CHVote: Sixteen Best Practices and Lessons Learned

Rolf Haenni, Eric Dubuis, Reto E. Koenig, and Philipp Locher

Bern University of Applied Sciences, CH-2501 Biel, Switzerland
{rolf.haenni,eric.dubuis,reto.koenig,philipp.locher}@bfh.ch

Abstract. The authors of this paper had the opportunity to closely accompany the CHVote project of the State of Geneva during more than two years and to continue the project after its abrupt stop in 2018. This paper is an experience report from this collaboration and the subsequent project continuation. It describes the lessons learned from this project and proposes some best practices relative to sixteen different topics. The goal of the paper is to share this experience with the community.

1 Introduction

Developing a verifiable Internet voting system is a delicate task. While conducting elections over the Internet seems intuitively like a simple matter of counting votes submitted by voters, it actually defines a unique combination of difficult security and privacy problems. As a response to these problems, numerous cryptographic protocols have been proposed to guarantee different combinations of often conflicting security properties. While many aspects of the general problem are solved today in theory, it turned out that transforming them into reliable practical systems is a completely different challenge. In fact, not many projects have been successful so far. In the Switzerland, which played a pioneering role in the early days of Internet voting, three completely untransparent systems were in used for pilot elections with a limited number of voters over more than a decade. They were all black-box system with no verifiability. One of them was the CHVote system from the State of Geneva.

1.1 Project Context

As a response to the third report on *Vote électronique* by the Swiss Federal Council in 2013 and the new requirements of the Swiss Federal Chancellery [1,16], the State of Geneva invited leading scientific researchers and security experts to contribute to the development of their second-generation system *CHVote 2.0*. In this context, a collaboration contract between the State of Geneva and the Bern University of Applied Sciences was signed in 2016. The main goal of this collaboration was the specification of a cryptographic voting protocol that satisfies the new requirements to the best possible degree. The main output of this project is the *CHVote System Specification* document [9], which is publicly available

at the *Cryptology ePrint Archive* since April 2017. In the course of the project, updated document versions have been released in regular intervals.

In November 2018, the council of the State of Geneva announced an abrupt stop of the CHVote 2.0 project due to financial reasons.[1] This implied that with the release of Version 2.1 of the specification document in January 2019, the collaboration between the State of Geneva and the Bern University of Applied Sciences came to an end. In June 2019, the State of Geneva released all the public material that have been created during the CHVote 2.0 project, including the Java source code.[2] The implemented cryptographic protocol corresponds to Version 1.4.1 of the specification document.

To continue the CHVote project independently of the support from the State of Geneva, a new funding from *eGovernment Switzerland* has been acquired by the Bern University of Applied Sciences in August 2019. The main goal of this project was to release a final stable version of the specification document and to update the cryptographic core of the protocol based on the code released by the State of Geneva. As a first project deliverable, the current Version 3.0 of the specification document has been released in December 2019 [9]. At the time of writing this paper, the developed *OpenCHVote* software is not yet complete. Since the project is in its final stage, the code is expected to be released soon under a non-proprietary license.[3] The general purpose of the project is to make the achievements available to others for pursuing it further.

1.2 Goals and Paper Overview

This paper presents a retrospective view of the CHVote project over the last four years. The paper is divided into three sections. The two main sections describe our experience and lessons learned from our work related to the specification document and the development of corresponding software, respectively, and the final section discusses some general aspects of the project. The whole paper contains our proposal for best practices on sixteen different topics. We present these topics project in chronological order. While we think that they all have played an important role for the success of our project, we do not claim that the given list is complete or that all points are directly applicable to all similar projects.

Nevertheless, we believe that our experience is worth to be shared with the community, who may struggle with similar problems in other e-voting projects. Sharing our experience with the community is therefore the general goal of this paper. As such, it should been seen as an experience report, which may be helpful in other projects as a guideline for achieving the required quality level in a shorter amount of time. Some of the proposed best practices may even set a certain minimal quality benchmark for e-voting projects in general.

[1] For further details about the reasons for abandoning the project, we refer to the State Council's press statement at https://www.ge.ch/document/12832/telecharger.
[2] See https://chvote2.gitlab.io
[3] See https://gitlab.com/openchvote

2 Specification

Item 1: Modeling the Electoral Systems

Democracies around the world use very different electoral systems to determine how elections and referendums are conducted. A major challenge in the design of CHVote was to cover the variety of electoral systems that exist in the Swiss context. On a single election day, democratic decisions are sometimes taken simultaneously on federal, cantonal, and communal issues, with election laws that differ from canton to canton. To cope with this complexity, we managed to map all electoral systems into a concise and coherent *electoral model* that is applicable to all possible situations. The core of this model is an *election event*, which consists of several independent *k-out-of-n elections*, in which voters can choose exactly k different candidates from a candidate list of size n. An election event is therefore defined by two vectors of such values k and n.

With this simple model, we were able to cover all electoral systems from the Swiss context with their specific properties, exceptions, and subtleties.[4] Elections of the Swiss National Council turned out to be the most complicated use case, but by splitting them into two independent elections, one 1-out-of-n_p party election and one cumulative k-out-of-n_c candidate election, they fit nicely into the general model [9, Section 2.3.2]. By reducing this complexity to essentially two public election parameters and by instantiating them to past election events in all regions of our country, we managed to determine upper limits $k_{\max} = 150$ and $n_{\max} = 1500$ for the overall problem size.

