DS-06-2017: Cybersecurity PPP: Cryptography

PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D1.2 – Validation Criteria

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

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

Lead contractor: Guardtime AS
Duration: 36 months

Project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020

Dissemination Level	
PU = Public, fully open	X
CO = Confidential, restricted under conditions set out in the Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC	

D1.2

Validation Criteria

Editor
Nikos Karagiannidis (IORESEARCH)
Contributors
Sven Heiberg (SCCEIV)
Sergei Kutenko (SCCEIV)
Ahto Truu (Guardtime)
Panos Louridas (GRNET)
Damian Nadales (IORESEARCH)
Nikos Karagiannidis (IORESEARCH)

Reviewers
Michele Ciampi (UEDIN)
Janno Siim (UEDIN)
Ivan Visconti (UNISA)
Vincenzo Iovino (UNISA)

31st December 2020
Revision 1.0

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

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

Executive Summary

In this document, we propose methods and detailed criteria for the validation of the prototype implementations developed in the context of the four use cases of the PRIViLEDGE project. The validation of *Decentralized Ledger Technology (DLT)* applications and blockchain systems pose some unique challenges compared to more traditional software systems, mainly due to the incorporated distributed protocols that need to operate with byzantine tolerance. Moreover, the utilization of advanced cryptography techniques for preserving the privacy of the data render the validation task even more challenging. It is true that there is a significant diversity in the implemented use cases, ranging from e-voting protocols with enhanced privacy, zero-knowledge protocols for verifying the integrity of sensitive medical data, university diplomas verification by authenticated verifiers, to the incorporation of a decentralized software update mechanism into the Cardano blockchain. In this document, we begin by laying out our arsenal of validation methods and tools for testing our implementations. Some of the techniques described are common and can be utilized by all use cases, while others are unique and specialized to specific use cases. Furthermore, we proceed by defining individual validation criteria for testing each and every aspect of the developed systems and applications. From the correctness and security of the implemented protocols to the usability of the deployed applications, a rich set of criteria has been recorded. These criteria may be expressed as properties, as unit tests, integration tests, as formal proofs, micro-benchmarks, or even as end-user usability tests depending of the nature of the feature that we try to validate. No matter what the form, or the type of each criterion is, the end-goal is to thoroughly test the developed prototypes and ensure their appropriateness for production use. This deliverable also sets the scene for the forthcoming deliverable (D1.3 “Use Case Validation”), where the actual validation results will be presented.

Contents

Chapter 1

Introduction

The validation of software systems, a ubiquitous topic in software development, is the process of answering the question "Are we building the right product?" [Boe81]. In the context of decentralized systems such as blockchains and DLT applications, this validation becomes a real challenge. This deliverable covers the validation of the implemented DLT systems and applications within work package 4 in the context of the four PRIViLEDGE use cases. In this document, we describe the tools and methods that will be utilized in order to collect the necessary evidence that the DLT applications and systems developed: a) fulfill the agreed business requirements, b) honor all functional and non-functional criteria, c) are secure, d) acceptable from a usability and user-friendliness perspective and e) finally capable to be deployed to production. Moreover, the document aims not only to generally define the methodologies to be used for the validation of each use case, but to define, in detail, each individual validation criterion, metric, property or Key Performance Indicator (KPI) that will guarantee the successful validation of each aspect of the system.

The PRIViLEDGE use cases uniquely combine DLT technology with privacy techniques to solve real world problems. In UC1, we focus on the auditing of election integrity under the condition of secret ballot. We are working with a voting protocol that allows to publish a cryptographic audit trail necessary for the integrity verification in such a manner that we can make it available for everyone on the public ledger without risking ballot secrecy even in the long term. The first validation challenge in this use case has to do with the validation of the requirements per se. The e-voting requirements set, must be validated against the EU recommendation on standards for e-voting [CoE17], where applicable. Moreover, the voting protocol proposed, which is the heart of the system, must be validated against such properties as correctness, integrity, privacy and verifiability. Finally, there are also other dimensions of validation that have to do with the usability of the developed application, as well as other non-functional requirements such as performance.

In UC2, the main goal is to enable a shift from activity-based costing to outcome-based costing and achieve better health outcomes for patients. To this end, we build a prototype for a health insurance system that enables care providers to report on the efficacy of treatments and the medication providers and payers to verify the correctness of those reports without disclosing the detailed health records of individual patients. This poses some unique challenges in the validation of a data-exchange service that allows posting of privacy-preserving commitments of medical records and providing zero-knowledge proofs that the aggregate performance reports are consistent with the underlying records, without the need to disclose the records themselves to either the payers or the pharmaceutical companies.

In UC3, we offer the solution to verify the authenticity of (paper-based) issued diplomas and issue diplomas digitally in a common format, while protecting the graduates privacy. This will decrease manual labour wasted to certify authenticity of past diplomas and decrease fraud. The specific implementation faces some unique validation challenges arising from the special requirements of the system. These pertain to functional requirements (e.g., the off-chain storage of diplomas), non-functional requirements (such as speed, security, etc.), the smooth interoperability of the prototype with existing university systems, and finally the usability and user-friendliness features of the developed application.

Finally in UC4, a prototype of a software update mechanism is implemented that handles software updates for blockchains in a completely *decentralized* manner, which is incorporated into the Cardano proof-of-stake blockchain system. One notable difference of this use case compared to the other three is that this is not an end-user-facing application that is being developed but rather an extension of the Cardano node, which is the software running the Cardano blockchain protocol. Therefore, we essentially have to deal with the validation of a *ledger system* rather than a ledger application. To this end, we have followed a requirements-driven approach and have defined detailed validation criteria that correspond to the requirements set. Testing a software update protocol is significantly complex and the integration with the Cardano ledger makes things much more challenging. For this reason, techniques such as state-machine modeling and *property-based testing* have been used to automatically produce thousands of scenarios, which are then recorded on traces, on which properties (as valid state transitions) can be defined and tested.

The primary goal of this deliverable is to provide a good description of the various validation techniques used in order to test the developed prototypes, present at an appropriate level of detail the individual validation criteria and of course set the scene for the forthcoming deliverable (D1.3 "Use Case Validation") where the actual validation results will be presented.

This deliverable is structured as follows: In chapter 2, we describe the validation methods that will be used. In particular, in section 2.1 we begin with a description of validation methods that are common to all use cases; then we continue with more specialized validation approaches that fit to the needs of the individual use cases. Next, in chapter 3, the validation criteria for the i-voting use case are presented. Following in chapters 4 and 5, a detailed description of the validation criteria used for the validation of the health-care and diplomas use case respectively are provided. Finally, in chapter 6 the validation criteria of the Cardano update system prototype are presented. The document ends with a wrap up and conclusions section in chapter 7.

Chapter 2

Validation Methodology

2.1 Common validation methods

For all use cases in PRIViLEDGE, we have set up a number of requirements that should be satisfied. These requirements have been to a large extend described in the corresponding deliverable *D1.1 Requirements and Interface Design*. Of course, there have been some updates since, but they still constitute a core set of initial requirements. These requirements and the ones added in the course of time, can be validated in different (not necessarily exclusive) ways. These validation methods can be potentially utilized by more than one use-case and so they are common in a sense. Therefore we have decided to describe them in a single section. We list these common validation methods below:

Formal proof This is the highest level of rigor and assurance we can demand. If a property of the system is formally proven, we know that, provided that the system implements it correctly, it will hold in any possible execution. The main impediment to using formal proofs for verifying properties of a system is that they are time consuming. Therefore we will use them as validation criteria very sparsely. Of course, all the cryptographic protocols used to realize the use cases have been formally proved in the context of the 3rd work package (WP3).

Property based testing In this validation method, the requirements are expressed as properties that can be checked by a computer. Then by means of random generation of test cases, the properties are automatically checked. Property based tests do not provide a mathematical proof, however they allow to check properties with a high level of rigor in only a small fraction of the time that a mathematical proof would have required. In addition, property based tests run on the actual implemented code, which means that if the code no longer satisfies a property, the tests will fail. By contrast, a mathematical proof cannot check for consistency between the system used in the proof and the actual implementation. One of the main drawbacks of property based testing is that no matter how thorough a property is checked, it still depends on how good the random test cases coverage is. If the test case generation fails to cover important cases, then we might have a false sense of security. Luckily, we can resort to techniques such as *test cases labeling* and statistic coverage analysis that can help mitigate this problem.

Case based testing In this case we simply manually define test cases by specifying the system's inputs and the expected output. This is less thorough than property based testing, but quite important nevertheless, since it allows to validate our expectations with very concrete examples. This broad category includes *unit testing* and *integration testing*, with the former focusing on the local correctness of a component and the latter on the global correctness of processes spanning multiple components.

Simulation In this validation method, we run the actual system but with some of its parts replaced by a simulated component. For instance, we can simulate the network component, or run it in an actual network but

simulating the participants' behavior. During or after a simulation run we inspect the system trace, either manually or automatically, and check that what we observe corresponds with our predefined expectations.

Asymptotic complexity analysis Some requirements such as execution speed, bandwidth utilization, or memory requirements can sometimes be statically determined without the need of running or simulating the actual system. While thorough, this kind of analysis requires manual audits of the code to make sure that the analysis reflects the actual implementation.

Microbenchmarking By identifying components of the system that are likely to become a bottleneck in terms of CPU usage, or memory consumption, it is possible to run micro-benchmarks that measure the execution of a specific component/task of the system and give actual measurements (e.g., execution time, memory allocation etc.) for different input sizes.

In the following sections we describe how each use case will utilize the aforementioned methodologies to validate the corresponding implementations.

2.2 Validation methodology—Use case 1: verifiable online voting with ledgers

The primary goals and more detailed requirements for the online voting use-case have been laid out in project deliverables [PRI18,PRI19]. The approach to achieve the goals and fulfill the requirements has been documented in the project deliverable [PRI19] and thesis [Avi19, Kuš20].

For the online voting use-case the result of the project is a pilot *implementation* of an online voting *protocol* using the DLT technology, fulfilling the *requirements*. This calls for the validation on three different levels—validation of the requirements, validation of the proposed protocol to satisfy the requirements and validation of the implementation of the protocol.

2.2.1 Requirements validation

The goal of the requirements validation is to

- analyze and document the compatibility of the requirements of online voting use-case with the EU recommendation on standards for e-voting ([CoE17]) where applicable;
- provide a basis for the traceability matrix to understand the completeness of protocol and implementation validation steps;
- document the extent that the pilot implementation is implementing the protocol and the requirements.

2.2.2 Protocol validation

Current protocol proposal as documented in [Avi19] formulates a set of voting protocol properties and heuristically proves that these properties are satisfied. However, rigorous definitions must be given and formal analysis carried out to completely assess the protocol.

