



DS-06-2017: Cybersecurity PPP: Cryptography


PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D1.3 – Use Case Validation

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

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

Lead contractor: Guardtime AS
Duration: 42 months

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

D1.3

Use Case Validation

Editor

Sven Heiberg (SCCEIV)

Contributors

Sven Heiberg (SCCEIV)
Sergei Kuštšenko (SCCEIV)
Ivo Kubjas (SCCEIV)
Jan Pentšuk (SCCEIV)
Ahto Truu (GT)
Panos Louridas (GRNET)
Foteinos Mergoupis-Anagnou (GRNET)
Athina Styliani Kleinaki (GRNET)
Vassilis Kefallinos (GRNET)
Kostas Papadimitriou (GRNET)
Maria Iliadi (GRNET)
Giorgos Korfiatis (GRNET)
Georgios Tsoukalas (GRNET)
Damian Nadales (IOResearch)
Nikos Karagiannidis (IOResearch)

Reviewers

Toon Segers (TUE)
Daniele Friolo (UNISA)

29th June 2021

Revision 1.0

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

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

©Copyright by the PRIViLEDGE Consortium

Executive Summary

In this deliverable, we report on the validation of the existing prototypes for verifiable online voting, health insurance, university diploma record management and Cardano update system based on the requirements [PRI18] and the specified validation criteria [PRI20a].

Contents

1	Introduction	1
2	Validation Results—Use Case 1: Verifiable Online Voting With Ledgers	2
2.1	Requirements validation	2
2.1.1	Validation of compatibility with EU recommendations	2
2.1.2	Validation of functional and non-functional requirements	3
2.2	Protocol validation	4
2.3	Implementation validation	7
2.3.1	Unit testing	7
2.3.2	End-to-end testing	8
	Testing setup phase	8
	Testing voting phase	10
	Testing bulletin board verification	11
	Testing tallying phase	12
2.3.3	Usability	14
2.3.4	Performance testing	16
2.4	Conclusions	17
3	Validation Results—Use Case 2: Distributed Ledger for Health Insurance	19
3.1	Functional validation	19
3.2	Security validation	20
3.3	Performance validation	21
3.4	Conclusions	22
4	Validation Results—Use Case 3: University Diploma Record Ledger	23
4.1	Functional Validation	23
4.2	Non-functional Validation	24
	Cryptographic Security	24
	Performance Metrics	25
4.3	Usability Validation	26
4.3.1	User Interface	26
	Design Toolkit	26
	e-diplomata Patterns	27
	Cross Application Heuristics	27
4.4	Interoperability Validation	30
4.5	Implementation Validation	31
4.6	Deployment Validation	32
4.7	Conclusions	32

5	Validation Results—Use Case 4: Cardano Update System	33
5.1	Property testing approach	33
5.1.1	Liveness properties	39
5.1.2	Coverage	39
5.2	Validation of the requirements	40
5.2.1	Open participation	40
	Authorship of proposals should be preserved	40
	Valid proposal commitments are not rejected	41
	Valid proposal revelations are not rejected	41
	Valid proposal votes are not rejected	42
	Valid implementation endorsements are not rejected	43
5.2.2	Decentralized decision making	44
	Votes are correctly tallied	44
	Endorsements are correctly tallied	44
	Decisions are honored by the protocol	44
5.2.3	Protocol driven	45
	The update system runs on Cardano	45
	The update protocol works as expected in Cardano	46
	Update events are eventually stored in the Cardano blockchain	46
	The proposal state transitions should be specified and verified	47
5.2.4	Transparent and auditable	47
5.2.5	Secure activation	47
	Secure Activation Protocols	47
	The adoption threshold is honored	49
	The update system preserves history across hard fork boundaries	49
5.2.6	Performant and scalable	49
	Transaction throughput is not significantly affected	49
	Transaction latency analysis	53
	Low-impact on processing time	54
	Low-impact on memory usage	56
	The system should scale	56
5.2.7	Metadata driven	56
	The voting period length is honored	56
	Version dependencies are honored	57
	Priorities are honored	57
	The deployment window is honored	57
5.2.8	Update logic consistency	57
	Consistent update logic	57
5.3	Validation results	58
5.4	Conclusions	64
6	Conclusions	66

Chapter 1

Introduction

In the PRIViLEDGE project, four prototypes have been developed focusing on using DLT or secure multi-party computation for the benefit of online voting, health insurance, university diploma record management and software updates in the Cardano stake-based ledger. These prototypes validate the feasibility of using DLT or secure multi-party computation to solve use-case specific problems.

The development of these prototypes started with identifying the requirements [PRI18], developing the architecture to satisfy these requirements [PRI20b, PRI20c], the definition of the validation criteria [PRI20a] and the actual implementation itself.

In this deliverable, we report on the validation of the existing prototypes based on the requirements [PRI18] and the specified validation criteria [PRI20a].

In Section 2, the results of requirements, protocol and implementation validation are presented for the online voting use-case.

In Section 3, the results of functional, security and performance validation are presented for the health-insurance use-case.

Validation of functional and non-functional aspects, together with usability, interoperability and deployment validation for the university diploma use-case is presented in Section 4

The report ends with the validation of the update system for the Cardano stake-based ledger, using the property testing approach in Section 5.

Chapter 2

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

2.1 Requirements validation

The Tiviledge online voting system uses the general-purpose permissioned blockchain Hyperledger Fabric (HLF) as the bulletin board technology, implementing the online voting auditing functionality as a smart contract (*chain-code* in HLF terminology).

Tiviledge provides secure, usable and transparent online voting by using a protocol that makes it possible, in addition to voter verifiability, to prove to an independent auditor, in a voter privacy preserving manner, that all accepted votes were stored, sent to the tabulation according to the election rules, and decrypted/tabulated correctly.

2.1.1 Validation of compatibility with EU recommendations

In the requirements validation process we have analyzed the compatibility of the requirements of online voting use-case with the EU recommendation on standards for e-voting ([CoE17]).

The EU recommendation contains 49 recommendations in following categories:

- Universal suffrage;
- Equal suffrage;
- Free suffrage;
- Secret suffrage;
- Regulatory and organisational requirements;
- Transparency and observation;
- Accountability;
- Reliability and security of the system.

Many recommendations are not directly about the voting technology, but more about the organization of the election. However, the voting technology can support the election organization in achieving the recommended goal. We strongly agree and endorse the recommendations and the Tiviledge is compatible with EU recommendations.

2.1.2 Validation of functional and non-functional requirements

Verification results of basic functional and non-functional requirements are provided in Table 2.1 and Table 2.2 respectively.

The requirements and the validation criteria can be found in the PRIViLEDGE deliverables [PRI18] and [PRI20a] respectively.

Table 2.1: Functional requirements verification.

ID	Requirement	Status	Reasoning
EA-1	The system must have a possibility to generate keys for elections.	Satisfied	Election Organizer can use Key application for key pair generation and vote decryption.
EA-2	The system must have an election bulletin board.	Satisfied	The election bulletin board is being held inside the Ledger with Election Organizer and Auditor organizations.
EA-3	The system must have a possibility to publish votes to the bulletin board.	Satisfied	Votes are saved locally and rerandomized commitments are published to the bulletin board by VoteCollector.
EA-4	The system must have a possibility to publish election configuration and results to the bulletin board.	Satisfied	These actions can be made with ElectionManager and their appearance on the bulletin board can be checked with VoteApp.
EA-5	The system must have a possibility to validate published votes and election results.	Satisfied	Verification of public records and election results are made in VoteCollector and Ledger. Whereas same verifications can be made by anyone in VoteApp.
VOT-1	The system must have a possibility to cast vote.	Satisfied	Voters are using VoteApp for vote casting.
VOT-2	The system must have a possibility for Voters to access the bulletin board.	Satisfied	Voters can access the bulletin board via VoteApp and make different verification to public records, election configuration and election results.
AUD-1	The system must be set up with Auditor participation.	Satisfied	Auditor participates in Ledger creation by representing his organization in the network.
AUD-2	The system must have a possibility for votes, election configuration and election results verification.	Satisfied	The same as in the EA-5 verifications are made with VoteCollector, Ledger and VoteApp.

Table 2.2: Non-functional requirements verification.

ID	Requirement	Status	Reasoning
NF-1	The applications web interface should be simple and intuitive to use.	Satisfied	The web interface is a wizard-like process, familiar to the voters from variety of online voting applications.
NF-2	The applications web interface should change it's dimensions according to the screen resolution.	Satisfied	VoteApp and ElectionManager change its user interface according to the screen size and additional testing was made with Google Mobile-Friendly test which concluded that the page is mobile-friendly (see Figure 2.7).
NF-3	The applications web interface should be mobile friendly.	Satisfied	The same as with NF-2 requirement, Figure 2.7 shows that both VoteApp and ElectionManager are mobile-friendly.
NF-4	The vote casting process for voter should take no more than 5 seconds.	Partially satisfied.	With a high number of choices casting time takes longer than 5 seconds, therefore this requirement is satisfied only with a small number of choices.

2.2 Protocol validation

We shall revisit each property from the validation criteria [PRI20a] and provide an update about its status in the prototype implementation. Not all properties are satisfied in the prototype, since the focus of the prototype is on the role of the DLT. Topics that might be covered differently in the production setting include voter verifiability, eligibility verification and election key management.

ID	Requirement	Status	Reasoning
P-1 (C)	A voter can post exactly 1 choice from set of n candidates as a vote to a ballot box.	Satisfied	The protocol ensures this property by applying ZKPs of well-formedness. This ZKPs have been implemented and tested.
P-2 (C)	A voter can post up to m unordered choices from set of n candidates as a vote to a ballot box.	Out-of-scope	Without the loss of generality this property is omitted from the scope of the prototype. In case a practical need arises for MofN instead of 1ofN, the zero knowledge proofs can be adjusted without changing the protocol itself.

D1.3 – Use Case Validation

P-3 (C)	Only persons from the set of eligible voters can post votes to a ballot box and bulletin board.	Satisfied	The protocol leaves the details of eligibility verification open, the implementation takes advantage of digital signature based one-time credentials. In production deployment there may be other eligibility verification methods to be supported, additionally the strength of the credential management process is crucial to achieving this property, since the online voting system does not determine eligibility directly, but rather relies on the validity of a credential. The property can be validated by an Auditor on the election basis.
P-4 (C)	Only votes cast during specified time (the voting period) can be accepted into the ballot box.	Satisfied	The property is implemented and tested, it can be validated by an Auditor on the election basis.
P-5 (C)	Voter may re-vote unlimited number of times during the voting period.	Satisfied	The property is implemented and tested.
P-6 (I)	Posted votes are available after the end of the voting period.	Satisfied	The property is implemented and tested. It can be monitored by an Auditor on the election basis, that the published votes shall go into the tally.
P-7 (I)	Tally results must not be published before the end of a voting period.	Partially out-of-scope	The property is implemented and tested, however – the election private key management is not in the scope of the prototype, therefore we do not apply any threshold decryption techniques. This means that a single election administrator could tally the preliminary result, in case access to the votes was provided. We consider the threshold decryption either with or without the trusted dealer a solved issue from the view point of online voting protocols, meaning that this property is relatively easy to support in a future production version.
P-8 (I)	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.	Satisfied	The property is implemented and tested. The fact that the latest vote is taken into account can be validated by the Voter herself on the election basis.

D1.3 – Use Case Validation

P-9 (I)	Votes may be excluded from the tally upon request.	Satisfied	The property is implemented and tested. The revocation can be validated by an Auditor on the election basis. Note, however, that the legitimacy of the record in the revocation list is not straightforward to validate. In the production, the revocation lists should be e.g. digitally signed by an authorized entity.
P-10 (P)	Voter's choice is not leaked for any of the stored votes.	Partially out-of-scope	Please refer to property P-7 (I). In the context of the online voting protocol where all votes are encrypted with some key, this property is a policy decision that can be enforced by measures such as threshold decryption.
P-11 (P)	No-one is capable of learning which choices a particular voter made for any of the posted votes.	Partially out-of-scope	For the data published on the ledger, this property follows from the proper implementation of the protocol. However, similarly to P-7 (I) and P-10 (P), this property must be enforced with measures such as threshold decryption to mitigate against malicious election officials with access to individual votes prior to the homomorphic aggregation.
P-12 (P)	Voter cannot prove to a third party the candidate she voted for.	Satisfied	This property follows from the protocol. Voter cannot construct a cryptographic receipt to prove voting for some candidate. Non-cryptographic proofs – e.g. screenshot of the voting application – can still be constructed, but these are not irrefutable.
P-13 (V)	Voter can verify that she communicates with the application/service which is in control of Election Official.	Out-of-scope	This property is relevant in the production setting, where there must exist trusted channels to publish the URLs and information about the validation of site certificates to the end-user.
P-14 (V)	Voter can verify that her posted vote correctly reflects her choice from a set of candidates.	Out-of-scope	The cast-as-intended property could be ensured by either using Benaloh challenges or Estonian style vote verification with different device. The protocol is compatible with both methods. The implementation of the prototype provides the voter with accepted-as-cast assurance.

P-15 (V)	System can detect that posted vote was not tampered with.	Satisfied	This property has been implemented and tested, it follows both from the use of digital signatures to sign the votes and the availability of election data for an audit.
P-16 (V)	System can detect that stored or posted vote is invalid.	Satisfied	The property follows from the protocol. The application of zero knowledge proofs makes it possible to detect invalid votes.
P-17 (V)	The auditor can verify that the ballot box and the bulletin board contains votes only from eligible voters.	Satisfied	Please see property P-3 (C) – the auditor gets access to the data necessary to verify this property on election basis.
P-18 (V)	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.	Satisfied	The property follows from the protocol, please refer to P-16 and P-17.
P-19 (V)	The auditor can verify that votes excluded from the tally were revoked on purpose.	Satisfied	Please see P-9 (I). The auditor has access to the data required for this validation.

2.3 Implementation validation

The pilot implementation of the protocol supports 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 exist.

This section describes testing methods used to verify that implemented online voting system fulfils desired functional and non-functional requirements.

Most of the testing processes were done on a local machine, see Table 2.4 for machine configuration. End-to-end testing was done in the Amazon Web Services cloud environment.

Table 2.4: Testing machine hardware and operating system info.

Operating system	Ubuntu 18.04.4 LTS
Operating system type	64-bit
Linux kernel version	4.15.0-88-generic
CPU	Intel i5-7260U
RAM	16 GB

2.3.1 Unit testing

The pilot implementation is developed mostly in the Go programming language. It consists of following modules with respective test coverage:

- HLF chaincode (80.1%)
- Common types module (97.2%)
- Bulletinboard module (96.3%)

D1.3 – Use Case Validation

- Commitment consistent encryption module (87.1%)
- Election definition module (100.0%)
- Voter list definition module (98.8%)
- Randomness module (66.7%)
- Voting protocol module (97.1%)
- Election key management tool (31.0%)
- Voter credential management tool (57.0%)
- Post-processing tool (77.3%)

There are altogether 4274 lines of code, 79.5% of which are covered with automated unit tests that focus on the correctness of any given module. These automated tests have proven to be useful in detecting the regression during the implementation phase.

We consider the overall percentage sufficient coverage, since the major use-cases and handling of main exceptions are covered. There is room for improvement in certain modules. However, automated unit testing is complemented by manual end-to-end testing.

2.3.2 End-to-end testing

The manual end-to-end testing was used to validate that the pilot implementation covers all major use-cases – setup, voting, tallying and auditing in repeatable manner. The end-to-end testing is focused on the interoperability of all components of the system.

For the basic end-to-end testing, the whole system was set up locally. Four different elections were created where the number of choices ranges from 2 to 1100 to represent different kind of elections. Multiple votes were cast in each election to understand how the number of choices affects computation time for the voter to cast a vote.

Following phases were tested – setup phase, voting phase, tallying phase. Makefiles, templates and Docker containers were used to automate the set-up of the test infrastructure both in the local and AWS environment.

Testing setup phase

For testing, the tester simulated the activities of participants in the setup phase.

Firstly, Election Organizer and Auditor set up a Ledger with two organizations, they both create a consortium where they agree on who can add, read and modify information inside the Ledger and on what conditions. Then Election Organizer makes different operations to finalize the system initialization:

1. Creates a Public Key Infrastructure (PKI) to give authentication and signing possibility to voters.
2. Generates identities for VoteCollector and ElectionManager by using Fabric CA which belongs to his organization.
3. Creates configuration files for VoteApp, VoteCollector and ElectionManager.
4. Compiles Go binaries (Key application, VoteCollector, ElectionManager server).
5. Creates application Docker images.
6. Initializes VoteApp, VoteCollector and ElectionManager as containers.

D1.3 – Use Case Validation

Because the set-up is a complex process these operations were automated.

After the system is initialized Election Organizer starts with election configuration 2.2 creation and CCE key pair generation.

Then Election Organizer proceeds with election management (see Figure 2.1).

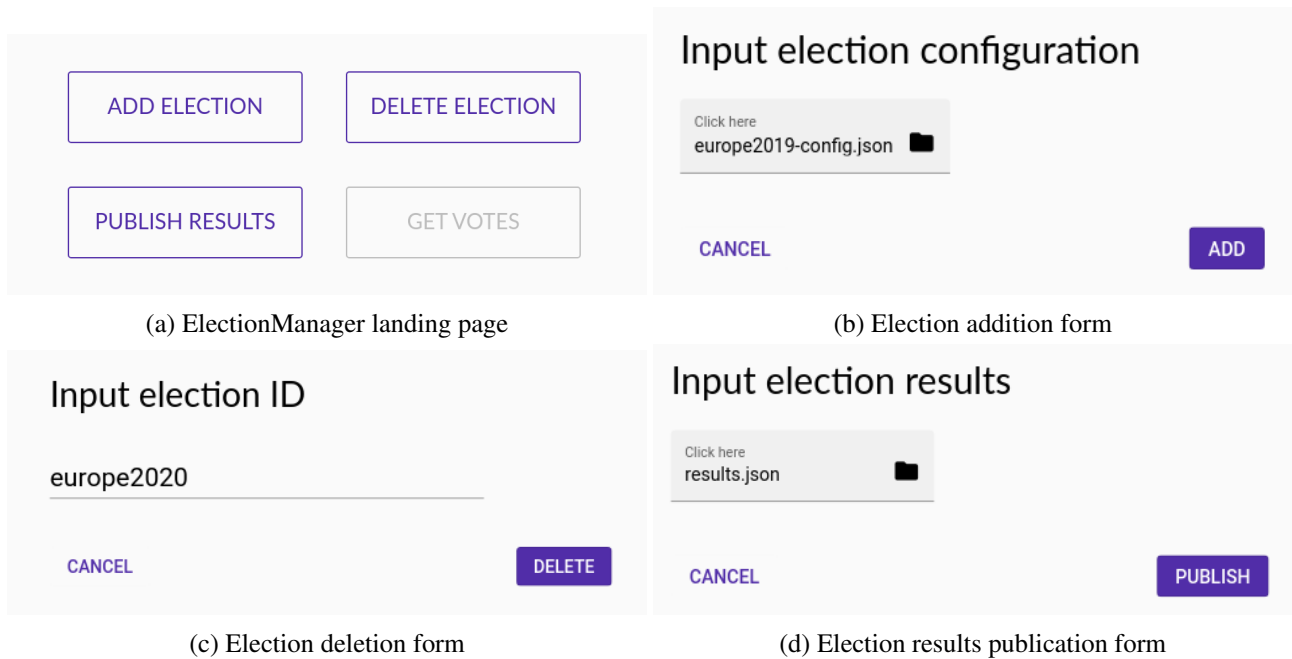


Figure 2.1: Election management demonstration in ElectionManager.