Defining a general electoral model and keeping it as simple and coherent as possible turned out to be a really important abstraction layer, which allowed us to design the cryptographic protocol independently of the variety of election use cases. The above-mentioned estimation of the maximal problem size defined important cornerstones for judging the suitability of cryptographic techniques and for anticipating potential performance bottlenecks. Therefore, we recommend to carefully design a suitable model of the electoral system as early as possible in projects like this.

Item 2: Modeling the Electorate

For a given election event in the given context of the CHVote project, an additional complication is the possibility that voters may not be eligible in all elections. This can happen for two reasons. First, since cantons are in charge of organizing elections, it may happen that elections are held simultaneously in different communes of a given canton, possibly in conjunction with cantonal and federal elections. In such cases, voters are equally eligible for federal and cantonal issues, but not for communal issues. Second, since non-Swiss citizens are allowed to vote in some canton and communes, they may be part of the electorate for cantonal or communal issues, but not for federal issues.

[4] We only had to admit one exception from the general model to allow write-in candidates in some cantons.

To map all possible cases of restricted eligibility into a general model, we introduced in CHVote the concept an *eligibility matrix*, which defines for a given electorate the eligibility of each voter in each election. By connecting this matrix with the two vectors from the general election event model, we can derive for each voter the number of admissible choices in each election. To ensure the correctness of an election outcome, it is absolutely critical for all involved parties to know these values at all times. This includes auditors performing the verification process in the aftermath of an election. The eligibility matrix is therefore a third fundamental public election parameter. Without taking it as additional input, the verification of an election result can not produce a conclusive outcome.

Item 3: Cryptographic Building Blocks

Given the central role of the cryptographic building blocks in a voting protocol, we recommend describing them in the beginning of the specification document. This lays the grounds for the whole document, for example by introducing respective terms and formal notations. By describing the building block next to each other, ambiguities and conflicts in the formal notations can be eliminated in a systematic manner. Given the overall complexity of the CHVote protocol, finding a coherent set of mathematical symbols and using them consistently throughout the whole document was a ongoing challenge during the project. Providing the highest possible degree of disambiguation improves greatly the document's overall readability.

Another important aspect of describing the cryptographic building blocks is to select from the large amount of related literature exactly what is needed for the protocol. Everything can be instantiated to the specific use case and underspecified technical details can be defined to the maximal possible degree. Examples of such technical details are the encoding methods between integers, strings, and byte arrays, or the method of computing hash values of multiple inputs. Another example of an often underspecified building block is the Fiat-Shamir transformation, which is widely applied for constructing non-interactive zero-knowledge protocols [6]. The significance of doing these things right is well documented [4, 15]. A separate chapter on these topics helps to present all important cryptographic aspects in a concise form.

Item 4: Cryptographic Parameters

The collection of cryptographic building blocks defines a list of cryptographic parameters for the protocol. This list of parameters is an important input for every participating party. In CHVote, it consists of a total of twenty parameters, which themselves depend on four top-level security parameters [9, Section 6.3.1 and Table 6.1]. In theory, proper parameterization is fundamental for defining the protocol's security properties in the computationally bounded adversary model, and in practice, proper parameterization provides the necessary flexibility for adjusting the system's actual security to the desired strength. Given its central role in the security model, we recommend making the cryptographic parameters as clear and visible as possible to everyone.

For building an even more solid basis for an actual CHVote implementation, explicit values are specified for all cryptographic parameters. We introduced four different security levels [9, Section 11]. Level 0, which provides only 16 bits of security, has been included for testing purposes. Corresponding mathematical groups are large enough for hosting small elections, but small enough to avoid expensive computations during the tests. Providing a particular security level for testing turned out to be very useful for the software development process. Levels 1, 2, and 3 correspond to current NIST key length recommendations for 80 bits (legacy), 112 bits, and 128 bits of security, respectively [2]. All group parameters are determined deterministically, for example by deriving them from the binary representation of Euler's number. Applying such deterministic procedures demonstrates that the parameters are free from hidden backdoors.

Item 5: Parties and Communication

Parties participating in a cryptographic protocol are usually regarded as atomic entities with distinct, responsibilities, abilities, goals, and attributed tasks. In the design of the protocol, it is important for the parties and their communication abilities to match reality as closely as possible. In CHVote, we decided to consider the voters and their voting devices as two separate types of parties with very different abilities. This distinction turned out to be useful for multiple purposes. First, it enables a more accurate adversary model, because attacks against humans and machines are very different in nature. Second, by including the tasks of the human voters in the abstract protocol description, it provides an accurate model for simulating human voters in a testing environment.

If a voting protocol depends on fully trusted parties, particular care must be applied in the design of their responsibilities and tasks. The *election administrator* and the *printing authority* fall into this category in CHVote. In both cases, we placed great emphasis on limiting their responsibilities to their main role in the protocol. The printing authority, for example, only applies a deterministic algorithm to assemble the inputs from multiple election authorities. The resulting voting cards, which are then printed and sent to the voters, are the only output of this procedure. The procedure itself can be executed in a controlled offline environment. After terminating this task, the printing authority is no longer involved in the protocol, i.e., all its resources can be freed for other tasks. In the aftermath of an election, the voting cards of all participating voters can be reconstructed from the publicly available information. In this way, possible frauds or failures by a corrupt printing authority can be detected. It also means that the printing authority does not need to protect any long-term secrecy.

