Swiss Post E-Voting

- Presentation by Ella Kummer

- Supervisor : Louis-Henri Merino
- Professor Bryan Ford
- DEDIS Laboratory

Plan of the presentation

1. Quick overview of the Swiss post e-voting system
2. Goal of the project
3. Summary of the reviews
3. Interesting reviews
4. Conclusion

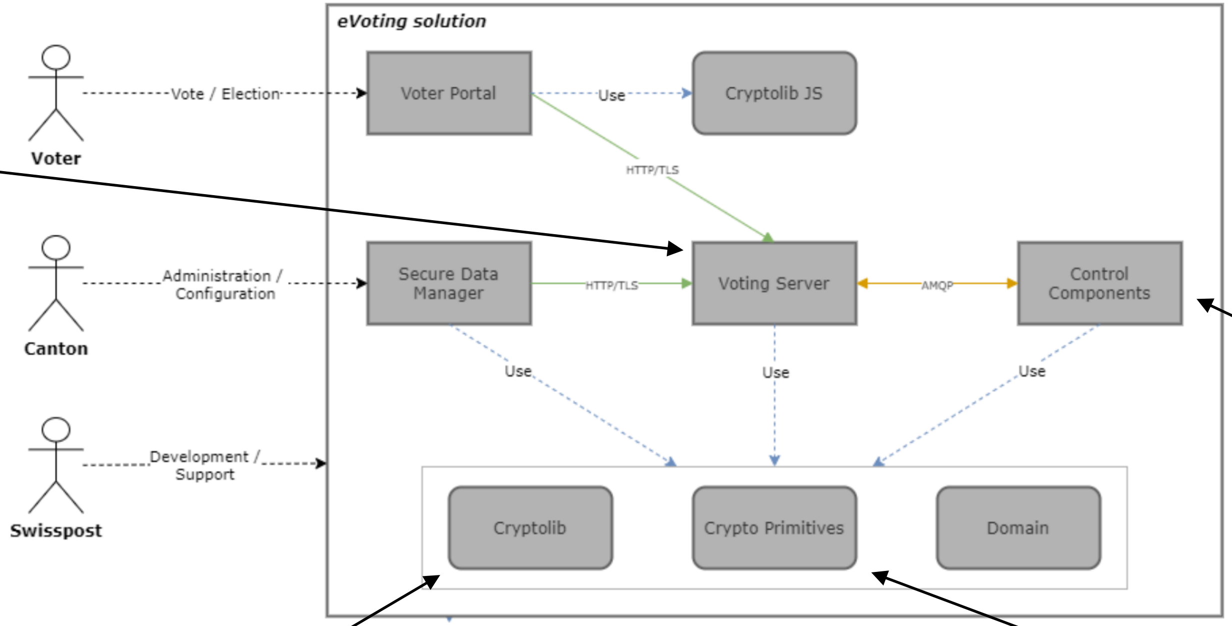
SECURITY OBJECTIVES

- **Individual verifiability** : Allows the voter to check that the vote was correctly transmitted and registered by the server by comparing a verification code that they receive with their voting documents to the verification code displayed online when they go to the ballot box.
- **Universal verifiability** : Allow voters or auditors to check that the election outcome corresponds to the registered votes using advanced cryptographic techniques such as non-interactive zero-knowledge proofs and verifiable mix-nets
- **Vote secrecy** : Do not reveal a voter’s vote to anyone. It preserves the privacy of the voter by encrypting votes end-to-end and splitting the decryption key among multiple entities.

DECOMPOSITION OF THE SYSTEM :

This application contains several microservices. Each microservice is responsible for one part of the voting process, i.e., authentication, election information, vote verification, etc.

It is considered untrustworthy



The Control Components compose a system in which they work together as a group.

There are two types of control components: the Return Codes Control Component (CCR) and the Mixing Control Component (CCM).

They generate the return codes, shuffle the encrypted votes, and decrypt them at the end of the election while guaranteeing the integrity of the voting protocol.

At least one of them must be trustworthy while three of them might be under an adversaries' control.

