SECURITY BY DESIGN IN SWISS POST’S NEW E-VOTING SYSTEM

WELCOME TO OUR EXPERT WEBINAR

10 June 2021

DIE POST

SPEAKERS

Xavier Monnat
Product manager e-voting
Swiss Post

Olivier Esseiva
Cryptographer
Swiss Post

Hadrien Renold
Software engineer
Swiss Post

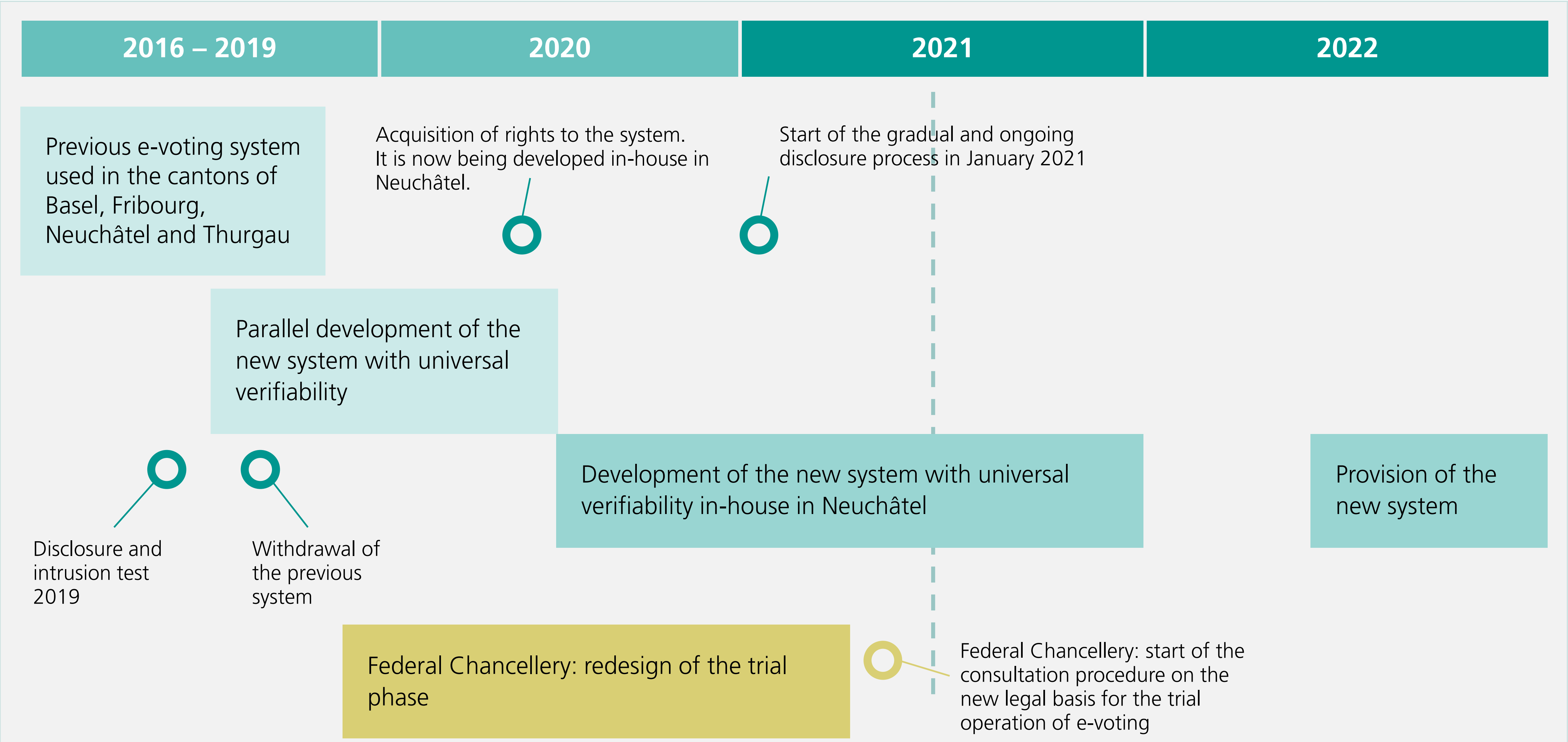
ORGANIZATION

- **Microphone**: Please put your microphone on mute during the presentations.
- **Video**: Please switch your camera on so we can engage in lively discussion despite the virtual event format.
- **Questions**: Please submit your questions in the chat during the presentations or ask them directly in the Q&A session at the end.
- **Recording**: The webinar will be recorded and then made available to you afterwards.

SWISS POST’S E-VOTING SYSTEM: CURRENT STATUS AND OUTLOOK

Xavier Monnat

Review and outlook



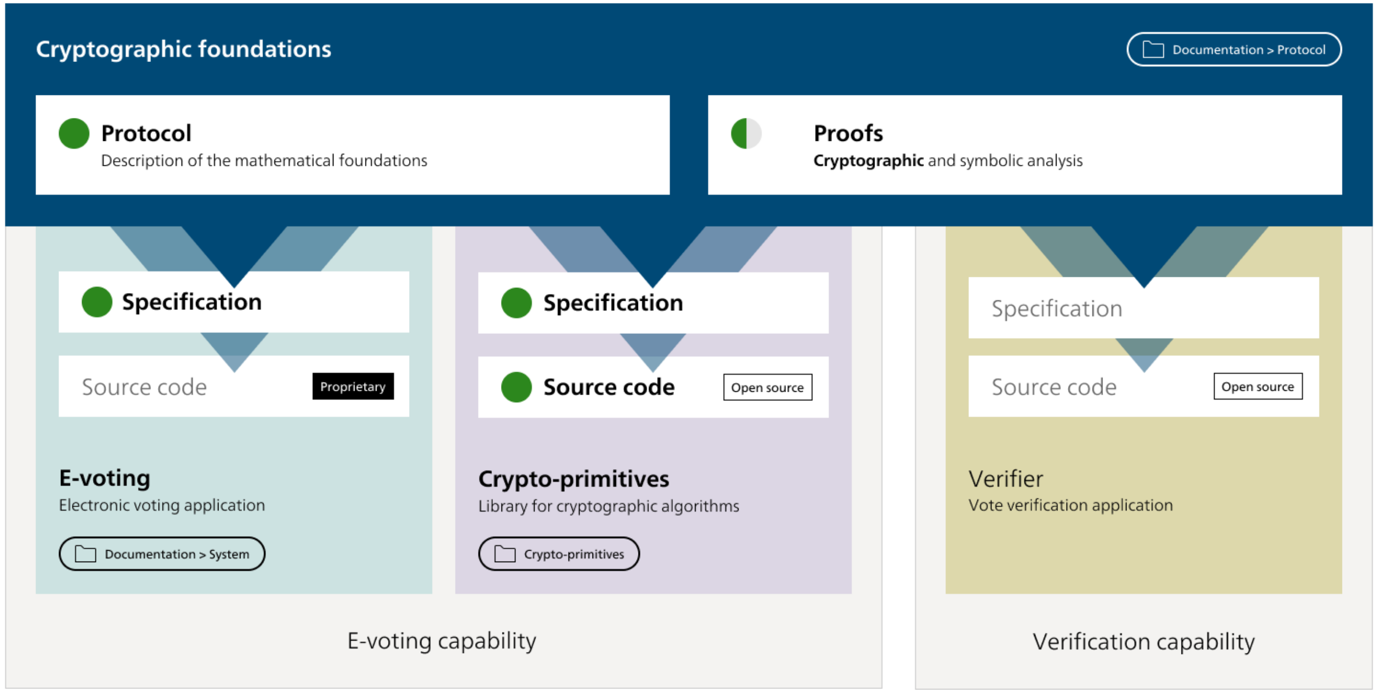
System disclosure
Community programme

- Swiss Post is gradually disclosing the beta version of its new system.
- When preparing the community programme, Swiss Post incorporated feedback from experts on the planned approach.
- Anyone who is interested can take part in this community programme.
- A concise Code of Conduct governs the conditions of participation.
- No registration is required.
- Full information can be found at www.swisspost.ch/e-voting-community.

