-is for protecting the integrity of the history of a private ledger via quantum-resistant commitments linked to external trust anchors.

4.3 BLT Signature Scheme

BLT signature scheme [BLT17] combines hash-then-publish time-stamping and hash-function based authentication with time-bound one-time keys to obtain a server-assisted signature scheme whose long-term value relies only on security of cryptographic hash functions and the assumption of secrecy of each one-time key up to the signing time.

Each of the signing keys z_1, z_2, \dots, z_n is generated as an unpredictable value drawn from a sufficiently large domain. Each key is pre-bound to a designated usage time by computing commitments $x_i = h(t_i, z_i)$, 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 .

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 cannot be used to generate any new signatures.

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

- checks that z was committed 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 .

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.

4.4 BLT Toolkit

The primary goal of the BLT toolkit is to provide a post-quantum replacement for asymmetric-key based digital signature schemes as transaction authentication mechanism in Hyperledger Fabric (HLF).³ The toolkit is implemented as a wrapper around the BLT Java SDK.

The toolkit also includes an extension of the HLF Membership Service Provider (MSP) component to enable use of BLT signatures as an alternative to the ECDSA signature scheme supported by the stock implementation.

¹<https://www.hyperledger.org/use/fabric>

²<https://github.com/guardtime/ksi-java-sdk>

³<https://www.hyperledger.org/use/fabric>

4.5 Availability

The KSI toolkit is available from <https://github.com/guardtime/ksi-hlf> under the Apache licence. Please note that its maturity is currently at the level of research prototype.

To use the toolkit, access to KSI service is needed. For research and testing purposes, access may be requested via the Guardtime KSI developer program at <https://guardtime.com/blockchain-developers>.

The BLT Java SDK is currently not public, and thus also the BLT toolkit cannot yet be released publicly. Please check <https://github.com/guardtime/> and <https://guardtime.com/> for future updates.

Chapter 5

Use Case: Decentralized Software Updates for Cardano Stake-Based Ledger

Software updates are a synonym to software evolution and thus are ubiquitous and inevitable to any blockchain platform. In the fourth use case of PRIViLEDGE, we propose a general framework for decentralized software updates in distributed ledger systems. Our framework is primarily focused on Proof of Stake blockchains and aims at providing a solid set of enhancements, covering the full spectrum of a blockchain system, in order to ensure a decentralized, but also secure update mechanism for a public ledger. We identify what are the critical decisions in the lifecycle of a software update and then propose a secure software update protocol that covers the full lifecycle of a software update from the ideation phase (the moment in which a change to the blockchain protocol is proposed) to the actual activation of the updated blockchain protocol, which enables decentralized decision making for all critical decisions. We propose a liquid democracy scheme based on experts for all the critical, but also deeply technical, decisions for software updates. We formally define what it means for a decentralized software update system to be secure and propose secure activation protocols [CKKZ20] with various trade-offs. We deal with the complexity of priorities, version dependencies, conflicts resolution and emergency handling for the activation of updates and propose an elegant design. We perform voting and activation threshold analysis, in order to achieve both properties of safety and liveness, which we define in detail. Finally, we implement our ideas into a research prototype that we have integrated with the Cardano node [CF15] based on the architecture that we have proposed in ([PRI20]).

In this deliverable, which is the last from a series of WP4 architecture/design/implementation deliverables, we present all the remaining technical details of our update mechanism that have not been already reported in the previous deliverables, namely D4.1 “Report on Architecture of Secure Ledger Systems” [PRI19] and D4.3 “Final Report on Architecture” [PRI20]. In particular, we start with an overview of the distinct phases of our update mechanism, which correspond to the lifecycle of a decentralized software update, then we provide a detailed design of our update governance scheme, then we present the technical details of the activation phase, which is the critical phase where the changes actually take effect on the chain and conclude with a threshold analysis that demonstrates the important trade-off between safety and liveness.

5.1 The Lifecycle of a Decentralized Software Update

A *software update (SU)* is the unit of change for the blockchain software. In Figure 5.1, we depict the full lifecycle of a software update following a decentralized approach. In this lifecycle, we identify four distinct consecutive phases: a) the *ideation phase*, b) the *implementation phase*, c) the *approval phase* and d) the *activation phase*. In the subsequent subsections, we provide a detailed description of each individual phase.

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. 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.

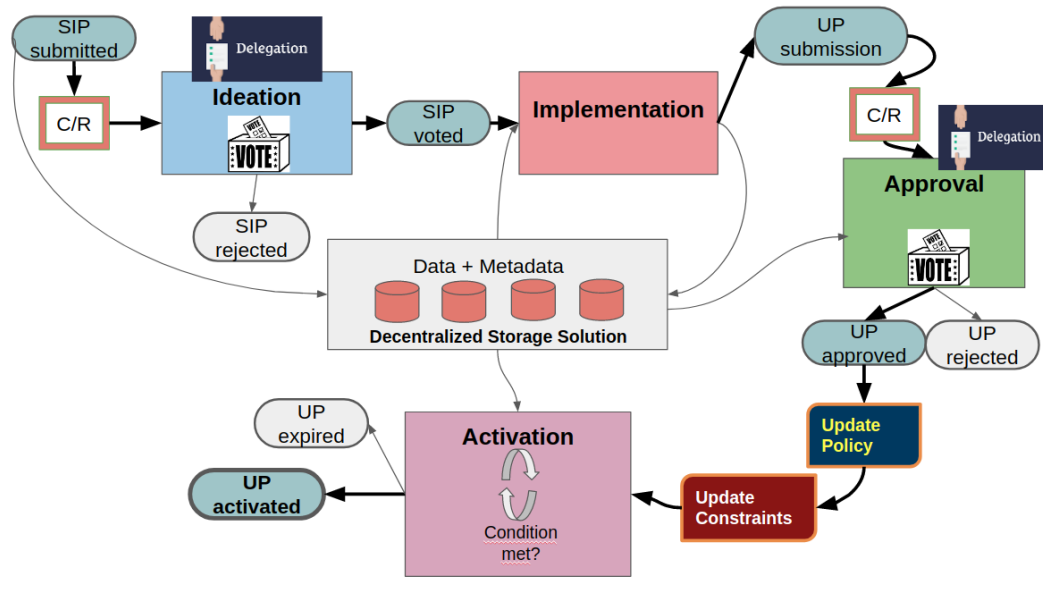


Figure 5.1: The lifecycle of a software update (a decentralized approach).

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.

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* (*Software¹ Improvement Document*). The SU life starts by submitting the corresponding SIP to the blockchain by means of a 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), the priority/criticality of the SU 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 other SUs (e.g., update dependencies or update conflict issues), or against any prerequisites required, in order to be applied. We call these requirements as *update constraints* and can be abstracted as a predicate, whose evaluation determines the feasibility of a software update.

We assume that the actual proposals are stored in some content addressable P2P storage system that ensures high availability. In parallel and for practical purposes, specifically for the Cardano case, we can assume that the proposals are also stored in a central server maintained by the Cardano Foundation. These are abstracted in Figure 5.1) with the component termed “Decentralized Storage Solution”. Since for practical reasons we have to use some off-chain decentralized solution for storing the actual proposals there are the questions of availability and integrity of the downloaded proposals. Fortunately, our protocol has a “natural protection” against unavailability and non-integrity of proposals at the phases of proposal voting and proposal activation. Naturally, an unavailable

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

proposal, or one that its hash does not match to the on-chain stored hash, will be rejected by the community (during Ideation, Approval or Activation).

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 fee-supported transactions, which are committed to the blockchain. More details for the proposed voting mechanism can be found in Section 5.2.1. Note that voters are not called to vote only for the SIP per se, but also for the various characteristics of the software update, such as the type of the change, the priority/criticality, etc., which are described in the corresponding metadata. These characteristics will drive the *update policy* adopted.

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 preserve the right to override the delegate's vote, if he/she wishes. More details about our delegation scheme are described in section 5.2.2.

Implementation.

The Implementation phase is an off-chain process, where an approved SIP is implemented. 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*² (*UP*). The newly created UP must be submitted for approval, in order to move forward.

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 to maintain the code-base (since there is not a central authority responsible for the maintenance of the code) and b) the conceptual equivalent to the submission of a pull-request (i.e., a call for approval by the developer to the code maintainer authority) must be realized.

In Figure 5.1, we depict the decentralized implementation phase. 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 code repository provided in the UP metadata. 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 done 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-review task to some group of experts (see the details of the proposed delegation mechanism in Section 5.2.2). Upon the conclusion of the implementation, 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 request to approval. This is accomplished with a fee-supported transaction, which represents the “decentralized equivalent” to the approval if a pull-request.