The definition of the parties in the abstract protocol model includes a description of their communication abilities. Properties of corresponding communication channels need to be specified, again in close accordance with a possible real-world setting. In CHVote, several authenticated and one confidential communication channel are needed to meet to protocol's security requirements [9, Figure 6.1]. This implies the existence of a public-key infrastructure (PKI), which needs to be precisely specified as part of the communication model. To minimize the size of

the PKI and the resulting key management overhead, we recommend keeping the number of participating parties (except the voters) as small as possible. Ideally, this PKI can be mapped one-to-one into an implementation of the system.

Item 6: Protocol Structure and Communication Diagrams

A precise and comprehensive description of the voting protocol is the most fundamental system design output. To cope with the overall complexity, we divided the CHVote protocol into three phases and a total of ten sub-phases. We drew protocol diagrams for each of these sub-phases. A portion of one of these diagrams is shown in Figure 1. Each diagram shows the involved parties, the relevant elements of the acquired knowledge, the messages exchanged between the parties, and all conducted computations. The description of the computations involves calls to algorithms, which are given in a separate section (see Item 11). To optimally connect these diagrams with the remaining parts of the document, we strictly applied our consistent set of mathematical notations and symbols (see Item 3). Keeping these diagrams up-to-date and ensuring their correctness and completeness was a constant challenge during the protocol design. Given their fundamental role in the whole system design, we recommend spending sufficient effort to achieve the best possible result. We see the communication diagrams of the protocol as the core of the system's master plan, which does not permit any lack of clarity or unanswered questions.

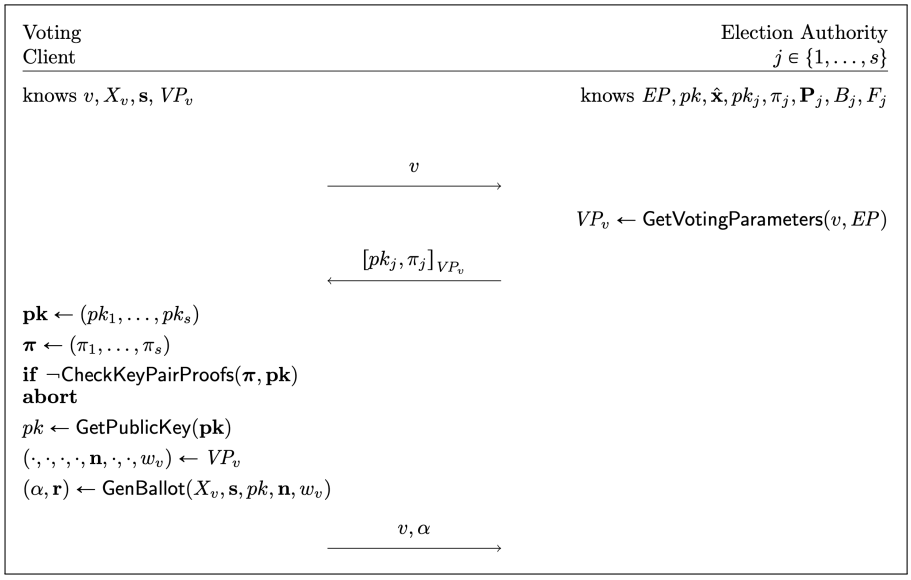

Fig. 1: Exemplary communication diagram: vote casting sub-phase (first part).

Item 7: Pseudo-Code Algorithms

To push the given amount of technical details to the limit, we decided in an early stage of the CHVote project to provide a full set of pseudo-code algorithms for

every computational task in the protocol [9, Section 8]. The current version of the protocol consists of a total of 79 algorithms and sub-algorithms for very different purposes, including primitives for converting basic data types, for computing hash values of complex mathematical objects, or for generating digital signatures. A large portion of the algorithms deals with the core of the CHVote protocol, which realizes a method for transferring verification codes obliviously to the voters in a distributed manner [8]. Other algorithms describe the verifiable mix-net and the distributed decryption process [10, 12]. By maintaining the consistent set of mathematical symbols and notation, this section of the specification document is smoothly integrated into the big picture of the cryptographic protocol. A tremendous amount of initial work, re-factoring, and housekeeping was necessary to reach the stability of the current set of algorithms. Like in regular code, we applied certain pseudo-code style guides to achieve a maximally consistent result. In Figure 2, the algorithm for generating a ballot is given as an example.