A library that provides key sharing and encryption capabilities to the voting protocol. It prevents incorrect, unsafe or insecure usage of cryptography algorithms and providers. It has a single-entry point that is configurable

An open-source server side library which implements cryptographic algorithms used as building blocks for the voting protocol. Focuses on the verifiable mix net and non-interactive zero-knowledge proofs.

PHASES

The cryptographic protocol divides the Swiss Post Voting System's "runtime" into **three parts**:

1. Configuration Phase : Generates the voter's codes that are subsequently sent to the voter by postal mail and generates the election public key that is used for encrypting the votes.

2. Voting Phase : First authenticates the voter. Then the voter can select the desired voting options and ensure individual verifiability, thus the vote can get confirmed.

3. Tally Phase : The voting server and the mixing control components decrypt the votes and compute the election result while protecting vote secrecy and guaranteeing universal verifiability.

GOAL OF THE PROJECT

- Review the source code and the documentation to look for potential vulnerabilities and security issues.

- Methodology :
  - Target a part of the system *(e.g. building block)* or part of the protocol *(e.g. phase)* for specific reasons *(e.g. complexity, newness)*
  - Analyse the documentation
    - Correct results
    - Correct security (also related to scheme used)
  - Analyse the implementation

SUMMARY OF THE REVIEWS

REVIEWED CODE

CRYPTOGRAPHIC IMPLEMENTATION :

- Multi-recipient ElGamal scheme
- Digital signatures (RSA-PSS)
- Randomness generation
- Hashing (SHA-256)

REVIEWED CODE
CONFIGURATION PHASE IMPLEMENTATION

GenKeysCCRj :

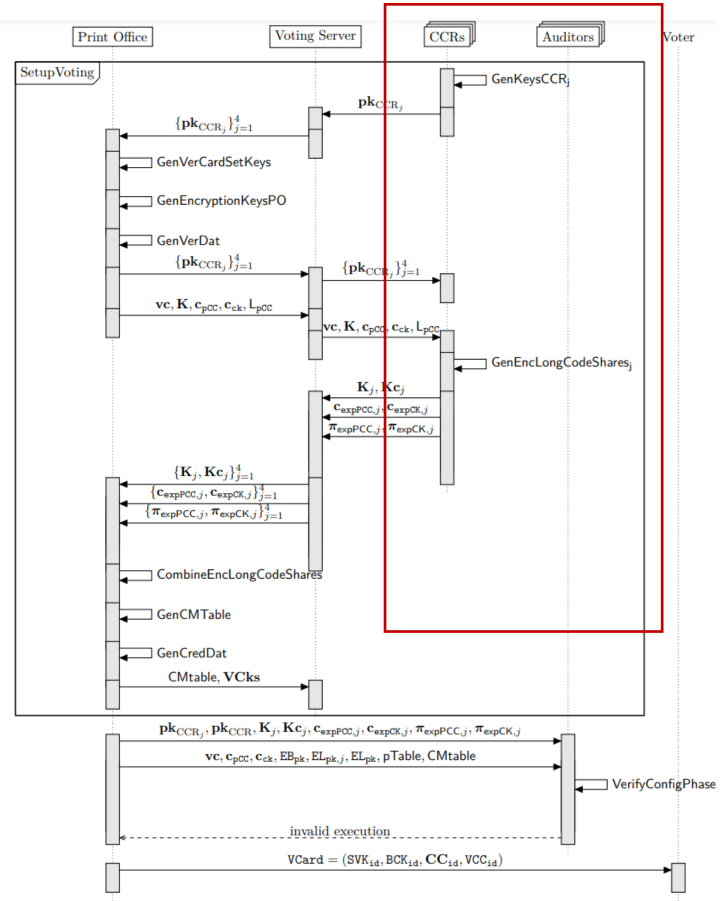
computes key pairs to later encrypt or derive new keys.

GenEncLongCodeShares :

creates shares of the long return codes.

SetupTallyCCMj :

computes the election key pair for each control component.