2.2.3 Implementation validation

The pilot implementation of the protocol must allow for the simulation of all core activities of the online voting—setup with DLT, voting, tallying and auditing. Both ceremonies to carry out these activities and tools to support the process must be present. Following validation methods from the standard set of software development methods are going to be used:

Automated unit testing shall be used to validate correctness of individual components. The automated nature of these tests provides the basis for regression testing.

Manual End-to-End testing / simulation shall be used to validate that all required functionalities are sufficiently documented in ceremonies and the achievement of the main goal—e.g. use of DLT for auditing purposes—can be demonstrated.

Benchmarking shall be used to measure throughput of the system under fixed set of parameters and for detection of the bottlenecks in the system.

2.3 Validation methodology—Use case 2: distributed ledger for health insurance

The requirements of this use case have been described in project deliverables [PRI18, PRI20]. Very briefly, these can be summarized as follows:

- Healthcare providers need to be able to post cumulative commitments of patient records and prove their intergity.
- Patients need to be able to review their own records and verify their consistency with the posted commitments.
- Payers need to be able to receive aggregate reports on treatment outcomes and verify theirs consistency with the posted commitments, without learning the details of individual patients.

The main deliverable of the project for this use case will be a prototype implementation of the system. Once the proof of concept has been validated, it will be the basis of adding such functionalities into products and services of Guardtime Health.[1]

2.3.1 Functional validation

As the functional requirements describe the ability of various types of users to perform certain operations (to add records, to view records, to generate reports), unit testing and integration testing will be used to validate that the functional requirements have been met.

To reduce the risk of exposure of confidential data of actual patients, both unit and integration testing will be performed on synthetic data.

2.3.2 Security validation

As the security requirements describe the inability of adversarial parties to perform certain operations, these cannot be validated by testing or bechmarking. Instead, static analysis methods will have to be employed. The security of the underlying cryptographic primitives and protocols will have to be proven by formal analysis.

In principle, correctness of the implementation could also be formally verified, but this is not realistic given the size of the system and the ability of current formal analysis tools. Instead, code reviews and static analysis tools will be used to validate the implementation of the protocol in the production implementation. As the prototype will only be used to prove the concept, the robustness requirements are much lower and thus only code reviews will be employed in the scope of this project.

2.3.3 Performance validation

The performance of the proposed protocols will first be analysed theoretically, in the asymptotic complexity model. The results will then be validated based on benchmarking on a representative set of test data. This is expected yield a robust model for estimating the performance of the system when operating on various real world data sets.

[1] `https://guardtime.com/health`

2.3.4 Usability validation

As the prototype to be implemented in the project is a proof of concept for back-end technology, and the user interface of real products will be provided by the respective medical information management systems, usability testing will not be performed in the scope of this project.

2.4 Validation methodology—Use case 3: university diploma record ledger

The primary goals of the University Diplomas use case has been described in project deliverables [PRI18,PRI20].

2.4.1 Functional validation

Functional validation refers to the assessment of the coverage of the functionalities required by the implemented system. This will be provided in the terms of a traceability matrix.

2.4.2 Non-functional validation

Non-functional validation refers to the assessment of non-functional requirements, such as speed, security, transaction volume. This will be provided by reporting on the behaviour of the system with respect to each one of the identified non-functional requirements.

2.4.3 Usability validation

An important factor for deciding whether the right product is build, in the context of UC3, is its usability as perceived by the various stakeholders. To evaluate the usability of the University Diploma Record Ledger we are going to follow a *Heuristic Evaluation* approach. As Heuristic Evaluation lies beyond cryptographic applications and DLTs, in the following we give an outline of the basics of the approach.

What is heuristic evaluation

Heuristic Evaluation [NM90, NM94] is a usability engineering method that is used to detect usability issues in an early stage of the design process of a software service. One or preferably more evaluators compare the user interface, its aesthetics and functionality, against a set of design principles, termed heuristics (a full list of the most used set of heuristics is described below). A group of evaluators that are not related with the development team takes the role of the user and tests the software service against this checklist of criteria to find usability issues that have gone unnoticed by the development team.

Advantages of heuristic evaluation

A non-exhaustive list of the advantages of heuristic evaluation is presented below:

- It is a fast and inexpensive method.
- Evaluators do not have to have formal usability training.
- It provides reliable results.
- It can help the team to gather feedback early in the development process.
- It can be combined effortlessly with other usability testing methodologies.
- It does not require a lot of advanced planning.
- The scope can be adaptable as every evaluation may contain one or more user journeys.

Disadvantages of heuristic evaluation

Potential drawbacks and caveats that will have to be taken into account when applying Heuristic Evaluation include:

- The evaluator should have some knowledge and experience of the evaluation process and the user experience in general to find the majority of the issues.
- Although even a single evaluator may find a sufficient percentage of issues, a greater number is better but this is not always achievable.
- The process may not identify as many usability issues as for example some usability testing with real users.
- It is not always the case that defining the problem can lead the evaluator to designate a solution.
- The evaluators, as they are not the users, may be biased and identify more minor issues and fewer major issues.

2.4.4 Usability heuristics

Jakob Nielsen's ten general principles for user interface design are undoubtedly the most widely used usability heuristics in the heuristic evaluation process [NM94].

Visibility of system status: The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Match between system and the real world: The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. It should follow real-world conventions, making information appear in a natural and logical order.

User control and freedom: Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Consistency and standards: Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Error prevention: Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Error-prone conditions should be eliminated or the system should check for them and present users with a confirmation option before they commit to the action.

Recognition rather than recall: The user's memory load should be minimized by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Flexibility and efficiency of use: Acceleratorsunseen by the novice usermay often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Users should be able to tailor frequent actions.

Aesthetic and minimalist design: Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Help users recognise, diagnose, and recover from errors: Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Help and documentation: Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

2.4.5 Conducting a heuristic evaluation

The Heuristic Evaluation process contains the following steps:

Preparation stage

During the preparation stage the evaluation team should be given information regarding the software service and the users that are going to use it. It would be very helpful for the evaluators to work with certain personas and user stories in order to examine specific scenarios during the evaluation of the user interface. Furthermore, it is essential for the evaluators to be familiar with the list of the heuristics that they are going to use during the evaluation process. Thus, the evaluators should be provided with a set of heuristics, a list of representative tasks for the application and a testing environment to conduct their assessment.

Execution stage

In this stage, the evaluators should examine every component and flow of the user interface according to the representative tasks and analyse them against the defined heuristics. It is essential to use a well organised way of providing their findings. As an example, a set of categories that should be filled during the Heuristic Evaluation process is shown below.

Section should be filled with the URL or the UI description of where the problem was encountered.

UI component description of the component the causes the violation (e.g., form, navbar , etc.).

Violated Heuristic note of one or more heuristics that has been violated.

Usability Issue detailed description of the encountered problem.

Severity assignment of severity code from a predefined list (e.g., low, medium, high).

The above can be recorded in a tabular format, with the categories in columns and rows for each UI component that is evaluated. In addition, the evaluator may choose to also suggest some fixes to the problems found.

Analysis stage

In the analysis stage, the evaluators should organise their findings in order to remove any duplicates and categorise them in accordance to their context and severity level. The final report should contain all the necessary information in a way that is efficient for the development team to easily understand the problems and correct them.

Remarks

Heuristic Evaluation is an effective method for a rapid evaluation of the usability status of a software service as it can help the development teams in the early detection of their implementation's faults. Although it has its limitations it is a sensible start for a team that wants to provide an above average quality of service and can't afford, or it is not possible, to invest in a professional user testing method.

2.4.6 Interoperability validation

Interoperability validation refers to the assessment of the interoperation between the ledger application of the use case and the information systems of higher education establishments.

2.4.7 Implementation validation

Implementation validation refers to the assessment of the correctness of the code implementing the use case. Absent a formal verification method, which will not be used in this use case, implementation validation will be performed using suitable testing.

2.4.8 Deployment validation

Deployment validation refers to the assessment of the mechanisms for effecting the transition from code to the deployment of a working system via Continuous Integration / Continuous Deployment approaches.

2.5 Validation methodology—Use case 4: Cardano update system

The detailed requirements of this use case have been described in previous deliverables [PRI18, PRI19]. In this section, we will give an overview of the validation techniques that we will use in order to validate our system with respect to the requirements set.

2.5.1 Formal proof

The first validation criteria for the Cardano update system will be done via a formal mathematical proof. Our system will be based on multiple cryptographic primitives, which are based on different assumptions. Examples of such assumptions are what is possible to compute efficiently, how some infrastructure is set up, who is trusted, what can we assume about hardware and so on.

The proof of our system will then follow the common approach in cryptography that relies on *reductions*. A reduction is an algorithm for transforming one problem into another problem. A sufficiently efficient reduction from one problem to another may be used to show that the second problem is at least as difficult as the first. In our specific case, we will show that if our protocol is insecure, then one of the cryptographic primitives that we have used to construct our protocol must be insecure, and this would lead to a contradiction.

Information-theoretic and computational security

We remind these two very basic notions for the sake of completeness.

Information-theoretic security is understood to be the kind of security that even an adversary with unlimited computing power cannot break. Essentially, the information is, information-theoretically speaking, not there.

This is contrasted with computational security. Computational security protocols come with some complexity parameter κ that describes the computational bounds of the adversary — it is said that an adversary whose computational powers are bound by κ can not break the scheme, while not making statements about computationally more powerful adversaries.

2.5.2 Property based testing

Validating a software using property based testing requires that a property is expressed in some target programming language. This property can be then checked by generating random inputs for the property and testing whether it holds for them. Cardano is implemented in Haskell, so we will be expressing our properties in Haskell, and using `QuickCheck` as our property based testing library.

A quintessential example of a property is that reversing a list twice should yield the original list. This can be readily expressed as follows:

```
prop_reverse :: [Int] -> Bool
prop_reverse xs = reverse (reverse xs) == xs
```

Testing this property requires that the test program has a way to generating random integers (`Int`).

When a property fails, a *counterexample* is produced. It is very important that said counterexample is small since this aids the developers in understanding the cause of the problem. For instance, consider `prop_reverse` above failing due to a wrong implementation of reverse, and giving as a counterexample a list of 1000 elements. If the problem can be reproduced with a list of three elements, then this would be highly preferred. To enable `QuickCheck` to give minimal counterexamples we need to provide a *shrink function* for our data types. A shrink function basically determines for a given value the ways in which it can be shrunk. Thus a good counterexample depends on good shrinking.

Another important aspect of property based testing is the degree of coverage our tests have. For instance, if we test a property of integer lists with only small integer values we might miss to uncover violations to this property caused by very large values. Therefore, besides testing our properties when doing property based test it is important to *test the coverage of the generators*. Luckily `QuickCheck` provides good support for writing and performing these tests. So for instance, we can write tests that state that 10% of our test cases should consist of lists with at least 5 large numbers.

So to summarize, using property based testing as validation methodology requires to:

- Express the properties in Haskell.
- Write generators for the input data.
- Write shrink functions for the input data.
- Write coverage tests for the generators.