Algorithm: $\mathsf{GenBallot}(X, \mathbf{s}, pk, \mathbf{n}, w)$
Input: Voting code $X \in A_X^{\ell_X}$
Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
Encryption key $pk \in \mathbb{G}_q$
Number of candidates $\mathbf{n} = (n_1, \ldots, n_t)$, $n_j \in \mathbb{N}^+$, $n = \sum_{j=1}^{t} n_j$
Counting circle $w \in \mathbb{N}^+$
$x \leftarrow \mathsf{ToInteger}(X, A_x)$ // see Alg. 4.8
$\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$
$\mathbf{p} \leftarrow \mathsf{GetPrimes}(n + w)$ // $\mathbf{p} = (p_0, \ldots, p_{n+w})$, see Alg. 8.1
$\mathbf{m} \leftarrow \mathsf{GetEncodedSelections}(\mathbf{s}, \mathbf{p})$ // $\mathbf{m} = (m_1, \ldots, m_k)$, see Alg. 8.24
$m \leftarrow \prod_{j=1}^{k} m_j$
if $p_{n+w} \cdot m \geqslant p$ **then**
  return $\perp$ // $\mathbf{s}$, $\mathbf{n}$, and w are incompatible with p
$(\mathbf{a}, \mathbf{r}) \leftarrow \mathsf{GenQuery}(\mathbf{m}, pk)$ // $\mathbf{a} = (a_1, \ldots, a_k)$, $\mathbf{r} = (r_1, \ldots, r_k)$, see Alg. 8.25
$r \leftarrow \sum_{j=1}^{k} r_j \bmod q$
$\pi \leftarrow \mathsf{GenBallotProof}(x, m, r, \hat{x}, \mathbf{a}, pk)$ // see Alg. 8.26
$\alpha \leftarrow (\hat{x}, \mathbf{a}, \pi)$
return $(\alpha, \mathbf{r})$ // $\alpha \in \mathbb{G}_{\hat{q}} \times (\mathbb{G}_q^2)^k \times (\mathbb{Z}_{2^\tau} \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q))$, $\mathbf{r} \in \mathbb{Z}_q^k$

Fig. 2: Exemplary pseudo-code algorithm: ballot generation.

To the best of our knowledge, enhancing the specification document of an e-voting system with a complete set of pseudo-code algorithms was a novelty in 2017—and still is today. Our experience with this approach is very positive in almost every respect. First, it added an additional layer to the protocol design, which created an entirely new perspective. Viewing the protocol from this perspective allowed us to recognize certain problems in the protocol design at an early stage. Without detecting them by challenging the protocol from the pseudo-code perspective, they would have come up later during code development.

Another positive effect of releasing pseudo-code algorithms in an early version of the specification document was the possibility of giving third parties the oppor-

tunity to inspect, analyze, or even implement the algorithms (see Item 15). Within a few months, we received feedback from two different implementation projects in different programming languages—from the CHVote developers in Geneva and from students of ours [13, 14]. This feedback was useful for further improving the quality of the specification document, but more importantly, it demonstrated that we managed to considerably reduce the complexity of developing the core tasks of the protocol in a suitable programming language. Our students, for example, who had only little experience in developing cryptographic applications, managed to fully implement all protocol algorithms from scratch in less than four months time. The resulting code from these projects also demonstrated how to almost entirely eliminate the error-prone gap between code and specification. This gap is a typical problem in comparable projects, especially when it comes to check the correctness of the code by external auditors. Without such a gap, auditors can enforce the focus of their inspection to software-development issues. In the light of these remarks, we learned in this project that providing pseudo-code algorithms defines an ideal interface between cryptographers and software developers.

Item 8: Usability and Performance

During the design of the CHVote protocol, we realized that parts of the overall complexity can be left unspecified without affecting the protocol's security properties. We separated some issues that only affect the usability or the performance of the system from the core protocol and discussed them in separate sections.[5] The general idea is to identify aspects that *can* be implemented in a real system or in a certain way, but with no obligation to do so. The benefit of separating them from the core protocol is a higher degree of decoupling in the specification document, which permits discussing corresponding aspects independently of each other. An example of such an aspect is the strict usage of unspecified alphabets for all the codes delivered or displayed to the voters [9, Section 11.1]. Since the actual choice of the alphabets only affects usability (not security), it is something that can be discussed from a pure usability perspective. The situation is similar for various performance improvements, which are optional for an actual implementation. By studying them in a more general context and by publishing the results, our work generated valuable side-products [10, 11].

3 Implementation

Item 9: Mathematical Library

The languages of mathematicians and computer scientists are fairly similar in many respects, but there are also some fundamental differences. One such difference comes from the stateless nature of most mathematical objects, which is very different from mutable data structures in imperative or object-oriented programming languages such as Java. Other differences stem from established

[5] The performance section of the specification document is currently under construction. It will be included in one of the next releases.

conventions. One example of such a convention is the index notation for referring to the elements of a list, vector, or matrix, which usually starts from from 1 in mathematics and from 0 in programming. If a complex cryptographic protocol needs to be translated into programming code, this difference makes the translation process error-prone.

To minimize in our CHVote implementation the difference between specification and code, we introduced a Java library for some additional immutable mathematical objects. The core classes of this library are `Vector`, `Matrix`, `Set`, `ByteArray`, `Alphabet`, and `Tuple` (with sub-classes `Pair`, `Triple`, ...). All of them are strictly generic and immutable. Applying generics in a systematic way greatly improves type-safety, for example in case of complex nested types such as

```
Triple<BigInteger, Vector<String>, Pair<Integer, ByteArray>>.
```

Working with immutable objects has many advantages. They are easier to design, they can always be reused safely, and testing them is much easier [5, Page 80]. `String` and `BigInteger` are examples of given immutable classes in Java. In our mathematical library, we adopted the convention of accessing the elements of a vector of size n with non-zero indices $i \in \{1, \ldots, n\}$, and similarly for matrices and tuples. This delegates the translation between different indexing conventions to theses classes and therefore eliminates the error-proneness of this process. It also creates a one-to-one correspondence between indexing variables in the specification and the code, which is beneficial for the overall code readability.