REVIEWED CODE
VOTING PHASE – CONFIRM VOTE IMPLEMENTATION

CreateConfirmMessage :

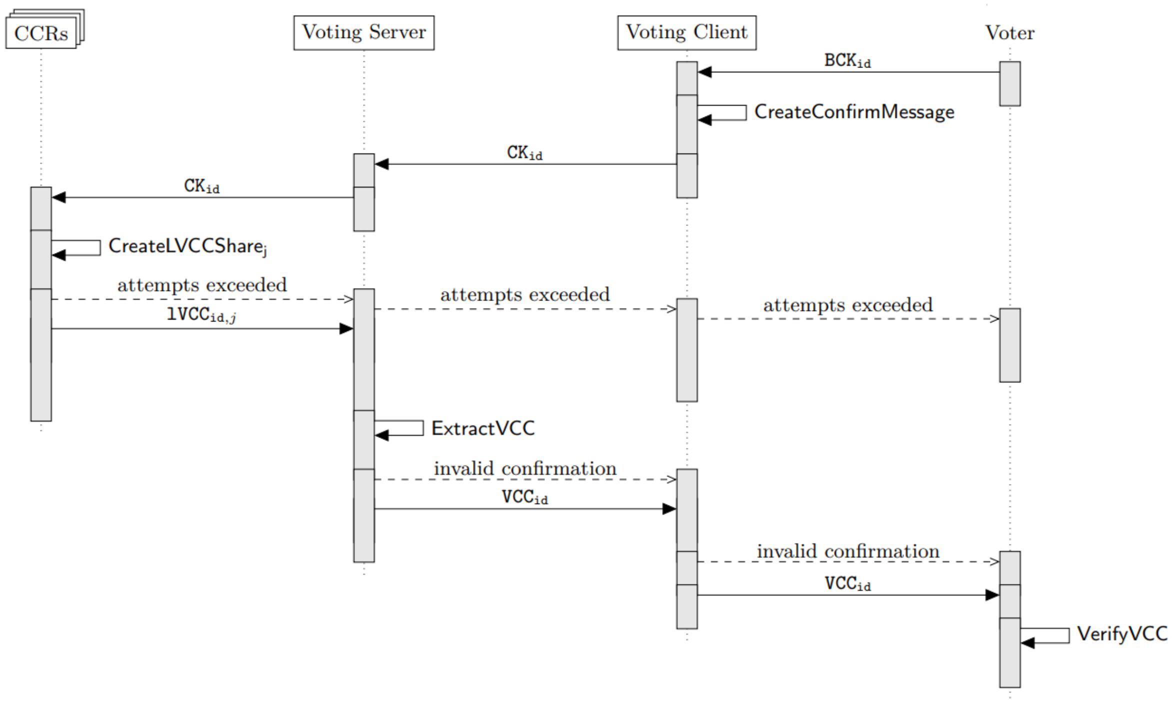
create a confirmation key.

CreateLVCCSharej :

derives codes for the next step.

ExtractVCC :

extracts a code to allow the voter to check that it is identical to the code printed on their voting card.



REVIEWED CODE
TALLY PHASE IMPLEMENTATION

GenVerifiableShuffle :

shuffles and re-encrypts the votes and proved a proof of the shuffle.

GenVerifiableDecryptions :

provides a verifiable partial decryption of a list of encrypted votes.

VerifyShuffle :

verifies the correctness of the shuffle argument.

VerifyDecryptions :

verifies the correctness of the decryption.

FINDS

- ✓ Necessary checks on inputs/data
- ✓ Groups operations
- ✓ Collision resistance for hash functions
- ✓ No insecure libraries
- ✓ Signed data
- ✓ Encryptions

- o Java best practises
- o Inconsistency between the documentation and the theory

FINDS

- Secure implementation of the cryptographic schemes

    - ✓ El Gamal encryption
    - ✓ Schnorr protocol for zero-knowledge proof (using Fiat-Shamir trick)
    - ✓ Pedersen commitment scheme
    - ✓ Bayer-Groth mixnet

FINDS

- Rely on java for
  - Randomness generation
  - Primality tests
  - Digital signature
  - Partly for hashing

