Bern University of Applied Sciences (BFH)

Bachelor Thesis

CHVote Demonstrator

Authored by Kevin Häni <kevin.haeni@gmail.com>
Yannick Denzer <yannick@denzer.ch>
Supervised by Prof. Dr. Rolf Haenni <rolf.haenni@bfh.ch>

Bern, November 12, 2017

Contents

1 Introduction

1.1 Electronic voting

Since 2015, it is possible for Swiss citizens registered in the cantons Geneva and Neuchâtel and living abroad, to vote electronically. However, these systems did not yet meet the requirements in terms of security and transparency, to be accepted as a secure E-Voting platform on a nationwide scale.

One of the requirements that is hardest to achieve, is that the system must ensure the voters privacy while at the same time, it must be verifiable that only valid votes have been counted.

A contract was formed between the state of Geneva and the Bern University of Applied Sciences to work out a new protocol which does meet the complex requirements set up by the government. In 2017, the resulting specification document written by Haenni Rolf, Philipp Locher and Reto E. Koenig has been officially published and a proof-of-concept / prototype has been successfully implemented by the State of Geneva.

1.2 Project task

Understanding such a complex protocol isn't easy and might be the reason why many people still do not trust electronic e-voting systems. In close consultation with the authors of the CHVote specification, we agreed to develop an application that allows users to get a hands-on experience with the CHVote e-voting system and makes it possible to show to an audience how the future of voting in Switzerland might possibly look like.

For this reason, we have implemented the protocol according to the specification and developed a web-based application on top of it, which allows to perform every step of an election, from generating the electorate data, to casting and confirming ballots from a voters point of view, to the post-election processes like mixing, decryption and tallying.

2 Management Summary

3 CHVote Protocol

3.1 Protocol

The protocol our solution is based on, does not originate from us! The concept and specification has been created by Rolf Haenni and his team at Institute for Security in the Information Society (RISIS) of the Bern University of Applied Sciences. For this project, we have implemented the protocol according to their specification. In this chapter we summarize the most important aspects of the protocol for better understanding our application.

3.2 Goals

As pointed out earlier, one of the big challenges an E-Voting protocol has to solve is the verifiability of the voting result while still ensuring the privacy of all voters. Another big problem e-voting systems are facing is the risk of a voting client being infected by malware which manipulates casting of a vote without the voters notice. Both of these issues are addressed by the use of modern cryptography.

3.3 Protocol Participants

3.3.1 Voter

A voter is a person who is eligible to vote in his state. Every voter must possess a voting card that has been sent to him prior to an election event and that contains several codes such as the voting code and the confirmation code, which are used to identify the voter during the vote casting process.

A voter uses a voting client (website) to form a ballot according to the protocol, by entering his selection, a voting code to cast, and later in the process, his confirmation code to confirm his vote. Verification codes and a finalization code are displayed by the voting client to ensure the voter that his vote has been cast as intended and hasn't been manipulated by a third party.

3.3.2 Election Administrator

The election administrator, typically a person of the government, sets up the election by providing the necessary information such as the candidates and the voters. He is also responsible for determining and publishing the final results of the election.

3.3.3 Election Authorities

The election authorities can be seen as some kind of observer.

3.3.4 Bulletin Board

3.3.5 Printing Authority

3.4 The basic idea of the CHVote protocol

Before the actual election, voting sheets are generated and printed for the whole electorate and delivered to the voters by a trusted mailing service. The voting sheets contain several codes, namely:

- voting code
- confirmation code
- finalization code
- one verification code for every candidate

The voting and confirmation code are authentication codes used to authenticate the voter.

The voter first selects candidates by entering their indices. The voting client then forms a ballot containing the voters selection encrypted with the authorities public key and authenticated with the voters personal voting code. Additionally, the ballot contains a query that queries the authorities for the corresponding verification codes of the selected candidates, without the server knowing which candidates the voter has selected. The voter then checks if the returned verification codes match the codes of the candidates he has chosen on the printed voting sheet. If the selection was somehow manipulated by malware, the returned verification codes would not match the printed ones and the voter would have to abort the vote casting process. This way the integrity of the vote casting can be assured even in the presence of malware. Privacy on the other hand cannot be protected since the malware will learn the plaintext of the voter's selection.

In order to verify that a voter has formed the ballot correctly by choosing exactly the number of candidates he is supposed to choose, the following trick is being used: the verification codes are derived from $n = \sum_{n=1}^{t} n_j$ random points on t polynomials (one for every election event j) of degree $k_j - 1$, that each election authority has chosen randomly prior to the election. By learning exactly $k = \sum_{n=1}^{t} k_j$ points on these polynomials, the voting client is able to reproduce these polynomials and therefore is able to calculate a particular point with $x = 0$ on these polynomials. The corresponding y values are incorporated into the second voting credential from which the confirmation code is derived. Only if the voter knows these values (by submitting a valid candidate selection), he will be able to confirm the vote that he casted.