The aim is to make it as easy as possible for independent experts to access and analyse the system, so that they can test it and report any vulnerabilities to us.

System disclosure
Current status

- The cryptographic protocol, the cryptographic primitives and the specifications have already been published.
- Swiss Post is publishing all reports that it receives on GitLab.
- It is in contact with the experts who have submitted reports.
- Around 20 reports have been received since the start of disclosure.
- Community research programme (BugBounty).



Swiss Post e-voting system

The community programme is up and running. Swiss Post has made various improvements based on the reports received.

What happens next?

- The beta version of the e-voting system will be completely disclosed in a few months.
- The permanent public bug bounty programme for e-voting is also being launched.
- Swiss Post is set to carry out an intrusion test in the autumn.
- **The scope of the community research programme will gradually be expanded.**
- Our aim is to make the system available for use in the cantons during the course of 2022.

SECURITY BY DESIGN AND PUBLIC SCRUTINY

Olivier Esseiva & Hadrien Renold

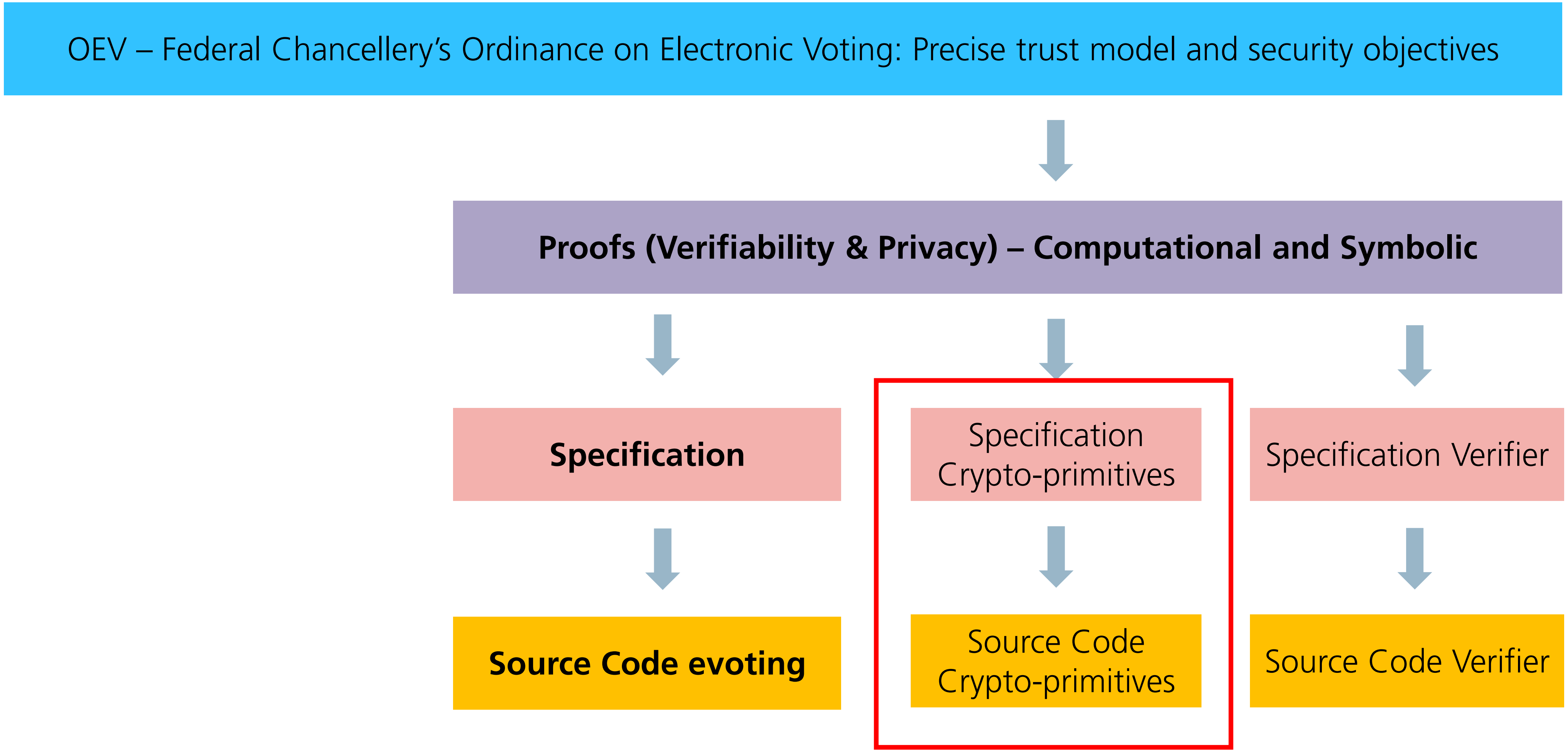
Agenda

- Security by design
  - How do we define a “secure” e-voting system?
  - How do we make sure that the code matches the design?
- Public Scrutiny in practice
  - Current experiences in our public GitLab Repository
  - Open-Source crypto-primitives