Cardano is divided intro three main components:

1. Network: which takes care of communication between nodes and clients (such as wallets).
2. Consensus: which runs the consensus algorithm. This layer communicates with the ledger layer.
3. Ledger: which encodes the logic for determining which transactions are valid.

The bulk of the update system will be implemented in the *ledger layer* of Cardano.

The ledger layer API consists of a set of state transforming functions: this is, given an initial ledger state and some data-value (like a transaction or a slot-number), the functions in this API return either an error (if the data could not be applied in that state) or the state resulting from applying the given value to the given state.

Given that the ledger API is organized as a set of state transforming functions, we can characterize *any* evolution of the ledger state as an interleaved sequence of state and actions: such sequence starts with some initial state, it is followed by a sequence of pairs of action and ensuing state. We call such sequence a *trace*.

Since we can characterize the behavior of the ledger in terms of set of traces, we can then express certain properties of the update system in terms of properties on traces. Thus, when testing the properties of the update system we will write:

1. Update system (ledger) trace properties.
2. Generators for these traces.
3. Shrink functions for traces.
4. Coverage tests on traces.

Note that properties that hold on (ledger) traces are transferable to the whole blockchain system since the consensus layer relies on the ledger to validate transaction, which means that in the blockchain we will only see sequences of transactions (traces) that are allowed by the ledger.

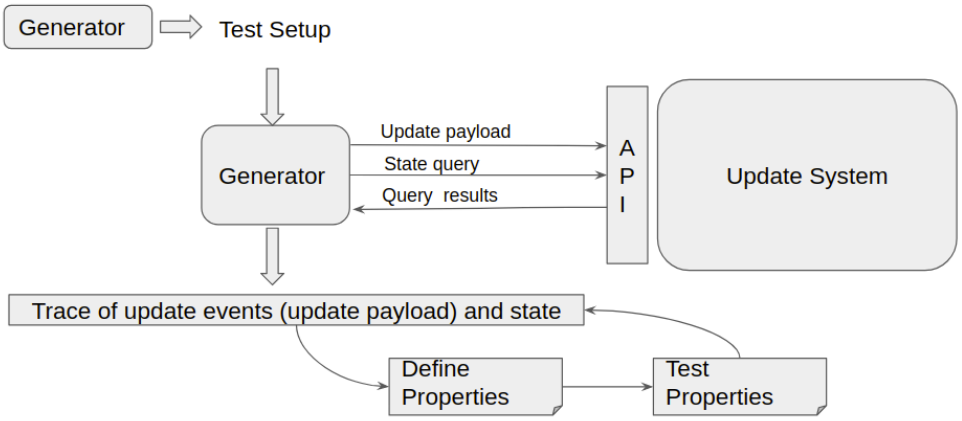

Figure 2.1: The testing framework for software updates.

In figure 2.1, we depict a high-level view of our testing framework. We can see the generator generating various test setups, which are fed into the main generator, which produces all the relevant update events (update payload). For each such payload the update system API is invoked, in order to handle it. The update system consumes the update payload based on the implemented update logic and consequently updates the *update state*. The main generator queries this state and records it in a trace along with the update payload generated in a chronological order. Finally, we define conditions (i.e., properties) over this trace of events, which constraint what state transitions are valid and with what update events this transition should take place. These conditions essentially form our validation criteria.

2.5.3 Unit testing

The structured-operational-semantics (SOS) [Wik20] rules framework of Cardano allows us to write tests in the form of "in this initial state, given this transaction we expect to see the following state". These tests will complement the property based tests and serve as a good source of examples and documentation for the update system. For running these tests we will be using a testing library for Haskell, most likely `tasty`.

2.5.4 Simulation

To simulate the update system we will integrate it directly to Cardano. This will not only allow us to validate additional properties of the system but to make sure that it is ready to be deployed.

Integrating the ledger layer to the other layers of Cardano requires that we satisfy properties which are also expressed as property based tests.

We will simulate the participants in the network by writing (update) transaction generators. Depending on the type of validation we want to perform on a simulation run, we will inspect the run of the system by analyzing

different artifacts such as log files or the blockchain stored on the nodes disc. The inspection will be manually done, and if time allows we will write automatic tests for simulation runs.

2.5.5 Asymptotic complexity analysis

Where applicable, we will write a formal analysis of different complexity metrics such as blockchain bandwidth usage, or execution time. For this we will use the size in bytes of the actual data structures used in the implementation, as well as audit the code that runs the various operations that we want to analyze.

2.5.6 Microbenchmarking

We will identify performance critical components in the update system and write micro-benchmark programs for them. We will use the `criterion`[2] library for this purpose. The benchmarks will be incorporated into the test suites, which developers can use for checking performance improvements or degradation as the system evolves.

[2] `http://www.serpentine.com/criterion/`

Chapter 3

Validation Criteria—Use Case 1: Verifiable Online Voting With Ledgers

3.1 Requirements validation criteria

Detailed requirements for the online voting use-case have been laid out in project deliverables [PRI18, PRI19]. The goal of the requirements validation process is to reassure the soundness and compelteness of these requirements, maintain potentially updated and detailed list of the requirements and provide basis for the validation of both protocol and the implementation.

The outcome of the project for UC1 is a pilot implementation of a cryptographic online voting protocol which is designed to be compatible with principles of democratic elections. This means that our requirements are inherited from different categories:

- requirements inherited from principles of democratic elections;
- online voting technology specific requirements;
- requirements deduced from the specific online voting protocol;
- implementation specific requirements;

In the requirements validation process we shall

- analyze and document the compatibility of the requirements of online voting use-case with the EU recommendation on standards for e-voting ([CoE17]) where applicable;
- maintain structured set of requirements for different categories with means to trace the origins of a particular requirement and its current implementation status;
- document the extent that the pilot implementation is implementing the protocol and the requirements.

3.2 Protocol validation criteria

The cryptographic protocol to achieve the target described in the requirements has been documented in the project deliverable [PRI19] and thesis [Avi19, Kuš20].

The protocol as documented in [Avi19] formulates a set of voting protocol properties and heuristically proves that these properties are satisfied. The goal of the protocol validation process is to give rigorous definition of the protocol and provide a high-level security proof.

The protocol validation properties come from one of the following categories:

Correctness Properties required to provide that data is valid or action is performed correctly.

Integrity Properties required to make sure that stored data has not been altered and there is auditable trail.

Privacy Properties required to provide that sensitive data is not given nor available to non-authorized parties.

Verifiability Properties required to ensure that with given data and/or proofs it is possible to verify that action was made correctly.

3.2.1 Correctness

Property 1. A voter can post exactly 1 choice from set of n candidates as a vote to a ballot box.

This property will be verified with the use of zero-knowledge proofs to ensure that the vote is well-formed. In case voter chooses more than one candidate, system must reject the vote from being added to the ballot box and bulletin board.

Property 2. A voter can post up to m unordered choices from set of n candidates as a vote to a ballot box.

This property is orthogonal mode of operation for the previous property. It will be verified with the use of zero-knowledge proofs to ensure that the vote is well-formed. In case voter chooses more than m candidates, the system must reject the vote from addition to the ballot box and bulletin board.

Property 3. Only persons from the set of eligible voters can post votes to a ballot box and bulletin board.

The proper implementation of this property can be checked by casting vote as an eligible and ineligible person. Whereas eligible person's vote must be successfully added to the ballot box and bulletin board and attempted vote by a ineligible person must be rejected by the system.
The system shall use digital signature based credentials to verify the eligibility.

Property 4. Only votes cast during specified time (the voting period) can be accepted into the ballot box.

It is necessary to remove possibility for early and late voting. In order to validate this property 3 votes will be made at a different times:

1. **before election start time** - system shall reject vote as election has not started yet;
2. **after election start, but before end time** - system shall accept vote and adds it to the ballot box and bulletin board;
3. **after election end time** - system shall reject vote as election ended.

Property 5. Voter may re-vote unlimited number of times during the voting period. Only one of re-votes (the last one) shall be taken into account in the tally.

This property will ensure that voter will have a possibility to re-vote, potentially for different candidate, in case she feels e.g. coerced. The proper implementation of this property can be checked by simulating the re-voting process and checking that the new vote was added to the ballot box and bulletin board.

3.2.2 Integrity

Property 6. Posted votes are available after the end of the voting period.

This property ensures the possibility to use posted votes for the tally.

We shall simulate voting with multiple voters and verify that when voting period ended it is possible to gather posted votes from ballot box and bulletin board.

Property 7. Tally results must not be published before the end of a voting period.

This property ensures that no-one will get intermediate results which can coerce or change voters motivations to vote.

We will simulate results publication before election start and end time. On both simulations system must forbid results publication.

Property 8. In case there are several votes for any single voter, only the last one, according to the bulletin board is taken into account in the tally.

This property ensures uniformity of the vote in case of re-voting being present. The proper implementation of this property can be verified by auditing the bulletin board.

Property 9. Votes may be excluded from the tally upon request.

This property is necessary to support the revocation of some votes (such as multiple votes that preceded the last vote of a re-voter). Revocation is also necessary in case of multi-channel voting, where voter may opt to use different voting channels simultaneously. In this case only one vote must be taken into account and other votes revoked.

We shall cast vote for particular voter and apply a revocation list for the same voter, whereas system must exclude voter vote from the tally.

3.2.3 Privacy

Property 10. Voter's choice is not leaked for any of the stored votes.

This property follows from the requirement of ballot secrecy. In order to ensure that it is impossible to leak any choice, the system forbids individual vote decryption. However, this policy can be enforced effectively only in case some distributed decryption mechanism is applied, which needs quorum of participants in order to make a decision to decrypt something.

Property 11. No-one is capable of learning which choices a particular voter made for any of the posted votes.

Same as previous property, but in order to ensure that public commitments on the bulletin board do not provide any information about each particular voters choice, mathematical description shall be provided.

Property 12. Voter cannot prove to a third party how she voted.

This property follows directly from the protocol and must be verified by protocol security proofs.

3.2.4 Verifiability

Property 13. Voter can verify that she communicates with the application/service which is in control of Election Official.

As this is a common problem with web based applications, it is important to guide voters to only valid URLs. We will publish trusted URLs of web applications and services via trusted channel.

Property 14. Voter can verify that her posted vote correctly reflects her choice from a set of candidates.

This property ensures that voter will be sure that her choice will not be altered during the process of casting. Voter can check her public part on the bulletin board and verify that it is correct and in case it's not re-vote and notify Election Organizer.

Property 15. System can detect that posted vote was not tampered with.

As we are using digital signature scheme and voter signs her vote, we will simulate a modification of the vote. It is expected, that the system detects the modification and removes vote from the tally and adds voter to the revocation list.

Property 16. System can detect that stored or posted vote is invalid.