Since there is still a connection between the encrypted ballot and the voter at this point, the encrypted candidate selection is extracted from the ballots before tallying. After that, every authority is shuffling/mixing these encryptions in order to make it impossible to find out which voter has submitted which encrypted ballot. This mixing of the encrypted votes is done by using the homomorphic property of the ElGamal encryption scheme. Re-encryption of the ballots multiplied with the neutral element 1 yields a new ciphertext for the same plaintext.

The public key that is used for encryption has been generated jointly by all authorities. Therefore in order to decrypt the result, all authorities must provide their share of the private key. The measure of multiple authorities participating in the whole e-voting process ensures the security of the whole election even if only one authority can be trusted.

4 Project Plan

4.1 Project Method

4.2 Requirements

4.2.1 General Requirements

	Description	Must	Priority	Iteration	Status
R1	The CHVote protocol is implemented as specified in the latest specification document. The only exclusion are the algorithms for channel security.	Must	High	1	Done
R2	The application is web-based shows updates within the same demo-election in real-time.	Must	High	1	Done
R3	The system supports 1-out-of-3 type of elections (e.g. elect 1 of 3 possible candidates)	Must	High	1	Done
R4	The system supports multiple parallel elections	Must	High	1	
R5	The system supports internationalization. Providing more than one language is not required.	Must	High	1	Part. done
R6	Users can create new elections	Must	High	1	Done
R7	The system can handle k-out-of-n type of elections	Can	Medium	1	

4.2.2 Election-Overview

	Description	Must	Priority	Iteration	Status
R8	The overview shows which phase the election is currently in	Must	High	2	
R9	A graphical scheme of the chVote protocol gives an overview of all participating parties	Must	Medium	2	

4.2.3 Election Administrator

	Description	Must	Priority	Iteration	Status

R10	An election can be set up by providing all required information such as the candidates, number of parallel voters, the number of voters and the number of selections (simplified JSON input)	Must	High	1	Done
R11	The election can be set up without entering the parameters in JSON format and allows easier set up of elections with multiple parallel election events	Must	Low	2	
R12	The election administrator view allows to perform the tallying and displays the final result of an election in numbers and a pie chart	Must	High	1	
R13	During election setup, the security parameters can be chosen from a set of predefined parameters	Can	Low	2	

4.2.4 Printing Authority

	Description	Must	Priority	Iteration	Status
R14	Users can generate and display voting cards for an election.	Must	High	1	Done
R15	Voting cards hide sensitive information behind a scratch card	Can	Medium	2	

4.2.5 Election Authority

	Description	Must	Priority	Iteration	Status
R16	The election authority view shows all information known to an election authority	Must	High	1	Done
R17	After a voter has submitted a ballot, all election authorities can check and respond to the voters submission	Must	High	1	Done
R18	In the post-election phase, all election authorities can perform the mixing and decryption tasks	Must	High	2	
R19	Each authority can optionally processes all tasks automatically	Can	High	2	Partially done

4.2.6 Voter

	Description	Must	Priority	Iteration	Status
R20	Users are able to go through the whole vote-casting process for every voter	Must	High	1	Done
R21	The voting card of a voter is displayed on screen. The voting and confirmation codes can be copied into the input textfields by double clicking	Must	Medium	1	

4.2.7 Bulletin Board

	Description	Must	Priority	Iteration	Status
R22	The bulletin board view shows what information is publicly available	Must	High	1	Done
R23	The bulletin board view is extended with verification-functionality	Can	Low	2	

4.3 Timeplan

4.4 Concept

TODO: Beschreiben, wie wir die Anwendung aus fachlicher Sicht aufteilen wollen (Unterteilung in eine View per Actor etc.)

4.5 Layout

TODO: Beschreiben wie wir die Anwendung optisch/layoutmässig geplant haben

4.6 Time schedule

Project tasks		38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
Documentation																		
Update documentation and journal	soll																	5
	ist																	
Concept																		
Working out goals / requirements	soll																	
	ist																	
Concept	soll																	
	ist																	
Implementation																		
Update CHVote cryptolibrary to the lastest specification	soll		1															
	ist																	
Prototyping (proof of concept)	soll				2													
	ist																	
Iteration 1																		
Implement backend API	soll																	
	ist																	
Implement frontend: Pre-election	soll																	
	ist																	
Implement frontend: Election	soll																	
	ist																	
Implement frontend: Post-election	soll										3							
	ist																	
Iteration 2																		
Improve backend API	soll																	
	ist																	
Improve frontend and implement can-criteria: Pre-election	soll																	
	ist																	
Improve frontend and implement can-criteria: Election	soll																	
	ist																	
Improve frontend and implement can-criteria: Post-election	soll																4	
	ist																	
Project completion																		
Poster, Final Day, Presentation	soll																	
	ist																	

Table 4.8: Project time schedule

5 Implementation Details

In this chapter, we will describe the details on how we implemented the application. We will start explaining the architecture from a high-level perspective with each component being a blackbox. Later sections of this chapter will then further describe the internals of each component of our system.