Approval.

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 guarantee its *correctness*. Overall, the approver approves the correctness (w.r.t. to the requirements described in the corresponding SIP) and safety of the submitted implementation.

²We use the terms *Update Proposal* and *Implementation* interchangeably.

Activation.

The final phase in the lifecycle of a software update 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 then synchronize with their peers before the changes take effect. Thus, the activation phase is clearly a *synchronization phase*. Its primary purpose is for the nodes to activate changes synchronously.

Why do we need such a synchronization 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, or even the installation of new hardware, 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, since different versions of the blockchain will be running concurrently. Of course, this is true *only* for those software updates that impact the consensus protocol (i.e., the validation rules for blocks and transactions and the consensus protocol per se). Indeed, any software update that does not change the consensus protocol, if applied in a non-synchronized manner, it could in-principle impact e.g., some non-functional characteristics (for example CPU related, or memory related), however the nodes will still be in consensus, since they will be running the same consensus protocol. Hence, in our proposal, synchronization is required only for software updates that do impact the protocol.

Finally, there is also one more case of software updates that we need to consider: *protocol parameter updates*. For example, the change of the block size is such an update. These are updates that do impact the consensus protocol but are *dynamic*, in the sense that no installation is required for their activation. In this case, the required synchronization can take place automatically without the need for signaling from the parties. Upon the approval of the change the update protocol will guarantee that the new parameter value will take effect at a specific time point (e.g., the next epoch).

Once the voting period of the Approval phase ends, the votes have been stably stored in the blockchain and the tally result is positive, then the activation period is initiated. In Figure 5.1, we depict the activation period in the decentralized setting. In the figure, we can see an implementation entering the activation phase with an approved *update policy* (e.g., priorities, deployment window etc.) and with the condition that certain *update constraints* (e.g., version dependencies, potential conflicts with other proposals etc.) are fulfilled. The first step in the activation phase is the installation of the software update. It is important to note that the new software is just installed but not activated. The node will continue to run the current version, 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 as a means to synchronization.

One popular method for signaling, that is used by Bitcoin [Nak08], is every new block issued to be stamped with the new version of the software, signifying their readiness for the new update. This approach is simple and straightforward to implement, but it restricts signaling only to *maintainer nodes*³, excluding other type of nodes like *full-node clients*, *light-node clients* etc. Moreover, the major drawback of this approach is that the activation of changes is delayed significantly by the block creation process, which is slow.

Since the adoption threshold is based on the stake that has signaled and not on the number of signaling blocks, another more flexible approach would be to signal by means of simple messages (i.e., fee-based transactions) that are issued from stakeholder keys and thus are bound to specific stake. In this way, we only have to wait for the stabilization of these messages into the blockchain and not for individual blocks to stabilize (a single block can store many such messages).

Furthermore, we could even use a separate consensus protocol, just for the purpose of agreeing on the binary value carried by these messages (Ready — Not Ready) (i.e., a binary Byzantine Agreement problem). The parties taking part in this new protocol would be any node (client, or maintainer) that actively participates in

³By *maintainer* we mean a node that runs the consensus protocol and can issue a new block (also called *minter*).

the underlying consensus protocol and thus needs to synchronize with its peers with respect to the activation of some change. The main assumptions (network, setup, computation as in Garay et al. [GK18]) of this consensus protocol, naturally will be identical to the assumptions of the underlying consensus protocol and therefore the *resiliency*⁴ of the protocol will also be the same.

All in all, by departing from the block signaling solution, we can distinguish between the readiness of different types of nodes and achieve a faster calculation of the adoption threshold. Of course, the downsides in this case are the added complexity and the extra fee that has to be paid for each activation signal.

In the absence of a central authority to set a deadline for the activation of changes, the parties need a way to synchronize, in order to avoid chain splits. Hence we need some sort of a synchronization mechanism. Signaling is indeed the most popular method for synchronization. However, signaling alone is not enough to protect from the risk of a chain split — therefore, a secure activation protocol is required that will enable a safe transition from the previous version to the new version of the consensus protocol (cf. Subsection 5.3.2).

5.2 Update Governance

With the term *update governance* we mean the processes used to control the software updates mechanism. In the centralized setting, the mere existence of a central authority (owner of the code) simplifies decision making significantly. On the other side, in the decentralized approach, we have seen that all decision-making procedures have been replaced by a voting process, where a decision is taken collectively by the whole stake. Thus, we are dealing with a *decentralized governance model*. Therefore voting and delegation are key components of a decentralized approach to software updates. In this section, we describe both these mechanisms.

5.2.1 Voting for Software Updates

Voting for SIPs and UPs

Voting is the main vehicle for driving democracy and in our case is indeed the main mechanism for decision making in a decentralized setting for software updates. A *vote* is a fee-based transaction that is valid for a specific period of time called the *voting period*. In particular, after the object of voting (a SIP or a UP) has been submitted to the blockchain (initially encrypted and then at a second step revealed - based on a commit-reveal scheme), then the voting period for this software update begins.

We acknowledge the fact that not all software updates are equal and therefore, we cannot have a fixed voting period. Therefore, the voting period must be adaptive to the complexity of the specific software update. We propose to have a *metadata-driven* voting period duration, based on the size, or complexity, of the software update.

We introduce a *vote* as a new transaction type, the *Software Update Vote Transaction (SUVT)* that can only be included in a block during the voting period. The core information conveyed by the vote transaction (i.e., included into the transaction Payload field) is summarized in the following tuple:

$$(H(\langle SIP/UP \rangle), SU_{Flag}, \langle confidence \rangle, vks_{source}, \sigma_{sk_{source}}(m))$$

$H(\langle SIP/UP \rangle)$ is the hash of the content of the SIP/UP and plays the role of the unique id of a software update. SU_{Flag} is a boolean flag (SIP/UP), which discriminates an SIP vote from an UP vote. $\langle confidence \rangle$ is the vote per se, expressed as a three-valued flag (for/against/abstain). vks_{source} is the public key of the party casting the vote. Finally, $\sigma_{sk_{source}}(m)$ is the cryptographic signature, signed with the private key corresponding to vks_{source} , on the transaction text m .

Anyone who owns stake has a legitimate right to vote and this vote will count proportionally to the owned stake. Furthermore, as we will describe in detail in the next section, the right to vote can also be delegated

⁴According to Garay et. al. in [GK18], the resiliency is the fraction (t/n) of misbehaving parties a protocol can tolerate, where t are the number of adversaries and n are the total number of parties. In our case, we can assume that t is the total adversary stake and n is the total stake.

to another party. However, we want to give the power to a stakeholder to override the vote of his/her delegate. Therefore, if within the same voting period appear both a private vote and a delegate's vote, for the same software update, then the private vote will prevail.

Moreover, we cannot exclude the possibility that a voter/evaluator changes his/her mind after voting for a specific SU (for example, the evaluator identifies a software bug after voting positively for a software update). We want to provide the flexibility to the evaluators to change their minds. Therefore, we allow for a voter to vote multiple times, within a voting period for a specific software update. At the end, we count only the last vote of a specific public key in the voting period for a specific software update.

Voting Results

After the end of the voting period for a specific software update and after we allow some stabilization period (in order to ensure that all votes have been committed into the blockchain), the votes are tallied and an outcome is decided. First, the non-delegated votes are counted. The tallying is performed as follows: For each slot within the voting period, if a block was issued pertaining to that slot, each SUV_T in that block is examined. If the staking key for that transaction has been tallied on a non-delegated vote previously, the previous vote is discarded and the new vote is counted. This allows voters to modify their votes until the end of the voting period. For every SUV_T which has been counted, the stake that votes for it is summed and this constitutes the *non-delegated stake in favour*, the *non-delegated stake against* and the *non-delegated abstaining stake*.

Subsequently, the delegated votes are counted. If the delegatee staking key for that transaction has been tallied on a delegatable vote previously, the previous vote is discarded and the new vote is counted. For each of the delegatable votes, the keys delegating to it are found. Each of the keys delegating is checked to ensure that the delegating key has not cast a non-delegated vote; if the delegating key has also cast a non-delegated vote, then the non-delegated vote is counted instead of the delegated vote. For each delegatable SUV_T which has been counted, the stake delegating to it which has not issued a non-delegated vote is summed and this constitutes the *delegated stake in favour*, the *delegated stake against*, and the *delegated abstaining stake*.