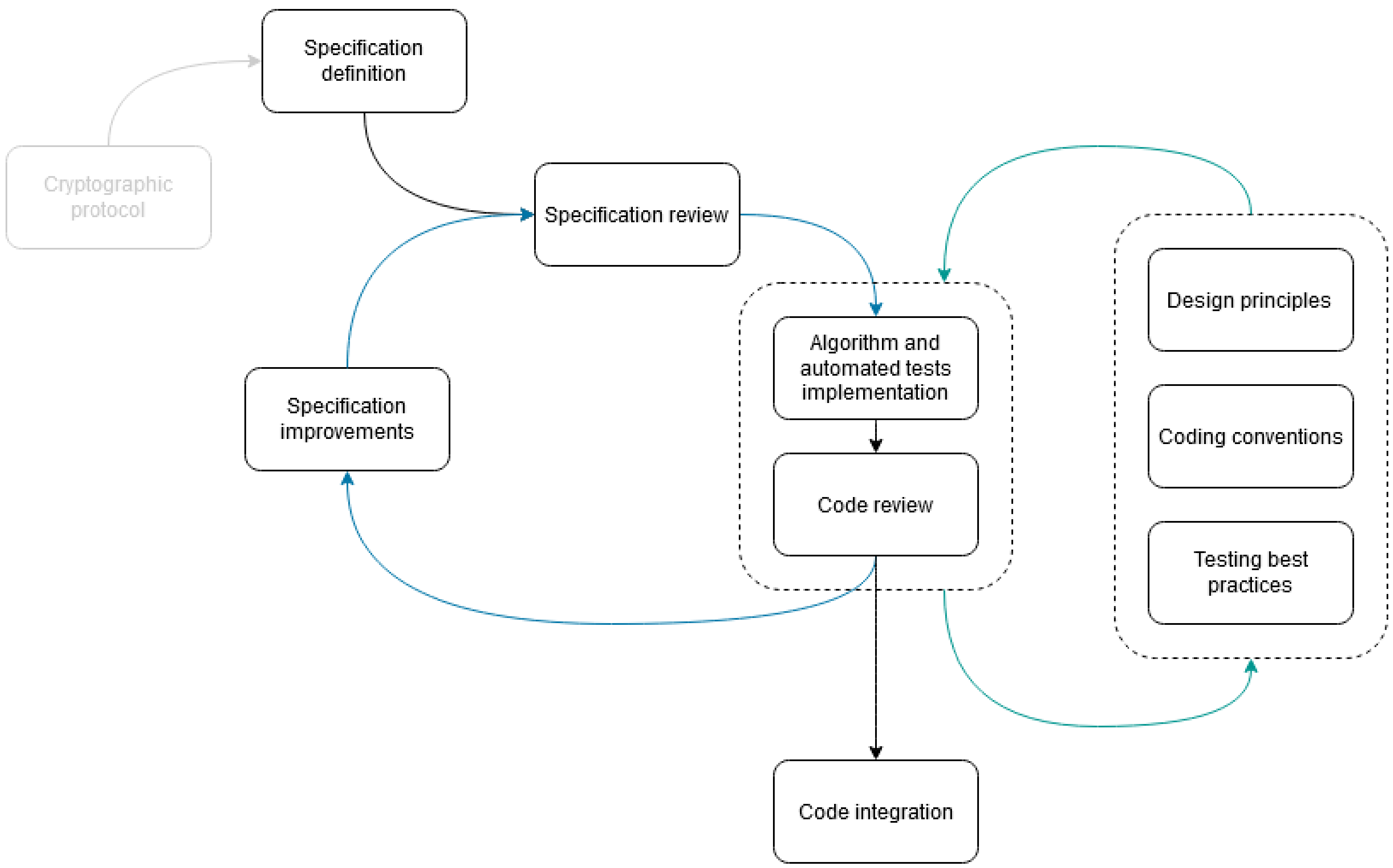
SECURITY BY DESIGN

Olivier Esseiva & Hadrien Renold

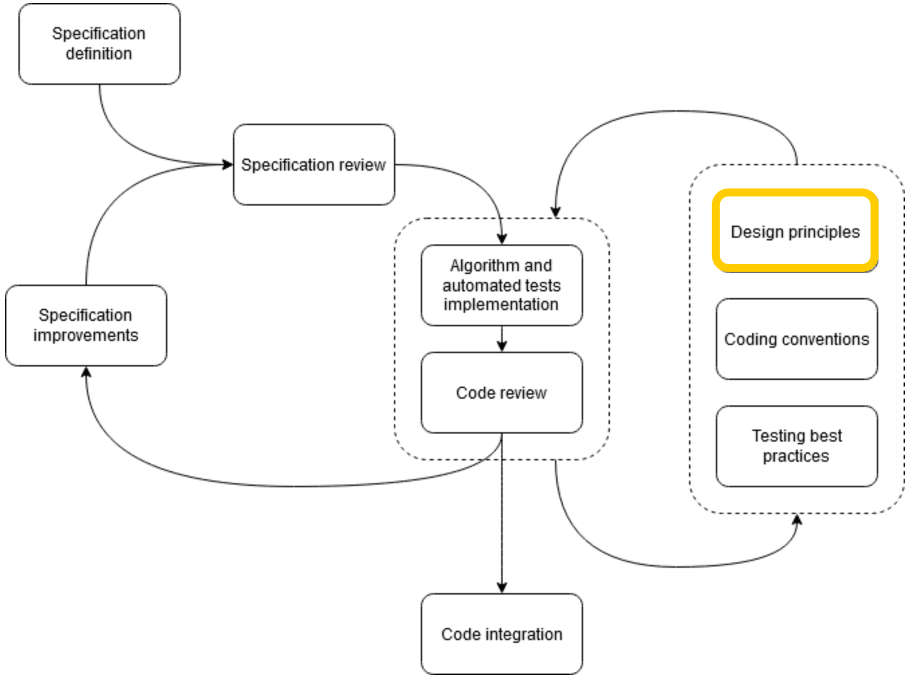
Hierarchy of artefacts



Crypto primitives software development process

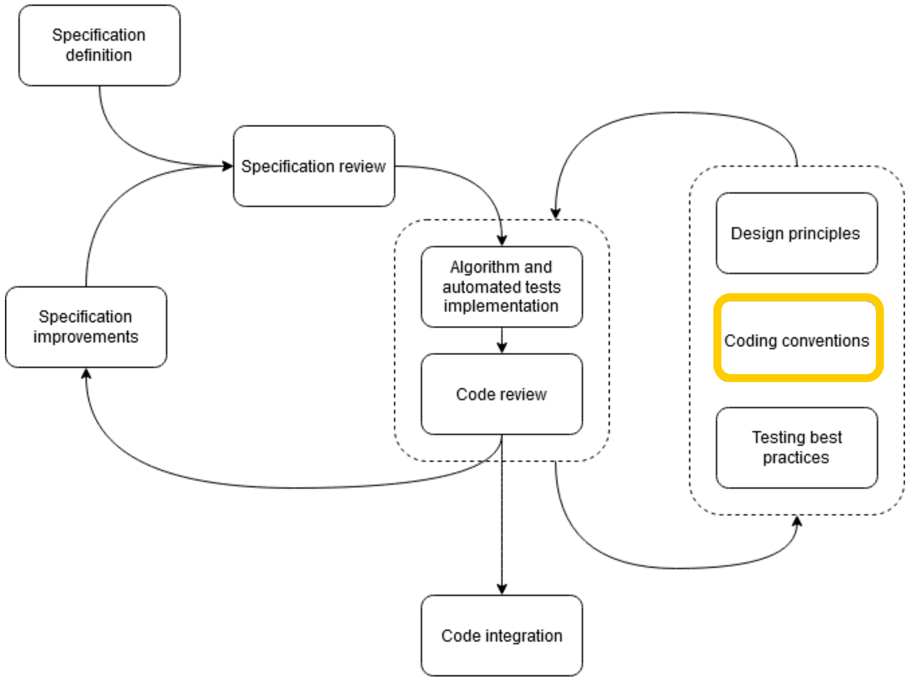


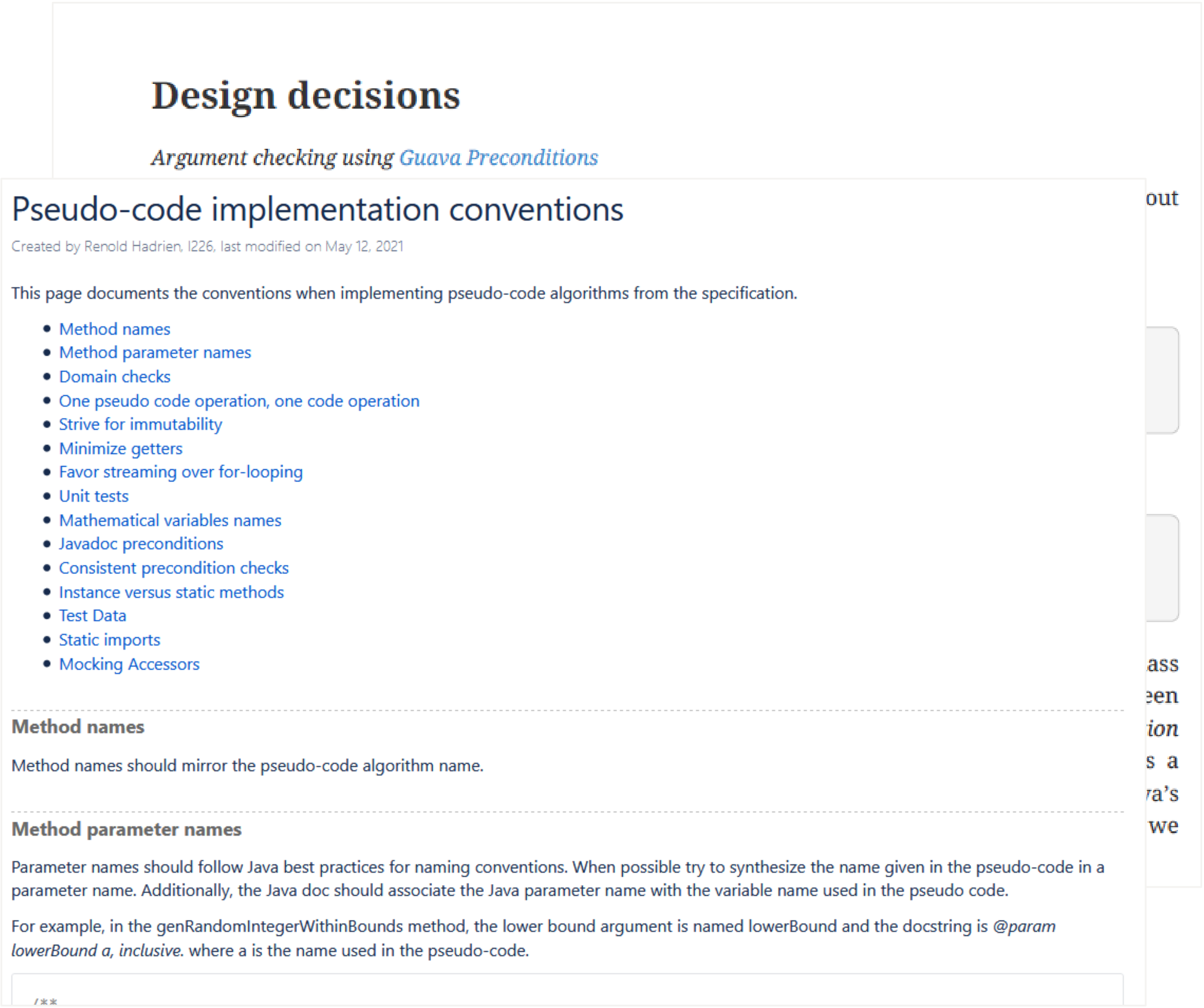
Design principles



- Auditability
- Easy mapping between specification and code
- Defence in depth
- Misuse-resistant
- Consistency
- Maintainability
- Immutability

Coding conventions





Design decisions

Argument checking using Guava Preconditions

Pseudo-code implementation conventions

Created by Renold Hadrien, I226, last modified on May 12, 2021

This page documents the conventions when implementing pseudo-code algorithms from the specification.

- Method names
- Method parameter names
- Domain checks
- One pseudo code operation, one code operation
- Strive for immutability
- Minimize getters
- Favor streaming over for-looping
- Unit tests
- Mathematical variables names
- Javadoc preconditions
- Consistent precondition checks
- Instance versus static methods
- Test Data
- Static imports
- Mocking Accessors

Method names

Method names should mirror the pseudo-code algorithm name.

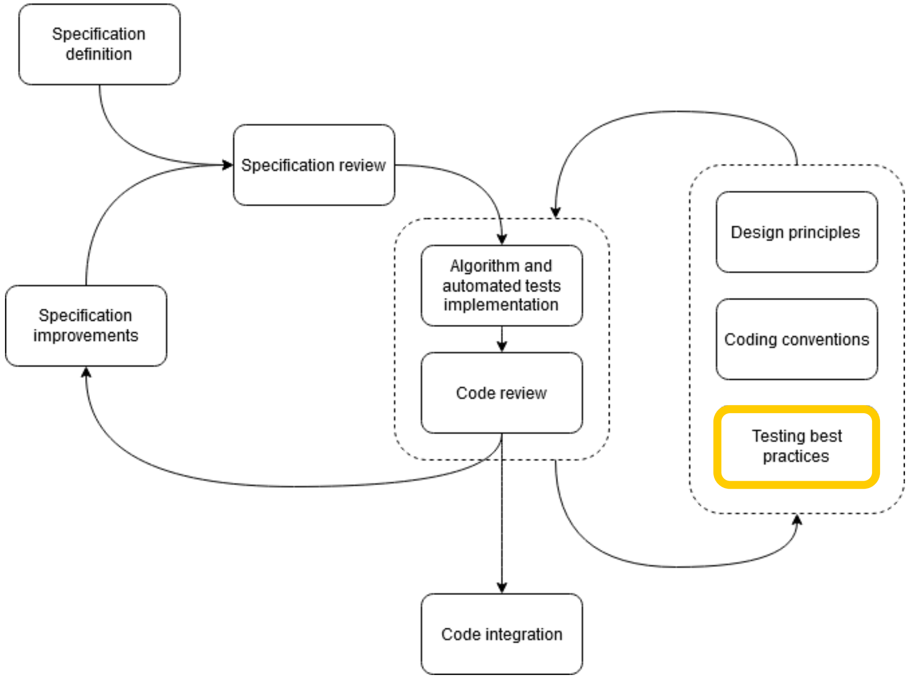
Method parameter names

Parameter names should follow Java best practices for naming conventions. When possible try to synthesize the name given in the pseudo-code in a parameter name. Additionally, the Java doc should associate the Java parameter name with the variable name used in the pseudo code.

For example, in the genRandomIntegerWithinBounds method, the lower bound argument is named lowerBound and the docstring is *@param lowerBound a, inclusive.* where a is the name used in the pseudo-code.

```
/**
```

Testing best practices



Types of tests:

- Unit tests
- Property based tests
- Hand computed values
- Independent implementation test data

Algorithm implementation
Example

Algorithm 5.11 GetShuffleArgument: compute a cryptographic argument for the validity of the shuffle

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \dots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$
- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$

The witness composed of
- permutation $\pi \in \Sigma_N$
- randomness $\vec{\rho} \in \mathbb{Z}_q^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$
The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$