5.1 Component Overview

From a high-level perspective, our application consists of three components which themselves may consist of multiple sub components:

- Web application
- Backend
  - REST Service
  - Datasync Service
  - VoteSimulation
- CHVote crypto-library

5.1.1 Technologies and Frameworks

Since the CHVote crypto-library has been already implemented in python during project II, python was also the obvious choice for the backend. **Python** offers a wide variety of frameworks for building webservices. Since we planed on building a singlepage-application for the client, we chose the lightweight micro-framework flask for building a restful webservice. **Flask** also offers support for websockets and the popular websocket framework socket.io, which we used also in our frontend. **Socket.io** simplifies the usage of websockets and offers fallback technologies such as longpolling in case websocket is not supported by the browser or the webserver. For persisting the state of an election between the stateless API/webservice calls, we use mongoDB to store the states of all parties of an election.

For our frontend (web application) we evaluated several singlepage application (SPA) frame-

works. **VueJS** is a new, modern and lightweight SPA framework that in contrast to Angular has a much flatter learning curve but still offered all the functionality that we needed. The VueJS addon Vuex enabled us to establish a datastore pattern in our frontend, which makes it possible to have a copy of the backend datastore in our web application which is synchronized in real-time through socket.io.

More about how these technologies and frameworks are used and the communication between these components is described in more details in the following sections.

5.2 Architecture

The core of our application is the VoteSimulation component in our backend which implements the evoting protocol by utilizing all algorithms of the CHVote crypto-library according to the CHVote specification. The VoteSimulation component internally holds the state of a whole CHVote election and exposes functions to manipulate this state at a granularity required by our web application to implement all usecases. For example: The VotingSimulation contains a list of ballots and exposes functions to cast a new ballot, which will generate a new ballot according to the protocol, by calling the CHVote crypto-library, and adds the ballot to the ballotlist.

On top of the VoteSimulation we have implemented a REST service that acts a facade to the VoteSimulation component and makes it's functionality available as an API for our webclient. The REST service also has to initialize the VoteSimulation by loading and persisting it's state from and to the database between each API call.

Since it's a requirement that all clients of a particular election must be notified of mutations of the state, we have implemented a data sync service which allows to push the state or parts of the state of the VoteSimulation to the webclients by using the websocket protocol. This service is utilized by the REST service after every API call that could have potentially manipulated the state.

To establish a proper separation of concerns, the state of the VoteSimulation is always sent to the client via the data sync service. The REST service only returns success or error codes or information that is required in response to some particular API call, and never state objects. On the other hand, the data sync service does never manipulate the state of the VoteSimulation and is solely responsible for sending data to the client.

From the clients point of view, the webclient contains a copy of the whole VoteSimulation state in a local datastore. This store is initially populated when the web application is initialized with an election. Whenever the state of the VoteSimulation changes, the data sync service is is called to push the new data to the web application. A local mutation handler is called inside the web application which writes the new data into the local datastore.

Since the components that the webpages of the web application are built of, are directly bound to the local datastore, all mutations automatically reflected to the user. From those

