

Security audit of the e-voting back-end

CLIENT **Swiss Federal Chancellery**
DATE **August 9, 2023**
VERSION **2.0**
GIT COMMIT **1e9af1b**
STATUS **Final**

CLASSIFICATION **Public**
AUTHOR **Philippe Oechslin**
DISTRIBUTION **Swiss Federal Chancellery**
MODIFICATIONS **Added appendix for version 1.3.2.1**



Contents

1	Introduction	1
1.1	Context	1
1.2	Execution of the work	1
1.3	Executive summary	2
2	Channel Security Signatures	3
2.1	Configuration Phase:	3
2.2	Voting Phase:	3
2.3	Tally Phase:	4
3	Creating error situations	5
3.1	Cast of a vote with an invalid pCC	5
3.2	Modifying a signed parameter	5
3.3	Modifying a signed parameter to create an inconsistency	6
4	Voting client signatures and Authenticated Data	7
4.1	Certificates and signatures	7
4.2	Authenticated data	8
5	Technical security tests	9
5.1	Analysis of Open Ports	9
5.2	Analysis of exposed REST endpoints	9
5.3	Scanning for vulnerabilities	10
6	Documentation of the Architecture	11
7	Conclusions	12
A	Additional tests of version 1.3.2.1	13
A.1	New authentication mechanisms and incorrect local time	13
A.2	Tally of invalid ballots	13
A.3	Modifying signed parameters	14



1 Introduction

1.1 Context

The goal of this audit was to examine the security of the server-side parts of the e-voting systems based on the publicly available end-to-end test system.

Swiss post publishes the end-to-end test system as a set of docker containers and configuration data¹. This makes it possible to run a complete election event on a local machine.

Obviously, a local end-to-end system does not have all the security controls that exist in the real production environment. But since it uses the same code, it allows testing security aspects that do not depend on the production environment.

1.2 Execution of the work

Version 0.15.2.1: The version of the end-to-end system that was used for this analysis was version 0.15.2.1, which was published mid of July 2022.

The tests were carried out in August and September on our premises, with the system installed on a Linux server, except for the three instances of the Secure Data Manager, which were run on a Windows machine.

We based our analysis on the following documentation (found on Swiss Post's public repository):

- Swiss Post Voting System, System Specification, v1.0.0
- Cryptographic Primitives of the Swiss Post Voting System, v1.0.0
- E-Voting Architecture Document, v.1.1.0

Version 1.3.2.1 Some aspects of the end-to-end system we tested in July 2023. given in appendix A.

We also adapted our analysis of the documentation to the new versions of the documentation.

- Swiss Post Voting System, System Specification, v1.3.1
- Cryptographic Primitives of the Swiss Post Voting System, v1.3.0
- E-Voting Architecture Document, v.1.3.0
- Infrastructure whitepaper of the Swiss Post evoting system, 2023-04-19

¹ <https://gitlab.com/swisspost-evoting/e-voting/evoting-e2e-dev>

1.3 Executive summary

The analysis of the results of the tests led us to the following conclusions:

The messages are effectively protected by signatures: We were able to verify that all messages that are required to be signed according to the specification are indeed signed in the implementation. In two instances we verified that the modification of a message was correctly detected.

Voters can verify that they use the correct software with the correct data: The voters can use a published hash to verify that they are using the correct software. This software verifies all parameters, including the public key used for encryption. The verification is done with a hash of all parameters that is stored in the authenticated data of the voter's keystore. Thus, when voter types in the Start Voting Key that is printed on their material, that key is used both to decrypt secret that will allow them to vote and to verify that the correct parameters will be used to vote.

Signature of parameters does not include keys: In version 0.15.2.1 there was an issue with fact that the encryption key was not part of the signed parameters. This had no impact, since the client software verifies the key, and the voters can verify that they have the correct client software (see above).

The signature of parameters has been completely removed in version 1.3.2.1.

No technical flaws detected: We discovered no vulnerabilities in the back-end systems that would allow an attacker to manipulate the systems.

Some inaccuracies found in the Architectural document: The document does not always describe the interconnections between the back-end systems exactly as they are. The detailed technical information about the interconnections can however be found in the Infrastructure whitepaper.