This property follows from the protocol definition, it must be validated with protocol definition and implementation validated with software development specific means. In case when stored vote is invalid system must detect it at vote aggregation and add voter to the revocation list. In case when posted vote is invalid it must be possible to be detected by any observer (voter, auditor, election organizer).

Property 17. The auditor can verify that the ballot box and the bulletin board contains votes only from eligible voters.

This property follows from the protocol definition, it must be validated with protocol definition and implementation validated with software development specific means.

Property 18. The auditor can verify that the set of posted votes sent for tallying is the same as the set of posted votes accepted from the eligible voters.

This property follows from the protocol definition, it must be validated with protocol definition and implementation validated with software development specific means. For this property validation we hold voter list publicly on the bulletin board which can be easily verified with the posted votes.

Property 19. The auditor can verify that votes excluded from the tally were revoked on purpose.

Same as previous, revocation list is published on the bulletin board.

3.3 Implementation validation criteria

The pilot implementation of the protocol must allow for the simulation of all core activities of the online voting—setup with DLT, voting, tallying and auditing. Both ceremonies to carry out these activities and tools to support the process must be present, usable and sufficiently efficient.

Following criteria must be filled by the implementation:

- Code coverage of automated test must be at least 70
- All ceremonies for major use-cases – setup, voting, tallying, auditing – must be documented in repeatable manner.

3.3.1 Usability

Voting application is a simple wizard that supports voter in the process of authentication, selecting and encrypting a candidate, confirming the selection with digital signature, casting the vote and acquiring the cryptographic material required for individual verifiability. This wizard must fulfill following criteria:

- the process can be cancelled by the voter any time before the confirmation;

- the voter is prevented from accidentally voting for an undesired candidate (by e.g. explicitly selecting / deselecting);
- the voter is prevented from overvoting or undervoting (in case election has m of n rule and voter marks only $m - 1$ choices), at all times she is informed about how many options she still has left;
- voter identity and voter choices are not displayed together on the same screen to fight against coercion;
- the ballot presentation (including candidate order) is aligned with the requirements of specific election;
- the information is presented in unbiased manner, voter has chance to review all options;
- the application follows adaptive/responsive design techniques to support various screen sizes and orientation;
- the application supports keyboard-only navigation;
- the application is compatible with assistive technologies.

3.3.2 Performance

From the perspective of the voter, the process has to be responsive - a voter would expect to have voted and optionally verified her votes in 5 minute time. This means that under test conditions we require the average query-response time to be within a couple of seconds and we would expect the confirmation of the vote on the bulletin board to take place in 15-20 seconds. However, these parameters heavily depend on the size of the election, dimensioning of the system and networking conditions. We expect the system to be ready to handle at least 30 000 voters (we have an intention to find the upper limit). In order to have better understanding of our capabilities we shall run several benchmarks under varying conditions:

- parallel voting rate;
- number of bulletin board participants;
- network bandwidth;
- available system resources.

We shall observe and document following parameters:

- communication footprint,
- latency,
- length of the voting process,
- time to confirm on the bulletin board,
- system storage, memory consumption, computational power.

Chapter 4

Validation Criteria—Use Case 2: Distributed Ledger for Health Insurance

The requirements of this use case have been described in project deliverables [PRI18, PRI20] and briefly summarized in Section 2.3. The main deliverable of the project for this use case will be a prototype implementation of the system. As explained in Section 2.3, the prototype will be used to evaluate the functionality, security, and performance properties of the underlying protocol.

4.1 Functional validation criteria

As detailed in [PRI20], the main required functions of the system will be to

- update patient records with new data (diagnoses, treatments, results);
- posting commitments of updated records;
- extracting records of individual patients, along with proofs of consistency with the posted commitment;
- aggregating records of treatment results of groups of patients, along with proofs of consistency with the posted commitment.

To improve both performance and business confidentiality, the updates to patient records will be posted in batches. This yields the following functional properties to be evaluated in the prototype implementation:

Property 1. It must be possible to compute a commitment for a set of events. As the events represent patient data, the commitments must not leak the details of the underlying events (Property 6). As the commitments will be the trust anchors for zero-knowledge proofs of data integrity and correctness of computations, the commitments must be binding (Property 5).

Property 2. It must be possible to compute a union commitment representing the union of two sets of events. The union commitment must additionally come with a correctness proof showing that the new commitment is consistent with the commitments of the two input sets, but without leaking details of the elements of the underlying sets (Property 7, Property 8).

Property 3. It must be possible to compute a subset commitment representing a subset of events (specified by the condition of having the value of the given attribute in the given range). Like the union commitment, the subset commitment must also come with a privacy-preserving correctness proof (Property 9, Property 10).

Property 4. It must be possible to compute simple aggregates (count of elements, sum of values of a given attribute) over the sets under the commitments. Like the union and subset commitments, the aggregate must also come with a privacy-preserving correctness proof (Property 11, Property 12).

4.2 Security validation criteria

Property 5. It must be infeasible to change the data under a commitment without breaking the link to the commitment.

Property 6. It must be infeasible to recover the underlying data from a commitment.

Property 7. When computing a union commitment, it must be possible to get a proof that the new commitment represents the union of the sets of events represented by the inputs.

Property 8. The proof of correctness of the union commitment must not leak the details of the underlying events.

Property 9. When computing a subset commitment, it must be possible to get a proof that the new commitment represents exactly the subset of the set of events matching the given filtering condition.

Property 10. The proof of correctness of the subset commitment must not leak the details of the underlying events. When the subset in question is records of one patient extracted for review by the patient or for referral, the zero-knowledge property is required only for the superset input. However, when the subset is used for generating a report, also the details of the entries in the extracted subset must not be leaked by the proof. This may present a performance trade-off (cf. Property 14, Property 16).

Property 11. When computing an aggregate of a subset under a commitment, it must be possible to get a proof that the aggregate was computed correctly.

Property 12. The proof of correctness of the aggregate must not leak the details of the underlying events.

Because the commitments and the related proofs leave the control of the party authorized to handle the patient data, the hiding property of the commitment scheme and the zero-kowledge property of the proof system must hold in long term. As the commitments protect the integrity of patient data, the binding property of the commitment scheme should also hold in long term. In contrast, the decisions on validity of the records under a commitment and the correctness of the computed aggregate results are taken close to immediately, so the binding properties of both the commitment scheme and the proof system only need to hold in short term. These observations may guide our choice of protocols and their parameters.

4.3 Performance validation criteria

Property 13. A busy regional hospital can generate tens of thousands of events in a day. As medical records are long-lived, the cumulative event counts are expected to reach hundreds of millions. The commitment scheme must be able to handle those sizes in daily update commitments. Computing and posting of the commitments is a background process, so a computation time in minutes is entirely acceptable, and even tens of minutes is still tolerable.

Property 14. Extraction of records of one patient is an on-demand activity and should run in a few seconds preferably, or a few tens of seconds at most.

Property 15. Verification of records of a patient delivered to either the patient or to a specialist the patient was referred to should ideally run in a few seconds. If that is not feasible, a fallback solution of displaying the records themselves immediately and having the verification run in the background in a few tens of seconds, or perhaps even in a few minutes, would also be tolerable.

Property 16. Reporting is a pre-planned activity expected to run on a monthly or even quarterly schedule, so for extraction of larger subsets of records and computing aggregates over them, computation time in tens of minutes per report is acceptable. However, generating a single report will typically involve multiple steps of filtering various subsets of events, joining different subsets, and computing aggregates on those subsets. Therefore, each single step should run in a few minutes.

Property 17. Also report verification is a pre-planned activity expected to run on a monthly or quarterly schedule, so performance expectations are roughly the same as for report generation.

Chapter 5

Validation Criteria—Use Case 3: University Diploma Record Ledger

The Diploma Use Case has been analysed in project deliverables [PRI18, PRI20]; a description of the validation approaches that will be used has been presented in the current deliverable in Section 2.3. The system development by the project will be used to assess the use case requirements.

In the interest of readability, we repeat some definitions given in the earlier deliverables. In the context of the diplomas use case, there are three distinct entities, namely: *Issuers*, *Holders*, and *Verifiers* (these concepts are in line with the World Wide Web Consortium's "Verifiable Credentials Data Model" [Con]). An Issuer is an organization (e.g., a university) that issues *titles*. Holders are persons that have titles and may want to present them to an interested entity. This entity can be an organization or a person that wishes to verify a title, i.e., a Verifier. To do so, the Verifier must contact the corresponding Issuer, who in turn, will provide the evidence needed.

5.1 Functional validation criteria

Property 1. Titles are private data, meaning that they should not be accessible to anyone else except for the ISSUER.

Property 2. The data inside the titles are personal data belonging to their HOLDERs. The HOLDERs should be able to control who and when gains access to their titles. In addition, HOLDERs should control which parts of the verifiable credentials can be presented to a VERIFIER (e.g., full transcript or bare title).

Property 3. All entities should be held accountable for all their actions. For instance, ISSUERs are not expected to issue forged titles. Furthermore, VERIFIERs should accept only valid credentials, and be held responsible for any proof they leak.

Property 4. An honest external auditor should be able to inspect all the recorded transactions.

Property 5. A HOLDER should be able to release any information they choose to any VERIFIER they choose. The system should protect against a corrupt VERIFIER extracting additional information.

Property 6. The system will not be able to tell if a qualification is forged or not, but it should guarantee that no forged qualification can be both acceptable and hidden.

Property 7. Proofs from the ISSUERs to the VERIFIERs should not be leaked, but the system should not prevent proof leaks. However, the system should guarantee that if a leak happens, the VERIFIER must always be held accountable. There will not be anonymous leaks.

Property 8. An auditor is special because they can request opening of arbitrary commitments.

Property 9. Although initially we want to commit whole diplomas, it is also desirable to be able to commit information at finer granularity (i.e., grades for specific courses).

Property 10. As awarded diplomas are committed to the blockchain, they can never be removed from there. If an award is revoked for some reason, it must not be possible for the Issuer to provide certificates for it from the point of revocation onwards. One way to guarantee that is to require revocations to be entered into the blockchain as well.

5.2 Infrastructure validation criteria

Property 11. A registration service for the VERIFIERs may be required.

Property 12. The cryptographic proofs must be persistent. Persistent storage should be guaranteed by the DLT, even if actual data are not stored on chain.

Property 13. It must be possible to query the blockchain to produce lists of awarded titles and proofs meeting specific criteria; the list will then be available for auditing purposes.

Property 14. The actual diplomas will not need to be entered to the blockchain, only encrypted hashes of them. Their individual hashes will be of small size. Apart from diplomas, cryptographic proofs will also be stored on the blockchain.