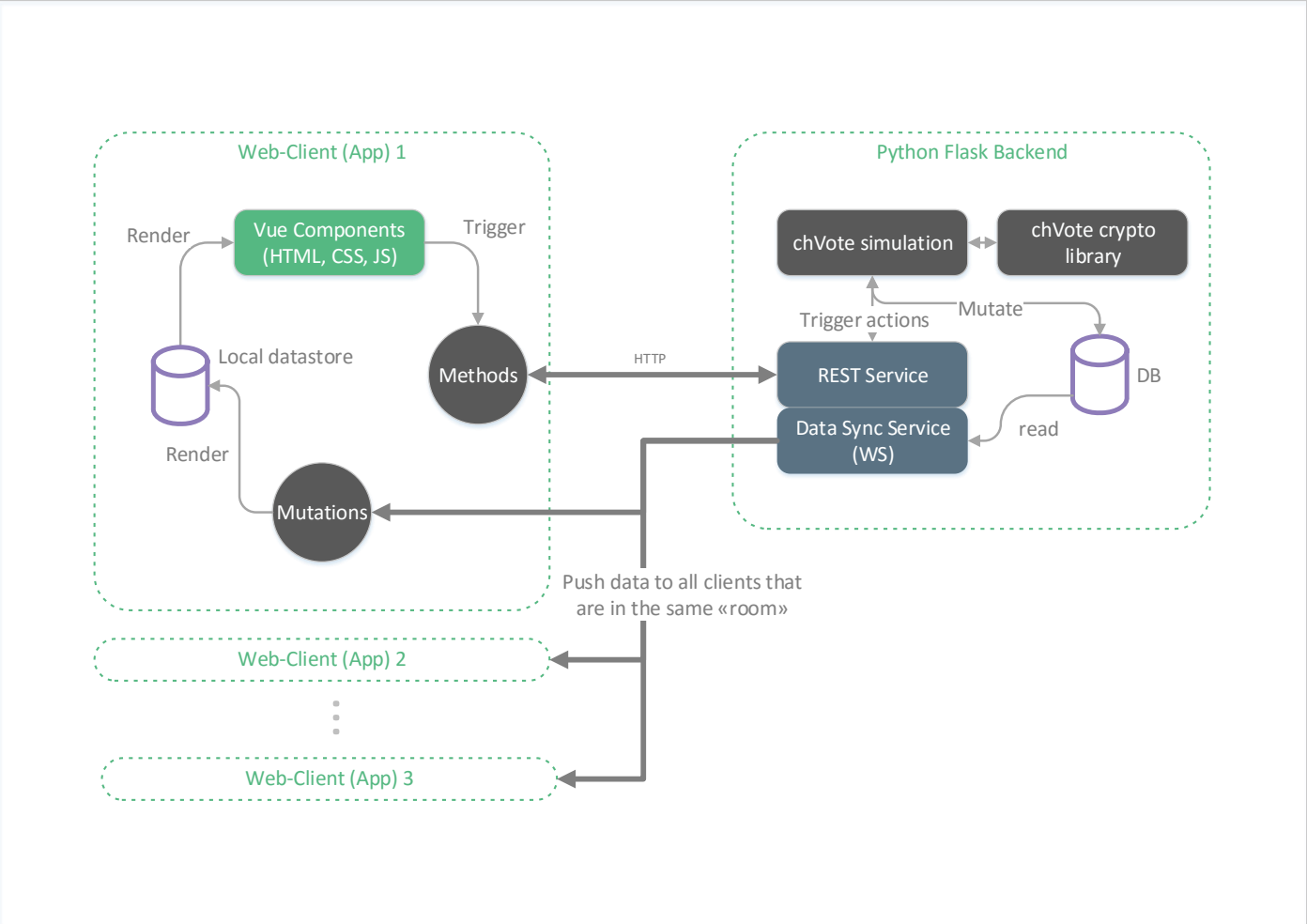

Figure 5.1: Architecture

pages, the REST API can be called to trigger some CHVote specific action on the backend. The resulting state change is again being pushed to all clients while the responsible client that performed the HTTPS request will additionally receive a success code, or an error message in case of an error.

5.3 Backend

The backend mainly consists of the VoteSimulation which implements the CHVote protocol by using the CHVote crypto-library.

5.3.1 VoteSimulation

We have divided the state of a CHVote election into the following classes:

- **BulletinBoardState**: Holds all data that is publicly available on the bulletin board (the number of candidates, the tallied result)
- **ElectionAuthorityState**: Holds all data that an election authority knows (e.g. the list of ballots, the secret key of an election authority)
- **VoterState**: Since there is no distinction made between a voter and a voting client in our application, the VoterState contains the data of both the voter (e.g. the voting card) and the data typically known to the voting client (e.g. the points returned by the oblivious transfer)
- **PrintingAuthorityState**: Holds the data known to the printing authority (e.g. the list of all voters private credentials and the voting cards)
- **ElectionAdministratorState**: Holds all data known to the election administrator

Since our application must handle multiple elections in parallel, the state cannot be kept in RAM, but needs to be persisted between every single request. For this reason we evaluated different database systems and concepts. We decided not to use a relational database system that requires us to define a database schema as we want our state objects to be the only place where the schema is defined. This makes it easier to apply changes to the protocol in future.

For our purpose, mongoDB seemed like a good choice. Since we do not need the ability to access and filter our data with arbitrary queries, but only need to be able to save and load a state object of a particular election, we simply store the whole state as a binary string in a mongoDB collection. The only additional attribute that is saved to the database alongside with the serialized state is the electionId which denotes which election a particular state belongs to.

An election contains multiple VoterStates and ElectionAuthorityStates. Therefore, these two states additionally require an electionAuthorityId and a voterId.