The sum of the non-delegated stake in favour and delegated stake in favour forms the *stake in favour*; similarly, we obtain the *stake against* as well as the *abstaining stake*.

Suppose that we define a voting threshold τ_V , then at the end of the tallying, a software update (i.e., a SIP or an UP) is marked one of the following:

- *Approved*. When the *stake in favour* $> \tau_V$
- *Rejected*. When the *stake against* $> \tau_V$
- *No-Quorum*. When the *abstaining stake* $> \tau_V$. In this case, we revote (i.e., enter one more voting period) for the specific software update. This revoting can take place up to $rv_{no-quorum}$ times, which is a protocol parameter. After that, the software update becomes *expired*.
- *No-Majority*. In this case none of the previous cases has appeared. Essentially, there is no majority result. Similarly, we revote (i.e., enter one more voting period) for the specific software update. This revoting can take place up to $rv_{no-majority}$ times, which is a protocol parameter. After that, the software update becomes *expired*.
- *Expired*. This is the state of a software update that has gone through $rv_{no-quorum}$ (or $rv_{no-majority}$) consecutive voting periods, but still it has failed to get approved or rejected.

In our proposal, we have chosen a three-value logic for our vote (for/against/abstain). In this way, apart from the actual result, we can extract the real sentiment (positive, negative, neutral) of the community for a specific software update. This is very important in a decentralized governance model, because it clearly shows the appeal of a software update proposal to the stakeholders. If we did not allow negative votes, then the negative feeling would be hidden under the abstaining stake. Moreover, the abstain vote can be also used as a way for the

evaluator to say that the evaluation of the SU has not finished, a conclusion can not be drawn yet and indirectly submit a request for a time extension (i.e., a new voting period).

5.2.2 Delegation to Experts ⁵

Each stakeholder has the right to participate in the software updates protocol of a proof-of-stake blockchain system. In this section, we discuss the delegation of the protocol participation right to some other party. As we will see next, this delegation serves various purposes and copes with several practical challenges.

Delegation for Technical Expertise

One of the first practical challenges that one faces, when dealing with the decentralized governance of software updates is the requirement of technical expertise, in order to assess a specific software update proposal. Indeed, even at the SIP level, many of the software update proposals are too technical for the majority of stake to understand. Moreover, during the UP approval phase, the approver is called for approving, or rejecting, the submitted source code, which is certainly a task only for experts.

Our proposal for a solution to this problem is to enable delegation for technical expertise. Stakeholders will be able to delegate their right to participate in the update protocol to an *expert pool*. The proposed delegation to an expert pool comprises the following distinct responsibilities:

- The voting for a specific SIP
- The voting for a special category of SIPs
- The voting for any SIP
- The approval of a specific UP
- The approval of a special category of UPs
- The approval of any UP

As you can see, we distinguish delegation for voting for a SIP document and that for approving an UP. We could have defined delegation for SIP voting to imply also the approval of the corresponding UP. However, since both have a totally different scope, there might be a need to delegate to different expert pools for these two. Indeed, a SIP is an update proposal justification document and the expert who is called to vote for, or against, a specific SIP, must have a good sense of the road-map of the system. On the contrary, the approval of a UP is a very technical task, which deals with the review and testing of a piece of code against some declared requirements (i.e., the corresponding SIP) and has nothing to do with the software road-map.

Delegation for Specialization

It is known that there exist special categories of software updates. Let us consider for example, security fixes. It is common sense, that security fixes are software updates that: a) have a high priority and b) require significant technical expertise to be evaluated. Therefore, by having a special expert pool as a *default delegate* for this category of software updates (both SIPs and UPs) enables: a) a faster path to activation and b) sufficient expertise for the evaluation of such SUs. The former is due to the omission of the delegation step in the process and that the evaluation (i.e., voting of SIPs/UPs) will take place generally in shorter times; exactly because this is a specialized and experienced expert pool that deals only with security fixes; we assume that they can do it faster than anybody else.

⁵Note that for the scope of our prototype implementation we have assumed delegation as an out-of-band solution. This is basically for two reasons: a) to reduce risks in the prototype implementation (especially of the Cardano integration part) and b) to defer introducing delegation to experts until a proper game-theoretic analysis of the expert's incentives has been completed.

We do not propose any specific set of categories in this deliverable. However, we do propose that: a) software updates are tagged with a specific category and b) to use delegation for enabling specialized treatment on special categories.

So, for software updates with a special tag, our proposal is, to have *default* specialized expert pools that will participate in the software updates protocol on behalf of the delegated stake. Of course, this default delegation based on SU tagging can be overridden. Any stakeholder can submit a different delegation for a specific SIP/UP regardless of its tag.

Default Delegation for Availability

Blockchain protocols based on the Proof-of-Stake (PoS) paradigm are by nature dependent on the active participation of the digital assets’ owners –i.e., stakeholders– (Karakostas et. al. [KKL18]). Practically, we cannot expect stakeholders to continuously participate actively in the software updates protocol. Some users might lack the expertise to do so, or might not have enough stake (or technical expertise) to keep their node up-and-running and connected to the network forever.

One option to overcome this problem, which is typical in PoS protocols, is to enable stake representation, thus allowing users to delegate their participation rights to other participants and, in the process, to form *stake pools*([KKL18]). The core idea is that stake pool operators are always online and perform the required actions on behalf of regular users, while the users retain the ownership of their assets ([KKL18]).

In this deliverable, we propose to utilize the stake pools mechanism for our software updates protocol in tandem with the consensus protocol. In particular, we propose to allow each stakeholder to define a default delegate for participating in the software updates protocol from the list of available stake pools that participate in the core consensus protocol. This will be a “baseline” representative of each stakeholder to the software updates protocol, just for the sake of maintaining the participation to the protocol at a sufficient level and minimizing the risks of non-participation. This delegate will coincide with the delegate for the participation in the consensus protocol. A stakeholder will be able at any time to override this default delegation. A delegation to an expert pool for a specific software update, or a specific category of software updates, due to specialization, described in the previous section, will override the default delegation to a stake pool.

Delegation Mechanics

For the realization of the stake pool delegation mechanism that we described above, we closely follow the work of Karakostas et al. [KKL18], so we refer the interested reader to this work for all the relevant details. In this subsection, we would like to focus on the most basic mechanics (i.e., technical details) that will enable such a delegation mechanism to work. Please note that many of our ideas are based on the design of the delegation mechanism for the Cardano blockchain system [KBC19].

Staking keys. Following the Karakostas et. al. [KKL18] approach, we separate for each address the control over the movement of funds (i.e., executing common transactions, such as payments) and that over the right for participation in the proof-of-stake protocol and consequently, in the software updates protocol, due to the ownership of stake. Intuitively, this separation of control is necessary, since we only want to delegate the management of stake to some other party, by means of participation in the software updates protocol and not the management of the funds owned by this stake. This is achieved in practice by assuming that each address consists of two pairs of keys: a) a *payment key pair* $K^p = (skp, vkp)$ and b) a *staking key pair* $K^s = (sk_s, vks)$. With the former a stakeholder can receive and send payments, while with the latter a stakeholder can participate in the proof-of-stake consensus protocol and in the software updates protocol. skp and sk_s are the secret keys for signing, while vkp and vks are the public keys used to verify signatures.

Stake Delegation

In its simplest form, delegation of stake from some party A to another party B (typically a stake pool) for participation in the proof-of-stake consensus protocol, also delegates the right for participation in the software updates protocol as well. The rationale of this has been described in Subsection 5.2.2 and it holds only on the assumption that there is no explicit delegation to some expert pool. So in the rest of this text, when we refer to stake delegation, we mean for the participation in both the proof-of-stake consensus protocol and the software updates protocol, unless an explicit statement is made for delegation to an experts pool.

At its core, the delegation of stake to some other party, essentially requires two things: a) stake registration and b) issuance of a delegation certificate:

Stake key registration. This step is a public declaration of a party that wishes to exercise its right for participation in the proof-of-stake protocol, due to its ownership of stake. In order for a stakeholder to exercise these rights, he/she must first issue a stake key registration certificate. This is a signed message stored in the metadata (i.e., Payload) of a transaction and thus it is published to the blockchain. The key registration certificate must contain the public staking key vk_s , and the signature of the text of the transaction m by the staking private key sk_s , which is the rightful owner of the stake. In other words, the key registration certificate r is the pair: $r = (vk_s, \sigma_{sk_s}(m))$. The signature σ of the certificate authorizes the registration and plays the role of a witness. Symmetrically, there is also a de-registration certificate for a stake key, which is a declaration that a party no longer wishes to participate in the proof-of-stake protocol.