2 Channel Security Signatures

The computational proofs rely on messages' authenticity which is not given in the communication channels. To achieve this property, the cryptographic protocol adds signatures to most messages. These signatures are detailed in section 7 of the System Specification.

We set up the system in a way that we could eavesdrop on all communications and verified the presence of the signatures listed in the specification. The signatures are given in tables 16, 17 and 18 of the System Specification.

2.1 Configuration Phase:

Message Name	Signer	Recipient(s)	Message Content	Context Data
1 ControlComponentPublicKeys	Online CC_j	Setup Comp., Auditors	$(\mathbf{pk}_{CCR_j}, \mathbf{EL}_{pk,j}, \pi_{\mathbf{EL}_{pk,j}})$	("OnlineCC keys", j, ee)
2 SetupComponentVerification-Data	Setup Comp.	Online CC_j , Auditors	$(\{vc_{id}, K_{id}, c_{pcc,id}, c_{ck,id}\}_{id=0}^{N_c-1}, L_{pcc})$	("verification data", ee, vcs)
3 ControlComponentCodeShares	Online CC_j	Setup Comp., Auditors	$(\{vc_{id}, K_{j,id}, K_{c,j,id}, c_{expPCC,j,id}, c_{expCK,j,id}, \pi_{expPCC,j,id}, \pi_{expCK,j,id}\}_{id=0}^{N_c-1})$	("encrypted code shares", j, ee, vcs)
4 SetupComponentLVCCAllowList	Setup Comp.	Online CC_j	L_{lvcc}	("lvcc allow list", ee, vcs)
5 SetupComponentCMTable	Setup Comp.	Voting Server	$CMtable^1$	("cm table", ee, vcs)
6 SetupComponentVerification-CardKeyStores	Setup Comp.	Voting Server	$VCKs$	("vc keystore", ee, vcs)
7 SetupComponentPublicKeys	Setup Comp.	Online CC_j , Tally CC, Auditors, Voting Server	$(\{\mathbf{pk}_{CCR_j}\}_{j=1}^4, \mathbf{pk}_{CCR}, \{\mathbf{EL}_{pk,j}\}_{j=1}^4, \mathbf{EB}_{pk}, \mathbf{EL}_{pk})$	("public keys", "setup", ee)
8 SetupComponentTallyData	Setup Comp.	Auditors, Tally CC	$(vc, K, pTable)$	("tally data", ee, vcs)
9 SetupComponentElectoralBoard-Hashes	Setup Comp.	Tally CC	$(hPW_0, \dots, hPW_{k-1})$	("electoral board hashes", ee)

Figure 2.1 Table 15 of the document: messages of the configuration phase



We numbered the messages from 1 to 9. We found the message 1-7 with their respective signature. Message 8 is sent to the verifier, which we did not test. Message 9 was absent due to the fact that the version of software in the end-to-end test system did not yet implement electoral board hashes. We verified that the CMtable is correctly ordered alphabetically according to the base64 value of its first column.

2.2 Voting Phase:



We observed all messages with their respective signatures.

Message Name	Signer	Recipient(s)	Message Content	Context Data
VotingServerEncryptedVote	Voting server	Online CC_j	$(E1, E2, \tilde{E1}, \pi_{Exp}, \pi_{EqEnc})$	("encrypted vote", ee, vcs, vc _{id})
ControlComponentPartialDecrypt	Online CC_j	Online $CC_{j'}$	$(d_j, \pi_{decPCC,j})$	("partial decrypt", j, ee, vcs, vc _{id})
ControlComponentLCCShare	Online CC_j	Voting Server	$(lCC_{j,id}, \pi_{explCC,j,id})$	("lcc share", j, ee, vcs, vc _{id})
VotingServerConfirm	Voting server	Online CC_j	CK_{id}	("confirmation key", ee, vcs, vc _{id})
ControlComponenthLVCC	Online CC_j	Online $CC_{j'}$	$hlVCC_{id,j}$	("hlvcc", j, ee, vcs, vc _{id})
ControlComponentlVCCShare	Online CC_j	Voting Server	$(lVCC_{id,j}, \pi_{explVCC,j,id})$	("lvcc share", j, ee, vcs, vc _{id})