D1.3 – Use Case Validation

```
{
  "electionID": "president2020",
  "electionName": "Presidential Election 2020",
  "electionQuestion": "Who will be the next president?",
  "startDate": "01.05.2020",
  "endDate": "12.05.2020",
  "choices": [
    {
      "id": "1",
      "text": "First Candidate"
    },
    ...
    {
      "id": "8",
      "text": "Eight Candidate"
    }
  ],
  "publicKey": {
    "EncKey": {
      "G": "AwRQ506ZtGO3OgyBMnEs9mi+kisZdDYWb3wpKglmU...",
      "Q": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAA///f//AGAAX/gX...",
      "Y": "AgGM/w01P14/y4O5uwJfMp6x2p6Ia3lzvPm1XOsDT..."
    },
    "H1": "AhDMVBOKBqUKmvZ5RTwnDIkvl8KQfteNRtWBpmgD8Ibb...",
    "H2": "AwVYEWU6UP Tb3iZRmry/JHaQFWleJ9cW4aVomhtdY4Oc..."
  },
  "rootCA": "-----BEGIN CERTIFICATE-----\nMIIBkzCCmgAwIBA..."
}
```

Figure 2.2: Election configuration example.

Testing voting phase

For testing, the tester simulated the activities of participants – voters and verifiers – in the voting phase.

Voter interaction with VoteApp to cast a vote:

1. Selects “CAST VOTE” button (Figure 2.3a);
2. Provides her credential;
3. Selects available election (Figure 2.3b);
4. Selects the desired choice (Figure 2.3c);
5. Receives success message of casting process (Figure 2.3d).

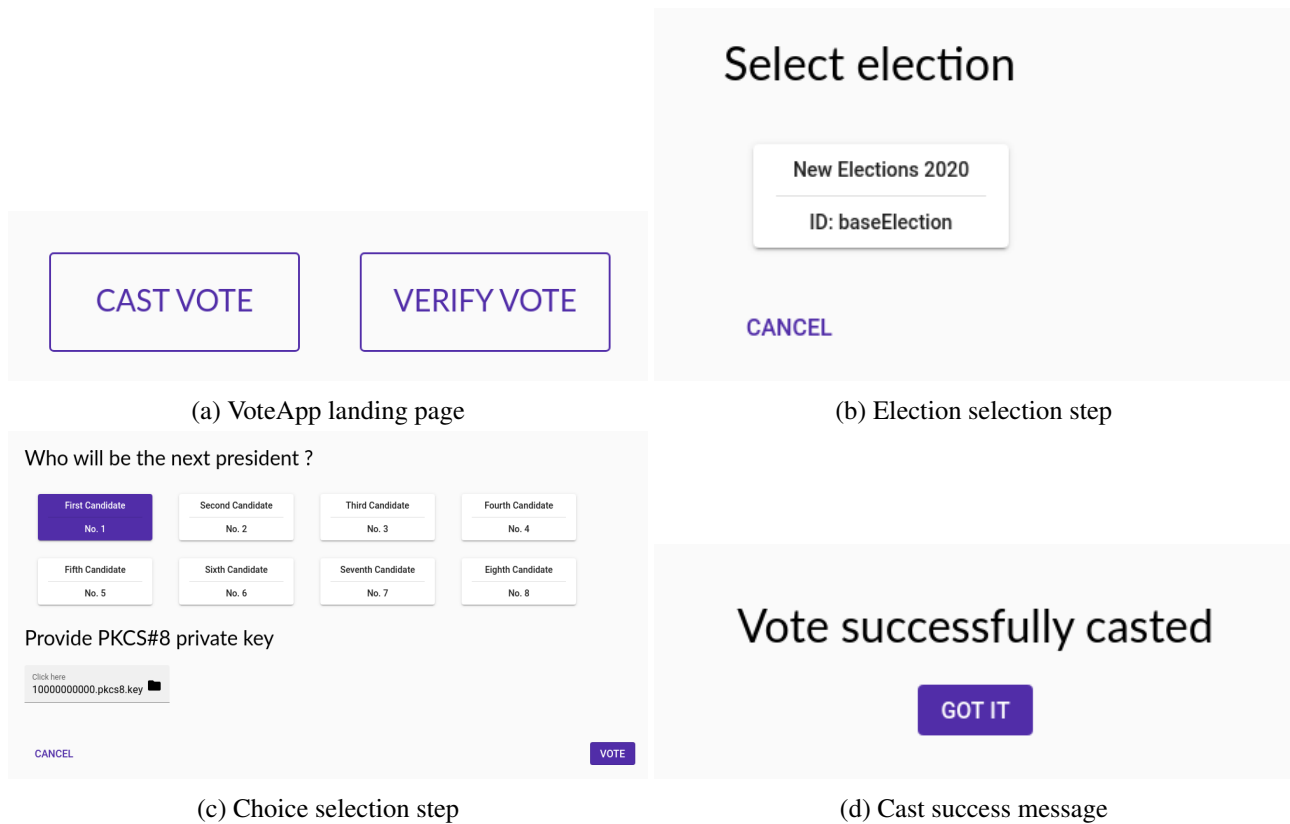
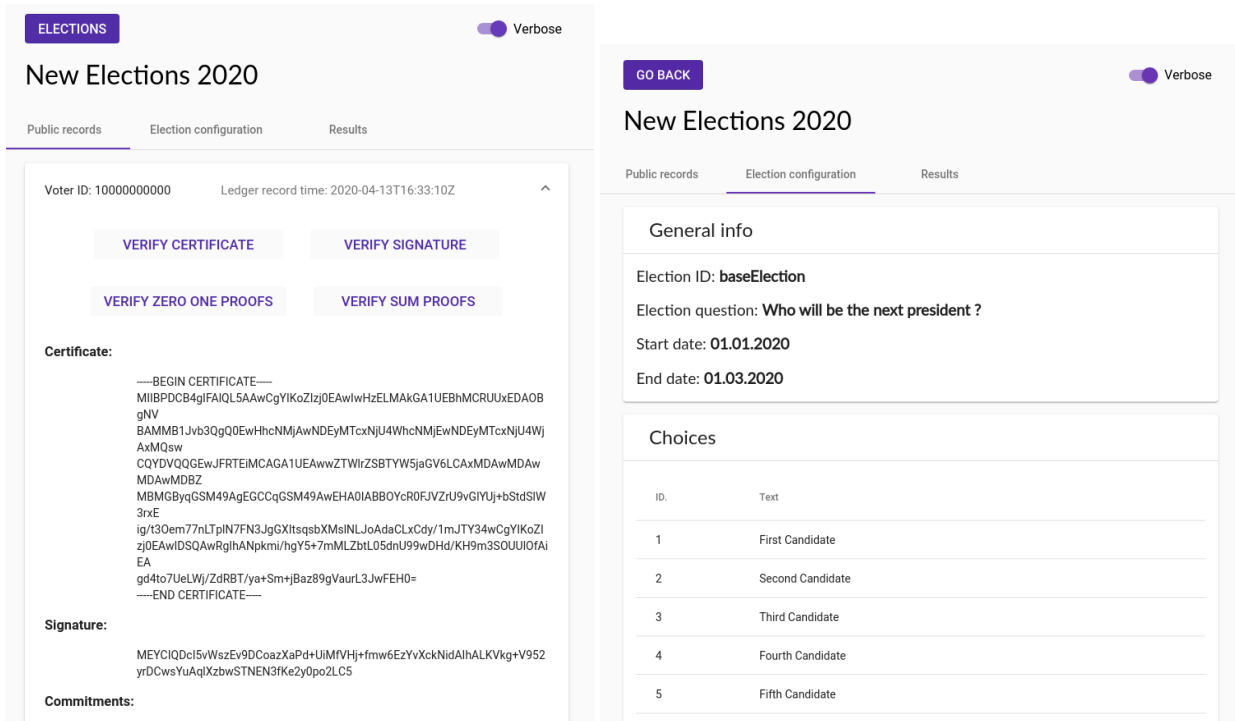


Figure 2.3: VoteApp casting process demonstration.

Testing bulletin board verification

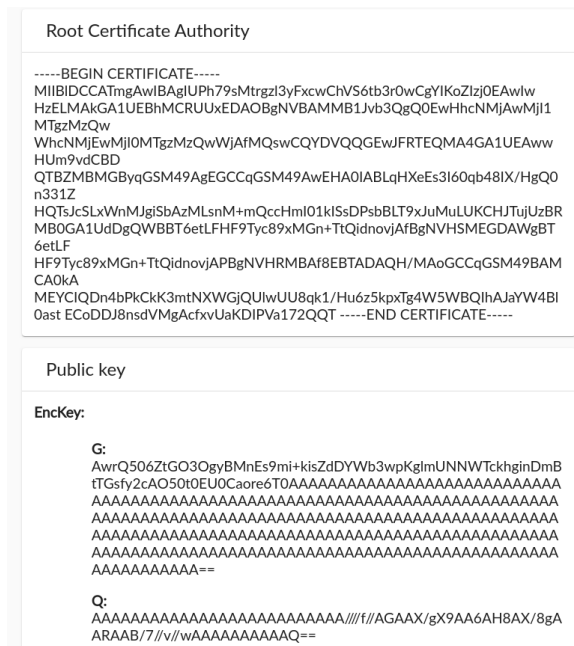
When voter finishes casting process, her public record can be found published on bulletin board and anyone can verify that this record is correct. Verifier interaction with VoteApp:

1. Selects “VERIFY VOTE” button (Figure 2.3a);
2. Selects available election (Figure 2.3b);
3. Verifier can now check any public record individually (if there are any) by making different verifications (Figure 2.4a):
 - VERIFY CERTIFICATE – checks that voter certificate was issued from root certificate in election configuration;
 - VERIFY SIGNATURE – verifies that signature on cast ballot is valid;
 - VERIFY ZERO ONE PROOFS – verifies that for each candidate there is at most one preference;
 - VERIFY SUM PROOF – verifies that the sum of cast preferences is one.
4. Verifier can check election configuration (Figures 2.4b, 2.4c);



(a) Public record on bulletin board

(b) Election configuration part 1



(c) Election configuration part 2

Figure 2.4: Public record and election configuration demonstration in VoteApp.

Testing tallying phase

For testing, the tester simulated the activities of participants in the tallying phase.

Election Organizer gathers all votes from VoteCollector container and performs decryption with additional opening extraction from aggregated commitments using Key Application 2.6. Then decryption results with openings are published on bulletin board by interacting with ElectionManager (Figure 2.1a, 2.1d).

Anyone can verify the published election results via VoteApp verification interface (Figure 2.3a, 2.5).

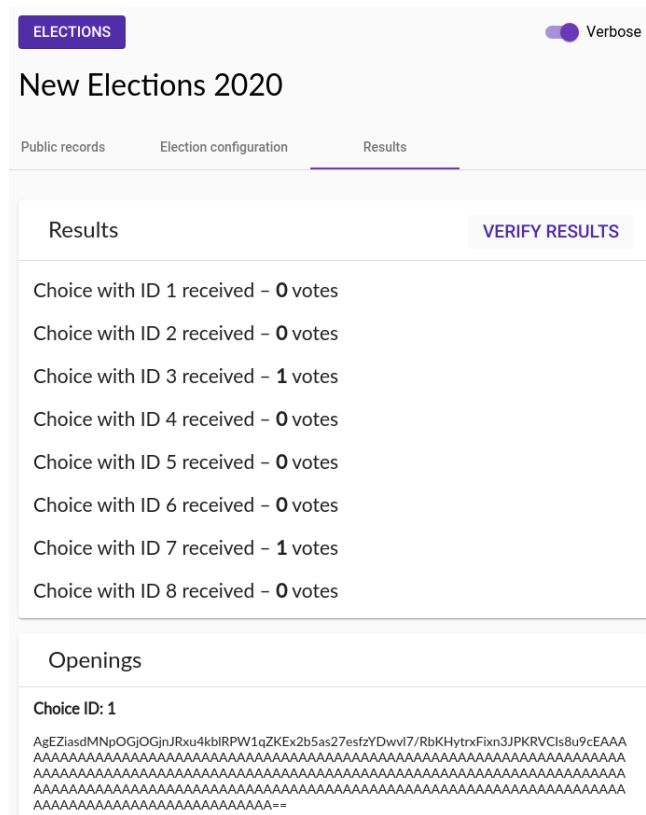


Figure 2.5: Election results published.

Verification is initiated by clicking on "VERIFY RESULTS" button 2.5. VoteApp aggregates public records into one vector, additionally, it makes the same checks as shown on Figure 2.4a. When aggregation is finished commitments are being verified with published openings and results. Verifier is notified if this process succeeded or failed.

D1.3 – Use Case Validation

```
$ ./key --decrypt --id president2020 --keys keys/president2020/ --votes
↪ votes/president2020/
Initiate vote decryption for 'president2020'...
Reading keys...
Aggregating votes...
Number of voters is: 3
Voter '10000000000' latest vote is 'vote~2020-04-07T08:27:43Z.json'
Voter '10000000001' latest vote is 'vote~2020-04-07T08:28:29Z.json'
Voter '10000000002' latest vote is 'vote~2020-04-07T08:30:17Z.json'
Votes are aggregated...
Aggregation took: 41.560649ms
Decrypting ciphertexts and extracting openings...
Decryption and openings extraction took: 170.552309ms
Election results are:
--- Choice ID - 1 got result - 2
...
--- Choice ID - 8 got result - 1
Election results with openings are saved to
↪ 'results/president2020-results.json'
```

Figure 2.6: Key Application decryption process.

2.3.3 Usability

Usability criteria were established in the [PRI20a]. A validation of the pilot implementation with the respect to the criteria was carried out with following results:

ID	Requirement	Status	Reasoning
UC-1	The process can be cancelled by the voter any time before the confirmation;	Satisfied	All steps in the online voting flow have explicit option to cancel the process without submitting the vote.
UC-2	The voter is prevented from accidentally voting for an undesired candidate (by e.g. explicitly selecting / deselecting);	Satisfied	Candidate must be explicitly selected or deselected.
UC-3	The voter is prevented from over-voting 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;	Partially satisfied	Current implementation supports 1 of n scenario. Voter can only vote when at least one selection is made, voter cannot make any more selections than one.
UC-4	Voter identity and voter choices are not displayed together on the same screen to fight against coercion;	Satisfied	Voter identity is only visible on the login page.

D1.3 – Use Case Validation

UC-5	The ballot presentation (including candidate order) is aligned with the requirements of specific election;	N/A	This requirement does not apply in the pilot setting, since there are no jurisdictional requirements (references to the legal texts, additional confirmations, ordering of the candidates etc.)
UC-6	The information is presented in unbiased manner, voter has chance to review all options;	Satisfied	Dependent on the number of candidates and screen size, scrolling may be needed to see all options.
UC-7	The application follows adaptive/responsive design techniques to support various screen sizes and orientation;	Satisfied	The application has been implemented using adaptive design techniques, it has tested to be mobile-friendly 2.7.
UC-8	The application supports keyboard-only navigation;	Partially satisfied	The application supports keyboard-only navigation and all functionality is available without mouse. This is important for the voters with disabilities. However, the usability validation suggested improvements to the current implementation, which we consider sufficient for the pilot.
UC-9	The application is compatible with assistive technologies;	Partially satisfied	The application supports assistive technologies as tested with screen-readers JAWS and NVDA. This is important for the voters with disabilities. However, the usability validation suggested improvements to the current implementation, which we consider sufficient for the pilot.

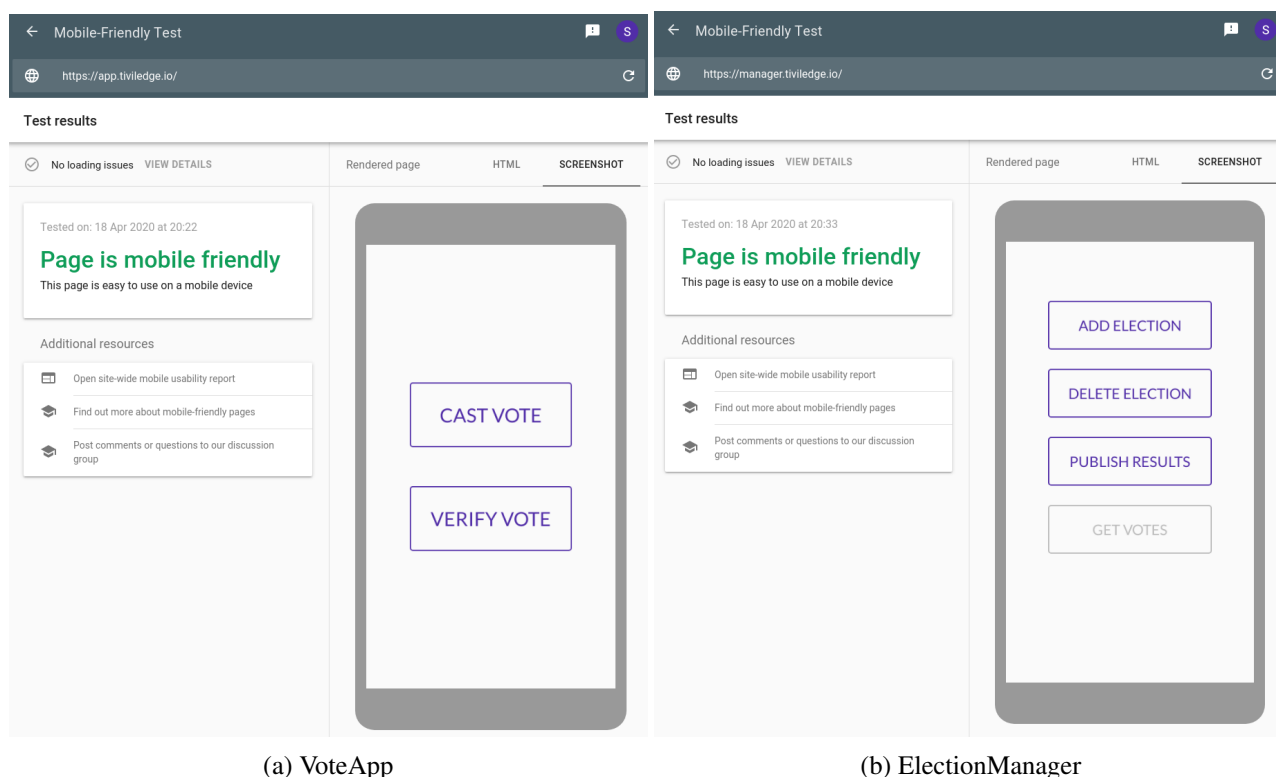


Figure 2.7: Google Mobile-Friendly Test results.

2.3.4 Performance testing

Performance testing consisted of computation time recordings of different processes inside components. Measured processes where:

- vote casting (Table 2.6);
- public record verification (Table 2.8);
- votes decryption (Table 2.7);
- election results publication to Ledger (Table 2.9);
- election results verification in VoteApp (Table 2.9).

Table 2.6: Vote casting time pure computation and total time with HTTP requests. All times are provided in seconds.

Number of choices	2	8	70	1100
Cast message creation - VoteApp	0.87	2.22	17.73	281.52
Cast message process - VoteCollector	0.28	1.58	9.58	147.88
Final message creation - VoteApp	0.53	1.65	13.87	222.73
Final message process - VoteCollector	0.12	0.41	3.57	68.11
Public record addition - Ledger chaincode	0.14	0.45	4.00	64.27
Total casting time	1.94	6.31	48.75	784.51
Total casting time with HTTP requests	3.82	8.44	51.02	799.65

Table 2.7: Votes tallying time results in Key application (with 5 votes). All times are provided in seconds.

Number of choices	2	8	70	1100
Vote aggregation time	0.11	0.16	1.25	20.77
Decryption with opening extraction	0.39	0.64	5.47	99.5
Total time	0.50	0.80	6.72	120.27

Table 2.8: Vote individual verification in VoteApp. All times are provided in seconds.