In our experience of implementing the CHVote protocol, the mathematical library turned out to be a key component for achieving the desired level of code quality in a reasonable amount of time. Given its central role in all parts of the system, we put a lot of effort into performance optimizations, rigorous testing, and documentation. We highly recommend the creation and inclusion of such a library in similar projects.

Item 10: Naming Conventions

Most programming languages have a well-established set of naming conventions. Generally, software developers are advised to "*rarely violate them and never without a very good reason*" [5, Page 289]. Not adhering to the conventions usually lowers the code readability and makes code maintenance unnecessarily complicated, especially if multiple developers are involved. In some situations, deviations from common conventions may even lead to false assumptions and programming errors. In Java, the naming convention for variables, fields, and method parameters is to use a connected sequence of words, with the first letter of each subsequent word capitalized (a.k.a. "camel case"), for example `maxVoterIndex`. Abbreviations such as `max` or single letters such as `i` are allowed, as long as their meaning in the given context remains clear.

In our implementation of the cryptographic protocol, we decided to deviate from general Java naming conventions. To achieve our goal of diminishing the gap between specification and code to the maximal possible degree, we decided to adopt the mathematical symbols from the protocol specification as precisely as possible in the code. This includes defining upper-case variable names in Java such

as `Set<Integer> X` for a set X of integers. In such cases, we prioritized project-internal naming consistency over general Java naming conventions. Tagged, boldface, or Greek variable names are spelled out accordingly, for example $\hat{\alpha}_{ij}$ as `alpha_hat_ij` or $\mathbf{k}'$ as `bold_k_prime`. We strictly applied this pattern throughout all parts of the code. Code that is written in this way may look quite unconventional at first sight, but it turned out to be a key element for making the Java code look almost exactly the same as the pseudo-code. As an example, consider our implementation of the algorithm GenBallot in Figure 3, which closely matches with the pseudo-code from Figure 2.

```java
public class GenBallot extends ch.chvote.algorithms.common.GenBallot {

    public static Pair<Ballot, Vector<BigInteger>>
        run(String X, IntVector bold_s, QuadraticResidue pk, IntVector bold_n, int w, Parameters params) {

        // PREPARATION
        int n = Math.intSum(bold_n);
        Precondition.checkNotNull(X, bold_s, pk, bold_n, params);
        Precondition.check(params.GG_q.contains(pk));
        Precondition.check(IntSet.NN_plus.contains(w));
        Precondition.check(Set.String(params.A_X, params.ell_X).contains(X));
        Precondition.check(Set.IntVector(IntSet.NN_plus).contains(bold_n));
        Precondition.check(Set.IntVector(IntSet.NN_plus(n)).contains(bold_s));
        Precondition.check(bold_s.isSorted());

        // ALGORITHM
        var x = ToInteger.run(X, params.A_X);
        var x_hat = Mod.pow(params.g_hat, x, params.p_hat);
        var bold_p = GetPrimes.run(n + w, params);
        var bold_m = GetEncodedSelections.run(bold_s, bold_p);
        var m = Math.prod(bold_m.map(QuadraticResidue::getValue));
        if (bold_p.getValue(n + w).getValue().multiply(m).compareTo(params.p) >= 0) {
            throw new AlgorithmException(GenBallot.class, AlgorithmException.Type.INCOMPATIBLE_MATRIX);
        }
        var pair = GenQuery.run(bold_m, pk, params);
        var bold_a = pair.getFirst();
        var bold_r = pair.getSecond();
        var r = Mod.sum(bold_r, params.q);
        var pi = GenBallotProof.run(x, Mod.prod(bold_m), r, x_hat, bold_a, pk, params);
        var alpha = new Ballot(x_hat, bold_a, pi);
        return new Pair<>(alpha, bold_r);
    }
```

Fig. 3: Exemplary Java code: ballot generation.

Item 11: Implementation of Pseudo-Code Algorithms

We already discussed our view of the pseudo-code algorithms as an ideal interface between cryptographers specifying the protocol and software developers implementing corresponding code (see Item 7). In such a setting, the implementation of the algorithms inherently defines an important bottom layer of the whole system architecture. To strengthen the overall clarity in our implementation of the algorithms, we decided the create separate utility class for all top-level algorithms. Each of them contains exactly one static method `run(<args>)`, which implements the algorithm (plus static nested classes for all sub-algorithms), for example `GenBallot.run(<args>)` for the algorithm GenBallot. This way of structuring

the algorithm module establishes direct links to the specification document. These links are clearly visible by inspecting the project's package structure. A section of this package structure is shown in Figure 4.