Delegation registration. In order to register the delegation of stake from one party (source) to another (target), a delegation certificate must be issued and posted to the blockchain by the source party. This certificate publicly announces to the network that the source party wishes to delegate its stake right (for participation in the proof-of-stake protocol) to the target party and this is recorded forever in the immutable history of the blockchain. At a minimum, a delegation certificate consists of the following information:

$$(H(vk_{s_{source}}), H(vk_{s_{target}}), \sigma_{sk_{s_{source}}}(m))$$

Where, $H(vk_{s_{source}})$ is the hash of the source party’s public staking key, $H(vk_{s_{target}})$ is the hash of the target party’s public staking key and $\sigma_{sk_{s_{source}}}(m)$ is the signature of the text m of the transaction (within which the delegation certificate is embedded) by the source party’s private staking key $sk_{s_{source}}$, which authorizes the certificate and plays the role of a witness for a formal definition of the signature scheme).

If at some point, the source party wishes to re-delegate to some other party, or even to participate in the protocol on its own, then it must simply issue a new delegation certificate. For self-participation in the protocol, a party must issue a delegation certificate to its own *private stake pool*⁶. If the source staking key is de-registered, then the delegation certificate is revoked.

Delegation to an Expert Pool

We have seen that by default, the participation right in the software updates protocol is delegated to the stake pool that the delegation for participation in the proof-of-stake consensus protocol has taken place. So by default, some stake pool will participate in the software updates protocol. Next, we will discuss the case where a stakeholder wants to override the default behavior and explicitly delegate to an expert pool.

An expert pool is an entity consisting of one or more experts, who are willing to participate in the software updates protocol as delegates of other stakeholders. Their main task is to vote for (or against) SIPs and to approve (or reject) UPs. We call them “experts”, because they need to have sufficient technical expertise, in order to evaluate a software update.

⁶A *private stake pool* is a trivial case of a stake pool. By treating self-staking as a special case of stake pool delegation is a design decision for the sake of simplicity [KBC19].

In order to enable delegation to an expert pool, we extend the delegation certificate presented above, with additional information. In particular, a delegation certificate to an expert pool is defined as the following tuple:

$$\begin{aligned} & (H(vk_{s_{source}}), \\ & \quad H(vk_{s_{target}}), \\ & \quad \sigma_{sk_{s_{source}}}(m), \\ & \quad \quad SU_{Flag}, \\ & \quad H(< SIP/UP >), \\ & \quad < category >) \end{aligned}$$

In this case, the $H(vk_{s_{target}})$ is the hash of the public staking key of the expert pool. We have extended the delegation certificate to include a boolean flag SU_{Flag} , which denotes, if the delegation pertains to a SIP, or an UP. We have explained previously (see subsection 5.2.2), the rationale for distinguishing the delegation for these two. Finally, the hash $H(< SIP/UP >)$ is the hash of the content of the SIP, or UP, in question and plays the role of the unique id for this SIP, or UP respectively. Note that if instead of a specific SIP/UP id, a special value is provided for this field (e.g., '*'), then this corresponds to a delegation for *any* SU of this type (SIP or UP). Finally, if the SU id field is empty (or $NULL$, it depends on the implementation), then we take into account the last field, which specifies the *category* of the SU (e.g., “security-fix”, “linux-update”, etc.) that, we wish to delegate for. This will be a simple string value chosen from a fixed set of values (a list of acknowledged SU categories).

In summary, with this certificate, a party can delegate its participation right in the software updates protocol to an expert pool: a) for a specific software update (SIP or UP), b) for a specific category of software updates, or c) for any software update. Of course, in order for this delegation registration to be valid, the target expert pool must have been appropriately registered first in the blockchain. This is the topic to be discussed next.

Expert Pool Registration

In order for someone to publicly announce his/her intention to play the role of an expert, or equivalently, to run an expert pool, two things are required: a) to issue an expert pool registration certificate and b) to provide appropriate *metadata* describing the expert pool.

Expert pool registration certificate. The certificate contains all the information that is relevant for the execution of the protocol. At its most basic form, this certificate comprises the following:

- $vk_{s_{expool}}$: This is the public staking key of the expert pool. This must be used as the target public key in the delegation certificate, as discussed in the previous subsection.
- $(< URL >, H(< metadata >))$: A URL pointing to the metadata describing the expert pool and a content hash of these metadata. The URL points to some storage server and the hash of the content retrieved must match the one stored in the certificate for the pool registration to be considered as valid.
- $\sigma_{sk_{s_{expool}}}(m)$: The certificate must be authorized by the signature σ of the expert pool $sk_{s_{expool}}$ on the text m of the transaction that includes the certificate.

Symmetrically, there should be also an *expert pool retirement certificate* for allowing an expert pool to cease to operate. This should include the public staking key of the expert pool, as well as a time indication (e.g., expressed in block number, or an epoch number etc.) of when the pool will cease to operate.

Expert pool metadata. The expert pool metadata are necessary information that describe sufficiently an expert pool, so as the stakeholders community can decide, which expert pool to choose for their delegations. Typically, this information will be displayed by the wallet application, in order to assist the users to select the expert pool of their choice. Examples of useful information to be included in the expert pool metadata are the name of the pool, a short description, the area of expertise, the years of expertise, preferences to specific SU categories, URLs to sites that exhibit the claimed experience and in general any information that can help the stakeholders to choose the appropriate delegate for the right software update.

Miscellaneous Considerations on Delegation

Chain delegation. Chain delegation is the notion of having multiple certificates chained together, so that the source key of one certificate is the delegate key of the previous one. In principle there is no reason to prevent the formation of delegation chains. However, an implementation of this proposal must take into account the possibility to form (deliberately or by accident) delegation cycles. This means that a target delegate ends up to be one of the sources. In this case, the delegation is essentially canceled and the system should detect it and prevent it pro-actively.

Certificate replay attacks. For all our certificates, namely: stake key registration, delegation registration and expert pool registration, we have provided signatures of the text of the encompassing transaction (the certificates are included as transaction metadata), signed by the party(ies) authorized to issue the certificate. This is a design choice made in [KBC19] that prevents against a certificate replay attack. In this attack, an attacker republishes an old certificate, in order for example to change a delegation to a new expert pool. In particular, since the certificate includes a signature on a specific transaction text, then this certificate is bound forever with the specific transaction, and just like in blockchains with a UTXO accounting model, a transaction cannot be replayed (a UTXO can be only spent once), similarly the specific certificate cannot be replayed either. For account based blockchains there are other approaches that one can follow, in order to prevent a replay attack, such as the *address whitelist* proposed in Karakostas et. al. [KKL18], where the transaction that includes the certificate must be issued from a specific whitelisted address. Of course, there are other common solutions like the counter-based mechanism (known as the *nonce*) used in Ethereum [B⁺14].

Identity theft. Significant expertise on difficult technical issues is not a skill that is easy to acquire. Moreover, experience comes after a long period (probably many years) of struggle with technical issues. Therefore, real specialists on a technical domain are hard to find and for this reason they are invaluable. Typically, these experts are well-known and well-respected figures in the community. Therefore, such a well-known expert is expected to receive a significant amount of delegations, if he/she chooses to register an expert pool. This fact makes expert pool registration susceptible to identity theft. This is the case where an expert pool falsely claims to be the famous “expert A”, just for the sake of receiving the delegations drawn from the fame of the expert. This identity assurance problem is external to the software updates protocol and also to the underlying consensus protocol and thus some out-of-band solution could be adopted. For example, a famous expert, could post his/her public key (or its fingerprint) to social media, so that the people who follow him/her, will know, which is the genuine key that they can delegate to. Of course, other similar in concept, solutions can be exploited as well. However, at the end of the day, in a decentralized setting, it is the stake via delegation that will be the ultimate judge of an expert pool.

5.3 The Activation of Changes

5.3.1 Overview

In section 5.1, we have described the lifecycle of a software update and there we have seen that once an implementation (UP) is approved, it enters the *Activation* phase. This phase is depicted in figure 5.2.