Number of choices	2	8	70	1100
Certificate check	0.17	0.10	0.18	0.12
Signature check	0.20	0.13	0.14	0.18
Zero One proof check	0.63	1.28	10.36	157.94
Sum proof check	0.18	0.24	0.91	12.04

Table 2.9: Vote universal verification in VoteApp and Ledger chaincode (5 votes). All times are provided in seconds.

Number of choices	2	8	70	1100
Public records aggregation with individual verification (s)	2.78	7.42	52.42	–
Opening verification	0.59	2.18	17.66	–
Total time VoteApp	3.37	9.6	70.08	–
Results verification inside Ledger chaincode	0.86	2.99	25.11	–

In Table 2.9 there are no time results for election with 1100 candidates. The reason is that the election results verification takes a lot of time (longer than 300 seconds), which triggers Orderer inside the Ledger to drop the connection. To resolve that some changes to Ledger network configuration must be done but at the time of writing and testing the right place where this modification must be done is not found.

Unfortunately with a high number of choices (greater than 8), vote casting takes a lot of time and as we see from the tables that by increasing the number of choices computation time increases linearly. Therefore modifications to the system are required to improve the performance.

2.4 Conclusions

Tiviledge provides secure, usable and transparent online voting by using a protocol that makes it possible, in addition to voter verifiability, to prove to an independent auditor, in a voter privacy preserving manner, that all accepted votes were stored, sent to the tabulation according to the election rules, and decrypted/tabulated correctly.

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

The underlying cryptographic protocol is based on verifiable homomorphic aggregation of rerandomized encrypted votes created with a commitment-consistent encryption (CCE) scheme. The protocol allows to publish

D1.3 – Use Case Validation

rerandomized commitments to the public ledger, providing third-party auditability and receipt-freeness. Unconditionally hiding commitment scheme is used.

We have validated the Tivledge from the perspective of requirements, protocol and implementation and we consider that it fulfills the validation criteria. However, we must note that current Tivledge implementation is a research prototype, its intended use is experimentation with the technology, not organizing an actual legally binding online voting event. There are following major areas of interest, important for any production-grade online voting system, that haven't received thorough attention in the development of Tivledge prototype:

- Compatibility with legal requirements of any particular jurisdiction—the Tivledge prototype implements 1-of-N ballot structure as a proof-of-concept, but is generalizable to M-of-N ballot-style, supports revoting and multi-channel elections. However, dedicated gap analysis is required in any instance of potential application.
- Voter eligibility verification—the Tivledge prototype implements a simple credentialing scheme, where voters authenticate and sign messages using one-time credentials that are stretched into ECDSA private keys. In practice the procedure and technology of eligibility verification is important cornerstone of election security to avoid e.g. ballot-box stuffing or breach of voter privacy and the simplistic PKI of Tivledge is not applicable.
- Election private key protection—while implementing the Tivledge prototype we've considered the election private key protection largely a solved problem, not requiring special attention in this project. Threshold decryption (with different dealer setups) and hardware security modules are the topics of interest here, since the file-based private keys with no access control, that we use in Tivledge are not suitable for real elections.
- Voter verification—the Tivledge prototype does not come with a verification application for cast-as-intended property, whereas both Benaloh-challenge and Estonian-style verification would be possible here.

We also note that the complexity of the system depends on the number of candidates and this makes Tivledge and homomorphic aggregation suitable in contexts where there are less candidates.

Chapter 3

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

The requirements of this use case have been described in project deliverables [PRI18, PRI20b]. The main deliverable of the project for this use case is a prototype implementation of the system. As explained in [PRI20a], the prototype was used to evaluate the functionality, security, and performance properties of the underlying protocol.

We model updates to patient data as a stream of events where each event is represented by a tuple of unsigned integers. The first value of each such tuple is the patient identifier, the second one the identifier of the event type and the remaining ones are parameter values whose meaning depends on the event type.

The prototype is built as an application on top of MPyC, a secure multi-party computation framework for the honest majority setting.¹ The prototype uses a private development snapshot of MPyC implementing the circuit satisfiability protocol of [AC20], which reconciles Bulletproofs with Sigma Protocol theory. Currently, only the classical addition and multiplication gates are supported by the verifiable computation framework. In particular, comparison gates are a work in progress not yet available to be used in our prototype.

To implement comparisons using the available gates, we need to pass the input data into the circuit as bit vectors instead of atomic integers, which causes a significant expansion of the circuit size. To control the expansion, we limit the patient identifiers to 20 bits (allowing for up to 1 048 576 patients), the event types to 10 bits (allowing for up to 1 024 distinct types) and the remaining attributes to three fields of 8 bits each (allowing for $256 \times 256 \times 256$ combinations).

3.1 Functional validation

As the functional requirements describe the ability of various types of users to perform certain operations, unit testing and integration testing were 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 were performed on synthetic data.

Requirement	Results
It must be possible to compute a commitment for a set of events.	We are using Pedersen vector commitments to represent the sets of events.
It must be possible to compute a union commitment representing the union of two sets of events.	With Pedersen commitments, the commitment of a union of disjoint sets is the product of the commitments of the input sets, and is thus trivial to compute and verify.

¹<https://github.com/lischoe/mpyc>

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).	We compute the required subset on plaintext data, but in parallel execute a verifiable computation to derive the commitment for the subset, including in the set commitment only the commitments of the elements that match the filter condition.
It must be possible to compute simple aggregates (count of elements, sum of values of a given attribute) over the sets under the commitments.	We run the counting or summing process as a verifiable computation and open the commitment of the count or sum value as the last step of the protocol.

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 benchmarking. Instead, the security of the underlying cryptographic primitives and protocols was analysed theoretically.

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 should be used to validate the implementation of the protocols in a production implementation. As the prototype was only used to prove the concept, the robustness requirements are much lower and thus only code reviews were employed in the scope of this project.

Requirement	Results
It must be infeasible to change the data under a commitment without breaking the link to the commitment.	Pedersen commitments used in the prototype are known to be computationally binding under the discrete logarithm assumption. This means they are vulnerable to quantum attacks and must be replaced with a post-quantum resistant solution when large-scale quantum computers become realistic. Note that this affects only the proof value of the commitments.
It must be infeasible to recover the underlying data from a commitment.	Pedersen commitments are known to be unconditionally hiding. This means they are not vulnerable to any advances in computing technology (including quantum attacks). Thus, the commitments posted today can not be used to attack the privacy of patient data in the future.
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.	With Pedersen commitments, the commitment of a union of disjoint sets is the product of the commitments of the input sets, and is thus trivial to compute and verify.
The proof of correctness of the union commitment must not leak the details of the underlying events.	The proof in this case is based only on the commitments themselves and thus can not leak any additional information.
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.	The verifiable computation protocol is based on [AC20], which is computationally knowledge sound under the discrete logarithm assumption.
The proof of correctness of the subset commitment must not leak the details of the underlying events.	The verifiable computation protocol used is based on [AC20], which is perfectly special honest-verifier zero-knowledge.
When computing an aggregate of a subset under a commitment, it must be possible to get a proof that the aggregate was computed correctly.	The verifiable computation protocol is based on [AC20], which is computationally knowledge sound under the discrete logarithm assumption.
The proof of correctness of the aggregate must not leak the details of the underlying events.	The verifiable computation protocol used is based on [AC20], which is perfectly special honest-verifier zero-knowledge.

It must be infeasible to recover the input data from the shares used in the multi-party computation.	The multi-party computation uses Shamir secret sharing which is known to be unconditionally hiding unless majority of the parties collude. Thus, the shares exchanged by any minority subset of the parties today can not be used to attack the privacy of patient data in the future.
--	--

3.3 Performance validation

The performance of the proposed protocols was first analysed theoretically, in the asymptotic complexity model. The results were then validated based on benchmarking on a representative set of test data. The experiments were performed using a single core of a 1.8 GHz Intel Core i7 CPU, running the Cygwin64 port of CPython 3.8 on top of Windows 10 Pro. For the multi-party computations, all parties were executed as threads within the same process. In all experiments, the performance was CPU-bound.

Metric	Target	Results
Cost of computing a set commitment, cost of computing a union commitment and its proof	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. 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.	Computing the commitments can be done in about 5 ms of CPU time per event, or in about 50 seconds for a batch of 10 000 events. The cost of merging the previous cumulative commitment and the commitment of a new daily batch is negligible. Overall, the performance is well on target.
Cost of verifying a set commitment, cost of verifying a union commitment	Verification of a set commitment is routinely done only when the records of one patient are extracted and their integrity verified. This is an on-demand activity and should run in a few seconds preferably, or a few tens of seconds at most.	The cost of verifying a commitment on a set of events is the same as computing the commitment. With one patients' records not expected to run into tens of thousands of events, the performance is on target.
	Exceptionally, the integrity of updates over some time period may have to be verified for auditing. This is a pre-planned activity, so computation time in tens of minutes is acceptable.	The cost of verifying a commitment on a set of events or a union commitment is the same as computing those commitments, so tens of thousands of events can be verified in minutes and thus the performance is on target.
Cost of computing a subset commitment and its proof.	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.	In the current model, proving the commitment of a subset is linear in the size of the superset from which the subset is extracted. In the best results so far, the process took about 3 seconds of CPU time per event in the input set for equality comparisons on patient identifiers and about 8 seconds for equality comparison on event type followed by a range comparison on an attribute value, which is not yet practical for using on general patient populations.

Cost of verifying a subset commitment.	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.	Computing the commitment of a subset for verification runs in time linear to the size of the subset, and verifying the proof of the computation of the commitment runs in time linear to the size of the proof, which itself is logarithmic in the size of the input set. For realistic data sizes, the verification cost is several orders of magnitude lower than the proving cost, and thus on target.
Cost of computing a count or a sum and its proof.	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.	In the current model, computing and proving a count or a sum is linear in the size of the input set and in the best results so far, the process took about 0.35 seconds of CPU time per event in the input set, which is much less than for the subset computations. Since any report will start from filtering the full data set, the aggregation performance is currently not a limiting factor for feasible dataset sizes.
Cost of verifying a count or a sum.	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.	Verifying the proof of the computation of a count or a sum runs in time linear to the size of the proof, which itself is logarithmic in the size of the input set. For any data sets where constructing the proof is realistic, the verification cost, which is several orders of magnitude lower than the proving cost, is on target.

3.4 Conclusions

Aside from the constraints imposed by the underlying cryptographic protocols, the verifiable MPC extensions of the MPyC framework currently have two technical limitations: (a) the cost of constructing the proofs is super-linear in the size of the circuit specifying the computation; and (b) it only implements the classical addition and multiplication gates, but lacks support for comparison gates. A subsequent iteration of the verifiable MPC extensions is expected to remove these limitations and speed up the execution of verifiable computations by a few orders of magnitude.

When the planned improvements to the verifiable MPC toolkit arrive, the computations should become feasible for smaller specialized subsets, such as cancer patients or transplant recipients. As the treatments in those segments are generally very costly and also quite experimental, it may well be the case that the technology will indeed be practically useful there even when it's not yet ready for the more general applications.

Chapter 4

Validation Results—Use Case 3: University Diploma Record Ledger

Use Case 3 implements a system for privacy-preserving diplomas (or more generally, credentials) validation. Three actors are involved:

- The ISSUER is the entity (typically, university) that awards degrees.
- The HOLDER is the entity (typically, a university alumnus or alumna) that is awarded the degree.
- The VERIFIER is the entity to which the HOLDER wants to provide proof of a degree awarded by the ISSUER.

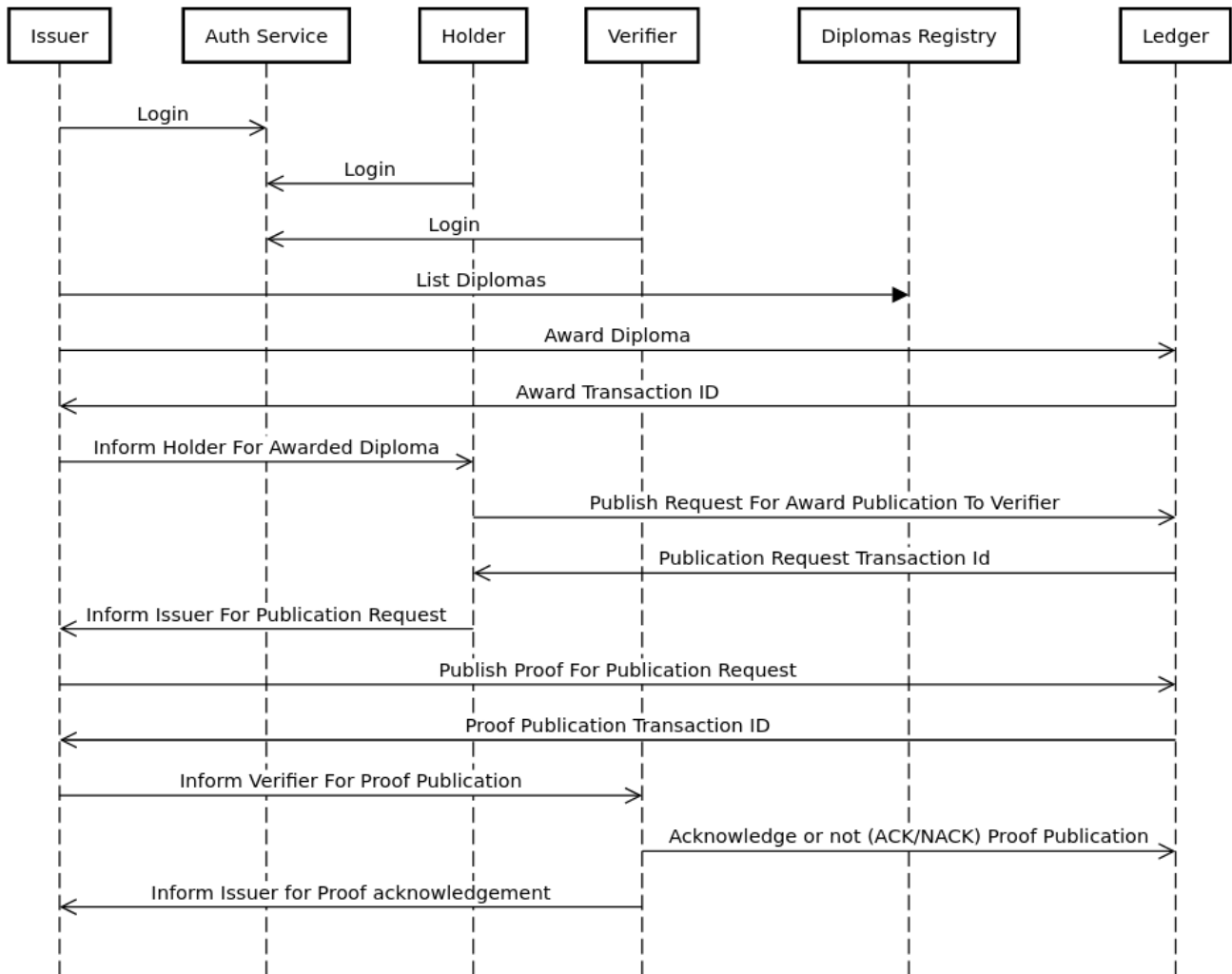
The three actors interact following the DIPLOMATA protocol described in [PRI20b]. The DIPLOMATA protocol has been implemented in the e-diplomata system, whose validation, following [PRI20a], we describe below.

4.1 Functional Validation

The following table provides a listing of the various actions performed by the involved parties at the application layer of the service.

Actor	Action	Component
ISSUER	Login	Authentication Service
ISSUER	List Diplomas	Issuer Service
ISSUER	Filter Diploma	Issuer Service
ISSUER	View Diploma	Issuer Service
ISSUER	Award Diploma	Issuer Service & Public Ledger
ISSUER	View Award Status	Public Ledger
HOLDER	Publish Request for Award Proof	Issuer Service & Public Ledger
HOLDER	View Publication Request Status	Public Ledger
ISSUER	List Publication Requests	Issuer Service & Public Ledger
ISSUER	Publish Proof for Publication Request	Issuer Service & Public Ledger
ISSUER	View Proof Publication Status	Issuer Service & Public Ledger
VERIFIER	Publish ACK/NACK/FAIL for Publication Proof	Verifier Service & Public Ledger
VERIFIER	List Verified/Rejected Publication Proofs	Verifier Service & Public Ledger

The sequence of the above actions is depicted in the following diagram.



Functional validation

4.2 Non-functional Validation

Cryptographic Security

Without making any particular assumptions about the ledger in use, the security of the DIPLOMATA protocol relies on the properties of an El-Gamal cryptosystem for generating signatures and commitments (Basic Crypto Layer) along with a symmetric encryption mechanism for interparty message communication (Transaction Logic Layer). In particular, the usual hybrid approach has been adopted, where the security features of an asymmetric cryptosystem (non-repudiation, zero-knowledge primitives, etc.) are combined with the advantages of symmetric cryptography (low cost of encryption). Both layers are built on top of appropriately chosen elliptic curves.

- (a) **Transaction Layer:** At the transaction layer, involved parties have the ability to potentially create common secrets for the purpose of exchanging sensitive information. For example, the ISSUER symmetrically encrypts part of the produced proof addressed to a VERIFIER (in particular: the proof of decryption along with the accompanying decryptor), so that no man-in-the-middle who eavesdrops the transaction layer and captures the proof-packet is able to verify it and publish an acknowledge. The symmetric encryption infrastructure makes calls to the NaCl public-key API, meaning that each involved party must own a key over the Curve25519 elliptic curve (128 bits of security/256 bits keys size) used for common secret agreement.

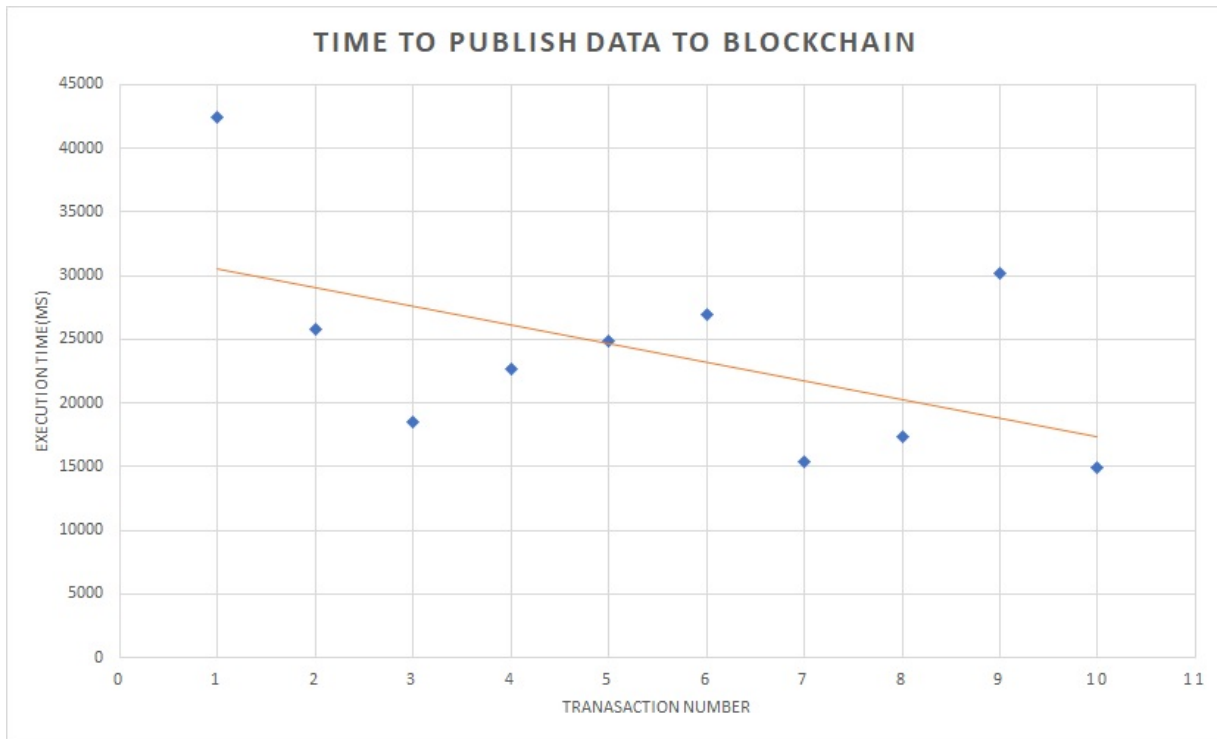
(b) **Basic Crypto Layer:** At this layer, the main interest is to

- (i) generate and verify zero-knowledge proofs; in particular, DDH proofs on behalf of the ISSUER along with Chaum-Pedersen proofs of reencryption are crucial for traceability of computations to knowledge of secret parameters and cross-checking integrity of documents.
- (ii) sign computations and verify these signatures; in particular, verifying the signatures which enter the ledger guarantees coherence and soundness (e.g., that a HOLDER cannot request a proof for an award they don't own, or that an ISSUER cannot create a proof without prior request on behalf of a HOLDER).