Property 15. Diplomas must be committed to the blockchain before the ISSUER can produce certifications. If we want to enter past diplomas in the blockchain, then a large number of diplomas, possibly thousands or tens of thousands, must be entered en masse. A similar, although less severe situation, will appear if we want to enter all diplomas issued at a certain point in time (e.g., end of semester). This requires that either transactions can be entered in the blockchain in large volumes, or that we can work with suitable data structures incorporating hashes of individual hashes.

Property 16. The readers of the blockchain can verify a transaction and hold corresponding actors accountable, i.e., we require *open verifiability*. Open verifiability means that blockchain readers can collectively ensure the validity of transactions and uphold the accountability of transaction actors.

Property 17. Designated auditors can verify a transaction after retrieving necessary secrets from actors, i.e., we require *forensic verifiability*. In an actual deployment, the retrieval may be enforceable by technological means (e.g., a key escrow), or by legal means (imposed by legal procedures).

Property 18. Open and private communication channels should be connected via the protocl used for titles issuing and verification. The blockchain itself is considered an open channel because a broad set of actors can read it. However, an open channel hinders both the private awarding of titles and the designation of verifier proofs. An open channel cannot hide secrets, while a private channel cannot be verified. In the case of awarding, the issuer must commit to a secret value for the title and communicate that value to the holder. This requires an open channel for the commitment and a private channel for the transfer of the title. In the case of proving, assuming that an issuer is reputable and hence considered to act honestly, a proof in the open channel that convinces the specified verifier should convince any other blockchain reader, because they all have the same information. Transferring the proof through a private channel between issuer and verifier supports designation but destroys (open) verifiability. A trade-off is to embed and commit the private channels to the open blockchain as a series of encrypted transactions. This enables proofs going through private channels to allow for forensic verifiability.

5.2.1 Performance validation criteria

Property 19. Speed (of network consensus system): Multiple transactions per second.

Property 20. Speed (of block confirmation times, i.e., finality): In minutes.

Property 21. Scalability: Not more than a few dozens of nodes.

Property 22. Confidentiality: Use of privacy-preserving cryptographic protocols.

Property 23. Identity services: Required, for all participants in a certification workflow. This requires: Issuers, Holders, Verifiers.

Property 24. Off-chain data storage: Diplomas will not be stored in the ledger; only encrypted hashes of data will be included in the ledger.

Chapter 6

Validation Criteria—Use Case 4: Cardano Update System

6.1 Requirements driven validation criteria

One of the most important goals of the Cardano blockchain is to enable *decentralized governance*. This is, the decisions for the evolution of the system should be made collectively by the Cardano stakeholders (a.k.a. the Cardano community) and the costs of evolving the system (e.g research and development) should also be paid by them, via a Treasury system that is collectively funded. The Cardano decentralized software update system developed in the context of PRIViLEDGE is undoubtedly a center piece in this decentralized governance puzzle.

Clearly, the design of the software update system is requirements-driven. The requirements are ultimately posed by the Cardano community and are eventually conveyed to our team via various internal (to IOHK) stakeholders; for example the Cardano Product Management. In deliverable *D1.1 Requirements and Interface Design* [PRI18], we have described an initial set of requirements. Since then, this set has been discussed internally, has matured and been enhanced and currently has been aggregated to eight *vision statements* that capture the essence of the to-be update system. We briefly outline these high-level requirements next. These will be our main driver for defining the validation criteria[1] in the rest of this section.

6.1.1 The Cardano software update system vision statements

Open Participation Enabled Anyone who can submit a common transaction should be able to submit an update proposal and vote for an update proposal.

Decentralized Decision-Making Enabled All major decisions typically made by central authorities (e.g., code maintainer, software owner etc.) should be made collectively by the community via a voting mechanism: a) What proposal will move forward? b) Do we accept the submitted implementation? and c) will the changes be activated?

Protocol-Driven The update mechanism should be an on-chain protocol driven process: Not based on informal discussions (social consensus[2]), but based on a protocol with specific security guarantees; a protocol that will leave behind an immutable (on-chain) trace of events

Transparent and Auditable Anybody should be able to answer *when*, *why* and *how* the system has evolved the way it has. The evolution history of the system should be open to everyone and form a globally-consistent tamper-evident public release log.

[1] In the rest of this text we will use the terms ”validation criteria” and ”properties” interchangeably.

[2] Although this is also a very useful preparatory step to utilize along with the update protocol.

Secure Activation Enabled The activation of changes on the blockchain system should be resilient to chain-splits and ensure a secure transition from the previous version of the consensus protocol to the new one. We need to formally define what is a *secure activation* and propose a protocol that enables such an activation.

Performant and Scalable The update mechanism should have a minimal impact on the transaction throughput and size of the underlying blockchain. Moreover, the update mechanism should be scalable to thousands (or even millions) of participants.

Metadata-Driven Update Logic All updates are not the same (criticality, impact on the system, time to deploy etc.). We should enable a metadata-driven update policy (voting periods length, activation priority, deployment window etc.). This update policy should efficiently handle priorities, version dependencies, update conflicts and emergencies, in order the system to always be at a consistent state.

6.2 An overview of the decentralized software update lLifecycle

In order to validate our update system, we need to closely follow a software update through out its whole lifecycle; from the very first phase, where it is born as an idea, to the last phase, where the changes are activated on the blockchain. In deliverable *D4.1 Report on Architecture of Secure Ledger Systems* [PRI19], we have provided a detailed description of all the steps in the said lifecycle. In this section, we only provide a brief overview (see figure 6.1), in order to set the context for the reader's convenience.

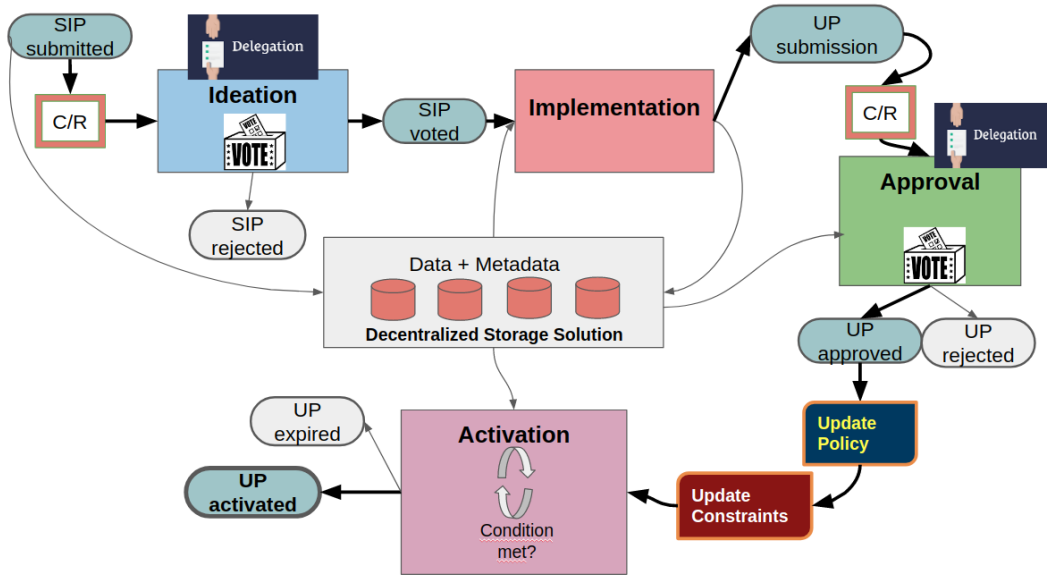

Figure 6.1: The Decentralized Software Update Lifecycle.

A software update starts its life as a *System Improvement Proposal (SIP)*, which is submitted to the blockchain. This submission takes place with a fee-based transaction, which carries the proposal in its metadata (*transaction payload*). A SIP is a structured document describing an update proposal. This submission goes through a typical *commit-reveal scheme*, in order to ensure the rightful authorship of the proposal. During the reveal phase, the actual proposal and the author of the proposal are revealed by submission on the blockchain and the update protocol checks, if the revealed info matches to the commitment submitted earlier.

As we have mentioned, the reason why the update proposal is issued on the blockchain in two stages, with a commitment first and a revelation later, is so that the rightful author can claim authorship of a particular Update Proposal. If the update proposal hash were to be revealed immediately, a dishonest party could create a competing transaction, signing the same proposal with their own key to claim authorship. If multiple valid

revelations pertaining to the same commitment are received, only the first is included in the blockchain. Only the first such transaction is necessary, as any two such revelations will necessary contain the same data due to it having been committed in the commitment and by the collision resistance property of the underlying hash function.

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 the a central server maintained by the Cardano Foundation. These are abstracted in figure 6.1 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 voting and activation. Naturally, an unavailable 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).

Once the SIP is revealed, a voting period starts for this SIP; the software update has entered the *Ideation phase*. The duration of the voting period is metadata-driven, i.e., defined in the SIP metadata[3].The purpose of the Ideation phase is to help the community decide which SIP will move forward to the next phase. Votes are also fee-based transactions with a special payload. Anyone who owns enough stake to make a transaction can potentially vote. Votes count proportionally to the owned stake. Votes are accepted only within the voting period and multiple proposals can compete in parallel. The outcome (verdict) of the voting process for a specific proposal might be: a) *Accepted*, when stake in favor is above a threshold, b) *Rejected*, when stake against is above a threshold, c) *No Quorum*, when stake abstaining is above a threshold; this outcome leads to revoting, d) *No Majority*, when none of the previous three results occurs, which also leads to revoting and e) *Expired*, when after the maximum number of revoting periods has been reached and still the proposal was neither been accepted or rejected. Finally, note that for the voting, we allow delegation of a stakeholder's voting right to an *expert*. For the scope of our prototype implementation we have assumed delegation as an *out-of-band solution*[4] and provided direct voting power to stakeholders. This is basically for two reasons: a) to reduce risks in the prototype implementation (especially of the Cardano integration part) and b) to defer introducing delegation to experts until a proper game-theoretic analysis of the expert's incentives has been completed.

The *Implementation* phase is an off-chain process, where an approved SIP is implemented. It ends with the submission of the implementation, which we formally call *Update Proposal (UP)*[5], to the blockchain. Once this UP is revealed, the we have entered the *Approval* phase. This is the phase where the community is called to approve a submitted implementation. The voting process and the technical details are similar to the Ideation phase. Once the UP is approved it enters the *Activation* phase depicted in figure 6.2.

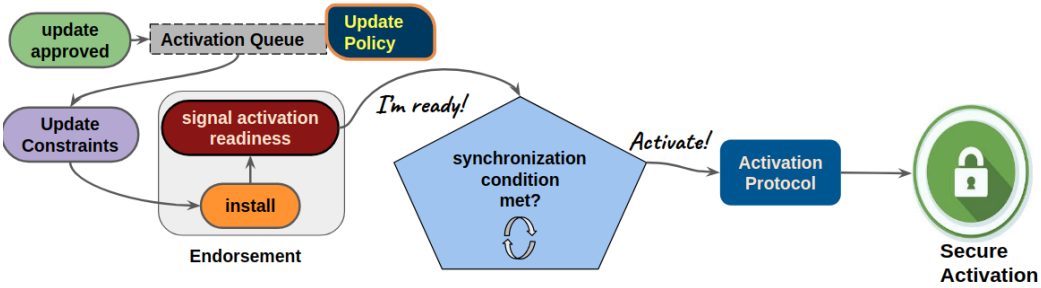

Figure 6.2: The Activation Phase.