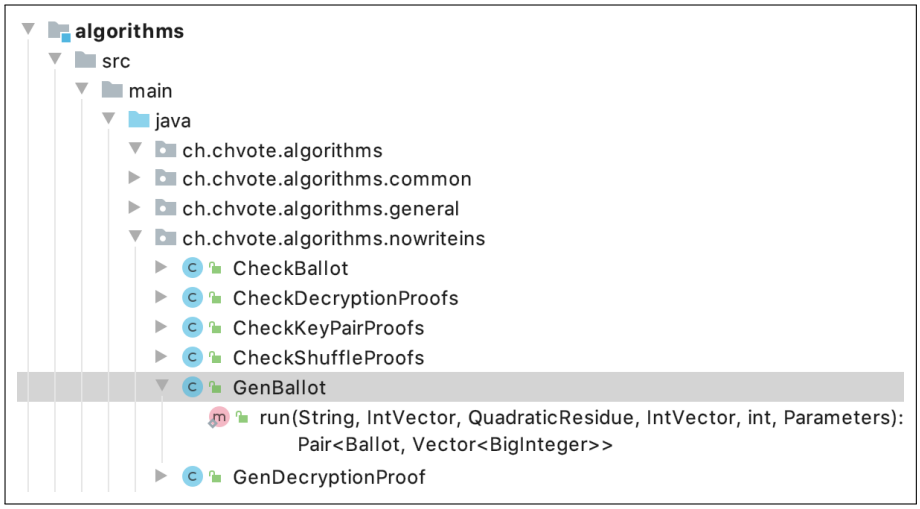

Fig. 4: Package structure of static utility classes for top-level algorithms.

Given the central role of the protocol algorithms for the whole system, we put extra care and effort into developing this part of the code. To obtain the best possible code consistency, we defined a set of project-internal coding style guidelines and applied them strictly to all algorithms. Each algorithm went through an internal reviewing and testing process over multiple rounds, which involved different persons according to the *Four Eyes Principle*. The result is a consistent set of Java methods that are perfectly aligned with the pseudo-code algorithms from the specification. The example shown in Figures 2 and 3 demonstrates how precisely the algorithms have been translated into code.

We see perfect alignment between specification and code as a quality criterion of highest priority. This implies that even the smallest change in either the specification or the code needs to be updated immediately on the other side. The general idea here is to view them as *the same thing*. This view enables third-party auditors that are familiar with the naming conventions and coding style guidelines to check the translation from specification to programming code at minimal costs. We believe that auditing the implementation of the algorithms remains a diligent (but mostly routine) piece of work, which does not necessarily require the involvement of cryptographic experts.

Item 12: Parameter Validity Checks

An important aspect of the proposed way of implementing the protocol algorithms is the introduction of systematic validity checks of all input parameters. These checks complement the built-in type safety obtained from strictly using the generic mathematical library (see Item 9). The domains of all input parameters

are specified in the pseudo-code algorithms, for example $X \in A_X^{\ell_X}$ in GenBallot for a string of characters from the alphabet A_X of length ℓ_X, which translates into the following line of Java code (see Figure 3, Line 36):

```
Set.Strings(params.A_X, params.ell_X).contains(X)
```

Provided that these checks are sufficiently strong for detecting all possibilities of invalid parameters—or invalid combinations of parameters—of a given algorithm, they ensure that the algorithm always outputs a meaningful result. In case of a failed check, it is clear that something else must have gone wrong, for example that a message with a corrupt content has been received or that some stored data has been modified. Every failed check therefore indicates some deviation from a normal protocol run. This is the reason for implementing them in a systematic way for all top-level algorithms (sub-algorithms do not require such checks).

To minimize the overhead of performing these checks each time an algorithm is called, we managed to entirely eliminate expensive computations such as modular exponentiations. To efficiently perform membership tests $x \in \mathbb{G}_q$ for the set $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of quadratic residue modulo a safe prime $p = 2q + 1$, we implemented the *membership witness* method proposed in [10]. The corresponding class `QuadraticResidue`, which realizes this test with a single modular multiplication, is part of our mathematical library. In Figure 3, the parameter `pk` is of that type, and its membership test is conducted in Line 38.

Item 13: Implementation of Protocol Parties

To implement the protocol based on the algorithms, we designed a software component for every involved party. These components share some code for various common tasks, but otherwise they are largely independent. For the design of each party, we derived a state diagram from the protocol description in the specification document. This diagram defines the party's behavior during a protocol run. Typically, receiving a message of certain type triggers the party to perform a transition into the next state. The transition itself consist of computations and messages to be sent to other parties. The computations, which we call *tasks*, can be implemented by calling corresponding protocol algorithms.

The left-hand side of Figure 5 shows the UML state diagram of the printing authority (printer), which consists of two states SP1 and SP2 and one error state EP1. In SP1, the printer expects messages of type MAP1 and MEP1. If all messages are received, the transition into SP2 (or EP1) is triggered. This involves computing task TP1 and sending two types of messages MPV1 and MAX1. The error state EP1 is reached in case of an exception of type AE (algorithm exception) or TE (task exception). This diagram represents the printer's view of the printing sub-phase [9, Protocol 7.2], which is the only sub-phase in which the printer is active. Similar state diagrams exist for all other parties and sub-phases. We defined further naming conventions and strictly applied them to all tasks and message types.

Modeling the parties using the (extended) state machine formalism turned out to be the ideal approach for structuring the parties' implementations in the most natural way. It also allowed us to apply the *state pattern*, one of the well-known

"Gang of Four" design patterns [7, Page 305]. This made our implementation very transparent from a general software-engineering perspective. The right-hand side of Figure 5 shows a section of the package structure, which illustrates for example that the party class `Printer` depends on three state classes `SP1`, `SP2`, and `EP1`, and one task class `TE1`. Every other party is implemented in exactly this way. Every task and every message type is connected to one of the sub-phase diagrams in the protocol specification, and vice versa.