~ *security*

INTERESTING DETAILED REVIEWS

RANDOMNESS GENERATION - UPDATE

SWISS POST

Class SecureRandom

java.lang.Object
    java.util.Random
        java.security.SecureRandom

: provides a cryptographically strong random number generator (RNG)

Inside Crypto Primitives

```java
public BigInteger genRandomInteger(final BigInteger upperBound) {
        checkNotNull(upperBound);
        checkArgument(upperBound.compareTo(BigInteger.ZERO) > 0, "The upper bound must a be a positive integer greater than 0.");
        final BigInteger m = upperBound;

        final int bitLength = m.bitLength();

        BigInteger r;
        do {
                // This constructor internally masks the excess generated bits.
                r = new BigInteger(bitLength, secureRandom);
        } while (r.compareTo(m) >= 0);

        return r;
}
```

```java
public BigInteger genRandomIntegerUpperBounded(BigInteger upperBound) {
        checkNotNull(upperBound);
        checkArgument(upperBound.compareTo(BigInteger.ZERO) > 0);

        int length = upperBound.bitLength();

        BigInteger random;
        do {
                random = genRandomIntegerByBits(length);
        } while (random.compareTo(upperBound) >= 0);

        return random;
}
```

Inside Cryptolib

RANDOMNESS GENERATION - UPDATE

- Some cryptographic primitives are implemented both in the crypto-primitives and the cryptolib (for instance the ElGamal encryption scheme). The implementations are functionally equivalent. We are continously replacing the cryptolib implementation with the more robust crypto-primitives one.

https://gitlab.com/swisspost-evoting/e-voting/e-voting

MixDecOnlinej function

The online Mixing control components CCM shuffle and re-encrypt the previous control component's ciphertexts and perform partial decryption.

If the control component is the first to mix (j = 1), the input list of ciphertexts corresponds to the cleansed encrypted votes.

Algorithm 6.2 $\mathsf{MixDecOnline}_j$

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election event ID $\mathsf{ee} \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}}}$
Ballot box ID $\mathsf{bb} \in (\mathbb{A}_{Base16})^{\mathtt{l_{ID}}}$
Control component index $j \in [1, 3]$
Number of allowed write-ins + 1 for this specific ballot box $\hat{\delta} \in \mathbb{N}^*$
List of shuffled and decrypted ballot boxes $\mathsf{L}_{\mathsf{bb},j}$

Input:

Partially decrypted votes $\mathbf{c}_{\mathsf{dec},j-1} \in (\mathbb{H}_l)^{\mathtt{N_C}}$
Remaining election public key $\overline{\mathbf{EL}}_{\mathbf{pk},j-1} \in \mathbb{G}_q^{\delta}$ ▷ $= \mathbf{EL}_{\mathbf{pk}}$, if $j = 1$
CCM_j election key pair $(\mathbf{EL}_{\mathbf{pk},j}, \mathbf{EL}_{\mathbf{sk},j}) \in \mathbb{G}_q^{\mu} \times \mathbb{Z}_q^{\mu}$

Ensure: $\mathtt{N_C} \geq 1$ ▷ The algorithm runs with at least one vote
Ensure: $l = \hat{\delta}$
Ensure: $0 < l \leq \delta \leq \mu$
Ensure: $\mathsf{bb} \notin \mathsf{L}_{\mathsf{bb},j}$

Operation:

```
i_aux ← (ee, bb, "MixDecOnline", IntegerToString(j))
if N_C > 1 then                                                   ▷ Shuffling requires at least 2 votes
    (c_mix,j, π_mix,j) ← GenVerifiableShuffle(c_dec,j−1, EL̄_pk,j−1)   ▷ See crypto primitives specification
    (c_dec,j, π_dec,j) ← GenVerifiableDecryptions(c_mix,j, (EL_pk,j, EL_sk,j), i_aux)   ▷ See crypto primitives specification
else                                                              ▷ If there is only 1 vote in the ballot box
    (c_dec,j, π_dec,j) ← GenVerifiableDecryptions(c_dec,j−1, (EL_pk,j, EL_sk,j), i_aux)
end if
EL'_pk,j ← CompressPublicKey(EL_pk,j, δ)                          ▷ See crypto primitives specification
EL̄_pk,j ← EL̄_pk,j−1 / EL'_pk,j mod p
L_bb,j ← L_bb,j ∪ bb
```