Ensure: $\forall i \in [0, N) : \vec{C}'_i = \mathsf{GetCiphertextProduct}(\mathsf{GetCiphertext}(\vec{1}, \rho_i, \mathbf{pk}), \vec{C}_{\pi(i)})$
Ensure: $N = mn$

Operation:

1: $\mathbf{r} \leftarrow \mathsf{GenRandomVector}(q, m)$ ▷ See algorithm 3.2
2: $A \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{\pi(i)\}_{i=0}^{N-1}, m, n))$ ▷ Create a $n \times m$ matrix. See algorithm 5.14 and algorithm 5.13
3: $\mathbf{c}_A \leftarrow \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$ ▷ See algorithm 5.8
4: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A))$
5: $\mathbf{s} \leftarrow \mathsf{GenRandomVector}(q, m)$
6: $\mathbf{b} \leftarrow \{x^{\pi(i)}\}_{i=0}^{N-1}$
7: $B \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\mathbf{b}, m, n))$
8: $\mathbf{c}_B \leftarrow \mathsf{GetCommitmentMatrix}(B, \mathbf{s}, \mathbf{ck})$
9: $y \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A))$
10: $z \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\text{"1"}, \mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A))$
▷ Both $\vec{C}$ and $\vec{C}'$ are passed in the vector forms here
11: $\mathsf{Zneg} \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{-z\}_{i=1}^{N}, m, n))$ ▷ Vector of length N, with all values being $q - z$
12: $\mathbf{c}_{-z} \leftarrow \mathsf{GetCommitmentMatrix}(\mathsf{Zneg}, \vec{0}, \mathbf{ck})$ ▷ A vector of length m, with all 0 values
13: $\mathbf{c}_D \leftarrow \mathbf{c}_A^y \mathbf{c}_B$ ▷ Entry-wise product
14: $D \leftarrow yA + B$
15: $\mathbf{t} \leftarrow y\mathbf{r} + \mathbf{s}$
16: $b \leftarrow \prod_{i=0}^{N-1}(yi + x^i - z)$
17: $\mathsf{pStatement} \leftarrow (\mathbf{c}_D \mathbf{c}_{-z}, b)$
18: $\mathsf{pWitness} \leftarrow (D + \mathsf{Zneg}, \mathbf{t})$
19: $\mathsf{productArgument} \leftarrow \mathsf{GetProductArgument}(\mathsf{pStatement}, \mathsf{pWitness})$ ▷ See algorithm 5.18
20: $\rho \leftarrow q - (\vec{\rho} \cdot \mathbf{b})$ ▷ Standard inner product $\sum_{i=0}^{N-1} \rho_i b_i$
21: $\vec{x} \leftarrow \{x^i\}_{i=0}^{N-1}$
22: $C \leftarrow \mathsf{GetCiphertextVectorExponentiation}(\vec{C}, \vec{x})$ ▷ See algorithm 4.6
23: $\mathsf{mStatement} \leftarrow (\mathsf{ToMatrix}(\vec{C}', m, n), C, \mathbf{c}_B)$ ▷ See algorithm 5.13
24: $\mathsf{mWitness} \leftarrow (B, \mathbf{s}, \rho)$
25: $\mathsf{multiExponentiationArgument} \leftarrow \mathsf{GetMultiExponentiationArgument}(\mathsf{mStatement}, \mathsf{mWitness})$ ▷ See algorithm 5.15