The only common functionality between every state object, is the ability to serialize the object to a JSON string. For this reason we had to write a custom transformer which tells the JSON parser how to serialize datatypes such as mpz, bytearrays and custom classes. Luckily, python offers a way to easily serialize any custom object. By calling `object.__dict__` we can convert an object into a dictionary, as long as transformer is able to serialize all properties of the object.

5.3.2 Actor/Party-classes

We described how the state classes are used to divide the data of the VoteSimulation into smaller units. Similarly, the functionality of the VoteSimulation class is separated into classes, one for every actor in the protocol.

The common functionality, namely, a function for loading the corresponding state from the

database and one for persisting the state to the database, are contained in an abstract base class.

5.3.3 REST Service

5.3.4 Data-Sync Service

The following sequence diagram shows how the vote casting use case is implemented within the backend.

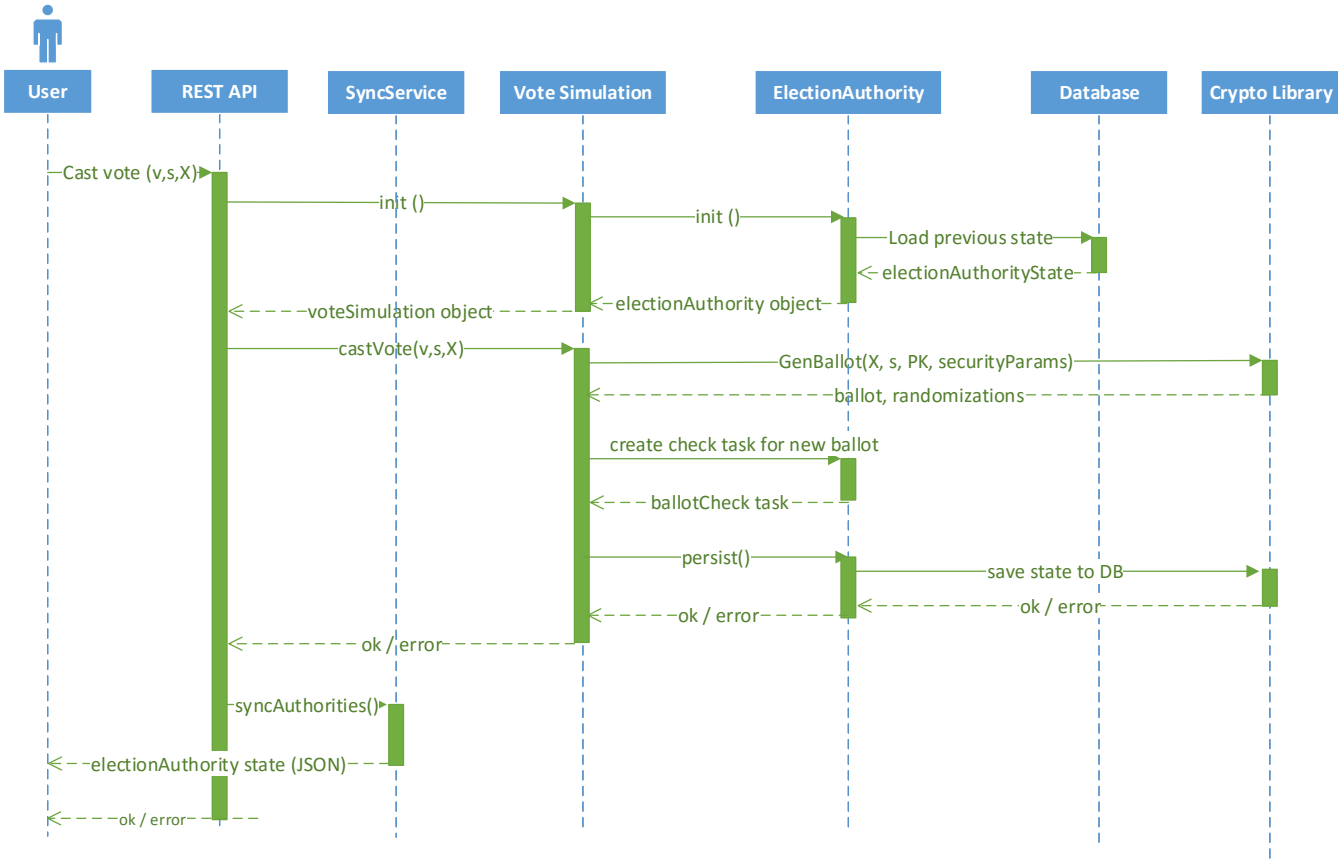

Figure 5.2: Vote casting sequence diagram

5.4 Crypto-library

5.5 Language choice

As both team members are working in different fields of employment and are experienced in different programming languages, there wasn't an obvious choice as in what programming language we would implement this project. Also, as Java has already been used by the team in Geneva, Java was out of question. Different programming languages have been taken into consideration and in the end Python seemed like a rather suitable language due to the following reasons:

- simple syntax and commonly known language features
- mature language and standard libraries
- python's syntax enables programs to be written in a compact and readable style
- native support for large integers (*BigInts*) and bindings for the GMP[1] library
- supports a lot of platforms
- many popular web development frameworks are implementated in Python

Throughout the project not all of the reason above turned out to be true or ideal. The drawbacks that we have experienced during the implementation of this project will be discussed at the end of this document.