The security of these operations depends on the hardness of the Discrete Logarithm Problem for certain elliptic curves. The currently suggested minimum key size is 384 bits, meaning that each involved party must own an El-Gamal key over the P-384 elliptic curve (192 bits of security). The adopted scheme for digital signatures is the ECDSA standard. In order to reduce the number of protocol steps and relax dependence on communication failures, the usual approach of making zero-knowledge proofs non-interactive was followed. This is attained by means of the Fiat-Shamir heuristic, which requires a secure hash function to be fixed and used by all parties. The bit length of the digest output should be at least equal to that of the order of the curve in use, so SHA384 has been chosen.

Performance Metrics

In order to examine the system's behaviour with respect to the ledger, the time needed to publish ledger entries was recorded. The following diagram shows the results of ten data publications to the distributed ledger. The ledger used for the tests was Ethereum Ropsten blockchain, but any other distributed ledger could be used as well. The vertical axis shows the time needed to confirm that data were published to the ledger and the orange line shows the linear regression derived from the recorded data. Based on the recorded values, the average time for the system to publish the data to the ledger is 23892.1 ms. Following that, the transaction block in which the data were published must be confirmed. For the Ethereum Blockchain, we can assume that this happens after the publication of 10 blocks. Consequently, the average time needed to confirm data publication to the ledger is roughly estimated to be $24 + 10 \times 24$ sec.



4.3 Usability Validation

In the following section we sum up the results of several User Interface (UI, UX) evaluations that took place during the e-diplomata service prototype design and implementation. Considering the early stage of the service, most of the evaluations were based on the cost effective method Heuristic Evaluation (HE) with a view to be able to continuously re-inspect the system usability after each iteration of the development process. Each evaluation resulted in a set of UX improvements. Each one, was categorized, based on the predefined set of usability heuristic(s), and then fed to the subsequent development iteration. In this context, heuristics refer to general guidelines for user interface design, such as Jacob Nielsen’s ten general principles [NM94]. For more details on Heuristic Evaluation see Section 2.4.3 of [PRI20a].

4.3.1 User Interface

Major focus to identify usability issues was given to the Single Page Application components that deliver the user interfaces of the service.

Design Toolkit

As is the case in modern UI development, a preliminary research regarding the use of an existing web based UI toolkit was conducted. A UI toolkit provides an assortment set of ready-to-use UI resources, providing

developers a way to efficiently build usable and accessible interfaces.

The e-diplomata prototype was based on digigov-sdk, an open-source toolkit that was built in the context of the digital transformation act of the Greek government. The tool consists of a Design System that was highly inspired by the corresponding project of gov.uk, and a prototype implementation of the system components and patterns using the react.js library. We additionally consider that e-diplomata user personas requirements closely match those targeted by the toolkit as it was designed with particular regard to meeting usability and accessibility standards for use in e-government services.

e-diplomata Patterns

The implementation of the design system of digigov-sdk is inspired from the Atomic Design methodology which indicates a set of rules regarding the classification of the system components. This allowed us to use existing basic (atom) components and a set of existing patterns (organisms) to build up majority of the e-diplomata views.

In the following sections we point out the set of usability heuristics that validate the overall design of the application. We additionally point out heuristics validation for a set of core patterns reused across application views.

Cross Application Heuristics

Match between system and the real world

- Express ledger concepts into basic wordings that stem from the domain model of e-diplomata. Prevent oversimplifying concepts, use multi-level state wordings when applicable.

State:

```
qualification award action pending for ledger transaction confirmation  
could be mapped to  
award pending / publishing
```

- Prevent hiding actions that are not yet available but are still to be executed at a later stage of the protocol. Use disabled styles instead.
- Hide resource actions that are no longer apt to be executed at a later stage of the protocol.

Consistency and standards

- Common 2/3 layout across views.
- Comply with the AA level Web Content Accessibility Guidelines (WCAG) success criteria.
- Responsive design for enhanced mobile UX.

Recognition rather than recall

- Common 2/3 layout across views.
- Comply with the AA level WCAG success criteria.
- Responsive design for enhanced mobile UX.

Recognition rather than recall

- Reuse of existing patterns when possible to mitigate user cognitive overload (resource listing, item details).

Aesthetic and minimalist design

- Minimalist layout design.

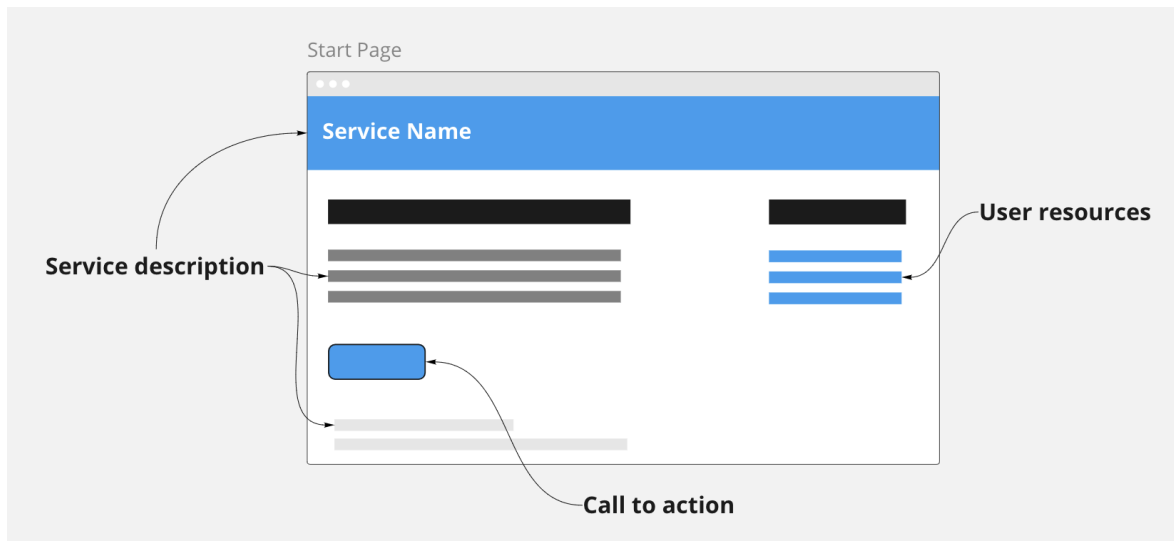
D1.3 – Use Case Validation

- Prevent excessive use of animations.
- Prefer use of comprehensive labels over icons/images.

User control and freedom

- Provide a link to the service index view from the header layout section.

Pattern 1. Start page / Layout



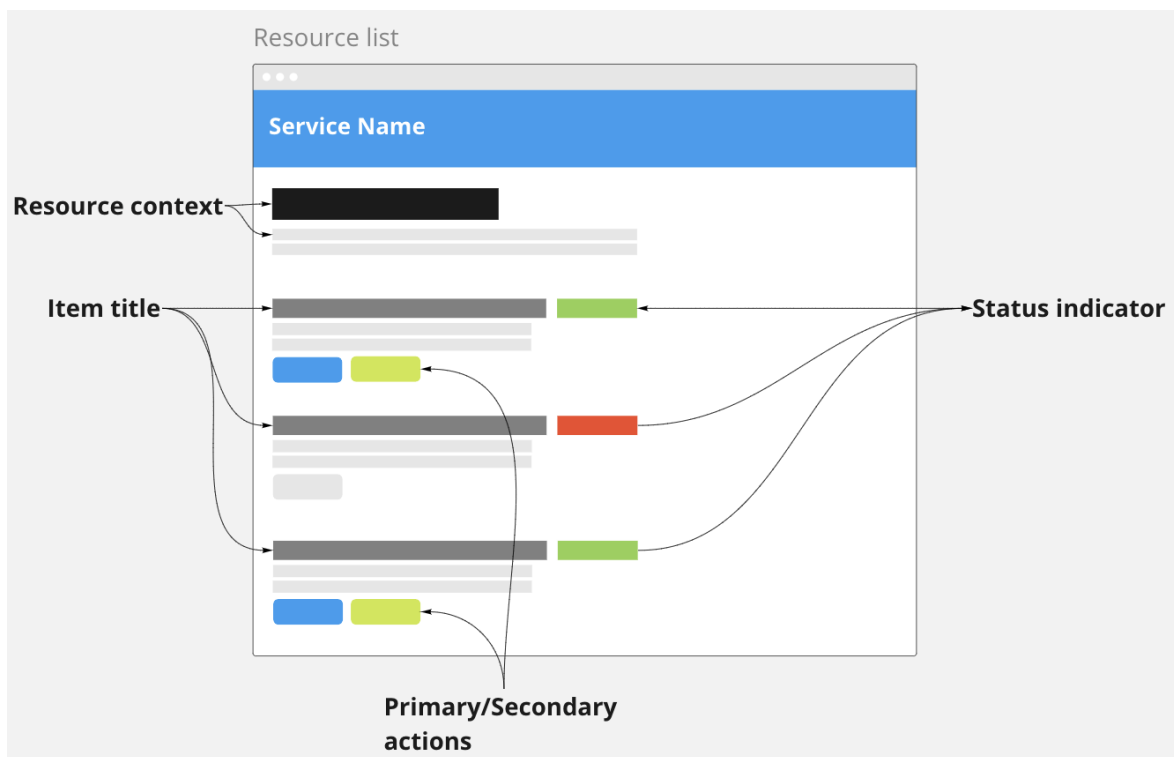
Aesthetic and minimalist design

- Prominent CTA (Call to Action) styles and positioning.

Help and documentation

- Primary service description.
- Secondary text to communicate next step eligibility details.
- Sidebar links to service documentation.

Pattern 2. Resource List



Visibility of system status

- Entry title and details.
- Styled status indicator for each item.
- Set of actions based on item status.

Recognition rather than recall

- Always map primary action to navigate to item details view.
- Use at most one available secondary action.
- Display non-applicable secondary actions using disabled styles.
- Separate styles for primary/secondary action.

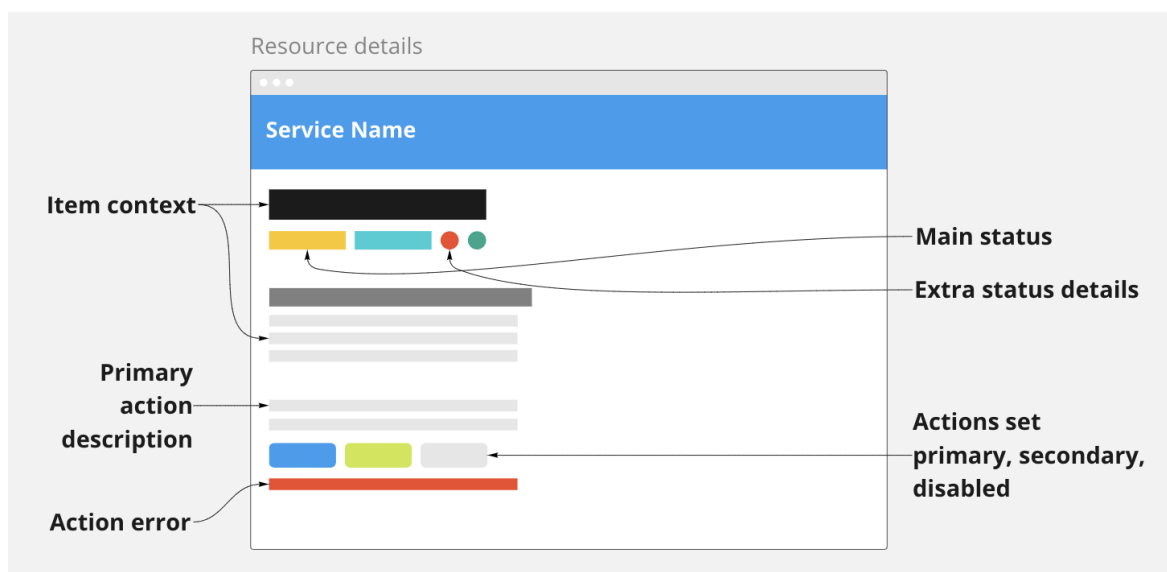
Aesthetic and minimalist design

- Cut down item details to 1–2 rows. Additional details will be provided in item details view.

Error prevention

- No irreversible resource action should be available in list view.
- Display a confirmation prompt for actions that alter the state of the resource.

Pattern 3. Resource Details / Actions



Visibility of system status

- Item title and detailed description.
- Primary item status.
- Additional status content if applicable, to map asynchronous ledger state.
- Set of actions based on item status.

User control and freedom

- Warn users regarding the irreversible result of actions that trigger ledger transactions.

Error prevention

- Detailed main action description.
- Display a confirmation prompt for actions that alter the state of the resource.

Flexibility and efficiency of use

- Elements that indicate the extended status of the resource may link to the public ledger system (e.g., <https://etherscan.io/txid>).

4.4 Interoperability Validation

Interoperability validation refers to the assessment of the ability of the server to integrate data from pre-existing sources. The objective is to ensure that the e-diplomata implementation can interoperate with existing university infrastructure which for our case is the GUnet server.

The interoperability part of the implementation consists of:

Authorization: An authorization token for communication with the GUnet server is stored for each issuer. The token is used to be able to ask the GUnet server for the issued diplomas and their respective title details.

Diplomas endpoint: Using the token a GET request to the provided GUnet diplomas endpoint (`http://toast.noc.uoa.gr:7770/diplomas?maintainerCode=uo`) is made.

The response contains information about the student, the degree and degree date for the issued title and a reference to the title id for the referred diploma.

Titles endpoint: Using the token a GET request to the provided GUnet titles endpoint (`http://toast.noc.uoa.gr:7770/title?maintainerCode=uo`) is made.

The response contains information about the title which includes the institution name and diploma title details.

Award functionality: For the award functionality the diploma information is duplicated on the diplomata issuer server storage component and is enhanced with crypto and ledger information.

4.5 Implementation Validation

Implementation validation refers to the assessment of the correctness of the code implementing the use case. The objective is to ensure the integrity and robustness of the service, and make sure that the implementation is successful and error-free.

In its current state, the project consists of approximately 5000 lines of code. It requires 2 physical CPU cores, 4GB of RAM and 20GB of disk space. With a standard Intel i7-7500U 2.70GHz processor, the average time needed to build the application is about 2 min and 20 sec.

Every project must describe a Continuous Integration pipeline, in order to ensure the quality of the project. The project has the following main jobs:

Test: All microservices must run and pass tests on all cases. Tests are the building block of the application, and they make sure that the application meets the needs of the users. Tests need to cover both business logic and unit tests.

Unit testing: the purpose of unit testing is to test the correctness of isolated code, such as methods, functions etc. It's done during the development of an application by the developers. A unit may be an individual function, method, procedure, module, or object.

Functional testing: functional tests ensure that the application works as expected from the user's perspective. Assertions primarily test the user interface.

Both unit and functional testing are implemented via Jest; a JavaScript testing framework designed to ensure correctness of any JavaScript codebase.

Test coverage: a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with high test coverage, measured as a percentage, has had more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage.

Lint: Since a project requires both active development and maintenance, the Lint job needs to ensure that the application stays aligned with the linting configurations used throughout the project, both for backend and frontend.

Linting is the automated checking of your source code for programmatic and stylistic errors. This is done by using a lint tool (otherwise known as linter). A lint tool is a basic static code analyzer.

Build: The build job generates artifacts, which are actually docker images, generated by the production grade Dockerfiles. These images need to be deployed in GRNET's docker registry, with the relevant docker tags. All projects need to follow and maintain the build job (which is maintained by the GRNET team).

4.6 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. Every project needs to have a pipeline in order to validate the quality, the readiness and the deployability of the project. There are specific rules that need to be met in order for the application to be deployed in an environment. The Continuous Integration and Continuous Deployment can be split into two main categories:

- Application specific jobs: the project defines its own jobs to ensure the quality of the project
- Deployment specific jobs: jobs that are required for a project to get deployed

In order for a project to get deployed in a specific environment (e.g., production, testing etc.), there are (at least) 2 required jobs:

- Build: Compiling the application (source code, libraries, configuration files, etc.) and producing shippable executable (could be of any extension such as .jar, .exe)
- Deploy: promoting the output from the Build phase in the intended environment, for instance, from Development to Testing environment.

Docker: Docker is a tool that allows developers to create, deploy, and run applications in containers. Containerization is the use of Linux containers to deploy applications. A container runs natively on Linux and shares the kernel of the host machine with other containers. It runs as a discrete process, taking no more memory than any other executable meaning it is very lightweight.

The application is divided into microservices, and each service has its own Dockerfile, so they can be launched and orchestrated from docker-compose.yml file. Access to resources (like networking interfaces and disk drives) are virtualized inside this environment, which is isolated from the rest of the system.

Kubernetes: The infrastructure of the service runs on a Kubernetes platform. The service is composed of several images, which run on the Kubernetes cluster, along with a set of configuration files which define how the service runs.

On Kubernetes, we have 2 clusters: demo and production. All the resources of the service are defined and handled via its API. So every component of the project is defined and sent to the API, in order to get deployed on Kubernetes.

4.7 Conclusions

The validation of the e-diplomata service spans a variety of different domains, both functional and non-functional. From the results of the evaluation, given above, we can draw a number of conclusions:

- The concrete implementation of a cryptographic protocol requires considerable engineering work, above and beyond any theoretical analyses of the protocol. Even the identification and selection of cryptographic libraries and parameters is a non-trivial task.
- For a service that is intended to have a wide audience, UI design is an important factor and calls for additional skills and technologies in addition to cryptography and security.
- In order for a diploma verification service, such as e-diplomata, to work in the real world, it must be connected to the institutions that award the actual diplomas. We have shown how this can happen in Greece, through the API offered by GUnet. However, actual adoption of a system like e-diplomata requires a buy-in from different institutions as well as cross-border interoperability. We are therefore currently following the Diplomas Use Case developed in the European Blockchain Services Infrastructures (EBSI).

Chapter 5

Validation Results—Use Case 4: Cardano Update System

This chapter describes how we validated the implementation of the update mechanism for decentralized software updates described in [PRI20d], which is summarized in [CKKZ20].

We designed a custom framework for validating the implementation of the update mechanism. Section 5.1 introduces this framework, which is based heavily in the property-testing [CH00] ideas.

After introducing our property testing framework, Section 5.2 presents an explanation on how we validated the requirements for the update mechanism, which are described in [PRI20a].

