Bern University of Applied Sciences (BFH)

Project II

CHVote prototype in Python

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

Bern, June 20, 2017

Contents

Chapter 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 electronic voting platform on a nationwide scale.

One of the requirements which is particularly difficult to achive is that the system must protect a voter's privacy while at the same time it must be verifiable that only votes have been counted.

A contract was formed between the state of Geneva and the Bern University of Applied Sciences (BFH) to work out a new protocol which does meet the complex requirements set up by the government. Some of the concepts behind this protocol are based on a Norwegian e-voting system. In 2017, the resulting specification document has been officially published and the protocol proved to be working correctly by the proof of concept implementation developed by the state of Geneva.

1.2 Project task

The task of "Project II"[1] was to implement all the algorithms specified by the "CHVote System Specification" document [1]. In addition, a primitive protocol implementation should be built which combines all the algorithms in order to simulate a small eectronic election event. Aside from the programming part, building knowledge about e-voting and specifically, the CHVote specification and its algorithms was the main task which enables us to further work on this topic as part of our bachelor thesis.

[1] BFH module BTI7302, see `https://www.ti.bfh.ch/fileadmin/modules/BTI7302-en.xml`

Another goal of our project was to prove that the CHVote specification can be implemented regardless of the programming language. Therefore, we decided to use a programming language that is different from the one used by the team in Geneva. While the state of Geneva has put a lot of emphasis on the performance of the system, performance is not of great relevance for our project.

Chapter 2

Introduction to CHVote

2.1 Protocol

As pointed out earlier, one of the big challenges the protocol is trying 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 the casting of a vote without the voters notice. Both of these issues are addressed by the use of modern cryptography.

2.2 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 the 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 had selected. The voter then checks if the

returned verification codes match the codes of the candidates he had 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. The privacy on the other hand cannot be protected since the malware will learn the plaintext of the voters 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 was 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.

Chapter 3

Project

3.1 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.

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

3.2 Project method

The first step was to get a basic understanding of the CHVote protocol by reading the specification document and refreshing our knowledge about the cryptographic primitives used in the protocol. We decided to follow a "learning by doing" approach: while implementing an algorithm, we tried to understand how it works and how it interacts with the other algorithms.

Before starting with the implementation we created a timetable in order to track the project's progress. The idea of the timetable was to keep track of each individual algorithm and its implementation status, as well as appropriate unit tests and the review status of each algorithm. We made sure that all algorithms were reviewed by a different person than the one who implemented it. This way we were able to eliminate a lot of careless mistakes.

Chapter 4

Implementation Details

4.1 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
- **Utils**: contains helper classes and miscellaneous utility functions
- **Protocol**: contains the protocol implementation
- **profiles**: contains JSON files that are used to define election parameters

4.2 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 are defined in a JSON file and simply read as an object which is directly accessed by the protocol. 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.

The following is a exmaple for the contents of a JSON file containing all the parameters:

```
{
  "autoGenerateVoters" : true,
  "numberOfVotersToGenerate" : 50,
  "voters" : [
    {
      "name" : "Voter1",
      "selection" : "1,5"
    },
    {
      "name" : "Voter2",
      "selection" : "0,5"
    },
    {
      "name": "Voter3",
      "selection": "0,5"
    }
```

```
  ],
  "t" : 2,
  "n" : [5, 3],
  "c" : ["Hillary Clinton", "Donald Trump", "Vladimir Putin", "Marine Le Pen",
   ↪  "May", "Yes", "No", "Empty"],
  "k" : [1, 1],
  "E" : [[1, 1], [1, 1], [1, 1]],
  "securityLevel" : 2,
  "deterministicRandomGeneration" : false
}
```

4.3 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.
```

```
24
25      Args:
26          X_bold (str):                       Voting Code X ∈ A_X^l_X
27          s (list of int):                    Selection s = (s_1, ... , s_k), 1
    ↪   <= s_1 < ... < s_k
28          pk (mpz):                           ElGamal key pk ∈ G_p \ {1}
29          secparams (SecurityParams):         Collection of public security
    ↪   parameters
30
31      Returns:
32          tuple:                              alpha = (r, Ballot) = (r, (x_hat,
    ↪   a, b, pi))
33      """
34
35      AssertMpz(pk)
36      AssertList(s)
37      AssertClass(secparams, SecurityParams)
38
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 emphatized (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)
```

4.3.1 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:

```
def foo():
   return (1, 2)

def main():
   a, b = foo()
```

For more complex data types or return values that are used more often, namedtuples were used:

```
Ballot      = namedtuple("Ballot", "x_hat, a_bold, b, pi")

def main():
   Ballot b = getBallot()
   x_hat = b.x_hat
```

The data type namedtuple are like lightweight classes and allow access to named properties.

By following this approach, we could avoid having hundreds of classes only used to pass data structures between the algorithms

4.4 Protocol

Upon completion of the algorithms implmentation, we have built a protocol-layer according to the specification. For that purpose, we created a seperate entity for every protocol participant, namely **VotingClient**, **ElectionAuthority**, **BulletinBoard** and **PrintingAuthority**. The following example shows the first step of the protocol 6.5:

```
def castVote(self, s, autoInput, secparams):
    self.pk = GetPublicKey(self.bulletinBoard.pk_bold,secparams)

    X = input('Enter your voting code: ')
    (self.alpha, self.r) = GenBallot(X, s, self.pk, secparams)

    return (self.alpha, self.r)
```

Finally, within a **VoteSimulation**, all these entities are instantiated and the perform their protocol steps in order. Example of the vote casting phase:

```
votingClients = [VotingClient(i, self.voters[i], self.rawSheetData[i],
↪  self.bulletinBoard) for i in range(len(self.voters))]
```

```
2  for votingClient in votingClients:
3      # Get selection (protocol 6.4)
4      s = votingClient.candidateSelection(autoInput, self.secparams)
5
6      # Generate ballot & send oblivious transfer query (protocol 6.5)
7      (ballot, r) = votingClient.castVote(s, autoInput, self.secparams)
8
9      # Generate oblivious transfer response & check ballot (protocol 6.5)
10     responses = [(authority.name, authority.runCheckBallot(votingClient.i,
       ↪  ballot, self.secparams)) for authority in self.authorities]
11     for res in responses: print("Ballot validity checked by authority %s: %r"
       ↪  % (res[0], res[1]))
```

Chapter 5

Summary

5.1 Summary

5.2 Python drawbacks

- Interpreted language (Performance issues)
- Function overhead
- Strongly dynamic typing vs. static typing
- BigInteger library isn't as fast as using the GMP library
- Large projects tend to turn out messy
- Little to no standard documentation regarding project structure
- No real standard for unit testing, documentation generation etc.

Based on the experience with Python that we have gathered throughout this project, we would not recommend to use Python for the use in similar project due to the reason mentioned above. 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.