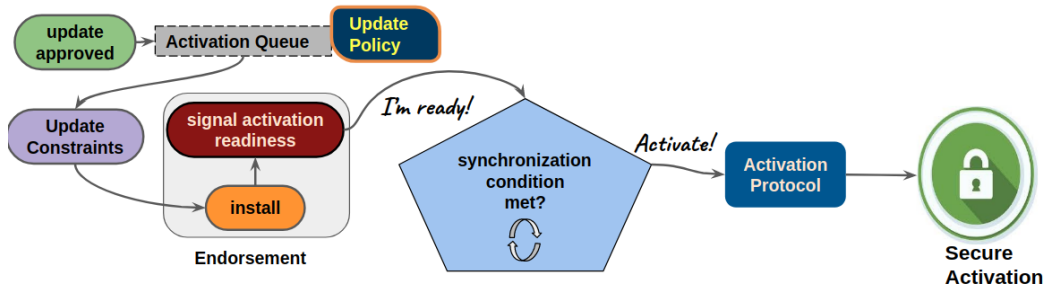


Figure 5.2: The Activation Phase.

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

- is approved,
- meets its dependencies,
- does not conflict with the current version,
- has the highest priority among competing proposals

If a proposal satisfies its update constraints it enters the *Endorsement Period*. This is the synchronization period discussed in section 5.1, where the *block issuers* download and install the update and *signal upgrade readiness*. We call this signal an *endorsement*. Note that only a *single* proposal can be endorsed at a time. The endorsement period lasts N number of *epochs*, which is a metadata-defined parameter, called the *safety lag*; the safety lag corresponds to the sufficient deployment time window required for the specific update proposal. Once the endorsements reach a specific stake threshold, called the *adoption threshold*, i.e., when the synchronization condition is met, then the activation gives the green light to the *activation protocol* to run. The activation protocol ensures the secure activation, i.e., the secure transition from the old ledger to the upgraded ledger, based on our formal definition of activation security and corresponding security proofs [CKKZ20].

Finally, note that all the metadata-defined information about an update proposal, such as the deployment window length, the proposal’s priority, the version dependencies etc., form the proposal’s *update policy* to be followed by the update system. This policy is accepted and confirmed by the community through the previous voting process during the Approval phase (see section 5.1). In section 5.3.3, we provide details on the design of the activation phase.

5.3.2 Secure Activation Protocols

A secure activation protocol is one that achieves a secure transition from the current version of the consensus protocol to a new one. We have formally defined what is a *secure activation* and have proposed two distinct protocols that provably achieve this. In a nutshell, *secure activation* means, the secure transition from the old ledger (L1) to the new ledger (L2) in a way where:

- L2 enjoys liveness [GKL15]
- L2 enjoys consistency [GKL15]
- L2 has L1 as a prefix

Our first activation protocol requires the structure of the current and the updated blockchain to be very similar (only the structure of the blocks can be different) but it allows for an update process more simple and efficient.

The second activation protocol that we propose is very generic (i.e., makes few assumptions on the similarities between the structure of the current blockchain and the updated blockchain). The drawback of this protocol is that it requires the new blockchain to be resilient against a specific adversarial behavior and requires all the honest parties to be online during the update process. However, we show how to get rid of the latest requirement (the honest parties being online during the update) in the case of proof-of-work and proof-of-stake ledgers. The interested reader can find more details on this topic in our paper [CKKZ20] and in Deliverable D3.3

5.3.3 Activation Phase Design

In this section we will provide details on the design of the activation phase over the Cardano blockchain system. This phase follows the approval phase where the submitted implementations are reviewed by the expert pools (who are delegated by the community). Through a voting process the experts decide whether an implementation of a software update will proceed to the activation phase or whether it will be rejected.

Such an update proposal (UP) can be of four types:

Parameters: which are updates that change the system parameters.

Consensus: which are updates that change the consensus rules. For instance changes in the transaction or block validation rules, or changes in the fork resolution rules. As an extreme example, even the underlying consensus protocol could be changed (for instance, going from Ouroboros classic to Ouroboros BFT).

Application: which are updates of applications in the Cardano ecosystem: nodes, wallets, explorers, etc. Application updates do not change the blockchain protocol.

Cancellation: which are special kind of update proposals that can be used to cancel other *approved* proposals⁷.

Application updates do not require synchronization among the nodes of the blockchain. Since this update does not affect consensus or might not even be related to a software update for a node, the community can upgrade to a new version of a given application as soon as it is (stably) approved. Note that if the upgrade process requires for the node to be off-line for some period of time, then this should not affect the security assumptions of the protocol, if this process does not take place at the same time for all nodes.

A protocol-parameters update comes into effect *at the beginning of a new epoch*. The nodes do not need to install new software, since the current running version can update the parameters as prescribed in an approved update proposal.

A consensus-update requires that the parties *synchronize* prior to its activation to avoid a chain split due to lack of consensus. In addition, before the activation, we need to make sure that certain security assumptions hold, like the fact that enough honest stake has upgraded. To this end, the activation phase synchronizes participants to make sure that at least a certain portion of the honest stake will activate the update at the same time, which is a necessary condition to ensure that the upgraded ledger will be secure. In this context, activation of an update means that the protocol version that it specifies becomes the current version of the blockchain.

The stake that can endorse is the stake delegated to stake pools, which is the stake that is considered for selecting block producers.

Note that a consensus-update can also specify parameters update. In this case the new software needs to be installed and run on the nodes, and the new software must apply, at the beginning of an epoch, the parameters updates that the update proposal dictates.

Additionally, consensus update must specify where the code and binaries that run the new protocol can be obtained.

All parameters and consensus proposals must specify:

- The version that they supersede. The proposal can only be applied to this version.

⁷For non-approved proposals the expert pools can simply reject said update

- The (hash of the) proposal they supersede. Update proposals in the approval phase can have the same version, so proposals need to disambiguate the proposal they supersede. This allows us to avoid a situation in which a proposal depends on a given version, but also (implicitly) on a particular implementation of that version. If we do not make this dependency implicit, then we risk having an update being applied on the wrong version. We cannot on the other hand, enforce uniqueness of versions in update proposals, since we risk having malicious actors organizing a denial of service attack that prevents honest proposals from entering the system. Note that having a proposal specify the proposal it supersedes require that we start with a Genesis (implementation) proposal.

When a parameters or consensus proposal is approved⁸ it enters a *priority queue* where it waits to be activated. The priority of the queue is determined by the update proposal's versions. A proposal with lower version will activate *before* a proposal with a higher version. We call the proposals in the activation queue the *approved proposals*.

Proposals *must increase* the current *protocol-version* that the chain supports. The protocol version consists of two components:

1. The major version, which is used to specify hard-forks.
2. The minor version, which is used to specify soft-forks.

The update system guarantees that the protocol versions can only strictly increase. In this way, the update system can rely on the update proposal's protocol-version and its declared predecessor version to:

- Determine priorities: a lower version (higher priority) is activated before a higher version (lower priority) one; specifying the version gives us a simple way to specify urgency.
- Resolve conflicts: a proposal declares *one and only one* version that it supersedes (its predecessor). Thus a proposal is in conflict with all others except the one that it supersedes.
- Check dependencies: a proposal depends only on the one it supersedes.

This design decision requires:

- The implementers to pick exactly one version their update is compatible with, and determine which version their update should receive.
- The experts to check the version that an update has, and the compatibility with the version it supersedes.

Protocol-versions of update proposals do not need to be unique.

If two proposals with the same version are in the approval phase and both are approved, then the last one entering the activation phase, will survive. In the unlikely case, where the two proposals coincide in their voting period ends and thus are approved at the very same slot, then we resolve this conflict by choosing the one with the higher stake in favor to enter the activation queue. In the even more unlikely case, where both have the same stake in favor, then we choose the proposal with the greatest id (proposal hash) to enter the activation queue.

The approved proposal at the front of the activation queue will enter the *endorsement period*, if it supersedes the current version. There can be *at most one proposal* in the endorsement period at any time. This is to avoid the stake of the stakeholders being split among competing proposals during endorsement, and thus, when changes take effect. We call a proposal that is in the endorsement period the *candidate proposal*.

If the new candidate proposal is a consensus-update, the block producers can start *endorsing* it. An endorsement is part of the transaction body. An endorsement signals the fact that a node has downloaded and installed the software that implements the update. If the proposal gathers enough endorsements, then it can be *scheduled-for-activation*. If the candidate proposal does not need endorsements, i.e. it is a protocol parameters update, it becomes immediately scheduled.

⁸Cancellation and application update proposals come into effect immediately after being approved.

A scheduled proposal waits to be activated till the first slot of the next epoch. In case of application updates, the proposal does not need to be activated. The ledger simply records its approval.

The next sub-sections describe the details of the activation protocol outlined above.

Entering the endorsement period.