Finally, Section 5.3 presents the results of the property tests, in which we detail several important bugs that we could catch thanks to our high assurance testing approach.

5.1 Property testing approach

In this section we explain the trace-based property-testing approach we took to validate some of the requirements that are part of the validation criteria defined in [PRI20a, PRI20d].

The update protocol described in [PRI20d, CKKZ20] was integrated to Cardano. The architecture of Cardano can be roughly split into three layers:

- Network, which is responsible for data transmission among the Cardano nodes.
- Consensus, which runs the Cardano’s consensus algorithm. It sends and receives data from the network layer, and uses the ledger layer to validate and process transactions.
- Ledger, which processes the transactions that are part of blocks.

In particular, the ledger layer is responsible for processing the update payload, and therefore this is the layer that implements the update logic proposed in [PRI20d, CKKZ20].

The ledger layer consists of a set of pure functions that transform the ledger state. That is, given an initial state and some “payload”, these functions return either an error if the payload could not be applied, or the state that results from applying the payload to that initial state. In Haskell such functions have type:

```
:: ( ... ) => st -> d -> Either err st
```

where (...) are constraints on the state `st` and payload data `d` (i.e. which information they need to provide or which operations they should support).

The payload passed to the ledger functions could be data such as:

- UTxO transactions, which specify which inputs to spend, how much to spend, fees to pay for the transaction, to which address the output should be assigned to, etc. In this case, the ledger function in charge of processing this payload will return an error under certain conditions such as inputs not having enough funds to cover the outputs, the fees being insufficient, the signatures not verifying, etc.
- Stake pool registration transactions, which are used to register and de-register stake pools.
- Update transactions, which are transactions that contain update information such as proposals (system improvement proposals (SIP) or implementations) commits or reveals, proposals votes, etc.

In addition to the state transforming functions discussed above, the ledger layer exposes a *query API*, which allows users to ask information about the ledger state, without having to know its internal structure (and thereby breaking encapsulation). In the case of the update protocol, we implemented an API in the ledger to query different kinds of information about the update proposals (SIP or implementation). For instance:

- Is the proposal known in the ledger state?
- Is the proposal *stably submitted*? (which means that the proposal was submitted long enough ago that is now permanent on the blockchain)
- Is the proposal approved or rejected?
- Is the proposal scheduled for activation?
- Is the activation of the proposal canceled?

When validating the implementation we must test not only the state transforming part of the ledger’s API but also the query API. This brings us to the question of how can we test the update API that is part of the ledger?

The blocks stored in the Cardano chain consist only of transactions that were validated by the ledger. These transactions must have been validated by the ledger (API functions). Therefore, one could characterize all possible evolutions of the blockchain by describing sequences of ledger state transformations together with the payload that caused this transformation. We call such evolution a *trace*.

A trace consists of an initial state, followed by a sequences of actions and ensuing state. In this context, an *action* represents either:

- the payload that is applied to the ledger state, e.g. UTxO transactions, update transactions,
- some external event that causes changes to the ledger state, e.g. the blockchain “clock” ticks or the stake distribution changes. Note that the classification of an event as “external” depends on the ledger component we are focusing on. For instance, the change in stake distribution is an event that is internal to the ledger component that maintains this information, but it is external to a component like the update mechanism.

Below we give a very simplified example of an update trace, which is represented as a Haskell data-structure:

```
Trace
{ initialState: State { slot: 0, commits: [], reveals: []}
, transitions:
  [ (Tick, State { slot: 1, commits: [], reveals: []})
  , (Tick, State { slot: 2, commits: [], reveals: []})
  , (Submit A, State { slot: 2, commits: [A], reveals: []})
  , (Tick, State { slot: 3, commits: [A], reveals: []})
  , (Reveal A, State { slot: 3, commits: [A], reveals: [A]})
  ]
}
```

In this trace we start with a state in which the initial slot is 0. The trace has 4 transitions, represented as a list. In Haskell lists are delimited by square brackets, and `[]` denotes the empty list. The transitions describe an evolution where the clock ticks twice, then a proposal `A` is submitted which results in a state that has `A` in its list of commits. After one more tick and the revelation of `A` we end up in the state that is in slot 3, and has `A` as the only commit and revelation.

Besides traces, we also define the notion of a *trace fragment* (encoded by the `TraceFragment` type in our tests). A trace fragment is a sub-sequence of the transitions in a trace, where only sub-sequences of contiguous elements are considered. So in the example above, the following transitions constitute a trace fragment:

```
[ (Submit A, State { slot: 2, commits: [A], reveals: []})
  , (Tick, State { slot: 3, commits: [A], reveals: []})
]
```

but the following transition do not, as the elements do not appear contiguously in the original trace:

```
[ (Tick, State { slot: 1, commits: [], reveals: []})
  , (Submit A, State { slot: 2, commits: [A], reveals: []})
  , (Reveal A, State { slot: 3, commits: [A], reveals: [A]})
]
```

We make use of `QuickCheck` [CH00] to define the generators of traces for the update mechanism. To generate traces, we first have to define generators for random update actions. Update actions consist of update payload and events external to the update system such as changes in the stake distribution. Given a randomly generated initial state and sequence of actions, we can elaborate a trace by feeding this initial state and first action in the sequence to the ledger's update API. This results in either an error, if the generated action is invalid, or a next state otherwise. Using either the initial state if the action is invalid or the next state if the action is valid, we can repeat this procedure until all the actions in the sequence are used.

The generation of invalid actions is fundamental to our testing approach since this allows us to test that the system does not reject valid inputs. One can imagine that it would be easy to write a very safe ledger by rejecting all transactions. By generating invalid actions, we can check in our tests that these invalid actions (as reported by the ledger) are indeed invalid according to our expectations. The other advantage of generating invalid actions is that we test the system also with invalid input. For instance, if we would only generate SIP revelations only after the corresponding SIP was stably submitted, we would not be able to detect errors in which the system wrongly accepts the revelation of an unstable commit.

In our testing framework, the actions of the trace are tagged according to their validity. Using our example above, we show an example of such tagged trace:

`Trace`

```
{ initialState: State { slot: 0, commits: [], reveals: []}
  , transitions:
    [ Valid (Tick, State { slot: 1, commits: [], reveals: []})
      , Valid (Tick, State { slot: 2, commits: [], reveals: []})
      , Invalid (Reveal A)
      , Valid (Submit A, State { slot: 2, commits: [A], reveals: []})
      , Valid (Tick, State { slot: 3, commits: [A], reveals: []})
      , Valid (Reveal A, State { slot: 3, commits: [A], reveals: [A]})
    ]
}
```

In the preceding example, we can see that the first revelation of proposal `A` is marked as invalid, since there is no corresponding stably submitted commit.

For more details on the implementation of our update-system trace generation, we invite the interested reader

to look at the code which is available in the project’s repository [NAK21].

So far we have seen what traces are and briefly explained how generate them. Since all possible evolutions of the Cardano blockchain have to correspond to a trace of the ledger, we can express *safety properties* of the ledger layer by defining properties on traces. Let’s now turn our attention to the properties we would like to express.

There are three main aspects of the update system we can model using the notion of traces:

1. the states an update can be in. For instance, the SIP is committed, the SIP was approved, an implementation of the SIP was rejected, etc.
2. the possible transitions between those states (i.e. a state machine specification of the update protocol). For instance, an implementation can be revealed only if its corresponding commit is submitted.
3. the assertions about the conditions under which an update can change from one state to the another. For instance, the SIP corresponding to a given update should enter the “approved” state only if we can find a previous tally event in which there is a sufficient amount of votes for that proposal. In this assertions we can also test the query API of the ledger by contrasting the query results against what we observe in the trace. For instance, if the query API reports an SIP as revealed, then we should be able to find the revelation payload in one of the actions of the trace.

During the implementation of the property tests, we found out that these properties are easier to test if we uniquely identify the updates in the system. Furthermore, for a given update, we would like to associate the SIP and implementation proposal that allows that update to take place¹. Therefore, we introduced in our property-testing framework the concept of *update specifications*. An update specification allows to uniquely identify updates in the system, and is a concept that exist only in our testing code. An update specification contains:

- an unique identifier,
- the SIP commit,
- the SIP revelation,
- the implementation commit, and
- the implementation revelation.

The above is coded in our implementation by the following data type:

```
data UpdateSpec =
  UpdateSpec
  { getUpdateSpecId :: SpecId
  , getSIPSubmission :: Submission MockSIP
  , getSIPRevelation :: Revelation MockSIP
  , getImplSubmission :: Submission MockImpl
  , getImplRevelation :: Revelation MockImpl
  }
```

where `SpecId` is simply the set of unsigned integers, `Submission MockSIP` is the set of (mock) SIP submissions, `Revelation MockSIP` is the set of SIP revelations, and `Submission MockImpl` and `Revelation MockImpl` are the sets of implementations submissions and revelations.

To be able to test the aforementioned three types of properties, we need to extract the parts of a trace that are relevant to a given update specification. To this end we “de-multiplex” traces into a sequence of events per-update specification. Let’s take a look next at what these events are.

¹There can be multiple implementations of an SIP, but we treat them as separate updates, and of course, only one of such update will be successful.

According to the update protocol proposed in [PRI20d,CKKZ20], which we implemented, a software update undergoes several state changes: for instance the SIP of a software update might get approved, the implementation of an update might get rejected, the update might get activated on the main chain. An update event represents the change in the state of an update proposal. If a proposal changed from state s to state t , the update event corresponding to this transition is the target state t , together with the trace fragment during which the update is in that state. This is encoded in Haskell as follows:

```
data UpdateEvent
  = E { eventState :: UpdateState
      , eventFragment :: TraceFragment UpdateSUT
      }

```

In addition to the events that are associated to state transitions, for each trace we can define an “initial event”, consisting of the initial state of the update in that trace together with the trace fragment during which the update is in that state.

We have talked about update states, but what are the states (`UpdateState`) a proposal might be in? We have identified the following:

1. The update proposal is unknown to the system (referred to as `Unknown` in the implementation), and in particular denotes the fact that no SIP corresponding to this update proposal was submitted.
2. The SIP commit for the update proposal was submitted (`SIP Submitted`).
3. The SIP commit for the update proposal is stable on the chain (`SIP StablySubmitted`).
4. The SIP is revealed (`SIP Revealed`).
5. The SIP is stably revealed on the chain (`SIP StablyRevealed`).
6. The SIP is rejected (`SIP (Is Rejected)`), which means that the SIP gathered enough votes against it.
7. The SIP has no quorum (`SIP (Is WithNoQuorum)`), which means that the SIP gathered enough abstentions to go into the next voting round.
8. The SIP is expired (`SIP (Is Expired)`), which means that there was no decision (for or against) reached after all the voting rounds are over.
9. The SIP is approved (`SIP (Is Approved)`), which means that the SIP gathered enough “for” votes.
10. The SIP has no verdict (`SIP (Is Undecided)`), which means that there are still voting periods that need to be completed, and we are awaiting for votes and a verdict.
11. The implementation commit for the update proposal was submitted (`Implementation Submitted`).
12. The implementation commit for the update proposal is stable on the chain (`Implementation StablySubmitted`).
13. The implementation is revealed (`Implementation Revealed`).
14. The implementation is stably revealed on the chain (`Implementation StablyRevealed`).
15. The implementation is rejected (`Implementation (Is Rejected)`), which means that the implementation gathered enough votes against it.
16. The implementation has no quorum (`Implementation (Is WithNoQuorum)`), which means that the implementation gathered enough abstentions to go into the next voting round.

D1.3 – Use Case Validation

17. The implementation is expired (`Implementation (Is Expired)`), which means that there was no decision (for or against) reached after all the voting rounds are over.
18. The implementation has no verdict (`Implementation (Is Undecided)`), which means that there are still voting periods that need to be completed, and we are awaiting for votes and a verdict.
19. The implementation is queued (`Queued`), which means that the implementation of an update proposal got approved, and is waiting in the queue for its turn to be endorsed for activation.
20. The activation of a proposal is expired (`ActivationExpired`), which means that the implementation did not get enough endorsements, and was therefore expired.
21. The activation of a proposal is canceled (`ActivationCanceled`).
22. The implementation of a proposal is unsupported (`ActivationUnsupported`), which means that the version to that the implementation upgrades from will never be active on the chain (for instance because it was already superseded by another version). This state also means that version the of update was activated at some point, and superseded by another update (with a higher version).
23. The implementation is being endorsed (`BeingEndorsed`).
24. The implementation is scheduled for activation (`Scheduled`), which means that we are past the slot inside an epoch where the endorsements of the proposal are counted and the implementation gathered enough endorsements.
25. The update is active (`Activated`), which means that the update is the current protocol version on the chain.

Once we have the trace de-multiplexed into sequences of events per-update specification (see figure 5.1), we can check:

- That only certain transitions can happen. For instance, it is legal for a proposal to go from the scheduled to the activated state, but it cannot go from the SIP submitted to the activated state.
- Which conditions should be satisfied in that transition. For instance, if a proposal goes enters the SIP submitted state, then the first action when the proposal enters that state should be the submission of the SIP.

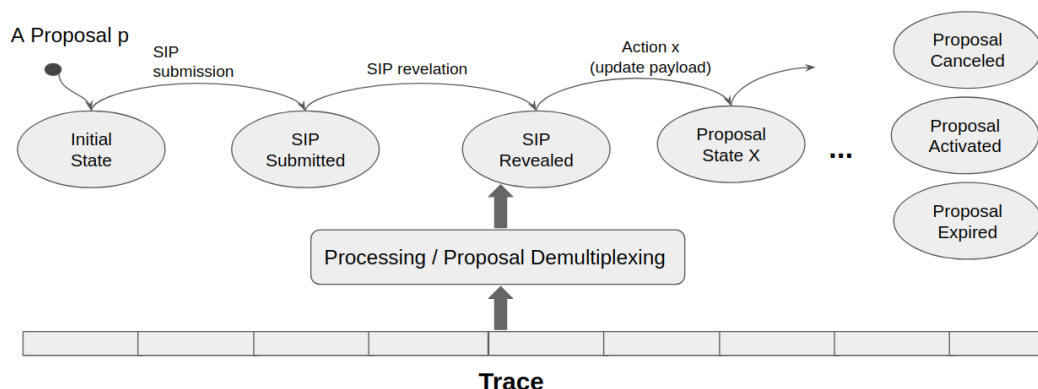


Figure 5.1: Valid state transitions of a specific update specification via trace de-multiplexing.

We will describe next how each generated trace is tested for conformance against our validation criteria.

In the test code, for each generated trace, we de-multiplex it into one sequence of events per update specification, and we check that said sequence is valid according to our specification. A sequence is considered valid if each pair of adjacent elements:

- is a valid transition
- it satisfies the conditions we expect for that transition to take place.

The notion of validity is implemented by a `validateTransition` function, which pattern matches on all the allowed transitions (and returns an error if a transition is invalid), and runs a series of checks depending on the matched transition. The definition of the function encodes the state-machine that captures the behavior of the update protocol, and has the following shape:

```
validateTransition
  (E Unknown _fragment)
  (E (SIP Submitted) fragment') = -- run some checks for this transition
validateTransition
  (E (SIP Submitted) fragment)
  (E (SIP StablySubmitted) fragment') = -- run some other check in this
  transition
-- ... etc.
```

We invite the interested reader to check the repository where the implementation of the update prototype is hosted [NAK21] for more details on how traces are de-multiplexed and fed into the function above, and also how it is implemented.

The reader familiar with tabular-specifications [BH97] might notice that the `validateTransition` function resembles an instance of such specification. Henceforth, we refer to the right hand side of `validateTransition`, i.e. the assertion, as the *transition-test*. In the following sections, when we describe how we checked certain validation criteria we will be showing and explaining parts of this function.

5.1.1 Liveness properties

We mentioned previously that since all possible evolutions of the Cardano blockchain have to correspond to a trace of the ledger, we can express *safety properties* of the ledger layer as properties on traces. The reader might ask herself, what about *liveness* properties? Here, liveness denotes the property of the system of allowing useful things to happen eventually, such as accepting proposals or votes, or activating an update proposal.

We address the problem of testing that the system allows useful things to happen by making sure that:

- No valid actions are rejected. In Section 5.2 we will show how we test that valid payload is not rejected by the system. See for instance Section 5.2.1.
- Important classes of traces can be observed in the system. For instance, the system should allow proposals to be activated. Section 5.3 provides further explanation on these kinds of tests.

5.1.2 Coverage

We have seen that we generate traces for a rather complex system, whose magnitude can be further appreciated in the state transition table of Table 5.1. However a question that might be raised is how can we have a reasonable degree of certainty that the generated traces are “complete”, under a certain notion of completeness.

The answer property-based testing provides to this question is *coverage testing*. In this approach, the idea is to identify the major scenarios we would like our tests to cover. For instance, we would like to test cases in which a proposal is waiting in the activation queue, or cases in which a vote is submitted outside the voting period. Once this scenarios are identified, we classify the traces according to the scenarios they represent (if

any). This is what in the property based testing jargon is known as *labeling*. Finally, by adding test suites that check that a certain percent of these cases are covered we can have a high (statistical) degree of certainty that the cases we are interested in will be generated during testing.

The `QuickCheck` library provides a `checkCoverage`², function that checks that all the coverage requirements are met, using a statistically sound test, and fail if they are not met. We used the default confidence parameters, where there's a 10e-9 probability that we will get a false positive regarding the coverage of a label, and a tolerance of 90% to a insufficient coverage result.

The results of our coverage tests are shown in Section 5.3.

5.2 Validation of the requirements

5.2.1 Open participation

Authorship of proposals should be preserved

In the tests we check that a proposal (SIP or implementation):

- can be marked as stably submitted only if the submission is stable on the chain
- can only transition to the revealed state from the stably submitted state

These checks are covered by the following cases of the transition validation, where the data constructor `E` can be ignored, and `do` (in this context) is a notation used to sequence assertions:

```
-- ...
validateTransition
  (E (SIP Submitted) fragment)
  (E (SIP StablySubmitted) fragment') = do
  -- The difference between the two states must be @stableAfter@.
  validateStabilityEvent fragment fragment'
-- ...
validateTransition
  (E (SIP StablySubmitted) _fragment)
  (E (SIP Revealed) fragment') = do
-- ...
```

in the code snippet above `validateStabilityEvent` asserts that at least `stableAfter` slots have elapsed between the first slot of the first fragment and the first slot of the second fragment, where `stableAfter` is the number of slots that must pass for a transaction to become stable in the chain. So this part of the state-transition property-test make sure that revelations only happen once they are stable on the chain, meaning that nobody can plagiarize the revealed content once it is observed in the chain.

In addition, we need to ensure that the key used for submitting a proposal is the same key used to reveal it. The update mechanism does not prescribe what the commit should be. In the implementation we defined the commit as follows³:

```
type Commit era =
  ( VKeyHash era -- submitter of the commit
  , Hash era (Int, VKeyHash era, Hash era (Proposal era))
  )
```

This definition means that a commit not only contains the standard commit (i.e. the hash part), but also the verification key of the submitter (`VKeyHash era`). Therefore, when checking that the submission `Commit` and

²<https://www.stackage.org/haddock/lts-17.13/QuickCheck-2.14.2/Test-QuickCheck.html#v:checkCoverage>

³where the `era` parameter can be ignored for the sake of the current discussion

the revelation `Commit` match, we make sure that the hash of the verification keys of the submission and revelation are the same, thus satisfying the validation criterion (6.a) of Section 6.3.1 found on deliverable [PRI20a], namely:

$$H(\text{salt}_1 || \text{vk}_2 || H(\text{proposal}_1)) = H(\text{salt}_0 || \text{vk}_0 || H(\text{proposal}_0)) \wedge \text{vk}_1 = \text{vk}_2$$

Valid proposal commitments are not rejected

Besides validating each event transition per-proposal, for each trace we also check that in the first event there are not unjust rejections of valid update transactions. In particular, we check that no SIP commits can be rejected *unless* the commitment signature does not verify, or the commitment was already submitted.