Output:

Shuffled votes $\mathbf{c}_{\mathsf{mix},j} \in (\mathbb{H}_l^{\mathtt{N_C}})$
Shuffle proof $\pi_{\mathsf{mix},j}$ ▷ See the domain of the Shuffle proof. Empty if $\mathtt{N_C} = 1$.
Partially decrypted votes $\mathbf{c}_{\mathsf{dec},j} \in (\mathbb{H}_l)^{\mathtt{N_C}}$
Decryption proofs $\boldsymbol{\pi}_{\mathsf{dec},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q{}^l)^{\mathtt{N_C}}$
Remaining election public key $\overline{\mathbf{EL}}_{\mathbf{pk},j} \in \mathbb{G}_q^{\delta}$

MixDecOnline function

SWISS POST

- The algorithm accepts the case **where only one vote is submitted to decryption**.
- This could be considered as a security issue as this case **does not preserve the voter anonymity**.
- As there is only one vote, **no shuffling** can be performed to break the link between the ciphertexts and the plaintexts and thus the voter.

Algorithm 6.2 $\mathsf{MixDecOnline}_j$

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Election event ID $\mathtt{ee} \in (\mathbb{A}_{Base16})^{l_{\mathtt{ID}}}$
Ballot box ID $\mathtt{bb} \in (\mathbb{A}_{Base16})^{l_{\mathtt{ID}}}$
Control component index $j \in [1, 3]$
Number of allowed write-ins + 1 for this specific ballot box $\hat{\delta} \in \mathbb{N}^*$
List of shuffled and decrypted ballot boxes $\mathsf{L}_{\mathsf{bb},j}$

Input:

Partially decrypted votes $\mathbf{c}_{\mathsf{dec},j-1} \in (\mathbb{H}_l)^{\mathtt{N_C}}$
Remaining election public key $\overline{\mathbf{EL}}_{\mathsf{pk},j-1} \in \mathbb{G}_q^{\delta}$ ▷ $= \mathbf{EL}_{\mathsf{pk}}$, if $j = 1$
CCM_j election key pair $(\mathbf{EL}_{\mathsf{pk},j}, \mathbf{EL}_{\mathsf{sk},j}) \in \mathbb{G}_q^{\mu} \times \mathbb{Z}_q^{\mu}$

Ensure: $\mathtt{N_C} \geq 1$ ▷ The algorithm runs with at least one vote
Ensure: $l = \hat{\delta}$
Ensure: $0 < l \leq \delta \leq \mu$
Ensure: $\mathtt{bb} \notin \mathsf{L}_{\mathsf{bb},j}$

Operation:

$\mathbf{i}_{\mathsf{aux}} \leftarrow (\mathsf{ee}, \mathsf{bb}, \text{"MixDecOnline"}, \mathsf{IntegerToString}(j))$
if $\mathtt{N_C} > 1$ **then** ▷ Shuffling requires at least 2 votes
$(\mathbf{c}_{\mathsf{mix},j}, \pi_{\mathsf{mix},j}) \leftarrow \mathsf{GenVerifiableShuffle}(\mathbf{c}_{\mathsf{dec},j-1}, \overline{\mathbf{EL}}_{\mathsf{pk},j-1})$ ▷ See crypto primitives specification
$(\mathbf{c}_{\mathsf{dec},j}, \boldsymbol{\pi}_{\mathsf{dec},j}) \leftarrow \mathsf{GenVerifiableDecryptions}(\mathbf{c}_{\mathsf{mix},j}, (\mathbf{EL}_{\mathsf{pk},j}, \mathbf{EL}_{\mathsf{sk},j}), \mathbf{i}_{\mathsf{aux}})$ ▷ See crypto primitives specification
else ▷ If there is only 1 vote in the ballot box
$(\mathbf{c}_{\mathsf{dec},j}, \boldsymbol{\pi}_{\mathsf{dec},j}) \leftarrow \mathsf{GenVerifiableDecryptions}(\mathbf{c}_{\mathsf{dec},j-1}, (\mathbf{EL}_{\mathsf{pk},j}, \mathbf{EL}_{\mathsf{sk},j}), \mathbf{i}_{\mathsf{aux}})$
end if
$\mathbf{EL}'_{\mathsf{pk},j} \leftarrow \mathsf{CompressPublicKey}(\mathbf{EL}_{\mathsf{pk},j}, \delta)$ ▷ See crypto primitives specification
$\overline{\mathbf{EL}}_{\mathsf{pk},j} \leftarrow \frac{\overline{\mathbf{EL}}_{\mathsf{pk},j-1}}{\mathbf{EL}'_{\mathsf{pk},j}} \mod p$
$\mathsf{L}_{\mathsf{bb},j} \leftarrow \mathsf{L}_{\mathsf{bb},j} \cup \mathsf{bb}$

Output:

Shuffled votes $\mathbf{c}_{\mathsf{mix},j} \in (\mathbb{H}_l^{\mathtt{N_C}})$
Shuffle proof $\pi_{\mathsf{mix},j}$ ▷ See the domain of the Shuffle proof. Empty if $\mathtt{N_C} = 1$.
Partially decrypted votes $\mathbf{c}_{\mathsf{dec},j} \in (\mathbb{H}_l)^{\mathtt{N_C}}$
Decryption proofs $\boldsymbol{\pi}_{\mathsf{dec},j} \in (\mathbb{Z}_q \times \mathbb{Z}_q{}^l)^{\mathtt{N_C}}$
Remaining election public key $\overline{\mathbf{EL}}_{\mathsf{pk},j} \in \mathbb{G}_q^{\delta}$

MixDecOnline function

- This is a special case not likely to happen as during a vote there is a very low chance of having only one voter.
- However, for the sake of security it should be acknowledge. Either it needs to be modified*, or they need to make clear that they allow this case which removes some anonymity.

* e.g. enforce to receive at least two ciphertexts to ensure 2-anonymity and have only 50% of chance of linking the voter to their vote.

MixDecOnline function

✓ Thankfully, as the verifier is trustworthy, it is not possible for an active attacker to try to reduce the list of ciphertexts in order to use this condition as an oracle to ask for the decryption of a specific ciphertext.

During the shuffling (and decryption) proof, the original vector of unshuffled ciphertexts is used as input, a difference between the lists' sizes would be directly detected by verifier.

DIGITAL SIGNATURE

"The Swiss Post Voting System uses the RSA-PSS signature scheme with 2048-bits key length "

Algorithm 8.16 SignCheckpoint: Generate a signature for the checkpoint log entry

---

Context:

the RSA signing key k_{sig}, defined by:
- the modulus $m \in \mathbb{P}$
- the public exponent $p \in \mathbb{N}^*$
- the private exponent $d \in \mathbb{N}^*$

Input:

The HMAC value of the previous entry $h_{-1} \in \mathcal{B}^{32}$ ▷ The empty byte array is used in the case of a first line log entry
The liberated session key $k_l \in \mathcal{B}^{32}$
The encrypted session key $k_e \in \mathcal{B}^*$
The maximal number of log lines between checkpoints $n \in \mathbb{N}$
The maximal time duration between checkpoints (in milliseconds) $d \in \mathbb{N}$
The timestamp of the log entry $t \in \mathbb{N}$
The log message $m \in (\mathbb{A}_{UCS} \setminus CR, LF)^*$
The HMAC of the checkpoint log entry $h \in \mathcal{B}^{32}$

Operation:

$n_b \leftarrow \mathsf{IntegerToByteArray}(n)$ ▷ See crypto primitives specification
$d_b \leftarrow \mathsf{IntegerToByteArray}(d)$
$t_b \leftarrow \mathsf{IntegerToByteArray}(t)$
$m_b \leftarrow \mathsf{StringToByteArray}(m)$ ▷ See crypto primitives specification
~~$p \leftarrow h_{-1}||k_l||k_e||n_b||d_b||t_b||m_b||h$~~
$s \leftarrow \mathsf{Sign}_{\mathsf{RSA-PSS}}(k_{\mathsf{sig}}, p)$ ▷ Uses standard RSA-PSS signature, with SHA-256 as a digest function.