An update proposal can enter the endorsement period if two conditions are met:

1. It is at the front of the priority queue (i.e. it is has the lowest version among the proposals queue).
2. The version and proposal hash that the proposal declares to supersede must be *the same* as the current version.

Once in the endorsement period, *it might return to the queue* if a new update proposal is approved and ends up in front of the priority queue (which means that the newly approved proposal has a lower version than the current candidate).

Endorsements of consensus updates.

Only a consensus-update needs endorsements, since it requires nodes to download and install new software, and signal that they are ready to run the new version. A consensus-update proposal has a *safety lag* associated with it, which is a time window that ensures sufficient time is provided to the parties to download and deploy the candidate proposal. This safety lag determines the duration of the endorsement period and *must be specified in number of epochs*. As a result, the end of the safety lag coincides with the end of an epoch.

When a consensus-update proposal becomes a candidate, block producers can start endorsing it. The stake associated with the keys endorsing the proposal is tallied $2k$ blocks before the end of an epoch, where k is maximum *number of blocks* the chain can roll back. We consider the stake *at the slot in which we tally*. We have two activation thresholds depending on the epoch in which the tally takes place:

- If the next epoch does not coincides with the end of the safety lag, then this threshold is the *adoption threshold* (τ_A).
- If the next epoch coincides with the end of the safety lag, then this threshold is set to 51%.

Once we sum up the stake endorsing a proposal there are three possibilities:

1. If the proposal has not gathered enough endorsing stake then:
 - (a) If the safety lag expires on the next epoch then the proposal is canceled.
 - (b) If the safety lag does not expire on the next epoch then the proposal can continue to be endorsed on the next epoch. The endorsements are carried over.
2. If the proposal has gathered enough endorsing stake, then it is *scheduled for activation* at the beginning of the next epoch.

Consensus and parameters updates get activated *at the beginning of an epoch*. The reason for this is that the ledger and consensus rules rely on these to be stable during any given epoch.

The cancellation⁹ or activation of a proposal causes its removal from the activation queue, and the next implemented proposal in the queue, if any, to become the new candidate and enter the endorsement period.

A proposal being endorsed can go back to the queue. In this case the endorsements for this proposal are discarded.

⁹Note that we classify expired proposals, i.e. proposals that did not get enough endorsements at the end of the safety lag, as canceled proposals.

Parameters update.

Parameters-update proposals determine new values for the (existing) protocol parameters. The current version of the software can deal with this sort of updates, and therefore there is no need for the nodes to download and install software, and thus no need for an endorsement period. However, they can only be activated at beginning of an epoch. Parameters-update must also wait in the activation queue.

A protocol parameters update must have a corresponding approved SIP, which justifies the need for changing the parameter values. The submitted implementation (UP) specifies the new values and the new protocol version associated to these parameters. Both SIP and UP describe the parameters change. The SIP does this in an abstract way, e.g. “change maximum block size to 1MB”, whereas the UP describes this change in a concrete way, e.g. `maxBlocksize = 1024`.

Note how, the activation of one parameter update does not automatically eliminate all the other pending parameters updates. We rely on the versioning scheme for resolving conflicts. For instance, assume the current protocol version is `1.0.0`. If updates with versions `1.1.0` and `1.2.0` both change a parameter `p`, activating `1.1.0` does not necessarily causes `1.2.0` to be discarded. It all depends on the version the `1.2.0` supersedes. If it supersedes `1.0.0`, then it will be canceled since the current version is now `1.1.0` and versions can only increase. However `1.2.0` could declare that it supersedes version `1.1.0`, in which case it can be activated after this version (for instance in the next epoch).

Application updates

Application updates are not related to protocol updates, and therefore they do not enter the activation phase. Upon approval of an application update the ledger simply records this event. The dependencies between applications and protocol versions has to be managed off chain, and it is the responsibility of developers (who implement the updates) and experts. The update system can provide metadata for specifying the dependencies and/or conflicts of an application with the rest of the ecosystem, but the update system will not check this.

Just like the other kind of updates, application updates must have a corresponding approved SIP and implementation.

Explicit cancellation

A proposal can be explicitly canceled. This allows the experts and community to cancel an approved implementation after a (significant) problem with it was discovered.

An explicit cancellation is submitted as a SIP that must justify the reason for canceling the proposals. When submitted as an update proposal (UP), the cancellation must specify the proposals that it will cancel if approved.

The explicit cancellation also goes through the ideation and approval phases, however, the explicit cancellation *immediately cancels* the proposals it refers to once it gets approved by the expert pools in the approval phase.

Cancellation proposals have no next-versions associated with it. They only specify the hash of the implementations they cancel.

Cancellations

There are several factors that can cause a proposal to be canceled:

- The proposal is explicitly canceled by a cancellation proposal that made it past the approval phase.
- The proposal is overridden by another proposal with the same version that made it through the approval phase. If the experts approve such proposal, then this gives a strong indication that the older proposal should be discarded. Note that even though the system guarantees that only one proposal per-version will get approved in any given slot, there might be proposals approved with the same version as a proposal approved in a previous slot.

- The proposal supersedes a version that can never follow. Protocol versions increase monotonically. So if an update depends on a version that is lower than the current version we know for sure that such version can never be seen on the chain, and therefore the proposal that depends on this version can be discarded.

If the candidate proposal already got the required endorsements (which means that the cancellation arrived at or later than the slot at which the tally occurred, this is $2k$ blocks before the end of an epoch), and is waiting to be activated, then the cancellation update cannot stop it. In this case, a new software update must be implemented and submitted that will correct the identified problem.

If a candidate proposal is a parameters update, which means that does not need endorsements, but the cancellation arrives *after* $2k$ blocks before the end of the epoch, then it cannot be canceled either for the reasons given in the below subsection "On cut-off points for endorsements".

Nodes that upgraded to a canceled version can continue to operate normally following the current version since the upgraded version *must* be able to follow the current ledger and consensus rules. Then it is up to the node operators to revert back to the previous version, or continue using the software version that can also validate the protocol version that got discarded. The ledger rules *shall ensure* that this discarded protocol version is never applied.

The queue.

Approved proposals enter an activation priority queue. The order of the queue is determined by the proposal's versions. This queue *shall not* contain any duplicated versions. If a proposal with the same version as a proposal from the queue gets approved, then:

1. If the old proposal (the one in the queue¹⁰) does not have enough endorsements by the time the new proposal is approved, then the old proposal is canceled.
2. If the old proposal has enough endorsements, then the new proposal is canceled.

If a proposal with a lower version than the current candidate enters the queue (i.e. the front of the queue changes), then we also have two situations:

1. If the candidate proposal does not have enough endorsements, then it is put back in the queue, and the new proposal becomes the new candidate. If this new candidate gets approved then the old proposal will be canceled (since it will supersede a version lower than the current version). However, since it is possible that the new proposal gets canceled (for instance due to lack of enough endorsements), we leave the old proposal in the queue.
2. If the candidate proposal has enough endorsements, then the new proposal gets discarded (since it will be superseding an older version that will never be adopted since versions increase monotonically).

On cut-off points for endorsements

The ledger layer must provide future information about a part of its state. In particular, it should be able to tell up to k blocks¹¹ in the future if a new protocol version will be activated. To this end, we require that the *last endorsement* required for meeting the adoption threshold is *stable* k blocks before the end of the epoch. This means that endorsements for a given proposal for a given epoch are considered up to $2k$ blocks before the end of that epoch. This is not to say that the endorsements after this cut-off point will not be considered, just that they will in the next epoch.

To see why this is required, consider a last endorsement (required to meet the threshold) arriving between $2k$ blocks and k blocks before the end of an epoch, which will happen at block number b_e . If the ledger is asked

¹⁰Note that only the proposal in front of the queue can receive endorsements.

¹¹where k is maximum *number of blocks* the chain can roll back

between blocks $b_e - k$ and b_e whether a new version will be activated at b_e , the answer depends on the stability of the last endorsement. If it is not stable, then replying “yes” might be incorrect, since we can roll back and switch to a fork where that endorsement never occurred. If we reply “no”, then this might also be incorrect if we continue in that fork.

5.4 Threshold Analysis

In the previous sections, we have described two separate voting phases in the decentralized software update life-cycle (figure 5.1), namely Ideation and Approval, where a proposal can be approved or rejected by the community. Moreover, during the Activation phase the community, or more accurately, the participants in the blockchain consensus protocol, signal *upgrade readiness* by endorsing a new version of the protocol. Both of these processes are based on a *threshold* of stake percent, against which, the gathered votes/signals are compared. Although the voting of a proposal compared to the activation of a proposal have different end-goals, there are still a lot of commons from a social dynamics perspective. In this section, we present an analysis towards a unified approach for defining an appropriate threshold for both of these two processes.