When the SIP that corresponds to a certain update has not been submitted to the system yet and we observe that it reports the SIP commit as invalid, then it must be because this commit has an invalid signature. Note that the first event of a proposal corresponds with the state in which said proposal is unknown to the system. We check that there are no other SIP submissions observed in that first trace fragment, and thus an SIP commit cannot be rejected because it was submitted at some earlier point in time, otherwise the state of the proposal would not have been unknown.

This requirement is tested by means of the following check that, for each update proposal, pattern-matches on the first element of an event trace (denoted as `x: _` in Haskell):

```
noUnrightfulRejectionsInUnknowState ((E Unknown fragment):_) = do
  -- The only action that we allow is implementation submissions. The update
  -- system has no way to determine if the submission corresponds to an
  -- approved SIP. This information is only available once we have a
  -- revelation.
  onlyImplementationSubmissionAllowed updateSpec fragment
  -- A valid SIP's submission corresponding to @updateSpec@ should  $\neg$  be
  -- rejected in fragment, since in @fragment@ the system under test reports
  -- that such submission didn't occur.
  forall (invalidActions fragment)
    (\act  $\rightarrow$  getSubmittedSIP act  $\neq$  Just (getSIPSubmission updateSpec)
       $\vee$   $\neg$  (signatureVerifies (getSIPSubmission updateSpec)))
  )
```

In this snippet `Just x` denotes the presence of an optional value (where `Nothing` denotes its absence). In the code above we check that if a SIP submission (`getSIPSubmission updateSpec`) is invalid, i.e. it belongs to `invalidActions fragment`, then it must be because its signature did not verify. This is expressed in our testing framework by the use of the universal quantifier `forall`: to check a predicate `p` over a certain trace (`fragment`) `t` we write:

```
forall t (\x  $\rightarrow$  p x)
```

where λ is a lambda abstraction (which allow us to write anonymous functions). In the snippet above the term to the right of the lambda abstraction arrow is a Boolean assertion that will cause the test to fail if it evaluates to false. Note that the proposal under consideration is unknown in the fragment (`E Unknown fragment`), and therefore it cannot be rejected due to the proposal being already submitted.

We perform a similar check for implementations, which can be found in the project's repository.

Valid proposal revelations are not rejected

This requirement is validated in the following transition check:

```
validateTransition
  (E (SIP Submitted) fragment)
```

D1.3 – Use Case Validation

```
(E (SIP StablySubmitted) fragment') = do
-- ...
forall (invalidActions fragment')
  (λ act → getRevealedSIP act ≠ Just (getSIPRevelation updateSpec)
  )
```

when a given proposal is in the stably submitted state, we know that there is a corresponding commit submitted to the chain that is also stable. So this means that the revelation of the proposal (`getRevealedSIP act`) must be accepted. Therefore, we check that all invalid actions in the trace fragment where a proposal is in the stably submitted state are not the revelation of that proposal.

We perform an analogous check for implementations revelations.

Valid proposal votes are not rejected

As soon a SIP proposal is stably revealed, this means that the vote period is open. From the stably revealed state, a proposal can go to only any of the following three states:

1. a verdict is reached on the proposal, but no implementation commit was submitted
2. a verdict is reached on the proposal, and an implementation commit was already submitted, but it is not stable yet.
3. a verdict is reached on the proposal, and an implementation commit is stably submitted.

in terms of test implementation, this means that we have the following three cases in our transition test:

```
validateTransition
(E (SIP StablyRevealed) fragment)
(E (SIP (Is what)) fragment') = do
  validateVerdictEvent (getSIP updateSpec)
    (getSIPVoteOf (getSIPId updateSpec))
    fragment
    (firstEvent fragment')
    what
-- ..

validateTransition
(E (SIP StablyRevealed) fragment)
(E (Implementation Submitted) fragment') = do
  validateVerdictEvent (getSIP updateSpec)
    (getSIPVoteOf (getSIPId updateSpec))
    fragment
    (firstEvent fragment')
    Approved
-- ..

validateTransition
(E (SIP StablyRevealed) fragment)
(E (Implementation StablySubmitted) fragment') = do
  validateVerdictEvent (getSIP updateSpec)
    (getSIPVoteOf (getSIPId updateSpec))
    fragment
    (firstEvent fragment')
    Approved
-- ..
-- ..
```

in each of these cases we validate the verdict event reached due to the actions occurred in the transition fragment. The `validateVerdictEvent` checks, among other things, that no votes are incorrectly rejected. The interested reader can check the code to find out more about the details of this function's implementation.

We omit the validation code for the implementation case since it is analogous to the SIP case. The code can be found in the project's repository.

Valid implementation endorsements are not rejected

We need to check that when a proposal is being endorsed there are no unexpected rejections of endorsements. A proposal being endorsed can go into the following states:

- scheduled, if the proposal gathered enough endorsements;
- expired, if the proposal did not get enough endorsement at the end of its endorsement period;
- queued, if a proposal with higher priority that can follow the current protocol version entered the activation phase; or
- canceled, if the proposal was explicitly canceled by an cancellation proposal.

these cases encoded in the `validateTransition` function:

```

validateTransition
(E BeingEndorsed fragment)
(E Scheduled fragment') = do
  validateActivationVerdictEvent updateSpec
    fragment
    (firstEvent fragment')
    Scheduled
-- ..
-- ..
validateTransition
(E BeingEndorsed fragment)
(E ActivationExpired fragment') = do
  validateActivationVerdictEvent updateSpec
    fragment
    (firstEvent fragment')
    ActivationExpired
-- ..
-- ..
validateTransition
(E BeingEndorsed fragment)
(E Queued fragment') = do
  validateActivationVerdictEvent updateSpec
    fragment
    (firstEvent fragment')
    Queued
-- ..
-- ..
validateTransition
(E BeingEndorsed fragment)
(E ActivationCanceled fragment') = do
  validateActivationVerdictEvent updateSpec
    fragment
    (firstEvent fragment')
    ActivationCanceled

```



```
-- ..
-- ..
```

In each of these cases we validate the activation verdict event using the `validateActivationVerdictEvent` function, which, in particular, checks that no endorsements are incorrectly rejected in each of the endorsement intervals of the fragment in which the proposal is being endorsed:

```
validateActivationVerdictEvent
  updateSpec endorsementsFragment tallyEvt updateState = do
  -- ...
  forall proposalEndorsementIntervals
    (λ (endorsementInterval, _, _) →
      forall (invalidActions endorsementInterval)
        noUnexpectedEndorsementRejection
    )
```

5.2.2 Decentralized decision making

Votes are correctly tallied

We have seen in Section 5.2.1 that we call function `validateVerdictEvent` in each fragment in which the voting period is open. This function also tallies the votes that are present in the trace fragment that is given to it, and compares this tally result against the tally result reported by the system.

Endorsements are correctly tallied

We have seen in Section 5.2.1 that we call function `validateActivationVerdictEvent` in each fragment in which a proposal is being endorsed. This function also counts the endorsements present in the given trace fragment, and compares this result against the verdict reported by the system.

Decisions are honored by the protocol

Our property-based tests check that only the transitions shown in Table 5.1 (Section 5.2.3) are allowed. Furthermore, when a verdict is reached we check that such verdict is correct. In particular, when we go from the “SIP stably revealed” to the “implementation (stably) submitted” state we also check, using the `validateVerdictEvent` function, that the SIP was approved. Similarly, when we go from the “implementation stably revealed” state to the “implementation is being endorsed” or “implementation is queued” states, we check that the implementation was approved, also via the `validateVerdictEvent` function. So this shows that our property test check that:

- an implementation can only be revealed if its corresponding commit is stably submitted. In turn, an implementation commit can only be submitted if the corresponding SIP was approved. When an SIP is not approved (is rejected, has no-quorum, or it expired) we can see that there are no transitions into any of the implementation states.
- an implementation can only go into any of the activation states (queued or being endorsed), only if the implementation is stably revealed and was approved (which is checked in our testing by the `validateVerdictEvent` function). If the implementation is not approved (is rejected, has no-quorum, or it expired) then we can see that this proposal does not enter any of the activation states.

5.2.3 Protocol driven

The update system runs on Cardano

We have incorporated the protocol in Cardano, and have run an end-to-end test that demonstrates the working integration on a testnet. The integration was done across different components (please refer to Privileged deliverable [PRI20c] for a detailed description of our prototype integration architecture to the Cardano node):

Ledger The update logic was incorporated to the ledger. We spent a considerable amount of work to adapt the ledger code so that it could work with any update mechanism, and those changes were merged in the master branch of the ledger, and are released into mainnet.

Consensus The consensus layer had to be adapted as well so that it could work with different update mechanisms than the one of Shelley (the release of Cardano that our integration was based).

Node The node needed to incorporate the new versions of consensus and the ledger.

Client The Cardano client is the main vehicle used to submit transactions when running low level tests like the ones we needed to run for verifying this validation criteria. We had to modify the client so that it could process the new update payload.

DevOps To deploy and run the tests on a testnet we had to elaborate a setup that developers could use to test new prototypes. These changes were also merged to the master branch of the Cardano DevOps repository.

We successfully ran a test on a testnet consisting of 8 nodes distributed across 4 countries located in 3 different continents. The test scenario involved a node submitting an SIP and its implementation, and all the nodes voting and endorsing the proposals. The proposal we activated on this testnet doubled the maximum block size.

In addition to the nodes submitting the update payload, we ran a transaction generator that submitted random UTxO transactions. The idea behind this generator was to show that the regular transactions could be successfully submitted before, during, and after an update took place.

In Figure 5.2 we show a screenshot of the demo’s output. Here we can see on the left side the part of the ledger state that shows the proposal became a candidate and was endorsed by the 8 pool nodes. On the right we can see the output of the process that periodically transfers a random amount of ADA to newly created addresses.



Figure 5.2: Scheduled update

```

Slot: 4010
Activation state
Current version: 77
Endorsed proposal: "NoProposal"
Endorsed proposal version: null
Endorsements:
Maximum block body size: 131072

[dev@dev-deployer:~/dnadales/cardano-ops]$
: {
  "value": 34,
  "address": "6038a0b2913b696d0813a815c632d22dee891be1c6dd30860d7f5d9f76"
}

Transferring 2 Lovelace to addr_test1vrhjj72usndljcenkq4jk5nvzfvpxqnxg0fz34uctam6fes4qtyld

Submit the signed transaction
Transaction successfully submitted.
Transaction submitted

{
  "38679ade3f291d6e859d05a761f319daf7019d2c1218a470843a3267da14fc3a#1"
: {
  "value": 2,
  "address": "60ef29795c84dbf96333b02b2b526c125813026643d228d7985f77a4e6"
}
}

```

Figure 5.3: Activated update

In Figure 5.3 we can see the update being activated when we are in a new epoch after the candidate proposal was endorsed by all the nodes in the network. We can see that the current version changed to the proposal's version (77), and that the maximum block size was doubled as dictated by the update proposal.

The code for the demo can be found in the Cardano DevOps repository [IOH21], commit with hash:

```
715f6766b7df5634ee430a38bf187e5d77da771e
```

The update protocol works as expected in Cardano

We run several manual and scripted tests after running the integrated system. We observed that updates could be successfully carried out (as described also in Section 5.2.3), and also that payload that violated the protocol was correctly rejected. For instance, when trying to reveal an SIP that was not stably committed, the system rejected this revelation as expected. Similarly, the system rejected votes that came outside the voting period.

Note that even though we tested the integrated system with a couple of cases, we ran hundred of thousands property-tests on the update implementation, which lies at the heart of this integration.

Update events are eventually stored in the Cardano blockchain

Our prototype incorporates the update payload in the transaction body:

```

data TxBodyRaw era = TxBodyRaw
{ inputs :: !(Set (TxIn (Crypto era))),
  outputs :: !(StrictSeq (TxOut era)),
  certs :: !(StrictSeq (DCert (Crypto era))),
  wdrls :: !(Wdrl (Crypto era)),
  txfee :: !Coin,
  vldt :: !ValidityInterval, -- imported from Timelocks
  update :: !(StrictMaybe (Update.Payload era)),
  adHash :: !(StrictMaybe (AuxiliaryDataHash (Crypto era))),
  mint :: !(Value era)
}

```

Since transactions are stored in blocks, and blocks are eventually stored in the blockchain, all the valid update events are eventually stored in the Cardano blockchain.

The proposal state transitions should be specified and verified

In Table 5.1 we specify the allowed state transitions in a proposal state, as reported by the update protocol. The meaning of these states was explained in Section 5.1. For proposals, (SIP or implementations) following there is a predefined sequence of events that relate to them: first it is committed onto the chain, after the commit is stable this proposal is revealed, once the revelation is stable, the voting period starts, and after the end of the voting period is stable the votes are tallied. These events are represented in Figure 5.4.

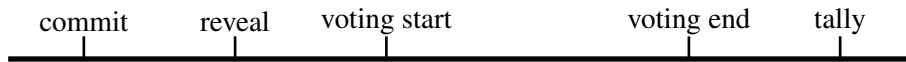


Figure 5.4: Proposal time line

An update proposal starts in the “Unknown” state, which means that the SIP commit corresponding to that update proposal has not been submitted yet. Most transitions seem sensible, however there might be some state changes that, at first sight, might not make sense to the reader. We will explain those transitions next.

- An update can go from the SIP stably revealed to the implementation (stably) submitted state because the update implementation commit could be submitted at any time, even if there is no corresponding approved SIP. This is because a commit is basically a hash, so there is no semantically relevant information that the system can validate, till that commit is revealed. So this transition can happen when an SIP was approved and the implementation corresponding to that SIP was already present in the system.
- There is no corresponding “implementation is approved” state, since once an implementation gets approved, it immediately enters the activation phase, so it either gets queued, becomes the candidate (“being endorsed”), is canceled (if a cancellation for that proposal was approved at the same time the proposal entered the activation phase), or gets marked as unsupported if the update supersedes a protocol version that can never be adopted.

The transition set specified in Table 5.1 is encoded by function `validateTransition`, which was introduced in Section 5.1,

Using property-based testing was crucial to reach a high degree of assurance on the completeness of the updates state-transition-specification. In several occasions `QuickCheck` was able to successfully find missing cases (transitions) in the specification after running the tests. In particular, some of the non-intuitive but legal transitions described above were found by our property tests.

5.2.4 Transparent and auditable

As described in [PRI20a], this validation criterion is covered by criteria 5.2.3, 5.2.2, and 5.2.2.

5.2.5 Secure activation

Secure Activation Protocols

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

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

From	To
Unknown	SIP submitted
SIP submitted	SIP stably submitted
SIP stably submitted	SIP revealed
SIP revealed	SIP stably revealed
SIP stably revealed	SIP is rejected
SIP stably revealed	SIP has no quorum
SIP stably revealed	SIP is expired
SIP stably revealed	SIP is accepted
SIP is rejected	SIP is stably rejected
SIP has no quorum	SIP has stably no quorum
SIP is expired	SIP is stably expired
SIP is accepted	SIP is stably accepted
SIP is approved	implementation is submitted
SIP is stably approved	implementation is submitted
SIP is stably revealed	implementation is submitted
SIP is stably revealed	implementation is stably submitted
implementation is submitted	implementation is stably submitted
implementation is stably submitted	implementation is revealed
implementation is revealed	implementation is stably revealed
implementation is stably revealed	implementation is rejected
implementation is stably revealed	implementation has no quorum
implementation is stably revealed	implementation is expired
implementation is stably revealed	implementation is accepted
implementation is rejected	implementation is stably rejected
implementation has no quorum	implementation has stably no quorum
implementation is expired	implementation is stably expired
implementation is accepted	implementation is stably accepted
implementation is stably revealed	implementation is queued
implementation is stably revealed	implementation is being endorsed
implementation is stably revealed	implementation activation was canceled
implementation is stably revealed	update is unsupported
implementation is being endorsed	implementation is scheduled
implementation is being endorsed	implementation activation expired
implementation is being endorsed	implementation is queued
implementation is being endorsed	implementation activation was canceled
implementation is queued	implementation activation was canceled
implementation is queued	update is unsupported
implementation is queued	implementation is being endorsed
implementation is scheduled	implementation was activated
implementation was activated	update is unsupported

Table 5.1: Update proposal allowed state changes

Our first activation protocol requires the structure of the current and the updated blockchain to be very similar (only the structure of the blocks can be different) but it allows for an update process more simple and efficient. The second activation protocol that we propose is very generic (i.e., makes few assumptions on the similarities between the structure of the current blockchain and the updated blockchain). The drawback of this protocol is that it requires the new blockchain to be resilient against a specific adversarial behavior and requires all the

honest parties to be online during the update process. However, we show how to get rid of the latest requirement (the honest parties being online during the update) in the case of proof-of-work and proof-of-stake ledgers. The interested reader can find more details on this topic, as well as all the relevant *formal proofs* that validate our results, in our paper [CKKZ20].

The adoption threshold is honored

As we shown in Table 5.1, an update proposal can only be activated after being scheduled, and our property tests validated that our implementation satisfies this property. Let’s turn our attention next to the conditions under which a proposal can become scheduled.

We have that, according to our transitions specification, an update can be marked as scheduled only after being endorsed. When this transition happens, in the tests we make sure that in the fragment that precedes the change to the “scheduled” state, we have enough stake endorsing this proposal:

```

validateTransition
(E BeingEndorsed fragment)
(E Scheduled fragment') = do
  validateActivationVerdictEvent updateSpec
    fragment
    (firstEvent fragment')
    Scheduled

```

In the snippet above, the `validateActivationVerdictEvent` function checks that if the update state was set to “scheduled”, then we should observe enough endorsements in the last endorsement interval of `fragment`. See [PRI20d], for an explanation of endorsements interval, and [NAK21] for details on how function `validateActivationVerdictEvent` is implemented.

The update system preserves history across hard fork boundaries

As described in [PRI20a], this validation criterion is covered by criterion 5.2.3.

5.2.6 Performant and scalable

Transaction throughput is not significantly affected

An important measurement of a blockchain system’s 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 section we estimate the impact on performance that the proposed update mechanism will have on a system’s *TBPS*. Our estimations are based on *worst case scenarios*, which allow us to determine upper bounds for the performance impact of the update mechanism.

Given a payload size in bytes (*psize*) that needs to be stored in a blockchain, the blockchain’s throughput measured in *TBPS*, and the duration of the update process (*duration_u*), we can calculate the percentage of the system’s *TBPS* (*usage_{pct}*) that will be used by the update payload as follows:

$$usage_{pct} = 100 \frac{psize}{TBPS \cdot duration_u}$$

An update consists of several phases (ideation, implementation, approval, and activation). In each phase, there are three types of messages being sent: commits, reveals, and votes.

Before the voting phases, where votes can be cast by the participants, each update requires only two messages spread across two stability windows, needed for transactions to stabilize in the chain. These stability windows

are quite large, e.g. in Cardano the stability window is 1 day and a half. As a result, only two messages need to be transmitted for the commit-reveal phase over a large period of time, which means that a blockchain system can easily handle this. This leaves us with the voting phase as the sole source for performance degradation that can be caused by the update mechanism.

In addition, note that we only need to consider the additional load introduced by the update mechanism during a single phase. It is in the voting period of each phase where the system should be able to handle the additional load, since the update mechanism introduces very little load between voting phases.