5.5.1 Project structure

We decided to put every algorithm of the specification in its own file together with related unit tests. The files are structured according to the actors of the protocol, for example:

- **Common**: contains common cryptopgraphic algorithms and the security parameters used by multiple algorithms
- **ElectionAuthority**: contains all the algorithms used by the election authority
- **PrintingAuthority**: contains all the algorithms used by the printing authority
- **VotingClient**: contains all the algorithms used by the voting client
- **ElectionAdministration**: contains all the algorithms used by the election administrator

[1] GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers, see `https://gmplib.org/`.

- **Utils**: contains helper classes and miscellaneous utility functions

5.5.2 Public parameters

There exist two types of public parameters:

The **security relevant parameters**, e.g:

- The order of the prime groups: p, $\prime p$, $\hat{p}$
- The length of the voting, confirmation, return and finalization codes
- The number of authorities: s

and **public election parameters**, e.g.:

- The size of the electorate: N_E
- The number of candidates: n
- The list of candidate descriptions: c

The security parameters are typically used within the algorithms and remain unchanged for a longer time period, whereas the public election parameters are only used by the protocol implementations and change with every election.

The object `SecurityParams` holds all security relevant parameters and is injected as an additional function argument to all algorithms. Several different `SecurityParams` objects are created initially, which contain all the parameters according to the recommendations in the CHVote specification document ("level 0" for testing purposes and "level 1" through "level 3" for actual use of the protocol). This approach allows us to use different levels of security during development of the algorithms and protocols. For simple unit testing we used "level 0" in order to inject the security parameters recommended for testing puposes. For actual test runs of the project the security parameters from "level 2" were used.

The public election parameters on the other hand are directly passed to the algorithms by the calling party. If an algorithm needs to know certain election parameters (like the size of the electorate N_E), these values are typically derived from vectors that they have access to, so they do not require specific knowledge of these parameters.

5.5.3 Coding style

The following source code sample shows a typical implemation of an algorithm (in this exmaple, algorithm 7.18 according to the CHVote specification).

```
import unittest
import os, sys
from gmpy2 import mpz
import gmpy2

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from Utils.Utils                    import AssertMpz, AssertList, AssertClass,
 ↪  AssertString
from Crypto.SecurityParams          import SecurityParams, secparams_l0
from Utils.ToInteger                import ToInteger
from VotingClient.GetSelectedPrimes import GetSelectedPrimes
from VotingClient.GenQuery          import GenQuery
from VotingClient.GenBallotProof    import GenBallotProof
from UnitTestParams                 import unittestparams
from Types                          import Ballot
from Utils.StringToInteger          import StringToInteger

def GenBallot(X_bold, s, pk, secparams):
    """
    Algorithm 7.18: Generates a ballot based on the selection s and the voting
 ↪  code X. The
    ballot includes an OT query a and a proof pi. The algorithm also returns
 ↪  the random
    values used to generate the OT query. These random values are required in
 ↪  Alg. 7.27
    to derive the transferred messages from the OT response, which itself is
 ↪  generated by Alg. 7.25.

    Args:
        X_bold (str):                   Voting Code X ∈ A_X^l_X
        s (list of int):                Selection s = (s_1, ... , s_k), 1
 ↪  <= s_1 < ... < s_k
        pk (mpz):                       ElGamal key pk ∈ G_p \ {1}
        secparams (SecurityParams):     Collection of public security
 ↪  parameters

    Returns:
        tuple:                          alpha = (r, Ballot) = (r, (x_hat,
 ↪  a, b, pi))
    """

    AssertMpz(pk)
    AssertList(s)
    AssertClass(secparams, SecurityParams)