Output:

shuffleArgument $(\mathbf{c}_A, \mathbf{c}_B, \mathsf{productArgument}, \mathsf{multiExponentiationArgument}) \in \mathbb{G}_q^m \times \mathbb{G}_q^m \times \dots \times \dots$
▷ See algorithm 5.18 and algorithm 5.15 for their respective domains

Algorithm implementation

Example

Algorithm 5.11 GetShuffleArgument: compute a cryptog

of the shuffle

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Group generator $g \in \mathbb{G}_q$

A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$

A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of

- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$

- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$

The witness composed of

- permutation $\pi \in \Sigma_N$

- randomness $\vec{\rho} \in \mathbb{Z}_q^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$

The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$

Ensure: $\forall i \in [0, N) : \vec{C}'_i =$ GetCiphertextProduct(GetCiphertext($\vec{1}, \rho_i, \mathbf{pk}$)

Ensure: $N = mn$

```java
/**
 * Computes a cryptographic argument for the validity of the shuffle. The statement and witness must comply with the following:
 *
 * <ul>
 *     <li>be non null</li>
 *     <li>the statement and witness values must have the same size</li>
 *     <li>the statement's ciphertexts group and the witness's randomness group must be of the same order</li>
 *     <li>re-encrypting and shuffling the statement ciphertexts C with the witness randomness and permutation must give the statem
 *     ciphertexts C'</li>
 *     <li>the size N of all inputs must satisfy N = m * n</li>
 * </ul>
 *
 * @param statement the {@link ShuffleStatement} for the shuffle argument.
 * @param witness   the {@link ShuffleWitness} for the shuffle argument.
 * @param m         the number of rows to use for ciphertext matrices. Strictly positive integer.
 * @param n         the number of columns to use for ciphertext matrices. Strictly positive integer.
 * @return a {@link ShuffleArgument}.
 */
ShuffleArgument getShuffleArgument(final ShuffleStatement statement, final ShuffleWitness witness, final int m, final int n) {
```

Algorithm implementation

Example

Algorithm 5.11 GetShuffleArgument: compute a cryptog
of the shuffle

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$
- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$

The witness composed of
- permutation $\pi \in \Sigma_N$
- randomness $\vec{\rho} \in \mathbb{Z}_q^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$
The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$

Ensure: $\forall i \in [0, N) : \vec{C}'_i = \mathsf{GetCiphertextProduct}(\mathsf{GetCiphertext}(\vec{1}, \rho_i, \mathbf{pk})$

Ensure: $N = mn$

```
/**
 * Computes a cryptographic argument for the validity of the shuffle. The statement and witness must comply with the following:
 *
 * <ul>
 *     <li>be non null</li>
 *     <li>the statement and witness values must have the same size</li>
 *     <li>the statement's ciphertexts group and the witness's randomness group must be of the same order</li>
 *     <li>re-encrypting and shuffling the statement ciphertexts C with the witness randomness and permutation must give the statem
 *     ciphertexts C'</li>
 *     <li>the size N of all inputs must satisfy N = m * n</li>
 * </ul>
 *
 * @param statement the {@link ShuffleStatement} for the shuffle argument.
 * @param witness   the {@link ShuffleWitness} for the shuffle argument.
 * @param m         the number of rows to use for ciphertext matrices. Strictly positive integer.
 * @param n         the number of columns to use for ciphertext matrices. Strictly positive integer.
 * @return a {@link ShuffleArgument}.
 */
ShuffleArgument getShuffleArgument(final ShuffleStatement statement, final ShuffleWitness witness, final int m, final int n) {
```

Algorithm implementation

Example

Algorithm 5.11 GetShuffleArgument: compute a cryptog

of the shuffle

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Group generator $g \in \mathbb{G}_q$

A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$

A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of

- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$
- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$

The witness composed of

- permutation $\pi \in \Sigma_N$
- randomness $\vec{\rho} \in \mathbb{Z}_q^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$

The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$

Ensure: $\forall i \in [0, N) : \vec{C}'_i = \text{GetCiphertextProduct}(\text{GetCiphertext}(\vec{1}, \rho_i, \mathbf{pk})$

Ensure: $N = mn$

```java
/**
 * Computes a cryptographic argument for the validity of the shuffle. The statement and witness must comply with the following:
 *
 * <ul>
 *     <li>be non null</li>
 *     <li>the statement and witness values must have the same size</li>
 *     <li>the statement's ciphertexts group and the witness's randomness group must be of the same order</li>
 *     <li>re-encrypting and shuffling the statement ciphertexts C with the witness randomness and permutation must give the statem
 *     ciphertexts C'</li>
 *     <li>the size N of all inputs must satisfy N = m * n</li>
 * </ul>
 *
 * @param statement the {@link ShuffleStatement} for the shuffle argument.
 * @param witness   the {@link ShuffleWitness} for the shuffle argument.
 * @param m         the number of rows to use for ciphertext matrices. Strictly positive integer.
 * @param n         the number of columns to use for ciphertext matrices. Strictly positive integer.
 * @return a {@link ShuffleArgument}.
 */
ShuffleArgument getShuffleArgument(final ShuffleStatement statement, final ShuffleWitness witness, final int m, final int n) {
```

Algorithm implementation

Example

Algorithm 5.11 GetShuffleArgument: compute a cryptog[...]
of the shuffle

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:
The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$
- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$

The witness composed of
- permutation $\pi \in \Sigma_N$
- randomness $\vec{\rho} \in \mathbb{Z}_q^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$
The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$

Ensure: $\forall i \in [0, N) : \vec{C}'_i = \mathsf{GetCiphertextProduct}(\mathsf{GetCiphertext}(\vec{1}, \rho_i, \mathbf{pk})$

Ensure: $N = mn$

```java
/**
 * Computes a cryptographic argument for the validity of the shuffle. The statement and witness must comply with the following:
 *
 * <ul>
 *     <li>be non null</li>
 *     <li>the statement and witness values must have the same size</li>
 *     <li>the statement's ciphertexts group and the witness's randomness group must be of the same order</li>
 *     <li>re-encrypting and shuffling the statement ciphertexts C with the witness randomness and permutation must give the stater
 *     ciphertexts C'</li>
 *     <li>the size N of all inputs must satisfy N = m * n</li>
 * </ul>
 *
 * @param statement the {@link ShuffleStatement} for the shuffle argument.
 * @param witness   the {@link ShuffleWitness} for the shuffle argument.
 * @param m         the number of rows to use for ciphertext matrices. Strictly positive integer.
 * @param n         the number of columns to use for ciphertext matrices. Strictly positive integer.
 * @return a {@link ShuffleArgument}.
 */
ShuffleArgument getShuffleArgument(final ShuffleStatement statement, final ShuffleWitness witness, final int m, final int n) {
```

Algorithm implementation

Example

Algorithm 5.11 GetShuffleArgument: compute a cryptog
of the shuffle

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$
- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$

The witness composed of
- permutation $\pi \in \Sigma_N$
- randomness $\vec{\rho} \in \mathbb{Z}_q^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$
The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$