Figure 2.2 Table 16 of the document: messages of the voting phase

2.3 Tally Phase:

Message Name	Signer	Recipient(s)	Message Content	Context Data
ControlComponentBallotBox	Online CC_j	Tally CC, Auditors	$(\{vc_{j,i}, E1_{j,i}, \tilde{E1}_{j,i}, E2_{j,i}, \pi_{Exp,j,i}, \pi_{EqEnc,j,i}\}_{i=0}^{M_E-1})$	("ballotbox", j, ee, bb)
ControlComponentShuffle	Online CC_j	Online $CC_{j'}$, Tally CC, Auditors	$(c_{mix,j}, \pi_{mix,j}, c_{dec,j}, \pi_{dec,j})$	("shuffle", j, ee, bb)
TallyComponentShuffle	Tally CC	Auditors	$(c_{mix,5}, \pi_{mix,5}, m, \pi_{dec,5})$	("shuffle", "offline", ee, bb)
TallyComponentVotes	Tally CC	Auditors	L_{votes}	("decoded votes", ee, bb)

Figure 2.3 Table 17 of the document: messages of the tally phase



We observed the signatures of the two first messages. Again, the last two were not seen as they are sent to the verifier, which we did not test.

3 Creating error situations

We created three error scenarios to verify the correct reaction of the system.

3.1 Cast of a vote with an invalid pCC

One of the new features added to the protocol is an allow-list that is used by the control components to let them verify whether an input they receive for the calculation of a return code, is among all possible valid inputs. This makes sure that the control components will not participate on calculations with manipulated parameters.

We used our own voting client to submit a vote with an invalid partial choice code. As a result, we got no answer from the voting server. We also got no answer from the server when we subsequently tried to submit a correct vote with the same voting card.



The behaviour of the system is correct in the sense that a vote with invalid pCCs will be detected when the control components run the CreateLCCShare algorithm, during the calculation of the return code. Subsequent submissions of a vote are blocked because the voting card has been recorded in the list L_{decPCC} of the card for which a pCC has already been decrypted.

We checked the logs of the e2e system to whether the absence of an answer was a correctly intercepted error or due to a crash of the program.



The error is correctly caught, as shown in the logs:

```
$ docker logs control-component-1 | less -R
Caused by: java.lang.IllegalStateException: Failed to obtain response payload
    at ch.post.it.evoting.controlcomponents.ExactlyOnceCommandExecutor.process(ExactlyOnceCommandExecutor.java:84)
    ~[classes!/:0.15.2.1]
    [...]
    ... 13 more
Caused by: java.lang.IllegalStateException: The partial Choice Return Codes allow list does not contain the partial Choice Return Code.
    at ch.post.it.evoting.controlcomponents.voting.sendvote.CreateLCCShareAlgoritih.createLCCShare
    (CreateLCCShareAlgoritih.java:152) ~[classes!/:0.15.2.1]
    [...]
```

3.2 Modifying a signed parameter

When calculating the finalisation code, the control components exchange their partial results and check that the combination of the results is in an allow-list. Only then do they reveal to the voting server the information needed to calculate the finalisation code.

In this experiment, we intercepted the traffic between the voting server and a control component. We modified the message containing the confirmation key CK_{id} , which is used as input for the CreateLVCC-Share algorithm. Note that this message is signed by the voting server.



The modification is detected and no finalisation code is generated. There is a timeout. The control component that received the manipulated value has the following log entry:

```
InvalidPayloadSignatureException: Signature of payload VotingServerConfirmPayload is invalid.  
The signature of Long Vote Cast Return Codes Share hash response payload is invalid
```

After tallying, we can confirm that the vote is not part of the votes that have been tallied.

3.3 Modifying a signed parameter to create an inconsistency

In this experiment we targeted the input to the next algorithm, `VerifyLVCCHash`. We modified one of the hash shares that are exchanged between the control components ($hLVCC_{id}$) for validation. Note that this message is signed by the control component that sends it.

The control component that receives the modified hash, will not register the vote as confirmed in the ballot box, whereas the three other components will register it as confirmed. This leads to the following situation:



The modification is detected and no finalisation code is generated. The control component that received the manipulated hash has the following log entry:

```
Signature of payload SetupComponentLVCCAllowListPayload is invalid
```

During the tally phase, the second control component generates the following log entry:

```
The initial ciphertexts vector and verifiable shuffles ciphertexts vector must have the same size.
```

Tallying stops.