```

```
39      x = mpz(StringToInteger(X_bold, secparams.A_X))
40      x_hat = gmpy2.powmod(secparams.g_hat, x, secparams.p_hat)
41
42      q_bold = GetSelectedPrimes(s, secparams)                    # q = (q_1,
        ↪  ... , q_k)
43      m = mpz(1)
44
45      for i in range(len(q_bold)):
46          m = m * q_bold[i]
47
48      if m >= secparams.p:
49          return None
50
51      (a_bold, r_bold) = GenQuery(q_bold, pk, secparams)
52      a = mpz(1)
53      r = mpz(0)
54
55      for i in range(len(a_bold)):
56          a = (a * a_bold[i]) % secparams.p
57          r = (r + r_bold[i]) % secparams.q
58
59      b = gmpy2.powmod(secparams.g,r, secparams.p)
60      pi = GenBallotProof(x,m,r,x_hat,a,b,pk, secparams)
61      alpha = Ballot(x_hat,a_bold,b,pi)
62
63      return (alpha, r_bold)
64
65  class GenBallotTest(unittest.TestCase):
66      def testGenBallot(self):
67          selection = [1,4]       # select candidates with indices 1,4
68          (ballot, r) = GenBallot(unittestparams.X, selection,
            ↪  unittestparams.pk, secparams_l0)
69          print(ballot)
70          print(r)
71
72  if __name__ == '__main__':
73      unittest.main()
```

All algorithms contain a short description, which was taken as-is from the specification document, as well as a comment (Google-style documentation string), which can be used to automatically generate code documentation. The algorithm itself is implemented as close to the specification as possible, using the same variable names and (as far as the language supports it) similar control structures:

- The suffix `_bold` for emphasized (bold) variables, e.g. `p_bold` for $\mathbf{p}$
- The suffix `_hat` for variables with a hat, e.g. `a_hat` for $\hat{a}$

- The suffix `_prime` for variables with a prime, e.g. `a_prime` for a'
- etc.

Each file also contains unit test relevant to the specific algorithm (if unit testing was considered useful for the particular algorithm).

The following example shows the similarities between the algorithm pseudo code and the actual implmentation in Python:

Algorithm: GenBallot$(X, \mathbf{s}, pk)$
Input: Voting code $X \in A_X^{\ell_X}$
Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k$
Encryption key $pk \in \mathbb{G}_q \backslash \{1\}$
$x \leftarrow \mathsf{ToInteger}(X)$
$\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$
$\mathbf{q} \leftarrow \mathsf{GetSelectedPrimes}(\mathbf{s})$
$m \leftarrow \prod_{i=1}^{k} q_i$
if $m \geqslant p$ **then**
  return $\perp$
$(\mathbf{a}, \mathbf{r}) \leftarrow \mathsf{GenQuery}(\mathbf{q}, pk)$
$a \leftarrow \prod_{i=1}^{k} a_i \bmod p$
$r \leftarrow \sum_{i=1}^{k} r_i \bmod q$
$b \leftarrow g^r \bmod p$
$\pi \leftarrow \mathsf{GenBallotProof}(x, m, r, \hat{x}, a, b, pk)$
$\alpha \leftarrow (\hat{x}, \mathbf{a}, b, \pi)$
return $(\alpha, \mathbf{r})$

```
x = mpz(StringToInteger(X_bold, secparams.A_X))
x_hat = gmpy2.powmod(secparams.g_hat, x, secparams.p_hat)
q_bold = GetSelectedPrimes(s, secparams)

m = mpz(1)
for i in range(len(q_bold)):
    m = m * q_bold[i]

if m >= secparams.p:
    return None

(a_bold, r_bold) = GenQuery(q_bold, pk, secparams)
a = mpz(1)
r = mpz(0)

for i in range(len(a_bold)):
    a = (a * a_bold[i]) % secparams.p
    r = (r + r_bold[i]) % secparams.q

b = gmpy2.powmod(secparams.g,r, secparams.p)
pi = GenBallotProof(x,m,r,x_hat,a,b,pk, secparams)
alpha = Ballot(x_hat,a_bold,b,pi)

return (alpha, r_bold)
```

5.5.4 Return types

In most cases, when an algorithm returns more than a scalar datatype, tuples are used. Tuples allow to return multiple values from a function:

```
1 def foo():
2     return (1, 2)
3
4 def main():
5     a, b = foo()
```

This way a lot of the source code looked very similar to the pseudo code in the CHVote specification. For more complex data types or return values that are used more often, named tuples were used. The data type "namedtuple" is like a lightweight class and allows access to named properties.

```
1 Ballot = namedtuple("Ballot", "x_hat, a_bold, b, pi")
2
3 def main():
```

```
4     Ballot b = getBallot()
5     x_hat = b.x_hat
```

By following this approach we can avoid having lots of container classes only used to pass data structures between the algorithms.

5.6 Frontend

Our frontend is a singlepage application built with Javascript, VueJS, HTML5 and CSS3. We tried to follow the design patterns and best practices proposed by the VueJS framework wherever possible.

Every site of our applications consists of at least one VueJS component which is activated when the user visits the corresponding route in the URL.