Lets assume that H is the percent of honest stake and T is the percent of adversarial stake that actively participate in the voting (or activation) process. We assume also that

$$H + T = 100 \quad (5.1)$$

We consider two types of possible attacks relevant to both processes:

Voting process

- **Malicious Proposal Attack:** The adversary manages with its own stake to approve a malicious proposal.
- **Denial of Approval Attack:** The adversary manages to block a good proposal approved by the honest stake.

Activation process

- **Rushing Adversary Attack:** The adversary rushes, by using its own stake, to signal upgrade readiness and cause the activation of the changes, in order to take over the control of the updated blockchain, before enough honest parties manage to upgrade.
- **Denial of Activation Attack:** The adversary manages to block the activation of a change signaled by the honest stake.

With respect to the two said attacks for each process, we define next, the notion of a secure voting (or activation) protocol:

Definition 5.4.1. A voting protocol is secure, if it enjoys both the safety and liveness properties.

- **Safety property for voting:** a software update that has achieved only the adversary stake approval, will never be approved.
- **Liveness property for voting:** a software update that has achieved sufficient honest stake L_b approval, will be eventually approved, i.e., cannot be blocked by the adversary stake.

Definition 5.4.2. An activation protocol is secure, if it enjoys both the safety and liveness properties.

- **Safety property for activation:** a software update that has not received signals of sufficient stake w.r.t. the security assumption of the upgraded protocol will never be activated.

- **Liveness property for activation:** a software update that has received signals of sufficient honest stake L_b , will be eventually activated, i.e., cannot be blocked by the adversary stake.

Clearly, the safety property protects us from the malicious approval/rushing adversary attack, while the liveness property protects us from the denial of approval/activation attack.

Our goal, is to define an appropriate voting/adoption threshold τ , such as the two said properties hold.

Definition 5.4.3. A voting threshold τ_V is an amount of stake such that any stake $p > \tau_V$ of positive votes can approve a proposal and any stake $n > \tau_V$ of negative votes can reject a proposal.

Similarly,

Definition 5.4.4. An adoption threshold τ_A is an amount of stake such that any stake $p > \tau_A$ of signals can activate a proposal.

Next we define the sufficient and necessary condition for the voting threshold and the adoption threshold in order for safety and liveness to hold

Theorem 5.4.1. For each voting protocol with a threshold τ_V both the safety and liveness properties hold iff:

$$T < \tau_V < H$$

Proof. Since $T < \tau$, any malicious proposal that is voted positively by the adversarial stake T , but by no honest stake at all, cannot be approved, because the condition for approval is $[positive\ stake] > \tau$. Therefore, the safety property holds. Inversely, if the safety property holds, then no proposal that has no positive votes from the honest stake can be approved even if the adversary stake T has voted in favor, which means that the positive votes must be below the voting threshold; thus $T < \tau$.

Similarly, since $\tau < H$, any proposal that has received L_b , or more, positive votes from the honest stake, where $\tau_V < L_b \leq H$ will be approved, because the condition for approval is $[positive\ stake] > \tau$. Inversely, if the liveness condition holds, then any proposal that has received at least L_b positive votes from the honest stake will be approved, which means that the following condition should hold $\tau < L_b$. \square

The adoption threshold τ_A should ensure that the required percent of stake that has signaled upgrade readiness, will guarantee that the new blockchain will kick off with sufficient honest stake, determined by the security assumption of the upgraded consensus protocol. It is easy to see that in this case, we need $\tau_A \geq \frac{1}{r_{Th}} \times T$, where r_{Th} corresponds to the theoretical adversary tolerance of the *upgraded protocol* (potentially different from the original protocol), so that for the upgraded stake, the required security property $\frac{AdversaryStake}{TotalStake_{upgraded}} < r_{Th}$ will still hold and thus the upgraded blockchain will be secure. Indeed, if we assume that the total upgraded stake percent is above the adoption threshold minimum value $\frac{1}{r_{Th}} \times T$ (e.g., $\frac{1}{r_{Th}} \times T + \delta$ where $\delta > 0$) and that all the adversary stake T has upgraded (so we have T percent of adversaries in the new blockchain too), then if we substitute, we see that the property $\frac{AdversaryStake}{TotalStake_{upgraded}} = \frac{T}{\frac{1}{r_{Th}} \times T + \delta} < r_{Th}$ holds for the upgraded protocol. So $\frac{1}{r_{Th}} T$ is the appropriate lower bound for the adoption threshold, in order to ensure safety. Here is the respective theorem for the adoption threshold.

Theorem 5.4.2. For each activation protocol with a threshold τ_A both the safety and liveness properties hold iff:

$$\frac{1}{r_{Th}} T < \tau_A < H$$

where r_{Th} is the theoretical adversary tolerance of our consensus protocol.

Proof. Since $\frac{1}{r_{Th}}T < \tau$, signals of any stake not sufficient w.r.t. the security assumption of the upgraded protocol, cannot cause the activation of the changes, because the condition for activation is $[signals' stake] > \tau_A$. Therefore, the safety property holds. Inversely, if the safety property holds, then any proposal that has not sufficient stake signaled cannot be activated, which means that $\frac{1}{r_{Th}}T < \tau$.

Similarly, since $\tau < H$, any proposal that has received signals of stake L_b or more, from the honest stake, where $\tau_A < L_b \leq H$ will be activated, because the condition for approval is $[signals' stake] > \tau_A$. Inversely, if the liveness condition holds, then any proposal that has received signals of stake at least L_b from the honest stake will be activated, which means that the following condition should hold $\tau_A < L_b$. \square

In the following, we propose a threshold function $\tau(\gamma)$, which depends on a parameter γ that we call the *safety strength* and adjusts the threshold towards liveness or safety. We provide also the sufficient conditions, in order for both safety and liveness properties to hold. Intuitively, if we increase the threshold, we increase the required amount of honest-stake positive votes required (or signals), in order to approve (or activate) a proposal. This means that we decrease liveness, but at the same time we increase safety, since it becomes more difficult for a malicious proposal to be approved (or for activation without a sufficient amount of stake to take place). The definition of the threshold function includes also a fixed amount of honest stake H_{min} , which is the minimum value that we want our threshold to have and essentially determines the minimum amount of honest stake that we wish not to be blocked (either for voting or activation), i.e., it determines the minimum amount of honest stake that we wish the liveness property to hold.

Although the threshold function definition is common for both voting and activation, since they differ on the required constraint, we present them in two separate theorems:

Theorem 5.4.3. If the threshold τ of a voting protocol is defined as:

$$\begin{aligned} \tau(\gamma) &= H_{min} + \gamma, \text{ where } 0 \leq \gamma < H - H_{min} \\ &\wedge \\ T &< H_{min} \end{aligned}$$

,where $0 \leq H_{min} \leq H$, then both the safety and liveness properties hold for any amount of honest stake L_b such that $H_{min} + \gamma < L_b \leq H$.

Proof. Since

$$\begin{aligned} T < H_{min} &\iff \\ T < H_{min} + \gamma &\iff \\ T < \tau(\gamma) & \end{aligned}$$

Also, since $H_{min} + \gamma < L_b \leq H$, then we have $\tau(\gamma) < H$. Therefore we have proved that $T < \tau(\gamma) < H$ and thus based on theorem 5.4.1 both safety and liveness properties hold. \square

Theorem 5.4.4. If the threshold τ of an activation protocol is defined as:

$$\begin{aligned} \tau(\gamma) &= H_{min} + \gamma, \text{ where } 0 \leq \gamma < H - H_{min} \\ &\wedge \\ \frac{1}{r_{Th}}T &< H_{min} \end{aligned}$$

,where $0 \leq H_{min} \leq H$, then both the safety and liveness properties hold for any amount of honest stake L_b such that $H_{min} + \gamma < L_b \leq H$.

Proof. Since

$$\begin{aligned} \frac{1}{r_{Th}}T < H_{min} &\iff \\ \frac{1}{r_{Th}}T < H_{min} + \gamma &\iff \\ \frac{1}{r_{Th}}T < \tau(\gamma) & \end{aligned}$$

Also, since $H_{min} + \gamma < L_b \leq H$, then we have $\tau(\gamma) < H$. Therefore we have proved that $\frac{1}{r_{Th}}T < \tau(\gamma) < H$ and thus based on theorem 5.4.2 both safety and liveness properties hold. \square

The γ parameter (safety strength) allows us to adjust the threshold towards liveness, or safety. This is depicted in figure 5.3. Low gamma values provide thresholds with a greater liveness, but reduced safety and inversely, high γ values enable higher safety, but less liveness. This applies to both the voting and the activation processes and this figure corresponds to both.