An approved update proposal enters the activation phase and is placed in the *activation queue*. At this point,

[3]There is an upper limit for the voting period duration in the implementation, to prevent DoS attacks.
[4]This means that a voter can still ask for the help of an expert but not in the context of an on-chain protocol.
[5]We will use the terms *UP* and *Implementation* interchangeably

the *update constraints* of the proposal will be evaluated. A proposal satisfies the update constraints when:

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

If a proposal satisfies its update constraints it enters the *Endorsement Period*. This is the period where the *block issuers* download and install the update and *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*, the activation gives the green light to the *activation protocol* to run. The activation protocol ensures the secure activation, i.e., the secure transfer from the old ledger to the upgraded ledger, based on our formal definition of activation security and corresponding security proofs [CKKZ20]. In a nutshell, *secure activation* means, the secure transition from the old ledger (L1) to the new ledger (L2) in a way where:

- L2 enjoys liveness
- L2 enjoys consistency
- L2 has L1 as a prefix

Finally, note that 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.

6.3 Open participation

The *open participation requirement* states that anyone who can submit a common transaction should be able to submit an *update payload*[6], provided that such payload is valid according to the protocol. The update payload refers to any of the following data:

- SIP commitments, revelations, or votes; or
- implementation commitments, or revelations (see about commit-reveal scheme in section 6.2), votes, or endorsements (i.e., upgrade readiness signals).

We consider that the system satisfies the open participation requirement when the following validation criteria are met:

- Authorship of proposals should be preserved.
- Valid proposals' commitments are not rejected.
- Valid proposals' revelations are not rejected.
- Valid proposals' votes are not rejected.

[6] An *update payload* refers to an update proposal(SIP/UP), a vote for an update proposal, a signal for upgrade readiness and in general any event that relates to the update protocol and is stored in the transaction's metadata called *payload*.

– Valid implementation endorsements are not rejected.

These validation criteria ensure that any update event (i.e., update payload) can be submitted to the blockchain, without it being rejected, as long as: a) it is valid and b) it has payed the necessary fees. They are detailed in the next section, along with the *validity conditions* for each type of update payload. Therefore conditioned to these constraints anyone can participate. It is important to note that in these validation criteria, we will not be mentioning fees. The fees are checked by the UTxO layer. We assume that the payload passed to the update system will be part of a transaction that spends at least the fees that the update payload must pay (which is determined by the UTxO layer[7].)

6.3.1 Authorship of proposals should be preserved

The protocol should prevent proposal authorship of proposals from being stolen. To this end, the protocol enforces a commit-reveal scheme for proposals. This validation criterion validates that the commit-reveal scheme indeed works correctly.

Proposals can be revealed only after a corresponding commitment is *stable* on the chain. A commitment should contain the commitment payload, as well as the verification key (i.e., public key) of the author, say vk, that submitted the commitment. The commitment payload should be of the form: $H(salt||vk||H(proposal))$, i.e., A salted commitment to the Update Proposal (either a SIP, or a UP) hash as well as the public key of the issuer, where H is a cryptographic hash function.

If the verification key of the commitment author is part of the information stored in the commitment, then the protocol can ensure that the commitment author and the revelation authors are the same. To see why this is the case, consider the following analysis. A commitment submission comprises:

– the commitment payload, which is of the form $H(salt_0||vk_0||H(proposal_0))$

– the verification key of the author, say vk_1

– a signature of the payload by vk_1

The corresponding revelation submission comprises:

– $proposal_1$ metadata

– $salt_1$

– vk_2

We assume that from the submitted proposal metadata we can retrieve the actual proposal. So if we compute $H(salt_1||vk_2||H(proposal_1))$ and this is equal to $H(salt_0||vk_0||H(proposal_0)$, then it must be the case that $vk_0 = vk_2$, and since we also checked that $vk_1 = vk_2$, we have that $vk_0 = vk_1$. Of course we assume that H is a collision resistant hash function.

Therefore for this validation criterion we have the following *validity condition*:

$$H(salt_1||vk_2||H(proposal_1)) = H(salt_0||vk_0||H(proposal_0) \wedge vk_1 = vk_2 \tag{6.1}$$

6.3.2 Valid proposal commitments are not rejected

In this validation criterion, we ensure that no proposal (SIP or Implementation) commitment can be rejected by the update system, unless the following validity condition is violated:

– the commitment signature verifies and

[7] Our prototype will be "plugged into" the Cardano ledger layer. This layer includes an UTxO layer that handles all this "UTxO logic"

– the commitment has not been already submitted.

If both of the conditions above hold, then the commitment should be accepted. If any of these two conditions is violated, then the commitment should be rejected. Note that in particular this means that the system allows an implementation commitment to be submitted at any time, as long as they are valid. This is true even in the case, where an implementation commitment does not match a previously approved SIP. Indeed, commitments are basically hashes so the system does not have the possibility of performing any semantic check on the commitment.

In summary, in all generated traces, and for all proposals' (SIPs/UPs) commitments submitted to the blockchain, no commitment should be rejected, unless the said validity condition is violated.

6.3.3 Valid proposal revelations are not rejected

In this validation criterion, we ensure that no proposal (SIP or Implementation) revelation can be rejected by the update system, unless the following validity condition is violated:

– the revelation matches a previously submitted commitment (see condition 6.1), and

– the matched commitment is stable on the chain

If both of the conditions above hold, then the revelation is valid and should be accepted. If any of these two conditions is violated then the revelation should be rejected.

In summary, in all generated traces, and for all proposals' (SIPs/UPs) revelations submitted to the blockchain, no revelation should be rejected, unless the said validity condition is violated.

6.3.4 Valid proposal votes are not rejected

In this validation criterion, we ensure that no vote for a proposal (SIP or Implementation) can be rejected by the update system, unless the following validity condition is violated:

– the signature of the vote is valid and

– the vote refers to a proposal known to the system and

– the voting period for the proposal is open

If all of the conditions above hold, then the vote is valid and should be accepted. If any of these conditions is violated, then the vote must be rejected.

In summary, in all generated traces, and for all proposals' (SIPs/UPs) votes submitted to the blockchain, no votes should be rejected, unless the said validity condition is violated.

6.3.5 Valid implementation endorsements are not rejected

In section 6.2, we have described that an *endorsement* is a signal of upgrade readiness issued by the block producers. In this validation criterion, we ensure that no endorsement for an Implementation can be rejected by the update system, unless the following validity condition is violated:

– the signature of the endorsement is valid and

– the endorsement refers to the *candidate proposal* (see next) and

– the endorsement period for the specific proposal is open.

If all of the conditions above hold, then the endorsement is valid and should be accepted. If any of these conditions is violated, then the endorsement must be rejected. Note that an approved proposal that has entered the activation queue (see figure 6.2), reaches at the top of the queue, if it has the *highest priority* (i.e., lowest version) among all the proposals in the queue. If the proposal that is at the top of the queue also supersedes the *current version* of the protocol, then immediately earns the right to be endorsed; we call the proposal being endorsed the *candidate proposal*. Therefore, a proposal in the activation queue becomes the candidate proposal if the following condition holds:

- It has the lowest version (highest priority) among all other proposals in the queue and
- it supersedes the current version.

A note on the endorsement of the candidate proposal. 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* (tally-slot). 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.

In summary, in all generated traces, and for all Implementations' endorsements submitted to the blockchain, no endorsements should be rejected, unless the said validity condition is violated.

6.4 Decentralized decision making

The *decentralized decision making* requirement states that the community should, via a voting mechanism, decide which proposals move to the next phase, and which proposals get activated. The voting power of the participants should be proportional to the stake they own.

We consider that the prototype satisfies the decentralized decision making requirement when the following validation criteria are met:

- votes are correctly tallied
- voting outcomes are honored by the protocol

Essentially, these validation criteria ensure that the decentralized decision making process in-place works correctly. This, on the one hand, translates to having a correct tally computation and on the other, the result of this process triggers the correct state transition of a software update. In particular, the transition of a proposal to a next phase in the lifecycle must always be backed up, by an approval outcome of the tally process.

The following sections provide a description of these validation criteria and define the corresponding validity conditions.

6.4.1 Votes are correctly tallied

The verdict (i.e., voting process outcome) that the system reports should coincide with the verdict we compute by looking at the trace. This means that for each verdict on a proposal the validity condition is the following:

- The verdict we compute at the tally slot should coincide with the verdict that the system reports.
- The verdict should be reached only at the right slot (tally slot). Here, the "right slot" corresponds to the end of the voting period for the specific proposal, which is determined by:
  - the slot at which the revelation occurred,
  - the voting period duration specified in the proposal metadata and
  - the maximum number of re-voting periods a proposal can go through (this is a protocol parameter).
- The verdict should be based on the stake distribution at the tally slot.

Note that, the vote tallying done in the tests on the trace, should use the same voting period duration as the one specified in the proposal's metadata. This means that if the system does not honor this duration, tests should be able to find a discrepancy between the test and systems' results.

6.4.2 Endorsements are correctly tallied

Similarly to the previous validation criterion, the decision on whether to activate an update proposal should coincide with the decision we arrive at by looking at the trace. This means that for each update proposal being endorsed (i.e., candidate proposal) the following validity condition should hold for the endorsement verdict to be correct:

- The decision we compute at the tally slot should coincide with the decision that the system took (activate or cancel).
- The decision should take place only at the right slot, which is the *tally slot* (metadata defined).
- The decision should be based on the stake distribution at the tally slot.

Note that, the endorsements tallying done in the tests on the trace, should use the same endorsement period duration as the one specified in the proposal's metadata. This means that if the system does not honor this duration, tests should be able to find a discrepancy between the test and systems' results.

6.4.3 Decisions are honored by the protocol

In this validation criterion, we want to ensure that only approved proposals are allowed to move to the next phase in the software update lifecycle. This means that the update payload for a given phase, for a given proposal is allowed only if the proposal was approved in the previous phase. More specifically we identify the following validity conditions:

- Implementations can be revealed only if the corresponding SIP was approved in the Ideation phase (see figure 6.1).
- Implementations can be queued, or start being endorsed only if they have been approved in the Approval phase (see figure 6.1).
- Implementations can be activated only if they have enough endorsements.

In summary, in all generated traces, and for all Implementations that have reached one of the states: Revealed, Stably Revealed, Queued, Being Endorsed, or Activated there should be a previous approval, which can be computed from the data on the trace that justifies the transition to this state.