We define the worst-case scenario for a voting period in terms of

- number of participants (n_p), e.g. voters (note that in the worst case scenario everybody will vote, regardless of their stake, which means that the stake distribution is irrelevant for this analysis)
- number of update proposals being voted at the same time (n_c), during the same period (note that in the worst case scenario multiple update proposals will coincide in the start and end of the voting period, otherwise the system would have a larger time interval to distribute the load).
- number of time a participant changes her vote (n_r), per-update proposal

Then, we can calculate the worst case scenario for the number of bytes that need to be transmitted as part of the vote payload ($psize_v$) as:

$$psize_v = s_v n_p n_r n_c$$

The size in bytes for s_v was obtained by calculating the size CBOR encoding [BH13] of the vote payload of our prototype. This payload includes:

- The hash of the voted SIP. We use 32 bytes hashes, so considering the 1 byte CBOR tag this gives us a total 33 bytes.
- The confidence (for, against, abstain), which can be encoded in 1 byte (which also included the CBOR tag).
- The key of the voter. We use 32 bytes keys, so this give us a total of 33 bytes, when we consider the CBOR tag.
- The vote signature. We consider 64 bytes signatures, which are accompanied by a 32 bytes key. This results in $64 + 32 + 1$ bytes required for the signature.

So a vote requires in total 164 bytes.

Table 5.2 shows the results of the worst case analysis for different parameter values, where $duration_v$ is the number of voting days, which was used to calculate the voting period duration. For this analysis we use the *TBPS* that Cardano currently achieves in mainnet: 3.2 Kb/s. This number is obtained from dividing the maximum block body size, 64 Kb, by the number of seconds per-block, 20.

Figure 5.5 shows the worst case usage as a function of the number of participants, assuming 10 concurrent update proposals, a 7 day voting period, and each participant changing her vote twice.

Looking at Table 5.2 and Figure 5.5, we can see that the usage percentage scales linearly in the number of participants⁴, i.e., a 10 times increase in the number of participants will increase 10 times the required usage percent. We can see that the impact of the update protocol on the system's performance is negligible up to 100,000 participants and 1 proposal being voted.

However, in spite of the usage percentage being a linear function of the number of participants, when we pass the million participants or 100,000 participants vote on 10 proposals at the same time, we start seeing a considerable impact of the update protocol on the system's performance. In case 1 million participants would vote on 10 proposals at the same time, the system could not process the extra payload. Nevertheless, this would

⁴Note the logarithmic scale used in Figure 5.5

n_p	n_r	n_c	$duration_v$	$usage_{pct}$
1000	2	1	7	0.017
1000	2	5	7	0.083
1000	2	10	7	0.166
10000	2	1	7	0.166
10000	2	5	7	0.828
10000	2	10	7	1.655
100000	2	1	7	1.655
100000	2	5	7	8.277
100000	2	10	7	16.555
1000000	2	1	7	16.555
1000000	2	5	7	82.773
1000000	2	10	7	165.546
1000000	2	10	14	82.773
1000000	2	10	30	38.627
10000000	2	1	7	165.546
10000000	2	5	7	827.729
10000000	2	10	7	1655.458

Table 5.2: Worst-case analysis TBPS for a voting period

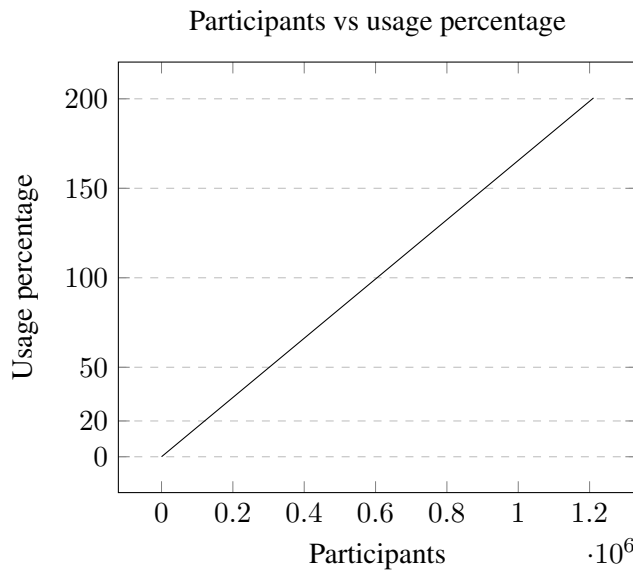


Figure 5.5: Worst case scenario analysis for system’s usage with 10 concurrent update proposals

require that the worst case conditions we assume in this analysis being met: 10 SIP’s being voted at the same time over the period of 7 days, where each participant votes twice. In practice we will have a much higher voting duration, and not all proposals will overlap exactly in their voting period, which means that the additional payload can be spread across a much larger time interval.

There are several alternatives to allow a blockchain system to accommodate more participants or concurrent update proposals:

- Use of expert pools. By having the participants delegate their voting rights, the number of voters can be substantially reduced, while increasing voter’s turnout.
- Increase in the duration of the voting period. We have used a very short voting duration (7 days). In

practice update proposals would require that voters are given a much longer time to decide, specially if the update proposal has a large impact on the network.

- Increase the maximum block size that the protocol allows, which will result in a larger TBPS. Note also that the current maximum block size in the Cardano protocol, which we use for our worst case analysis, is quite small but sufficient for accommodating the blockchain current usage. If the blockchain would need to process additional payload, experiments show that Cardano can increase its maximum block size up to 1Mb, which would result in a tenfold increase in the TBPS, and therefore a tenfold increase in the number of participants that the update protocol can accommodate.

The back-of-the-envelope calculation described in this section was put to the test against the data generated by actual runs of the update protocol on a testnet. We present next the setup of our experiments and the results we obtained.

Our goal was to measure the impact of the voting process on the blockchain's transaction throughput. To this end, we fixed the protocol parameters of the network, and ran experiments in which we gradually increased the number of participants that voted on a proposal.

We used the testnet setup described in Section 5.2.3, which consisted of 10 nodes running a stake pool each. The nodes were deployed on AWS machines, distributed across 4 countries and 3 continents.

The protocol parameters were chosen in such a way that they resembled the mainnet conditions, while allowing us to run experiments in a reasonable amount of time. This means that in our testnet:

- like the Cardano network, blocks were produced every 20 seconds.
- unlike the Cardano network, the stability window was set to 36 minutes (in Cardano this is 36 hours).

Using these parameters we could run each experiment in about 2 hours, depending on the number of voters, since more voters required more time to register staking keys, which we used as voting keys.

In each node that ran a stake pool (pool node) we ran:

- 50 threads that were constantly submitting UTxO transactions to the network.
- a varying number of threads that submitted the votes on a given SIP proposal.

Each experiment carried out the following steps:

1. Start the UTxO transaction submission threads.
2. Register stake keys that will be used to vote on the SIP.
3. Submit an SIP commit, wait for it to become stable on the chain, and then reveal the SIP.
4. Wait for the voting period for the submitted SIP to open and then have all the participants in all the nodes vote in parallel.
5. After all participants voted, check that the number of voters matched the expected number of participants to make sure that each vote went through.

After running the experiment we analyzed the script and node logs to count the number of UTxO transactions that were submitted during the voting period. Table 5.3 shows the results we obtained, and this information is also plotted on Figure 5.6.

The results shown in Table 5.3 coincide with our back-of-the-envelope calculation regarding the impact of the voting phase on the transaction throughput. We can observe a linear relationship between TPS (or equivalently usage) and number of participants.

Voters per node	TPS	Usage %
1	16.64764543	0.036
10	16.5933518	0.362
100	16.04376731	3.662
250	15.14626039	9.051
500	13.64155125	18.087
750	12.13795014	27.115
1000	10.63822715	36.121
1250	9.127977839	45.189

Table 5.3: Experimental results on TPS

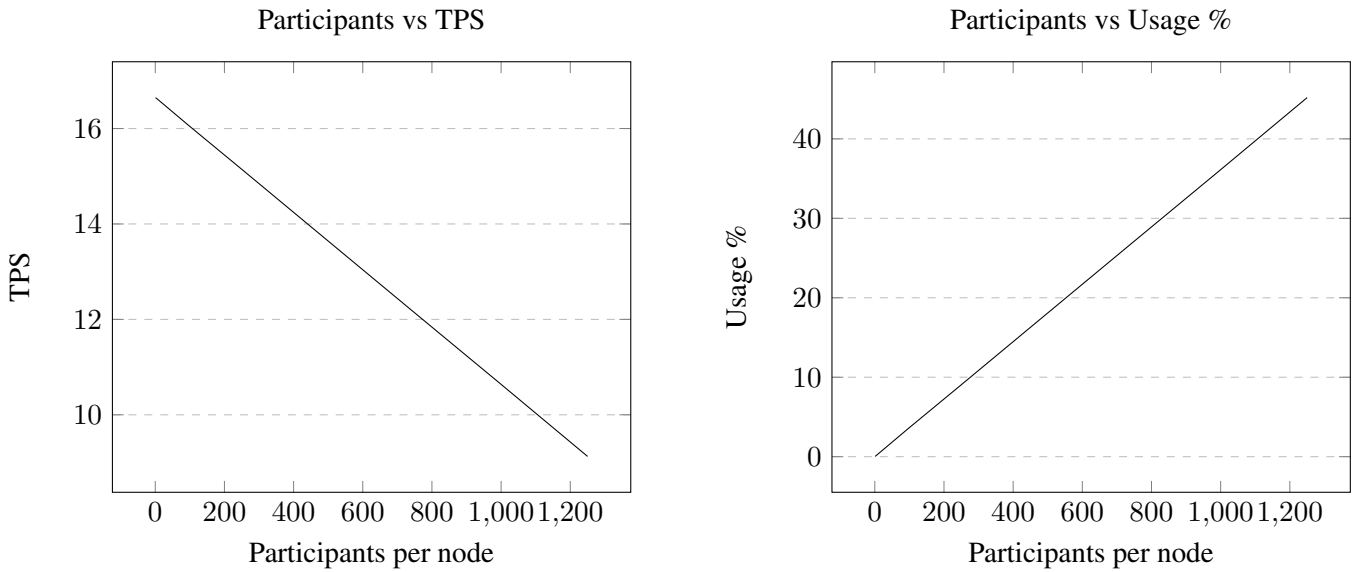


Figure 5.6: Experimental results on TPS

Transaction latency analysis

Additionally to the throughput, in our testnet experiments, we have measured the impact on the *transaction latency*. By “latency”, we mean the *average end-to-end time* from the transaction submission until it appears in the ledger. To this end, we have used the same setup as in the throughput experiments. In other words, a 10 node geographically distributed network, where 50 threads per node constantly submit transactions and we utilize a varying number of voters that try to vote in a 30 minute voting period window.

In table 5.4, we record our main results. Initially, with a single voter per node and as we are submitting transactions almost at maximum capacity, we observed an average latency of around 25 seconds. Then, as we gradually increase the number of voters, we observe the latency increasing. In the rightmost column of table 5.4, we have recorded the factor by which this latency-increase takes place for each increase of the voters load. In figure 5.7, we observe a quadratic increase of the latency with the increase of the load. Note that when the voting period is open, all participants vote at the same time. As a consequence a queue is formed which causes the latency to increase. The size of this queue will be dependent on the number of participants. Clearly, all participants voting at the same time is unrealistic, but even in this extreme situation we can accommodate a large number of votes (12500) in 30 minutes. Moreover, apart from the simultaneous voting, a more realistic voting duration (e.g., of some days, or weeks) would make the latency even more tolerant to the increase of the load.

Voters per node	(Average) Latency (sec)	Times increase
1	25.90	1
10	25.88	1
100	25.36	1
250	46.73	1.8
500	105.99	4.1
750	214.54	8.3
1000	397.76	15.3
1250	692.38	26.7

Table 5.4: Experimental results on Latency

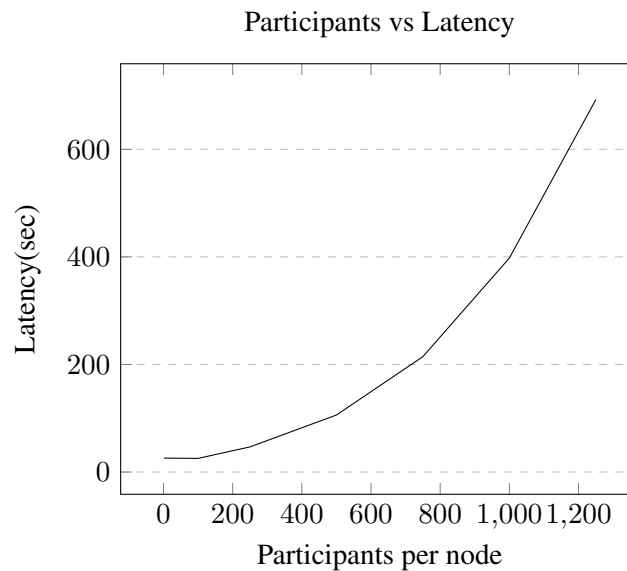


Figure 5.7: Experimental results on Latency

Low-impact on processing time

Clearly the most processing intensive task of the update mechanism is the tally phase. It is the phase where all the collected votes are counted in order to reach a decision for a specific proposal.

We start with a theoretical time complexity analysis where we assume a worst-case scenario, where we have n participants that all of them vote by submitting a single vote. Also we assume that we have a single proposal, so that within a voting period, the number n of participants coincides to the number of submitted votes.

In the following, we try to break up the operations during the tally phase. In the heart of the tally phase lies the following function call, which is called for each proposal.

Listing 5.1: Tally phase initial function call

```

tallyStake confidence result ballot stakeDistribution adversarialStakeRatio
=
if stakeThreshold adversarialStakeRatio (totalStake stakeDistribution)
<
stakeOfKeys votingKeys stakeDistribution
then Just result
else Nothing
where

```

```
votingKeys = Map.filter (== confidence) ballot
```

Function `stakeThreshold` is constant whereas `Map.filter`⁵, is $O(n)$ so `votingKeys` is $O(n)$, where n is the number of votes, which as we have said coincides to the number of participants. At this point, we have a single pass (loop) over n votes.

The remaining function used in the definition of `tallyStake`, `stakeOfKeys`, calculates the stake associated to the given key-map, and is defined as follows:

Listing 5.2: Code example

```
stakeOfKeys
keyMap
StakeDistribution
{ stakeMap
}
= Map.foldl (+) 0 (stakeMap `Map.intersection` keyMap)
```

The *intersection* function in the worst-case is $O(n)$ ⁶. Therefore this is a second pass (loop) over the data of length n . Finally, we call `foldl` with a constant time operation, `+`, which means that this call is also $O(n)$ ⁷. This is a third pass (loop) over the data of length n . Thus from the above analysis we see that we have for a single proposal a call of `tallyStake`, where in each such call we have three passes over the data of length n . So in total for a single proposal we do 3 passes over the data of length n . That is $3n$ operations, which means that the tally time complexity is $O(n)$.

This result is also confirmed by the experimental evaluation shown in Figure 5.8, where we see that the processing time increases linearly with the number of participants. In addition, we see that it takes almost one tenth of a second to process the votes of 1 million participants. These results correspond to a single-core execution of the tally algorithm on a i7 CPU laptop with 32GB of RAM.

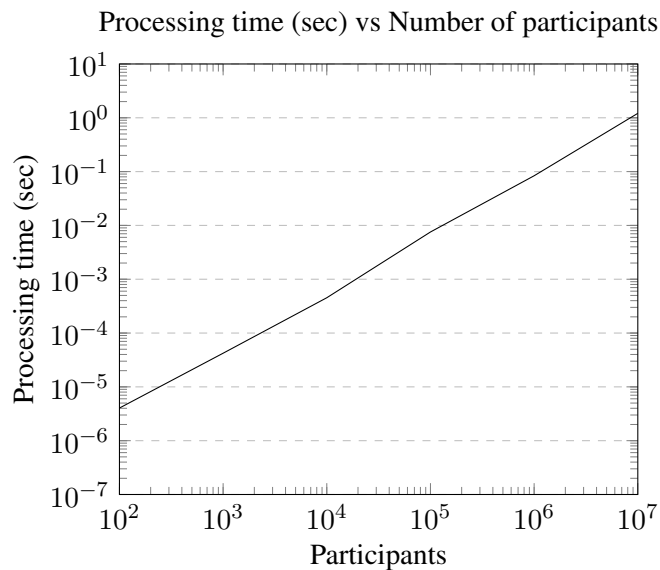


Figure 5.8: Worst case scenario analysis for tally phase processing time

⁵<http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Strict.html#g:25>

⁶<http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Strict.html#v:intersection>

⁷<http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Strict.html#v:foldl>

Low-impact on memory usage

Finally, we present the measurements of the memory consumption during the tally phase. Again as the graph in Figure 5.9 shows, the memory allocated scales linearly in the number of participants. Moreover, our measurements show that the allocated memory essentially corresponds to the space required for storing the 256 bit hashes of the public keys of the participants in a Haskell map structure⁸.

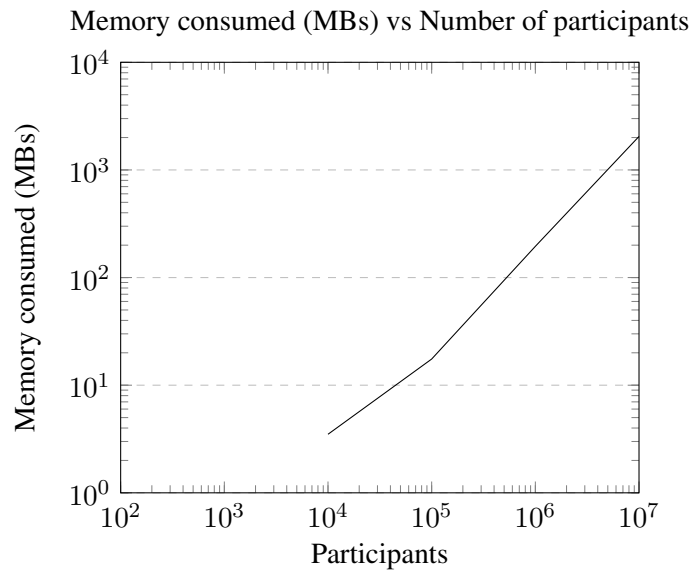


Figure 5.9: Worst case scenario analysis for tally phase consumed memory

The system should scale

Based on the results shown in the previous sections, we can see that overall the system scales quite efficiently with the number of participants. With respect to transaction throughput, processing time and memory consumption, we have observed a linear relationship with the number of participants, while for transaction latency our measurements showed a slow quadratic increase under the (unrealistic) burst condition of simultaneous voting, which the system can handle.

5.2.7 Metadata driven

The voting period length is honored

In each transition where a voting period can take place, we use function `validateVerdictEvent` to make sure that the voting period length is honored by the implementation of the update protocol. This function computes the voting intervals based on the voting period duration of the proposal. Using these interval, the voting results computed by the testing code is contrasted with the voting results reported by the system. If there would be a mismatch between the voting period duration used by the test and the implementation, assuming a sufficient coverage of our property tests, then we should be able to find a counterexample where due to the different voting intervals there would be a mismatch between either:

- the voting results, or
- the validity of a vote

⁸<http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Strict.html>

Version dependencies are honored

We guarantee that for all updates that enter the “implementation is being endorsed” phase, the protocol version and id that the candidate supersedes coincides with the current protocol version and id. So in the transitions that lead to the `BeingEndorsed` state, we check:

```

getCurrentProtocolVersion (firstState fragment')
  ==! supersedesVersion (getProtocol updateSpec)
getCurrentProtocolId (firstState fragment')
  ==! supersedesId (getProtocol updateSpec)

```

Priorities are honored

As with the check in the previous section, in all proposal state-transitions that lead to the `BeingEndorsed` state, we check that the new candidate has the highest priority among the queued proposals during the trace fragment in which said proposal is being endorsed:

```

validateTransition
  (E Queued _fragment)
  (E BeingEndorsed fragment') = do
  -- ..
  forall
    (candidatesIn fragment' `withIdDifferentFrom` updateSpec)
    (\protocol →
      version (getProtocol updateSpec) < version protocol)

```

The assertion above checks that all the queued candidates in `fragment'`, which are not the update proposal being endorsed, have a protocol version higher than the protocol version of the proposal being endorsed.