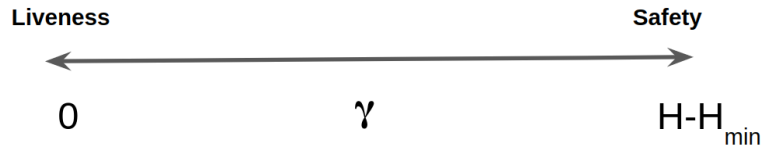


Figure 5.3: The γ parameter versus liveness and safety for either voting or activation.

This parametric definition of the threshold gives us the ability to adjust the threshold based on the context of each individual proposal and not follow a one size fits all approach. For example, for a very critical update we might require a higher γ value in order to ensure a greater safety for the result by sacrificing some of the liveness and demanding a stronger consent from the community.

Table 5.1 shows some examples of different threshold functions $\tau(\gamma)$, expressed by means of the adversary stake ratio r , $r = \frac{\text{Adversary Stake}}{\text{Total Stake}} = \frac{T}{T+H} = \frac{T}{100}$, along with the corresponding constraint on r , derived from the $T < H_{min}$ constraint and $\frac{1}{r_{Th}}T < H_{min}$ constraint for voting and activation respectively. The choice of H_{min} recorded on the second column determines the resulting threshold function, as well as the required constraint, appearing in the respective columns. Our analysis is based on the assumption that we can have an estimation of the adversary ratio r . From Garay et. al. in [GK18], we know that the ratio r is always upper bounded by the theoretical adversary tolerance r_{Th} of our consensus protocol, i.e., $r < r_{Th}$ (e.g., $r < 1/2 = r_{Th}$, or $r < 1/3 = r_{Th}$ etc.).

Process	H_{min}	Threshold $\tau(\gamma) = H_{min} + \gamma$, where $0 \leq \gamma < H - H_{min}$	Constraint $T < H_{min}$ (for voting) $\frac{1}{r_{Th}T} < H_{min}$ (for activation)
Voting	$\frac{H}{2}$ $T + 1$	$50(1 - r) + \gamma$ $100r + 1 + \gamma$	$r < 1/3$ Any $r < r_{Th}$
Activation	$\frac{H}{2}$ $\frac{1}{r_{Th}}T + 1$	$50(1 - r) + \gamma$ $\frac{100r}{r_{Th}} + 1 + \gamma$	$r < \frac{r_{Th}}{2+r_{Th}}$ ($r < 1/5$ for $r_{Th} = 1/2$) Any $r < r_{Th}$

Table 5.1: Examples of different threshold definitions.

In table 5.2, we provide examples of threshold values for different values of γ and different adversary ratios r for a specific choice of the $\tau(\gamma)$ function and a constraint relevant to the voting process. In particular, for the

function $\tau(\gamma) = H/2 + \gamma = 50(1 - r) + \gamma$ and $r < 1/3$ (i.e., $H_{min} = \frac{H}{2}$ and $T < \frac{H}{2}$). For a specific adversary ratio r , as we increase γ , we require stronger honest stake majority, in order to avoid the denial of approval attack, however this provides us with more safety. In this particular case of $\tau(\gamma)$, the allowed γ values lie in the range $0 \leq \gamma < H/2$. So in the table we have chosen the lowest possible γ value ($\gamma = 0$), the maximum γ value ($\gamma = \frac{H}{2} - 1$) (assuming for the sake of the example that γ takes integer values) and an intermediate value $\gamma = T$.

So for instance, in the line where $r = 0.1$, we can see that the allowed range of values for the threshold is [45%, 89%]. As long as the threshold is in this range, we can have both liveness and safety (assuming $r = 0.1$). We observe that for low values of γ ($\gamma = 0$), $H/2$ of honest stake is enough to have liveness. In contrast, for high values of γ ($\gamma = H/2 - 1$), we essentially require honest stake unanimity in order to have liveness.

Adversary ratio r ($r < 1/3 \approx 0.33$)	$\gamma = 0$	$\gamma = T$	$\gamma = \frac{H}{2} - 1$
0.1	45%	55%	89%
0.2	40%	60%	79%
0.3	35%	65%	69%

Table 5.2: Threshold values for different γ and adversary ratios for a specific choice of $\tau(\gamma)$ ($\tau(\gamma) = 50(1 - r) + \gamma$), relevant to the voting process.

5.5 Availability

Our work on decentralized software updates, as well as the prototype code is available open-source at <https://github.com/input-output-hk/decentralized-software-updates> under Apache 2.0 license. Its maturity level is research prototype, but it has been:

- Fully integrated with the Cardano node,
- Deployed and tested on an geographically distributed AWS testnet,
- Has been extensively property-based tested (in the context of Deliverable D1.3).

Chapter 6

Conclusions

This deliverable presented the final report on toolkits, use cases and prototype solutions for secure ledger systems developed in PRIViLEDGE. In summary, D4.4. presented:

- the final report on two PRIViLEDGE toolkits developed by IBM: a toolkit for anonymous authentication (Chapter 2) and a toolkit for flexible consensus in Hyperledger Fabric (Chapter 3),
- a toolkit for post-quantum secure protocols for distributed ledgers developed by Guardtime (Chapter 4), and
- the report on decentralized software updates for Cardano stake-based ledger use case by IOResearch (Chapter 5).

The system implementation details of the presented toolkits and use cases can be found in respective deliverable D2.x and D3.x. All software is available open-source under permissive licenses (MIT/Apache 2.0).

Bibliography

- [B⁺14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
- [BB04] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 56–73, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [BCET19] Dmytro Bogatov, Angelo De Caro, Kaoutar Elkhiyaoui, and Björn Tackmann. Anonymous transactions with revocation and auditing in hyperledger fabric. *Cryptology ePrint Archive*, Report 2019/1097, 2019. <https://eprint.iacr.org/2019/1097>.
- [BKL13] Ahto Buldas, Andres Kroonmaa, and Risto Laanoja. Keyless signatures’ infrastructure: How to build global distributed hash-trees. In *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*, pages 313–320, 2013.
- [BLT17] Ahto Buldas, Risto Laanoja, and Ahto Truu. A server-assisted hash-based signature scheme. In *Secure IT Systems - 22nd Nordic Conference, NordSec 2017, Tartu, Estonia, November 8-10, 2017, Proceedings*, pages 3–17, 2017.
- [BN05] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*, pages 319–331. Springer, 2005.
- [CDD17] Jan Camenisch, Manu Drijvers, and Maria Dubovitskaya. Practical uc-secure delegatable credentials with attributes and their application to blockchain. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS ’17*, page 683–699, New York, NY, USA, 2017. Association for Computing Machinery.
- [CF15] Cardano-Foundation. Cardano white paper, 2015. <https://www.cardano.org/en/philosophy/>.
- [CKKZ20] Michele Ciampi, Nikos Karayannidis, Aggelos Kiayias, and Dionysis Zindros. Updatable blockchains. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*, volume 12309 of *Lecture Notes in Computer Science*, pages 590–609. Springer, 2020.
- [CKS09] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanisław Jarecki and Gene Tsudik, editors, *Public Key Cryptography – PKC 2009*, pages 481–500, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [GK18] Juan A. Garay and Aggelos Kiayias. Sok: A consensus taxonomy in the blockchain era. *IACR Cryptology ePrint Archive*, 2018:754, 2018.

D4.4 – Tools for Secure Ledger Systems

- [GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310, 2015.
- [Gro15] Jens Groth. Efficient fully structure-preserving signatures for large messages. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 239–259, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- [KBC19] Philipp Kant, Lars Brunjes, and Duncan Coutts. Engineering design specification for delegation and incentives in cardano – shelley - an iohk technical report. 2019.
- [KKL18] Dimitris Karakostas, Aggelos Kiayias, and Mario Larangeira. Account management and stake pools in proof of stake ledgers. 2018.
- [Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [PRI19] First report on architecture of secure ledger systems (d4.1). Technical report, PRIViLEDGE project, 2019.
- [PRI20] Final report on architecture (d4.3). Technical report, PRIViLEDGE project, 2020.
- [Sco] Michael Scott. The Apache Milagro Crypto Library.