6.5 Protocol driven

This requirement states that the update mechanism must be an on-chain protocol driven process, that is enforced automatically by the system. The update mechanism should not be based on informal discussions (social consensus). Furthermore, the protocol should leave behind an immutable (on-chain) trace of events.

To honor this requirement, we have to incorporate our update protocol into a blockchain. In particular, we will integrate our update protocol into the Cardano blockchain. Our update events will be stored in the transactions' metadata (a.k.a, transaction payload). All transactions are stored in the ledger. The ledger layer does not know how to handle the update payload and thus it must interact with the update system that must process the update payload and then update the ledger state accordingly. The validation criteria in this section, try to validate the integration of the update protocol to the Cardano blockchain system. To this end, we define the following validation criteria that we detail next.

- The update system runs on Cardano; and in particular, it is integrated with the Cardano ledger layer.
- The update protocol works as expected in Cardano
- Update events are eventually stored in the Cardano blockchain

6.5.1 The update system runs on Cardano

The update system should be integrated with Cardano. An new enhanced version of the Cardano node (ledger layer) software will be produced that will include the update protocol. This will be deployed on a *testnet*.

In summary, on this testnet, we will validate that the integration works by submitting transactions with update payload and observing the system changing versions, i.e., being updated by the update protocol.

6.5.2 The update protocol works as expected in Cardano

To validate that the integration to Cardano is successful we will deploy the integrated node on a testnet and run an end-to-end update from version 1 to version 2. In addition, we plan to validate the integrated version of our update protocol against more detailed scenarios. Therefore, from the existing set of unit tests that we have run on the single mode version, we will choose a sufficient subset to be executed also on the testnet (manually, or automatically based on the time constraints). Following is an indicative list of such test scenarios that we could utilize:

- The protocol version is changed once.
- The protocol version is changed twice.
- An update is preempted by another with higher priority.
- An old proposal gets canceled by a new one with the same version.
- A proposal with lower priority (higher version) gets queued.
- A proposal that does not meet its dependencies is queued, until the *current version* of the system becomes compatible with it.
- A candidate proposal without enough endorsements is not adopted.
- Endorsements on non-candidate proposals are not allowed.

6.5.3 Update events are eventually stored in the Cardano blockchain

When the update mechanism will be tested on a testnet, various update events will be generated (proposals, votes, endorsements etc.). As we have seen these comprise the so-called update payload. We need to ensure that this payload is stored in the Cardano blockchain successfully. Therefore, this validation criterion will validate that the update payload used in the scenarios of the validation criterion 6.5.2 is stored in the blockchain maintained by the Cardano node software.

6.5.4 The proposal state transitions should be specified and verified

The states of a proposal and the transitions between them should be specified. This will constitute a *state-machine model* of the system that will precisely determine what states the protocol enforces a proposal to go through. Furthermore, there should be tests that demonstrate that the system only takes the transitions that are specified.

In figure 6.3, we depict an example of this state machine model, which defines the valid state transitions for a proposal. At the bottom we see the trace storing all the update events generated for all proposals. We can say that the events of all proposals are *multiplexed* on a single trace. Therefore, by demultiplexing (i.e., extracting) the state transition events and the states for a specific proposal, we can create a state machine model for each proposal. This is depicted in the figure.We see how the various state transitions for a specific proposal take place and how these are demultiplexed from the total trace of events. Having isolated the individual proposal transitions, it is relatively easy to apply the appropriate conditions and test the validity of the state transitions in each case.

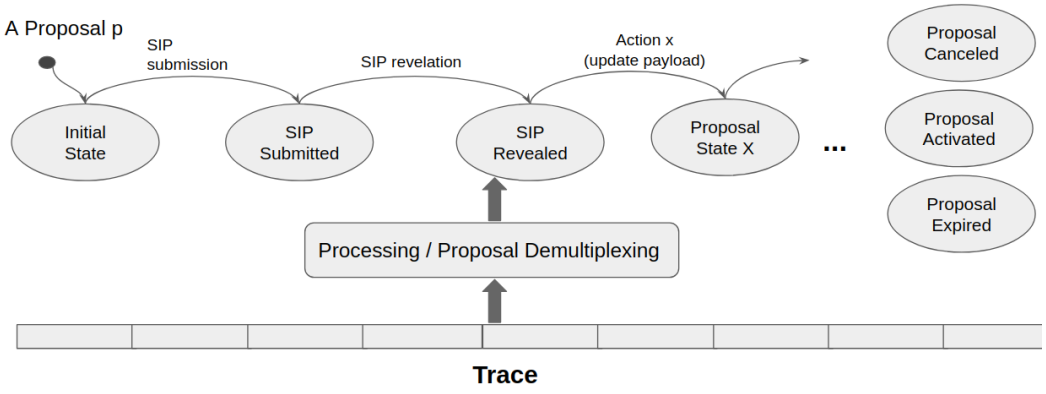

Figure 6.3: Valid State Transitions in the Software Update Lifecycle.

6.6 Transparent and auditable

This requirement states that anybody should be able to answer *when*, *why*, and *how* the system has evolved the way it has. The evolution history of the system should be open to everyone and form a *globally-consistent tamper-evident public release log*.

The first thing that we need to validate in this case is that the update events are eventually stored in the blockchain. This will be the result of the integration of the update system to Cardano. This validation criterion is covered in 6.5.3, which checks that the update protocol is transparent.

The second thing that we need to validate is auditability. To this end, we must ensure that all the reported outcomes from the update system can eventually be computed from the data stored on chain. This validation will be covered from criteria 6.4.1 and 6.4.2. Note also that for all the on-chain data (i.e.,update payload), auditability is guaranteed by our update mechanism, however for the actual proposals (e.g., implementation code) that are

stored off-chain, auditability is ensured only under the assumption that the utilized P2P storage system guarantees data availability.

6.7 Secure activation

For this requirement we need to define what is a *secure activation* and validate that our protocol enables such an activation. The formal definition of secure activation on blockchain systems can be found in our paper [CKKZ20]. In summary, in order for an activation of changes, which will enable the transition from a ledger L_1 to a ledger L_2, to be secure the following properties must hold:

- The updated ledger L_2 enjoys the *liveness* property
- The updated ledger L_2 enjoys the *consistency* property
- L_2 includes L_1 as a prefix.

The first two properties ensure that L_2 is indeed a secure ledger. This means that if we were to start running L_2 from scratch then these two properties would be enough to guarantee the security of the blockchain protocol. The main prerequisite for this to be true, i.e., for L_2 to enjoy liveness and consistency and thus to be secure, is to ensure that the *security assumption* A_2 of the L_2 protocol holds. In our update protocol, this is guaranteed via the *adoption threshold*. The definition of the adoption threshold embeds "the logic" that enough honest parties must have signaled upgrade readiness, before we allow the activation of changes to take place. "Enough" here means with respect to the security assumption A_2.

The third property has to do with the fact that L_2 is not a new ledger running from scratch, but it should be a valid *continuation* of the previous ledger L_1. This very important from a security perspective since it ensures that all fund transfers made in the past are still valid and the state of ledger L_1 will not just get wiped out at the hard fork point to L_2. Thus we must guarantee that during the transition from L_1 to L_2, history is preserved. The secure activation protocol that we use in order to make a secure transition from L_1 to L_2, will be described in the forthcoming deliverable D3.3 "Revision of Extended Core Protocols".

All in all, in order to ensure the said secure activation properties, we define the following validation criteria, which we detail next.

- The adoption threshold is honored
- The update system preserves history across hard fork boundaries

6.7.1 The adoption threshold is honored

The adoption threshold is the threshold used during the endorsement period (see section 6.2). When the stake endorsing a candidate proposal is above this threshold, then we can be sure that enough honest parties have signaled upgrade readiness and thus the security assumption of the upgraded ledger will hold upon activation. This validation requirement is covered by the validation criterion 6.4.3. Indeed, in this criterion we validate that all proposals that have reached the state of *Activated*, they must have been previously approved. This means that enough endorsements with respect to the adoption threshold have been submitted by the block issuers.

In summary, in all generated traces and for all candidate proposals that have reached the status of *Activated*, there should be endorsement stake previously submitted that exceeds the adoption threshold. This result must be able to be computed from the data on the trace and thus justify the transition of the candidate proposal to the activation state.

6.7.2 The update system preserves history across hard fork boundaries

In this criterion we need to validate that after the update system gives the green light for an activation - based on the adoption threshold discussed before- the ledger stays intact, before and after the hard fork slot. We should observe ledger 1 (L_1) before the hard fork slot and ledger 2 (L_2) after. This criterion requires the integration with the Cardano ledger layer. In this integrated version of our update protocol we will observe the ledger transitioning from the previous version to the new one. It is essentially the validation criterion 6.5.1.

In summary, on the integrated (with Cardano) version of the update system, we provide appropriate update payload that will update the ledger from L_1 (version 1) to L_2 (version 2). We should be able to observe this transition at the hard fork slot (epoch boundary) and validate that the ledger forms a continuous and valid historical record of transactions regardless of the version update. To this end, we must show that the first block of the new version, will eventually stabilize and also that it "points" to a stable block of the previous protocol version. The secure activation protocol that we use in order to make a secure transition from the previous version to the new one, will be described in the forthcoming deliverable D3.3 "Revision of Extended Core Protocols".

6.8 Performant and scalable

This requirement is about showing that the update mechanism should not impact negatively the performance of the underlying blockchain. For evaluating the performance of the blockchain we utilize the following measures:

- Transaction bytes per second (TBPS)
- Processing time and
- Memory usage

Finally, the update mechanism must be scalable. This means that as the number of participants in the update protocol increases and eventually reaches the scale of a global blockchain system, the update mechanism resource consumption should scale up efficiently, i.e., ideally linearly to the number of participants. To validate all the above we define the following validation criteria:

- Transaction throughput is not significantly affected.
- Low-impact on processing time.
- Low-impact on memory usage.
- The system should scale.

We detail each one next.

6.8.1 Transaction throughput is not significantly affected

An important measurement of a blockchain systems performance is the number of *transaction bytes per-second (TBPS)* it can sustain. Unlike the more commonly used metric, transactions per-second, this number is not dependent on the chosen size of a transaction (which would allow to manipulate it at will by choosing different transactions sizes). Running an update mechanism on the blockchain should not result in a substantial performance degradation. Therefore, in this validation criterion, we estimate the impact on performance that the proposed update mechanism will have on a systems TBPS. Our estimations should be based on *worst case scenarios*, which determine upper bounds for the performance impact of the update mechanism.

The amount of data required by the update payload associated to a proposal should be have little impact on the blockchain protocol. A worst case analysis should be made that shows the TBPS impact of a proposal as a function of the following variables:

- Number of participants
- Number of voting rounds
- Duration of the update process

Different systems will have different requirements on the maximum TBPS impact that they can tolerate. The aforementioned function should provide an easy way to determine whether they can accommodate the additional usage requirements of the update protocol. In practice, we will validate what percent of the available TBPS throughput is consumed by the update mechanism and how this percent scales as the number of participants increases.