5.6.1 Datastore

The web application's datastore is divided into multiple modules, one datastore module for each corresponding state of the backend.

TODO: Grafik die zeigt, wie VueX Datastores mit Backend States zusammenhängen

5.6.2 Websocket room concept

TODO: Konzept mit den Rooms / Election IDs beschreiben

5.6.3 Development Environment

TODO: Webpack, ESlint etc. beschreiben

6 Conclusion

During the first few weeks we felt as if we have been thrown into cold water. Reading and understanding the protocol wasn't easy at first, because we had to get used to the notation and memorize a large amount of variables used by the many algorithms. While some of the cryptographic primitives were taught in previous courses, most of them were new and unknown to us. We focused on getting a good understanding of the protocol on a higher level rather than learning about each and every algorithm in detail, as this was sufficient for implementing and understanding the protocol.

Additionally, programming algorithms isn't something we are doing on a daily basis. Therefore, the first few algorithms took us quite some time to implement. After a few weeks, we could greatly increase our productivity and in the end, we could implement even the larger algorithms in not much more time than the simple ones in the beginning of the project.

From our perspective, the project has been extremely interesting and we are still impressed by the ideas presented and specified in the CHVote specification. From simply implementing the protocol we could learn a lot about the CHVote protocol and E-Voting in general and could improve both our knowledge of more advanced cryptographic topics and get practise in implementing cryptographic algorithms.

6.1 Python drawbacks

During the project we have experienced a few issues with the programming language that we used to implement the specification in, Python. In particular, we have observed the following issues:

- Performance issues due to Python being an interpreted language
- Function overhead: function calls in Python seem to be quite slow
- Strongly dynamic typing vs. static typing: the Python interpreter needs to inspect every single object during run time (be it an integer or a more complex object)
- The *BigInteger* library surprisingly isn't as fast as using directly the GMP library
- Larger projects tend to turn out messy
- Little to no standard documentation regarding project structure

- No real standard for unit testing, documentation generation etc.

For detailed information regarding the performance issues that we have experienced see [2] and [3]. Based on the reasons above we would not recommend to use Python for the use in similar or larger project. Python is indeed a very handy language to write quick prototypes and proof of concepts, but issues become more frequent in larger projects.

Bibliography

[1] "CHVote System Specification", by Rolf Haenni, Reto E. Koenig, Philipp Locher and Eric Dubuis, April 11, 2017.

[2] "Why Python is Slow: Looking Under the Hood", by Jake VanderPlas, see `http://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/`

[3] "Python speed: performance tips", from the official Python wiki, see `https://wiki.python.org/moin/PythonSpeed/PerformanceTips`

7 Journal

7.1 Week 1

7.1.1 Kickoff Meeting

During our kickoff meeting we discussed the possibilities of our bachelor thesis based on the spadework of the previous "project 2" module and broke them down into two options: A realistic prototype of the whole CHVote system which includes everything a real implementation would need, like signatures, channel security, distributed election authorities with docker etc., or to build a demonstrator tool (an application that allows to demonstrate the functionality of the chVote protocol in a more visual manner).

7.2 Week 2

In week 2 we have made the decision to build the demonstrator mainly because the final product would potentially be more attractive visually than a prototype where the main work lies in the background which is not visible to an outsider. We have also started thinking about what technologies and frameworks to use and to build a few sketches and mockups to have some basis for discussion for our next meeting.

We have also started updating our implementation of the CHVote crypto-library to the latest specification. Since we ran into a few problems, this took us almost the whole week.

7.2.1 Reflexion

At this stage we have yet been very unsure about how the application should look like, what audience we should have as our main target and what functionality the application should offer.

7.3 Week 3

During our second meeting we discussed the further elaborated the goals, the structure of our project and talked about the audience.

- In essence, the application should allow to demonstrate a chVote election from the view of every party participating in the election process.
- The application should be a real-time webapp that updates the views automatically as soon as something changes and without having to reload the page

7.4 Week 4

In the fourth week we continued describing the goals and further worked on the system architecture. We also made some first experiences with the envisaged frameworks and technologies (VueJS, socket.io, Flask, MongoDB).

7.4.1 Reflexion

- working with socket.io and VueJS has been very intuitive and looked very promising and suitable for our project
- We were not yet sure whether or not mongoDB is the right technology for our needs.

During prototyping, we observed that our first architecture approach of doing everything over websockets, turned out to be a bad decision.

7.5 Week 5

In the fifth week we started with the real implementation.

7.5.1 Reflexion

- working with socket.io and VueJS has been very intuitive and looked very promising and suitable for our project
- We were not yet sure whether or not mongoDB is the right technology for our needs.

During prototyping, we observed that our first architecture approach of doing everything over websockets, turned out to be a bad decision.