6: $s \leftarrow \mathsf{Sign}_{\mathsf{RSA-PSS}}(k_{\mathsf{sig}}, p)$ ▷ Uses standard RSA-PSS signature, with SHA-256 as a digest function.

```
final Signature signature = Signature.getInstance("SHA256withRSA");
Signature.update(…);
Signature.sign();
```

Class Signature

java.lang.Object
java.security.SignatureSpi
java.security.Signature

“The Signature class is used to provide applications the functionality of a digital signature algorithm.

The signature algorithm can be, among others, the NIST standard DSA, using DSA and SHA-256.

These algorithms are described in the Signature section (*) of the Java Cryptography Architecture Standard Algorithm Name Documentation.”

…
https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#Signature
…

The signature algorithm with SHA-* and the RSA encryption algorithm as defined in the OSI Interoperability Workshop, using the padding conventions described in PKCS #1 (*Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*)

CONCLUSION

CONCLUSION

- The system is still under development and some future work is planned
- Through the work done in this project, we have a high level of confidence on the security of the Swiss post e-voting system

SWISS POST

Future Work

We plan the following work for future releases.

- Explicitly implement the DecodePlaintexts algorithm. Currently, the output of the system consists of the encoded voting options (prime numbers). Please note that the verifier checks the decoding of the prime numbers.
- Reduce the number of code smells and increase test coverage. We will prioritize code that implements important elements of the cryptographic protocol.
- The voting server is *untrusted*: we distributed many functionalities to the mutually independent control components in the current protocol. However, for historical reasons, the voting server still performs additional validations not strictly necessary from the protocol point of view. Moreover, the voting server uses the JavaEE framework, while the other parts of the solution use SpringBoot. To improve maintainability, we want to reduce the voting server's responsibility to the strict minimum and align it to SpringBoot.
- Ensure reproducible builds, see the section on reproducible builds.

Known Issues

The current version of the source code has the following known issues:

- The source code is not fully aligned to the specification version 0.9.7:
  - The GenVerDat and CreateLCCShare algorithms do not implement the partial Choice Return Codes allow list. This point refers to Gitlab issue #7
  - The control components do not check the list of ballot boxes that they already partially decrypted. This refers to Gitlab issue #11
  - The DecryptPCC_j algorithm is not implemented. Currently, the voting server decrypts the PCC and sends them to the control components.
  - The control components do not receive the CCR_j Choice Return Codes Encryption public keys during the configuration phase. They only received the combined Choice Return Codes Encryption public key during voting (signed by the administration board key)
- We plan for a typescript implementation of the crypto-primitives implementation (open-source) that follows the crypto-primitives specification and implements the voting client's zero-knowledge proofs. The typescript implementation is going to address the following points:
  - The EncodeVotingOptions method is currently implemented outside the *CreateVote* algorithm
  - The implementation's zero-knowledge proof currently do not include the complete statement of the specification and use a different hash function
- Write-ins are currently not supported
- Some cryptographic primitives are implemented both in the crypto-primitives and the cryptolib (for instance the ElGamal encryption scheme). The implementations are functionally equivalent. We are continously replacing the cryptolib implementation with the more robust crypto-primitives one.
- The voter portal (a component considered untrustworthy in our threat model) is built using AngularJS. Even though there are long-term support options, covering both security weaknesses and future browser compatibility support, ideally, the frontend would be migrated to / rewritten in Angular.
- In some cases, publicly writable directories are used as temporary storage. The security risk associated is mitigated as we run such services in a containerized environment, where we control all the running processes. We plan to remove the use of temporary storage completely.