The deployment window is honored

As explained in Section 5.2.2, the test code computes the endorsement results and checks them against the result reported by the implementation. To compute the endorsement results, the test code uses the deployment window of the candidate proposal. Therefore, if there was a discrepancy between the deployment window used by the implementation and the expected deployment window (given by the proposal’s metadata), the tests would find a counterexample showing a mismatch in the endorsement results.

5.2.8 Update logic consistency

Consistent update logic

The test code checks that a proposal is activated when ([PRI20d]):

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

Let’s dig into the details of how the test code performs these checks.

The transitions specification of Table 5.1, which we property-test that the implementation satisfies, only allows a proposal to enter the activation phase if the implementation was previously approved. When a proposal

enters the activation phase for the first time it can only be in one of the “queued” or “being endorsed” states: if the proposal has the highest priority among other proposals in the queue and it can follow the current version, then its state will be “being endorsed”, otherwise its state will be “queued”. Similarly, by inspecting this table we can see that we can enter the implementation phase only when the corresponding SIP was approved, as we explained in Section 5.2.2.

Dependency and conflict resolution is achieved by protocol version checking: a proposal declares the protocol version it supersedes. This ensures that an update has only one dependency: the only version to which it can be applied. Conflicts between two updates that want to upgrade to the same version are resolved by choosing the update with the highest priority, which is determined by the protocol version (the lower the version the higher the priority). In the test code we make sure that an update can only be endorsed if the protocol version and id that it declares to supersede coincides with the current version, as explained in Section 5.2.7.

In the test code we also check that an endorsed proposal has the highest priority among competing proposals, as explained in Section 5.2.7.

Finally, we can see that the transitions specification of Table 5.1, only allows a proposal to be activated after being scheduled. When a proposal enters the scheduled state, we check, using function

```
validateActivationVerdictEvent
```

that the proposal got enough endorsements.

```
validateTransition
  (E BeingEndorsed fragment)
  (E Scheduled fragment') = do
    validateActivationVerdictEvent updateSpec
      fragment
      (firstEvent fragment')
      Scheduled
-- ...
```

5.3 Validation results

We implemented and ran our property tests. The test that checks our validation criteria according to the methodology described before (“Changes in the state of update proposals are valid”) passed after running 100,000 test cases, each of which tested a trace of length between 0 and 1,000. Figure 5.10 shows the output of the execution of our property tests. In particular, the state change validation tests ran the 100,000 cases in about 3.6 minutes. Note that the large amount of test cases should be typically done in CI, upon a pull request, when we want higher assurance on the correctness of the implementation. While maintaining the code, a smaller number of tests cases can be run.

The screenshot of Figure 5.10 also shows a number of tests that output “OK, failed as expected”. This is QuickCheck terminology⁹ and should be interpreted as “the test case appeared in the trace as expected”. These tests provide a way of testing the coverage of our trace generation. For instance, we want to make sure we generate traces where SIP are expired, or where implementations are waiting in the activation queue, but without explicitly defining instances of these traces by hand. These tests make sure that these cases are actually covered. If a particular case is found (e.g. “Implementations are expired”), QuickCheck will regard this as a successful test, whereas if such case is not found it will report a failure.

The tests whose output is shown in Figure 5.10, validate that the update system allows certain important scenarios to happen. Next to these basic coverage tests, we also classified the traces according to the scenarios they represented, and used QuickCheck to make sure that the generators always generated said scenarios with

⁹It really means that Quickcheck found a counterexample violating a certain property.

```

777 Property tests
SIP's are expired: OK
+++ OK, failed as expected. Falsified (after 4 tests and 23 shrinks):
SIP's are rejected: OK
+++ OK, failed as expected. Falsified (after 4 tests and 32 shrinks):
SIP's get no-quorum: OK (0.02s)
+++ OK, failed as expected. Falsified (after 16 tests and 35 shrinks):
SIP's are approved: OK (0.02s)
+++ OK, failed as expected. Falsified (after 4 tests and 29 shrinks):
Implementations are expired: OK (0.06s)
+++ OK, failed as expected. Falsified (after 69 tests and 31 shrinks):
Implementations are rejected: OK (0.06s)
+++ OK, failed as expected. Falsified (after 52 tests and 38 shrinks):
Implementations get no-quorum: OK (0.07s)
+++ OK, failed as expected. Falsified (after 127 tests and 30 shrinks):
Implementations are activated: OK (0.25s)
+++ OK, failed as expected. Falsified (after 345 tests and 48 shrinks):
Implementations are queued: OK (0.02s)
+++ OK, failed as expected. Falsified (after 2 tests and 38 shrinks):
Updates are discarded due to being expired: OK (1.18s)
+++ OK, failed as expected. Falsified (after 1991 tests and 28 shrinks):
Updates are discarded due to being unsupported: OK (0.22s)
+++ OK, failed as expected. Falsified (after 247 tests and 55 shrinks):
Changes in the state of update proposals are valid.: OK (221.16s)
+++ OK, passed 100000 tests.
Data is correctly serialised to CBOR: OK (3.39s)
+++ OK, passed 5000 tests.

All 32 tests passed (226.52s)

```

Figure 5.10: Property tests results

a high degree of probability. Figures 5.11 and 5.12 show the part of the output of our coverage tests run. There we can see for instance that:

- in more than 96% of the traces, a trace is generated where the SIP revealed when its corresponding commit is not stable in the chain.
- in more than 68% of the traces, an implementation is voted when the SIP is stably revealed, but no corresponding implementation has been revealed yet (which cannot be the case since the SIP was not yet approved).
- in more than 0.0823% of the traces there is a proposal whose activation is canceled. This might not seem like a lot of cases, but one has to consider the large amount of traces that are generated. For instance, we test our properties in 100,000 traces, which means that about 82 traces will contain an example of a proposal that is canceled. This is already quite impressive if one considers the fact that we did not guide the generation of traces, i.e. actions are randomly generated, and the fact that we need a lot of events occurring at the right time for a proposal to be canceled. For instance we need at least two proposals with the same version, which have to go through two approval phases, and have to be approved at about the same time so that the proposal that is approved last can cancel the proposal that is approved first.

We have identified 106 important scenarios. For the sake of brevity we do not describe them in this report, and invite the interested reader to look at the implementation for the full details.


```

OK (747.13s)
+++ OK, passed 409600 tests:
 96.2341% Implementation is revealed when SIP commit is not stable
 96.2332% SIP is revealed when SIP commit is not stable
 96.2302% Implementation is voted when SIP commit is not stable
 96.1914% SIP is voted when SIP commit is not stable
 93.4824% SIP is revealed when SIP is unknown
 93.4661% Implementation is voted when SIP is unknown
 93.4565% SIP is voted when SIP is unknown
 93.4346% Implementation is revealed when SIP is unknown
 80.1011% Implementation is voted when the SIP is revealed
 80.0869% SIP is voted when the SIP revealed
 80.0598% SIP is revealed when the SIP is already revealed
 80.0547% Implementation is revealed when the SIP is revealed
 79.7781% SIP is voted when the SIP is not yet revealed
 79.7063% Implementation is revealed when the SIP is not yet revealed
 79.6580% Implementation is voted when the SIP is not yet revealed
 68.4961% SIP is revealed when the SIP is already stably revealed
 68.4775% Implementation is voted when the SIP is stably revealed
 68.4722% Implementation is revealed when the SIP is stably revealed
 33.3035% Implementation is voted when a verdict on its SIP was reached
 33.2842% SIP is submitted when a verdict on it was reached
 33.2676% SIP is revealed when a verdict on it was reached
 33.2668% Implementation is revealed when a verdict on its SIP was reached
 33.2610% SIP is voted when a verdict on it was reached
 26.6541% SIP is revealed when a verdict on it was stably reached
 26.6331% SIP is voted when a verdict on it was stably reached
 26.6248% SIP is submitted when a verdict on it was stably reached
 26.6140% Implementation is revealed when a verdict on its SIP was stably reached
 26.6116% Implementation is voted when a verdict on its SIP was stably reached
 12.8357% Implementation is submitted when it is already in the stably submitted state

```

Figure 5.11: Coverage test results

Using `QuickCheck` to find examples of traces is also useful to investigate the behavior of the protocol through the examples that this tool finds. For instance, when `QuickCheck` found an example of a trace where a proposal was activated, we realized that the protocol allowed an implementation to be committed before its corresponding SIP was approved. Although counterintuitive, this behavior is sensible since a commit is basically a hash, and as such, there is no semantically relevant information that the protocol can check, other than the fact that the commit was not already submitted. An example contains information about the participants, their stake distribution, the consensus protocol parameters such as the stability-window, etc. In case a test failure is found, this example becomes our “counterexample”. Below we show some parts of the example that demonstrates that our protocol allows proposals to be activated, in particular the stability window tsK , the maximum number of voting periods, the slot at which the tests start, the number of slots per-epoch, the list of participants and the stake they possess, and the list of actions that are run in the trace:

```

UpdateTestSetup
{ tsK = BlockNo { unBlockNo = 1 }
, tsMaxVotingPeriods = VotingPeriod { unVotingPeriod = 1 }
, tsCurrentSlot = SlotNo { unSlotNo = 0 }
, tsSlotsPerEpoch = SlotNo { unSlotNo = 5 }
-- ..
, tsParticipants =
[ ( Participant
  ( ParticipantId 3 )

```

```

0.1714% Implementation is submitted when it is already in the submitted state
0.0823% Canceled
0.0728% Unsupported
0.0615% SIP is submitted when its implementation is canceled
0.0601% SIP is revealed when its implementation is canceled
0.0588% SIP is voted when its implementation is unsupported
0.0583% Implementation is revealed when its implementation is canceled
0.0581% SIP is revealed when its implementation is unsupported
0.0579% SIP is voted when its implementation is canceled
0.0576% Implementation is voted when its implementation is canceled
0.0566% Implementation is submitted when its implementation is unsupported
0.0566% Implementation is voted when its implementation is unsupported
0.0554% Implementation is revealed when its implementation is unsupported
0.0554% SIP is submitted when its implementation is unsupported
0.0549% Implementation is submitted when its implementation is canceled
0.0356% Activation expired
0.0273% Implementation is revealed when its implementation is activation expired
0.0273% SIP is submitted when its implementation is activation expired
0.0261% Implementation is voted when its implementation is activation expired
0.0256% SIP is voted when its implementation is activation expired
0.0251% SIP is revealed when its implementation is activation expired
0.0239% Implementation is submitted when its implementation is activation expired

```

Figure 5.12: Coverage test results (cont)

```

, 2
)
]
, tsActions =
[ SIPCommit
  ( SpecId { unSpecId = 1 } )
, JustTick
, JustTick
, SIPReveal ( SpecId { unSpecId = 1 } )
, JustTick
, JustTick
, ImplCommit
  ( SpecId { unSpecId = 1 } )
, JustTick
, SIPVote
  ( MockVote
    { voteVoterId = MockVoterId
      { unMockVoterId = ParticipantId { unParticipantId = 3 } }
    , voteCandidate = MPId 1
    , voteConfidence = For
    , voteSignatureVerifies = True
    }
  )
, JustTick
-- ...

```

In addition to the property tests, we have written and run a series of *unit tests* to check and also to illustrate different aspects of the update protocol. Below we show a snippet of a test-case that checks that an implementation is approved:

```
-- Precondition: the update SIP should be stably approved
```

D1.3 – Use Case Validation

```
approveImplementation :: UpdateSpec → TestCase
approveImplementation update = do
  stateOf update `shouldBe` SIP (IsStably Approved)
  submit `implementation` update
  tickTillStable
  reveal `implementation` update
  stateOf update `shouldBe` Implementation Revealed
  tickTillStable
  approve `implementation` update
  stateOf update `shouldBe` Implementation StablyRevealed
  tickFor (Proposal.votingPeriodDuration (getImpl update))
  tickTillStable
  stateOf update `shouldBe` Implementation (Is Approved)
```

Figure 5.13 shows the results of the unit tests results. The complete set of unit tests can be found in the project repository [NAK21].

The screenshots of Figures 5.10 and 5.13 show all the tests green. This was of course not always the case. During the implementation of the prototype, the property tests allowed us to discover several bugs, which we will discuss next.

Zero threshold We implemented a function that calculated the approval threshold for update proposals (SIP or implementations) based on the adversarial stake ratio and the total stake. This function was defined as follows:

```
stakeThreshold r_a totalStake =
  round (1/2 * (r_a + 1) * fromIntegral totalStake)
```

Our tests found that proposals were approved without votes, and activated without endorsements. This is because this function returns 0 when r_a is 0 and the total stake is 1. Even though this is a rare corner case, it could have been a serious problem in a permissioned blockchain in which one node was in charge of carrying out updates.

Cutoff slot The cutoff slot is the last slot in which endorsements are considered for an epoch. We calculated the cutoff slot as:

```
nextEpochFirstSlot - 2 * stableAfter
```

The problem is that this calculation causes the protocol to fail if

```
slotsPerEpoch < 2 * stableAfter
```

Our tests managed to detect this error.

Entering the endorsement period at the wrong time A proposal can be endorsed only when ([PRI20d]), after being approved:

- Has the highest priority among all the queued proposals
- The protocol version it supersedes equals the current version
- The protocol version id it supersedes equals the current version id

However, our tests found that in the implementation we did not consider the protocol id. This error was found by the following assertion in our transition test:

```

💡 Ideation phase unit tests
  SIP approval succeeds: OK
    +++ OK, passed 1 test.
👍 Approval phase unit tests
  Implementation gets expired: OK
    +++ OK, passed 1 test.
  Implementation gets no quorum: OK
    +++ OK, passed 1 test.
  Implementation gets rejected: OK
    +++ OK, passed 1 test.
  Implementation gets accepted: OK
    +++ OK, passed 1 test.
  Implementation gets accepted in the second voting round: OK
    +++ OK, passed 1 test.
  Votes from previous voting periods are not carried over: OK
    +++ OK, passed 1 test.
  Implementation reveal without approved SIP fails: OK
    +++ OK, passed 1 test.
  Revelation of an unstable submission fails: OK
    +++ OK, passed 1 test.
  Votes before the start of the voting period are rejected: OK
    +++ OK, passed 1 test.
  Votes before the end of the voting period are rejected: OK
    +++ OK, passed 1 test.
⚡ Activation phase unit tests
  Protocol version is changed once: OK
    +++ OK, passed 1 test.
  Protocol version is changed twice: OK
    +++ OK, passed 1 test.
  Update is preempted by one with higher priority: OK
    +++ OK, passed 1 test.
  Old proposal with same version gets canceled: OK
    +++ OK, passed 1 test.
  A proposal with lower priority (higher version) gets queued: OK
    +++ OK, passed 1 test.
  A proposal without its dependencies met gets immediately queued: OK
    +++ OK, passed 1 test.
  A candidate without enough endorsements is not adopted: OK
    +++ OK, passed 1 test.
  Endorsements of non-candidates are not allowed: OK
    +++ OK, passed 1 test.

```

Figure 5.13: Unit test results

```

validateTransition (E (Implementation StablyRevealed) fragment)
  (E BeingEndorsed fragment') = do

```

```
-- ...
getCurrentProtocolVersion (firstState fragment')
  ==! supersedesVersion (getProtocol updateSpec)
getCurrentProtocolId (firstState fragment')
  ==! supersedesId (getProtocol updateSpec)
```

Votes on rejected or expired proposals We found out that the implementation allowed votes on rejected or expired proposals. This was detected by the following assertion:

```
validateTransition (E (SIP StablyRevealed) fragment)
  (E (SIP (Is what)) fragment') = do
  validateVerdictEvent (getSIP updateSpec)
    (getSIPVoteOf (getSIPId updateSpec)
      fragment
      (firstEvent fragment')
      what
    -- We shouldn't see any actions in @fragment'@, save for implementation
    -- submissions, which the system can check.
    onlyImplementationSubmissionAllowed updateSpec fragment')
```

Queued proposals that should have been removed If a proposal is queued, then it must be possible for the protocol version it supersedes to be activated. If this is not the case, we know that this queued proposal can never be activated either. Our tests found that the implementation left proposals in the queue that could never be adopted. The following assertion led to the discovery of this bug:

```
validateTransition (E (Implementation StablyRevealed) fragment)
  (E Queued fragment') = do
  -- ...
  -- the current version, or there is a candidate proposal with higher or
  -- the same priority.
  ( getCurrentProtocolVersion (firstState fragment')
    < supersedesVersion (getProtocol updateSpec)
  ∨
  exists
    (candidatesAtTheBeginningOf fragment' `withIdDifferentFrom` updateSpec)
    (λ protocol →
      version protocol < version (getProtocol updateSpec)))
```

5.4 Conclusions

In this section we have seen how the requirements defined in [PRI20a, PRI20d] were validated by means of:

- unit tests
- property based tests
- back of the envelope calculations
- micro-benchmarks
- integration tests

D1.3 – Use Case Validation

- benchmarks of the integrated prototype

We found the unit tests to be quite useful for illustrating how the protocol works when communicating with other stakeholders (project and product managers, engineers, researchers, etc).

For writing and running the property based tests, we developed a custom framework for property testing system traces. This framework improves upon the work done at IOHK, and can be readily used for testing other ledger components.

With our extensive set of unit and property test we were able to detect several bugs before the integration with Cardano took place. After testing the integrated prototype we did not find any errors, which attest to the importance of testing early.

The results obtained in our back of the envelope calculations were confirmed by our experimental results.

Chapter 6

Conclusions

This document provided a validation report for all the use-cases in the PRIViLEDGE project, about achieving the validation criteria [PRI20a] based on the requirements [PRI18]. All contributions rely on DLT or secure multi-party computation to achieve use-case specific goals in the context where the privacy is paramount. Different aspects were validated depending on the use-case: compliance, functional requirements, security, usability, performance, interoperability. Different methods for validation were applied as chosen to be fit by the use-case partners. The report covers four mature research prototypes with potential to evolve into a part of the given ecosystem, be it software updates for Cardano, online voting with TIVI or university diploma / health insurance records management.

Bibliography

- [AC20] Thomas Attema and Ronald Cramer. Compressed Σ -protocol theory and practical application to plug & play secure algorithms. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 513–543. Springer, 2020.
- [BH97] Ramesh Bharadwaj and Connie Heitmeyer. Applying the scr requirements specification method to practical systems: A case study. 02 1997.
- [BH13] C. Bormann and P. Hoffman. Concise binary object representation (cbor). RFC 7049, RFC Editor, October 2013.
- [CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.
- [CKKZ20] M. Ciampi, N. Karayannidis, A. Kiayias, and D. Zindros. Updatable blockchains. *Lecture Notes in Computer Science*, vol 12309, 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).
- [GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310, 2015.
- [IOH21] IOHK. Cardano ops. <https://github.com/input-output-hk/cardano-ops>, 2021.
- [NAK21] Damian. Nadas Agut and Nikos Karagiannidis. Decentralized software updates. <https://github.com/input-output-hk/decentralized-software-updates>, 2021.
- [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.
- [PRI20a] Validation Criteria (D1.2). Technical report, PRIViLEDGE project, 2020.
- [PRI20b] Report on Architecture for Privacy-Preserving Applications on Ledgers (D4.2). Technical report, PRIViLEDGE project, 2020.
- [PRI20c] Final Report on Architecture (D4.3). Technical report, PRIViLEDGE project, 2020.
- [PRI20d] Report on Tools for Secure Ledger Systems (D4.4). Technical report, PRIViLEDGE project, 2020.