6.8.2 Low-impact on processing time

The processing time required by a node that runs the update protocol should not be significantly impacted by the it. The bottlenecks of the protocol should be identified, and for each one of them:

- an asymptotic time complexity analysis should be made, which should provide the resource requirements in terms of processing time.
- micro-benchmarks should be run that show the CPU time required for the most CPU-intensive parts of the update protocol, as a function of the input size. The benchmarks results should show that the processing time will not affect overall processing time of a node, assuming a reasonable input size.

6.8.3 Low-impact on memory usage

The memory usage requirements to store the update payload should be analyzed. As with the previous validation criterion, micro-benchmarks should be executed that show the consumption of memory by the update protocol. Memory usage should be as small as possible (given the amount of data being manipulated by the protocol).

6.8.4 The system should scale

The resource consumption required by the update protocol should scale up linearly with the number of participants. The results obtained in the previous requirements can be used to demonstrate a linear relation between number of participants and resource consumption (CPU, memory, impact in TBPS).

6.9 Metadata driven

Metadata are very important in a software updates system, because they provide the context of a software update. Without metadata all updates would look the same. In our update protocol, the proposals should be able to specify as part of its metadata the following information:

- voting period length
- dependencies with other versions
- activation priority
- deployment window

and the protocol should make sure that these are honored as part of the update logic implemented. The voting period duration corresponds to the voting processes in the Ideation and Approval phases (see figure 6.1) in the lifecycle of a software update and is attached to the submission of SIPs and UPs respectively. It is the number

of slots during which votes are accepted for a specific proposal. Dependencies refer to the protocol version upon which an update is based. It is the version that a specific update is compatible with and can be safely deployed on (i.e., the version that an update supersedes). Priorities are declared in the proposal's metadata in the form of software versions. Since the update system guarantees that the protocol versions can only strictly increase, the lower the version is, the higher is the priority of a proposal. Ultimately, the community decides (through the available voting processes), which proposal will move forward, in the case of proposals with the same version (i.e., priority). Finally the deployment window, which is also called *safety lag*, is defined in number of epochs and it is a safe time window that allows sufficient time to stakeholders to download the update and upgrade their software.

The following validation criteria will validate if the update protocol indeed honors the proposal's metadata.

- The voting period length is honored
- Version dependencies are honored
- Priorities are honored
- The deployment window is honored

We detail these validation criteria next:

6.9.1 The voting period length is honored

The update prototype should provide a mechanism for specifying the duration of the voting period, and ensure that a proposal is granted the voting period duration it specifies. Validation criterion 6.4.1 requires that the vote result that computed over the generated trace of events is the same as the result reported by the system. In this computation, we will use the voting period defined in the proposal's metadata. If the update system does not honor the voting duration in the metadata, our property-tests should find a case in which comparison will fail. Therefore this validation criteria is subsumed in validation criteria 6.4.1.

6.9.2 Version dependencies are honored

The update prototype should provide a mechanism for specifying version dependencies, and ensure that a protocol update should always be applied to a protocol version that it depends on. This validation criterion will ensure that every update is applied only on the version that (based on its metadata) it supersedes. As we mentioned earlier, in our prototype every software update supersedes one and only one protocol version.

In summary, for all trace of events generated, and for all updates that have been activated superseding a version X, X should match the dependency version (i.e., to be superseded) specified in the corresponding metadata of the said update.

6.9.3 Priorities are honored

The update protocol should provide a way for update proposals to specify their priority, and once updates get enough endorsements, they should be activated according to their priority. As we mentioned earlier, priorities are defined by means of software versions. The lower the version, the higher the priority. Protocol versions increase monotonically. This validation criterion will ensure that updates are activated based on their (metadata-defined) priorities, which are of course confirmed by the stakeholders approval.

In summary, for all generated traces of events and for all updates activated, two conditions must hold:

- The version that the system upgrades to, coincides with the version of the update defined in the metadata
- The versions of the system increase monotonically

6.9.4 The deployment window is honored

The update protocol should ensure that each proposal in the endorsement period (see section 6.2) is allowed to have the endorsement window that this proposal specifies. The endorsement window is specified in number of epochs and defines the time window during which participants can upgrade their software and signal (i.e., endorse) their upgrade readiness (see note on endorsements in section 6.3.5). As in the case of the validation criteria in section 6.9.1, the tests will compute (and thus double check) the results of an endorsement period. If there is a discrepancy between the proposal-specified deployment window and the deployment window allowed by the system, then the property tests should find a case in which the two endorsement-tally results do not match. Therefore, this validation criteria is subsumed in validation criteria 6.4.2.

6.10 Update logic consistency

The update system must implement an *update logic* that handles software dependencies, update priorities, software conflicts, and emergencies in a smooth way that always guarantees the *consistency* of the blockchain software.

In our prototype, every software update supersedes one and only one protocol version. This is the version that it *depends* on. Moreover, every update has a version. It is the to-be version of the protocol after the update is activated. Our update mechanism ensures that protocol versions increase strictly-monotonically. In our model, we allow only a single base version per-update proposal. A conflict is the case where two updates are applied on the same base version. The update mechanism implements an update logic that resolves such conflicts and allows only a single proposal to supersede the current version each time.

Priorities are defined by means of software versions. The lower the version, the higher the priority. In the case of multiple updates with the same priority, the community is the ultimate decision maker of which one will move forward to the next phase. 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.

Emergency handling is realized by means of *cancellations*. 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.

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. 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 following validation criterion ensures that update system implements a consistent update logic and is detailed next.

6.10.1 Consistent update logic

The update protocol should implement a consistent update logic. This means that it handles dependencies, conflicts and priorities in such a way that the blockchain software always reaches a consistent state. In particular, the protocol should allow a proposal to be activated if and only if:

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

In summary, for all traces generated and for all proposals activated the above validity conditions should hold. In addition, carefully selected unit tests will validate the correctness of the cancellation mechanism.

6.11 Validation criteria summary

In table 6.1, we summarize all the aforementioned validation criteria. In the leftmost column, we present the high-level requirements, or vision statements. Validation criteria are grouped in four different categories:

- **Liveness**: criteria that ensure that something ”good” eventually happens.
- **Safety**: criteria that ensure that nothing ”bad” happens.
- **Performance**: criteria that validate the performance and scalability of the update mechanism
- **Integration**: criteria that validate the integration with the Cardano blockchain system.

Table 6.1: Summary of all validation criteria.

Vision Statements	**Validation Criteria**			
	Liveness	Safety	Performance	Integration
Open Participation Enabled		6.3.1 Authorship of proposals should be preserved 6.3.2 Valid proposal commitments are not rejected 6.3.3 Valid proposal revelations are not rejected 6.3.4 Valid proposal votes are not rejected 6.3.5 Valid implementation endorsements are not rejected		
Decentralized Decision-Making Enabled	6.4.3 Decisions are honored by the protocol	6.4.1 Votes are correctly tallied 6.4.2 Endorsements are correctly tallied		
Protocol-Driven	6.5.3 Update events are eventually stored in the Cardano blockchain	6.5.4 The proposal state transitions should be specified and verified		6.5.1 The update system runs on Cardano 6.5.2 The update protocol works as expected in Cardano
Transparent and Auditable	6.5.3 Update events are eventually stored in the Cardano blockchain	6.4.1 Votes are correctly tallied 6.4.2 Endorsements are correctly tallied		
Secure Activation Enabled	6.7.2 The update system preserves history across hard fork boundaries	6.7.1 The adoption threshold is honored		

continued on next page

continued from previous page

Vision Statements	**Validation Criteria**			
	Liveness	Safety	Performance	Integration
Performant and Scalable			6.8.1 Transaction throughput is not significantly affected 6.8.2 Low-impact on processing time 6.8.3 Low-impact on memory usage 6.8.4 The system should scale	
Metadata-Driven Update Logic	6.10.1 Consistent update logic	6.9.1 The voting period length is honored 6.9.2 Version dependencies are honored 6.9.3 Priorities are honored 6.9.4 The deployment window is honored		

Chapter 7

Conclusions

In this deliverable, we have proposed methods and detailed criteria for the validation of the prototype implementations developed in the context of the four use cases of the PRIViLEDGE project. The four use cases differ significantly. They range from end-user facing decentralized applications such as e-voting with advanced privacy, zero-knowledge verification of sensitive medical data, privacy-preserving verification of university diploma ownership to blockchain infrastructure implementations that incorporate decentralized software updates to the Cardano proof-of-stake blockchain. This diversity has led to a differentiation in the validation methods, although there is a core set of common techniques that all use cases have proposed. In any case, the validation of DLT systems and applications is a real challenge, mainly because of the multiparty protocols operating in a decentralized setting with participants that potentially exhibit byzantine behavior. In this document, we have identified the most appropriate methods and tools for validating such systems. Moreover, we have proposed a rich set of validation criteria that aim to test the developed prototypes in all aspects of their use; from the cryptographic protocols employed, the end-user experience, to the performance and resource consumption. Detailed descriptions for each criterion have been provided that essentially set the scene for the forthcoming deliverable D1.3 ”Use Case Validation”, where the actual results of these tests will be presented.

Bibliography

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

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, 1981.

[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.

[CoE17] Council of Europe Committee of Ministers: Recommendation CM/Rec(2017) 51 of the committee of ministers to member states on standards for E-voting, 2017. `https://search.coe.int/cm/Pages/result_details.aspx?ObjectId=0900001680726f6f`. (13.09.2020).

[Con] World Wide Web Consortium. Verifiable credentials data model 1.0. Expressing verifiable information on the Web. `https://w3c.github.io/vc-data-model/#dfn-verifiable-credentials`. Accessed: 2018-02-02.

[Kuš20] Sergei Kuštšenko. Implementation of election bulletin board using HyperLedger Fabric. BSc. thesis, University of Tartu, 2020.

[NM90] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 249–256, New York, NY, 1990. Association for Computing Machinery.

[NM94] Jakob Nielsen and Robert L. Mack, editors. *Usability Inspection Methods*, chapter Heuristic Evaluation. John Wiley & Sons, New York, NY, 1994.

[PRI18] Requirements and Interface Design (D1.1). Technical report, PRIViLEDGE project, 2018.

[PRI19] First Report on Architecture of Secure Ledger Systems (D4.1). Technical report, PRIViLEDGE project, 2019.

[PRI20] Report on Architecture for Privacy-Preserving Applications on Ledgers (D4.2). Technical report, PRIViLEDGE project, 2020.

[Wik20] Wikipedia. Structural operational semantics. wikipedia.org, 2020. `https://en.wikipedia.org/wiki/Operational_semantics#Structural_operational_semantics`.