Ensure: $\forall i \in [0, N) : \vec{C}'_i =$ GetCiphertextProduct(GetCiphertext($\vec{1}, \rho_i, \mathbf{pk}$)

Ensure: $N = mn$

```java
checkNotNull(statement);
checkNotNull(witness);

checkArgument(m > 0, "The number of rows for the ciphertext matrices must be strictly positive.");
checkArgument(n > 0, "The number of columns for the ciphertext matrices must be strictly positive.");

final GroupVector<ElGamalMultiRecipientCiphertext, GqGroup> ciphertextsC = statement.getCiphertexts();
final GroupVector<ElGamalMultiRecipientCiphertext, GqGroup> shuffledCiphertextsCPrime = statement.getShuffledCiphertexts();
final Permutation permutation = witness.getPermutation();
final GroupVector<ZqElement, ZqGroup> randomness = witness.getRandomness();

// Cross dimensions checking.
checkArgument(ciphertextsC.size() == permutation.getSize(),
                "The statement ciphertexts must have the same size as the permutation.");

// Cross group checking.
checkArgument(ciphertextsC.getGroup().hasSameOrderAs(randomness.getGroup()),
                "The randomness group must have the order of the ciphertexts group.");

// Ensure the statement corresponds to the witness.
final GqGroup gqGroup = ciphertextsC.getGroup();
final ZqGroup zqGroup = randomness.getGroup();
final int N = permutation.getSize();
final int l = ciphertextsC.get(0).size();

checkArgument(l <= publicKey.size(), "The ciphertexts must be smaller than the public key.");

final ElGamalMultiRecipientMessage ones = ElGamalMultiRecipientMessage.ones(gqGroup, l);
final List<ElGamalMultiRecipientCiphertext> encryptedOnes = randomness.stream()
                .map(rho -> getCiphertext(ones, rho, publicKey))
                .collect(toList());
final List<ElGamalMultiRecipientCiphertext> shuffledCiphertexts = permutation.stream()
                .mapToObj(ciphertextsC::get)
                .collect(toList());
```

Algorithm implementation

Example

Algorithm 5.11 GetShuffleArgument: compute a cryptog
of the shuffle

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:
The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$
- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$

The witness composed of
- permutation $\pi \in \Sigma_N$
- randomness $\vec{\rho} \in \mathbb{Z}_q^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$
The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$

Ensure: $\forall i \in [0, N) : \vec{C}'_i =$ GetCiphertextProduct(GetCiphertext($\vec{1}, \rho_i, \mathbf{pk}$)
Ensure: $N = mn$

```java
checkNotNull(statement);
checkNotNull(witness);

checkArgument(m > 0, "The number of rows for the ciphertext matrices must be strictly positive.");
checkArgument(n > 0, "The number of columns for the ciphertext matrices must be strictly positive.");

final GroupVector<ElGamalMultiRecipientCiphertext, GqGroup> ciphertextsC = statement.getCiphertexts();
final GroupVector<ElGamalMultiRecipientCiphertext, GqGroup> shuffledCiphertextsCPrime = statement.getShuffledCiphertexts();
final Permutation permutation = witness.getPermutation();
final GroupVector<ZqElement, ZqGroup> randomness = witness.getRandomness();

// Cross dimensions checking.
checkArgument(ciphertextsC.size() == permutation.getSize(),
                "The statement ciphertexts must have the same size as the permutation.");

// Cross group checking.
checkArgument(ciphertextsC.getGroup().hasSameOrderAs(randomness.getGroup()),
                "The randomness group must have the order of the ciphertexts group.");

// Ensure the statement corresponds to the witness.
final GqGroup gqGroup = ciphertextsC.getGroup();
final ZqGroup zqGroup = randomness.getGroup();
final int N = permutation.getSize();
final int l = ciphertextsC.get(0).size();

checkArgument(l <= publicKey.size(), "The ciphertexts must be smaller than the public key.");

final ElGamalMultiRecipientMessage ones = ElGamalMultiRecipientMessage.ones(gqGroup, l);
final List<ElGamalMultiRecipientCiphertext> encryptedOnes = randomness.stream()
                .map(rho -> getCiphertext(ones, rho, publicKey))
                .collect(toList());
final List<ElGamalMultiRecipientCiphertext> shuffledCiphertexts = permutation.stream()
                .mapToObj(ciphertextsC::get)
                .collect(toList());
```

Algorithm implementation

Example

Algorithm 5.11 GetShuffleArgument: compute a cryptog
of the shuffle

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$
- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$

The witness composed of
- permutation $\pi \in \Sigma_N$
- randomness $\rho \in \mathbb{Z}_q$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$
The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$

Ensure: $\forall i \in [0, N) : \vec{C}'_i = \mathsf{GetCiphertextProduct}(\mathsf{GetCiphertext}(\vec{1}, \rho_i, \mathbf{pk})$
Ensure: $N = mn$

```java
checkNotNull(statement);
checkNotNull(witness);

checkArgument(m > 0, "The number of rows for the ciphertext matrices must be strictly positive.");
checkArgument(n > 0, "The number of columns for the ciphertext matrices must be strictly positive.");

final GroupVector<ElGamalMultiRecipientCiphertext, GqGroup> ciphertextsC = statement.getCiphertexts();
final GroupVector<ElGamalMultiRecipientCiphertext, GqGroup> shuffledCiphertextsCPrime = statement.getShuffledCiphertexts();
final Permutation permutation = witness.getPermutation();
final GroupVector<ZqElement, ZqGroup> randomness = witness.getRandomness();

// Cross dimensions checking.
checkArgument(ciphertextsC.size() == permutation.getSize(),
                "The statement ciphertexts must have the same size as the permutation.");

// Cross group checking.
checkArgument(ciphertextsC.getGroup().hasSameOrderAs(randomness.getGroup()),
                "The randomness group must have the order of the ciphertexts group.");

// Ensure the statement corresponds to the witness.
final GqGroup gqGroup = ciphertextsC.getGroup();
final ZqGroup zqGroup = randomness.getGroup();
final int N = permutation.getSize();
final int l = ciphertextsC.get(0).size();

checkArgument(l <= publicKey.size(), "The ciphertexts must be smaller than the public key.");

final ElGamalMultiRecipientMessage ones = ElGamalMultiRecipientMessage.ones(gqGroup, l);
final List<ElGamalMultiRecipientCiphertext> encryptedOnes = randomness.stream()
                .map(rho -> getCiphertext(ones, rho, publicKey))
                .collect(toList());
final List<ElGamalMultiRecipientCiphertext> shuffledCiphertexts = permutation.stream()
                .mapToObj(ciphertextsC::get)
                .collect(toList());
```

Algorithm implementation
Example

$\mathbf{r} \leftarrow \mathsf{GenRandomVector}(q, m)$
$A \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{\pi(i)\}_{i=0}^{N-1}, m, n))$ ▷ Create a $n \times m$ π
$\mathbf{c}_A \leftarrow \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$
$x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A))$
$\mathbf{s} \leftarrow \mathsf{GenRandomVector}(q, m)$

```java
// Algorithm operations.

final BigInteger p = gqGroup.getP();
final BigInteger q = gqGroup.getQ();

// Compute vector r, matrix A and vector c_A
final GroupVector<ZqElement, ZqGroup> r = randomService.genRandomVector(q, m);
final GroupVector<ZqElement, ZqGroup> permutationVector = permutation.stream()
                .mapToObj(BigInteger::valueOf)
                .map(value -> ZqElement.create(value, zqGroup))
                .collect(toGroupVector());
final GroupMatrix<ZqElement, ZqGroup> matrixA = permutationVector.toMatrix(m, n).transpose();
final GroupVector<GqElement, GqGroup> cA = getCommitmentMatrix(matrixA, r, commitmentKey);

// Compute x.
final byte[] xHash = hashService.recursiveHash(
                HashableBigInteger.from(p),
                HashableBigInteger.from(q),
                publicKey,
                commitmentKey,
                ciphertextsC,
                shuffledCiphertextsCPrime,
                cA
);
```