Using the state pattern, we achieve close correspondence between specification and code also on the abstraction layer representing the parties. Again, we see the code and the specification related to the parties as essentially *the same things*, which means that the slightest change on one side needs to be updated immediately on the other side. In this way, we tried to achieve a similar level of structural clarity and code quality as for the algorithm implementation. The state pattern was also useful for establishing the flexibility of running multiple election events simultaneously (possibly using different protocol versions).

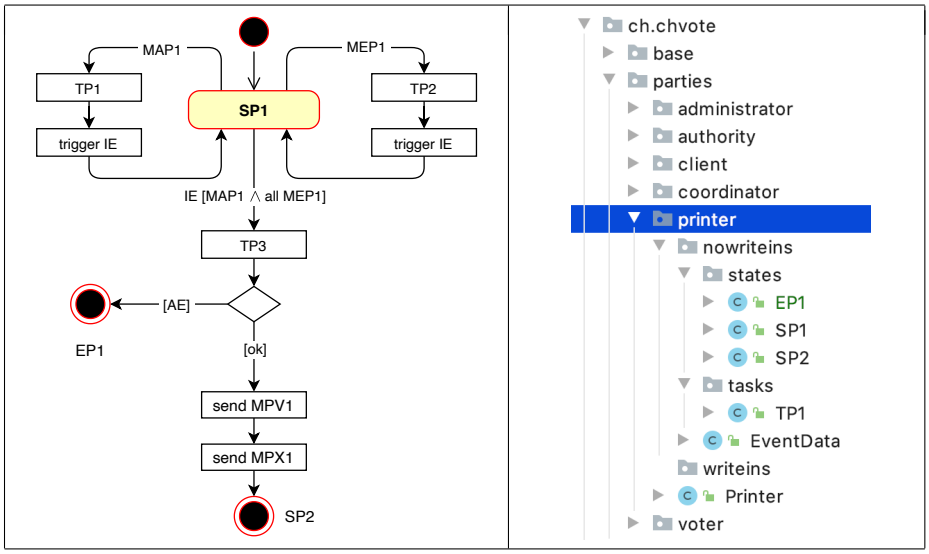

Fig. 5: State diagram of the printer (left) vs. package structure of party classes (right).

Item 14: Cryptographically Relevant Code

Providing code for all algorithms and all parties concludes the implementation of the cryptographically relevant part of the protocol. This is where flaws in the code can cause critical errors or vulnerabilities. Generally, we recommend structuring the software design into *cryptographically relevant* and *cryptographically irrelevant* components and to link them over suitable interfaces. Our current implementation of the CHVote protocol is limited to the cryptographically relevant part of the system, but we provide the required interfaces, for example for connecting our code to concrete high-performance messaging and persistence services.

For testing purposes, we only implemented these interfaces in a rudimentary way, but this turned out to be sufficient for simulating even the most complex election use case from top to bottom. Such a simulation can be conducted on a single machine using any common development environment, i.e., no complex installation of a distributed test environment over multiple servers is required. This is an efficient environment for running all sorts of functional tests with a clear focus on the cryptographic protocol. With almost no communication overhead, it is also ideal for analyzing and optimizing the overall protocol performance. A precondition for establishing a complete test run is the implementation of all protocol parties, including the (human) voters. Even if corresponding code will obviously not be included in a real-world deployment of the system, we see it as an indispensable component of our implementation.

Given its central role in the overall security of the system, we tried to make the cryptographically relevant part of the code accessible to the broadest possible audience. For that, we decided to avoid dependencies to complex third-party libraries or software frameworks as far as possible. We only admitted two dependencies to the widely used native GMP library for efficient computations with large numbers and to the Spock framework for enabling data-driven tests. Both libraries are almost entirely invisible in our implementation, i.e., there is no need to familiarize reviewers with these technologies (except for reviewing the tests). Generally, we see complex frameworks based on annotation, reflection, or injection mechanisms as unsuited for developing cryptographically relevant code. They are great for implementing enterprise software components at minimal costs, but they often tend to obscure the general program flow. This reduces the overall code readability and makes static code analysis more difficult.

4 Project Management

Item 15: Transparency

We started this project from the beginning with the mindset of maximal transparency. At an early stage of the project in 2017, we published the first version of our specification document [9]. At that time, we had already published a peer-reviewed paper describing the cryptographic core of the protocol [8]. The feedback that we received, mostly from members of the e-voting community, was very useful for improving the protocol and its security properties. The most important feedback came from Tomasz Truderung on April 19, 2017, who found a subtle but serious flaw in the construction of our protocol. This flaw had been overlooked by the reviewers of the published paper. After a few weeks, we were able fix the problem to a full extent and update the protocol accordingly. In the meantime, the success of the entire project was at stake.

We recall this anecdote here for making two important points. First, releasing specification documents of an e-voting project usually launches a public examination process in the community. The outcome of this process is sometimes unpredictable, but the received feedback has the potential of greatly improving the quality of the protocol. At the time of writing this document, we have not yet

released the source code for public examination, but we expect a similar amount of interest and feedback from the community. Second, a cryptographic protocol without formal security definitions and rigorous proofs provides not a sufficiently solid foundations for building a system. In CHVote, a different group of academics was contracted by the State of Geneva to perform this task. The outcome of this sister project was released in 2018 [3]. The high quality of their work leads one to suppose that the above-mentioned flaw would have been detected in their analysis. Unfortunately, their report has not yet been updated to the current version of the protocol.