Algorithm implementation

Example

```
r ← GenRandomVector(q, m)
A ← Transpose(ToMatrix({π(i)}_{i=0}^{N−1}, m, n))  ▷ Create a n × m m
c_A ← GetCommitmentMatrix(A, r, ck)
x ← ByteArrayToInteger(RecursiveHash(p, q, pk, ck, C⃗, C⃗′, c_A))
s ← GenRandomVector(q, m)
```

```java
// Algorithm operations.

final BigInteger p = gqGroup.getP();
final BigInteger q = gqGroup.getQ();

// Compute vector r, matrix A and vector c_A
final GroupVector<ZqElement, ZqGroup> r = randomService.genRandomVector(q, m);
final GroupVector<ZqElement, ZqGroup> permutationVector = permutation.stream()
                .mapToObj(BigInteger::valueOf)
                .map(value -> ZqElement.create(value, zqGroup))
                .collect(toGroupVector());
final GroupMatrix<ZqElement, ZqGroup> matrixA = permutationVector.toMatrix(m, n).transpose();
final GroupVector<GqElement, GqGroup> cA = getCommitmentMatrix(matrixA, r, commitmentKey);

// Compute x.
final byte[] xHash = hashService.recursiveHash(
                HashableBigInteger.from(p),
                HashableBigInteger.from(q),
                publicKey,
                commitmentKey,
                ciphertextsC,
                shuffledCiphertextsCPrime,
                cA
);
```

Algorithm implementation

Example

$\mathbf{r} \leftarrow \mathsf{GenRandomVector}(q, m)$

$A \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{\pi(i)\}_{i=0}^{N-1}, m, n))$ ▷ Create a $n \times m$ m

$\mathbf{c}_A \leftarrow \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$

$x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A))$

$\mathbf{s} \leftarrow \mathsf{GenRandomVector}(q, m)$

```java
// Algorithm operations.

final BigInteger p = gqGroup.getP();
final BigInteger q = gqGroup.getQ();

// Compute vector r, matrix A and vector c_A
final GroupVector<ZqElement, ZqGroup> r = randomService.genRandomVector(q, m);
final GroupVector<ZqElement, ZqGroup> permutationVector = permutation.stream()
                .mapToObj(BigInteger::valueOf)
                .map(value -> ZqElement.create(value, zqGroup))
                .collect(toGroupVector());
final GroupMatrix<ZqElement, ZqGroup> matrixA = permutationVector.toMatrix(m, n).transpose();
final GroupVector<GqElement, GqGroup> cA = getCommitmentMatrix(matrixA, r, commitmentKey);

// Compute x.
final byte[] xHash = hashService.recursiveHash(
                HashableBigInteger.from(p),
                HashableBigInteger.from(q),
                publicKey,
                commitmentKey,
                ciphertextsC,
                shuffledCiphertextsCPrime,
                cA
);
```

Algorithm implementation
Example

$\mathbf{r} \leftarrow \mathsf{GenRandomVector}(q, m)$
$A \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{\pi(i)\}_{i=0}^{N-1}, m, n))$ ▷ Create a $n \times m$ m
$\mathbf{c}_A \leftarrow \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$
$x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A))$
$\mathbf{s} \leftarrow \mathsf{GenRandomVector}(q, m)$

```java
// Algorithm operations.

final BigInteger p = gqGroup.getP();
final BigInteger q = gqGroup.getQ();

// Compute vector r, matrix A and vector c_A
final GroupVector<ZqElement, ZqGroup> r = randomService.genRandomVector(q, m);
final GroupVector<ZqElement, ZqGroup> permutationVector = permutation.stream()
                .mapToObj(BigInteger::valueOf)
                .map(value -> ZqElement.create(value, zqGroup))
                .collect(toGroupVector());
final GroupMatrix<ZqElement, ZqGroup> matrixA = permutationVector.toMatrix(m, n).transpose();
final GroupVector<GqElement, GqGroup> cA = getCommitmentMatrix(matrixA, r, commitmentKey);

// Compute x.
final byte[] xHash = hashService.recursiveHash(
                HashableBigInteger.from(p),
                HashableBigInteger.from(q),
                publicKey,
                commitmentKey,
                ciphertextsC,
                shuffledCiphertextsCPrime,
                cA
);
```

PUBLIC SCRUTINY IN PRACTICE

Olivier Esseiva

Gitlab issues

Public scrutiny

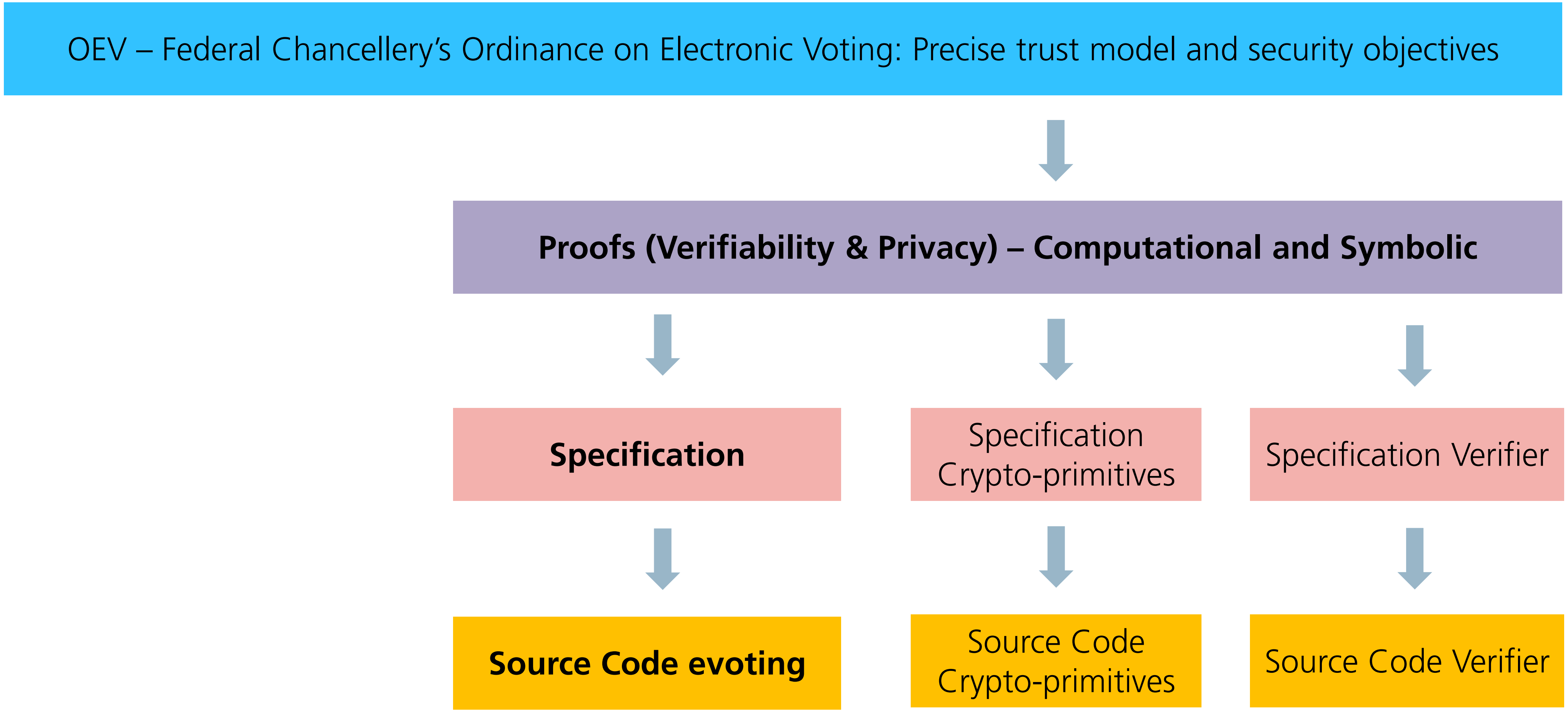
- Swiss Post is publishing all reports received on GitLab.
- It is in contact with the experts who have submitted reports.