In this project, our mindset of maximal transparency always allowed us to openly discuss all aspects of our work with many different people, including students of ours who developed various prototypes [13, 14]. This created a permanent challenge for the cryptographic protocol, which forced us to constantly question our design decisions and improve our technical solutions. We conclude that releasing all cryptographically relevant documents as a matter of principle was fundamental for the success of the project. More generally, we see it as an important trust-establishing measure.

Item 16: Verifier

The last point we want to mention in this paper is an important aspect for a verifiable e-voting system. Unfortunately, we were not yet able to cover it in this project. It's about specifying the verification software—sometimes called *the verifier*—for the proposed protocol. In the original project setting of the State of Geneva, it was planned to outsource the specification and development of the verifier to a third-party institution. To establish a certain degree of independence between the protocol and the verifier, this decision of the project owners was perfectly understandable. We never questioned this decision, but it prevented us from paying enough attention to this important topic. When the project was dropped in fall 2018, the outsourced verifier project had started, but it was not yet very advanced. This finally led to the current situation, where the specification and the implementation of the e-voting protocol are both very advanced, but almost nothing is available for the verifier. Even though, the e-voting protocol describes how to verify certain cryptographic aspects, but that is not to be confused with the complete verification of the whole voting process.

We believe that in projects like this, it's best to let the specification of the protocol and the verifier go hand in hand, and to apply the same level of preciseness and completeness to both of them. We see the verifier as the ultimate way of challenging the protocol run, both in the abstract setting of the specification document and in the concrete setting of executing the code on real machines. So far, this challenge is missing in our project.

5 Conclusion

In software development, best practices are available in many areas. They are very useful for developers to avoid bad design decisions and typical programming

mistakes. This certainly also holds for developing an e-voting system, but the delicacy of implementing a cryptographic protocol makes the situation a bit more complicated. We therefore believe that the e-voting community should come up with its own set of best practices and define respective minimal standards. This paper makes a first step into this directions based on our experience from the CHVote project. Among the discussed sixteen topics, we believe that the advice of providing all algorithmic details in pseudo-code is the most important one, together with structuring the source code into a cryptographically relevant and a cryptographically irrelevant part.

References

1. *Verordnung der Bundeskanzlei über die elektronische Stimmabgabe (VEleS) vom 13. Dezember 2013 (Stand 1. Juli 2018).* Die Schweizerische Bundeskanzlei (BK), 2018.
2. E. Barker. Recommendation for key management. NIST Special Publication 800-57, Part 1, Rev. 5, NIST, 2020.
3. D. Bernhard, V. Cortier, P. Gaudry, M. Turuani, B. Warinschi. Verifiability analysis of CHVote. *IACR Cryptology ePrint Archive*, 2018/1052, 2018.
4. D. Bernhard, O. Pereira, B. Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. *ASIACRYPT'12, 18th International Conference on the Theory and Application of Cryptology and Information Security*, LNCS 7658, pages 626–643, Beijing, China, 2012.
5. J. Bloch. *Effective Java.* Addison-Wesley, 3rd edition, 2018.
6. A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. *CRYPTO'86, 6th International Cryptology Conference on Advances in Cryptology*, LNCS 263, pages 186–194, Santa Barbara, USA, 1986.
7. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
8. R. Haenni, R. E. Koenig, E. Dubuis. Cast-as-intended verification in electronic elections based on oblivious transfer. *E-Vote-ID'16, 1st International Joint Conference on Electronic Voting*, LNCS 10141, pages 277–296, Bregenz, Austria, 2016.
9. R. Haenni, R. E. Koenig, P. Locher, E. Dubuis. CHVote system specification – version 3.0. *IACR Cryptology ePrint Archive*, 2017/325, 2020.
10. R. Haenni and P. Locher. Performance of shuffling: Taking it to the limits. *Voting'20, FC 2020 International Workshops*, Kota Kinabalu, Malaysia, 2020.
11. R. Haenni, P. Locher, N. Gailly. Improving the performance of cryptographic voting protocols. *Voting'19, FC 2019 International Workshops*, LNCS 11599, pages 272–288, Frigate Bay, St. Kitts and Nevis, 2019.
12. R. Haenni, P. Locher, R. E. Koenig, E. Dubuis. Pseudo-code algorithms for verifiable re-encryption mix-nets. *Voting'17, FC 2017 International Workshops*,, LNCS 10323, pages 370–384, Silema, Malta, 2017.
13. K. Häni and Y. Denzer. CHVote prototype in Python. Project report, Bern University of Applied Sciences, Biel, Switzerland, 2017.
14. K. Häni and Y. Denzer. Visualizing Geneva's next generation e-voting system. Bachelor thesis, Bern University of Applied Sciences, Biel, Switzerland, 2018.
15. S. J. Lewis, O. Pereira, V. Teague. How not to prove your election outcome. Technical report, 2019.
16. U. Maurer and C. Casanova. Bericht des Bundesrates zu Vote électronique. 3. Bericht, Schweizerischer Bundesrat, 2013.