Reports received

18 reports have been received since the start of disclosure.

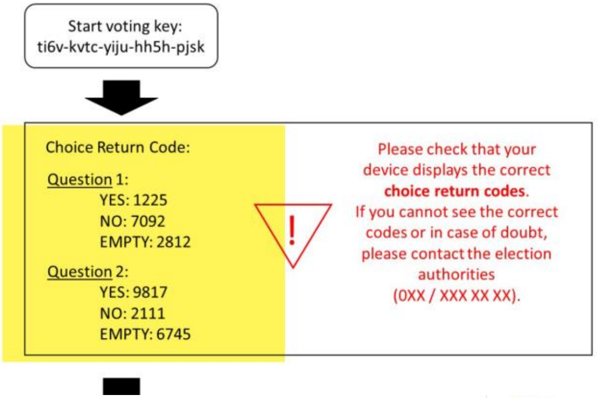
  - 1 high-priority finding
  - 1 medium-priority finding
  - 13 improvements
  - 2 comments
  - 1 question

Hierarchy of artefacts



Gitlab issues

#2 Issue by Thomas Haines



6. Generate the Return Codes Mapping table CMtable:

- Compute hashes $\mathtt{hlCC}_{\mathtt{id},i} \leftarrow \mathsf{H}(\mathtt{lCC}_{\mathtt{id},i})$ and $\mathtt{hlVCC}_{\mathtt{id}} \leftarrow \mathsf{H}(\mathtt{lVCC}_{\mathtt{id}})$. Then derive the symmetric encryption keys $\mathtt{skcc}_{\mathtt{id},i}$, $\mathtt{skvcc}_{\mathtt{id}}$ using the key derivation function KDF (see section 6).

$$\mathtt{skcc}_{\mathtt{id},i} \leftarrow \mathsf{KDF}(\mathtt{hlCC}_{\mathtt{id},i}, 16) \quad \forall i \in (1,\ldots,n),\ \forall \mathtt{id} \in \mathcal{ID}$$

- Symmetrically encrypt the short Choice Return Codes $\mathbf{cc}_{\mathtt{id}}$ with the Choice Return Code encryption symmetric keys $\mathbf{skcc}_{\mathtt{id}}$ and the short Vote Cast Return Code $\mathtt{VCC}_{\mathtt{id}}$ with the Vote Cast Return Code encryption symmetric key $\mathtt{skvcc}_{\mathtt{id}}$.

$$\mathtt{ctCC}_{\mathtt{id},i} \leftarrow \mathsf{Enc}_s(\mathtt{CC}_{\mathtt{id},i}; \mathtt{skcc}_{\mathtt{id},i}) \quad \forall i \in (1,\ldots,n),\ \forall \mathtt{id} \in \mathcal{ID}$$

$$\mathtt{ctVCC}_{\mathtt{id}} \leftarrow \mathsf{Enc}_s(\mathtt{VCC}_{\mathtt{id}}; \mathtt{skvcc}_{\mathtt{id}})$$

- Map the long Return Codes (long Choice Return Codes **lCC** and long Vote Cast Return Code $\mathtt{lVCC}_{\mathtt{id}}$) to the encrypted short Return Codes (short Choice Return Codes $\mathbf{cc}_{\mathtt{id}}$ and short Vote Cast Return Code $\mathtt{VCC}_{\mathtt{id}}$)

$$\mathsf{CMtable}_{\mathtt{id}} \leftarrow \left\{ \{[\mathsf{H}(\mathtt{lCC}_{\mathtt{id},i}), \mathtt{ctCC}_{\mathtt{id},i}]\}_{i=1}^{n}, [\mathsf{H}(\mathtt{lVCC}_{\mathtt{id}}), \mathtt{ctVCC}_{\mathtt{id}}] \right\}$$

- Set the Return Codes Mapping table $\mathsf{CMtable} \leftarrow \{\mathsf{CMtable}_{\mathtt{id}}\}_{\mathtt{id} \in \mathcal{ID}}$.
- Shuffle the table's entries to avoid trivial correlation.

SwissPost Dev @Swisspost-DEV · 4 weeks ago

Reporter

Version 0.9.9 of the computational proof fixes the issue as described above:

Protocol of the Swiss Post Voting System — Version 0.9.9
Computational Proof of Complete Verifiability and Privacy — © 2021 Swiss Post Ltd.

- Derive the symmetric encryption keys $\mathtt{skcc}_{\mathtt{id},i}$, $\mathtt{skvcc}_{\mathtt{id}}$ using the key derivation function KDF (see section 6).

$$\mathtt{skcc}_{\mathtt{id},i} \leftarrow \mathsf{KDF}(\mathtt{lCC}_{\mathtt{id},i}, 16) \quad \forall i \in (1,\ldots,n),\ \forall \mathtt{id} \in \mathcal{ID}$$

$$\mathtt{skvcc}_{\mathtt{id}} \leftarrow \mathsf{KDF}(\mathtt{lVCC}_{\mathtt{id}}, 16) \quad \forall \mathtt{id} \in \mathcal{ID}$$

- Symmetrically encrypt the short Choice Return Codes $\mathbf{cc}_{\mathtt{id}}$ with the Choice Return Code encryption symmetric keys $\mathbf{skcc}_{\mathtt{id}}$ and the short Vote Cast Return Code $\mathtt{VCC}_{\mathtt{id}}$ with the Vote Cast Return Code encryption symmetric key $\mathtt{skvcc}_{\mathtt{id}}$.

$$\mathtt{ctCC}_{\mathtt{id},i} \leftarrow \mathsf{Enc}_s(\mathtt{CC}_{\mathtt{id},i}; \mathtt{skcc}_{\mathtt{id},i}) \quad \forall i \in (1,\ldots,n),\ \forall \mathtt{id} \in \mathcal{ID}$$

$$\mathtt{ctVCC}_{\mathtt{id}} \leftarrow \mathsf{Enc}_s(\mathtt{VCC}_{\mathtt{id}}; \mathtt{skvcc}_{\mathtt{id}})$$

- Map the long Return Codes (long Choice Return Codes **lCC** and long Vote Cast Return Code $\mathtt{lVCC}_{\mathtt{id}}$) to the encrypted short Return Codes (short Choice Return Codes $\mathbf{cc}_{\mathtt{id}}$ and short Vote Cast Return Code $\mathtt{VCC}_{\mathtt{id}}$)

$$\mathsf{CMtable}_{\mathtt{id}} \leftarrow \left\{ \{[\mathsf{H}(\mathtt{lCC}_{\mathtt{id},i}), \mathtt{ctCC}_{\mathtt{id},i}]\}_{i=1}^{n}, [\mathsf{H}(\mathtt{lVCC}_{\mathtt{id}}), \mathtt{ctVCC}_{\mathtt{id}}] \right\}$$

Q&A SESSION

FEEDBACK

Summary

We are developing our e-voting system at the IT center in Neuchâtel, adopting an iterative approach and working with external specialists.

We started the gradual disclosure of a beta version of the system in January 2021, with the aim of identifying vulnerabilities and to continually improving the system.

Parts of the system will be open source.

We have opted for permanent disclosure with a bug bounty programme and annual intrusion tests.

The development of the system is based on Security by Design in cryptography and software engineering.

As part of the community programme, we engage in direct dialogue with the experts who submit reports.

THANK YOU!

www.swisspost.ch/e-voting-community

DIE POST