Note that is manipulation could be done once for each control component, but on different votes. They would thus all have the same number of votes, although they would be missing a different one. This situation would not be detected by the consistency check that compares the number of initial votes with the number of shuffled votes.

It would however be detected when the control components verify the mixing proofs of the first mixing step (`VerifyMixDecOnline` algorithm in the system specification). Since the proofs of the first mixing step are checked against the locally stored initial list of votes ($c_{init,j}$), the check would fail if the control component running the check does not have the same list of votes as the first control component.

4 Voting client signatures and Authenticated Data

4.1 Certificates and signatures

When the voter accesses the home page of the voting server, a few javascript files included in the page. One of them contains the code of the voting client, that will execute the cryptographic protocol for the voter. Another file is a root certificate that is used to sign some elements that are downloaded later.

For both elements the server includes an integrity tag which lets the browser automatically check the integrity of the javascript files. Moreover, these tags are also published on a cantonal website, to allow voter to check for themselves if the downloaded files are the intended files.

Figure 4.1 shows the relation between the elements that are downloaded in the homepage and the two successive requests to the server. The elements pointed by an arrow are signed by the element from which the arrow emerges.

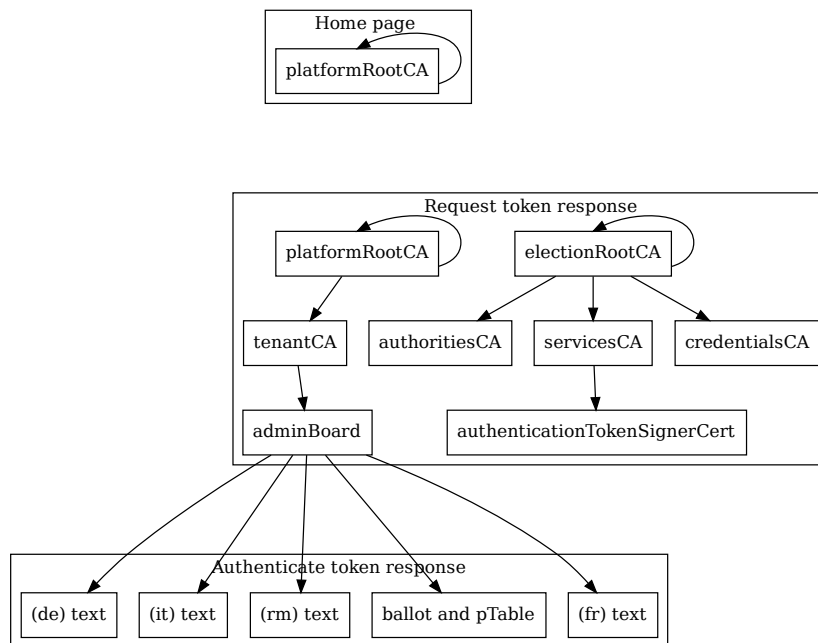


Figure 4.1 Certificates downloaded in the first three requests, and their relation



The goal of providing a root CA with an integrity tag is to give the voter the tools needed to convince themselves that the texts of the questions and their mapping to numbers are the intended one (as listed in the pTable). As we see in figure 4.1 the platform Root CA provided in the home page does not actually sign these elements, but rather an intermediate certificate (tenantCA) that signs yet another intermediate (adminBoard) that finally signs the elements. Thus, even if the voter has certainty to have the correct platform root CA, they still must trust in whoever signed the intermediate certificates.

If the certificate provided with an integrity tag in the home page had been the adminBoard certificate, then the voters would need to trust nothing else to verify the texts and the pTable.

Moreover, basic cryptographic elements like the definition of the group (p and q) and the public key used to encrypt the votes are not signed.

Fortunately, another way has been introduced to convince the voters that all the parameters they received are authentic, as we will see in the next section.

The imperfections of the signing scheme have thus no impact.

In version 1.3.2.1 of the software, there are no more certificates or signatures.

4.2 Authenticated data

An alternative way of verifying the parameters received is given by the way the voter's keystore is encrypted. The keystore contains the private key that the voter needs to carry out the cryptographic protocol. The keystore is encrypted with a key derived from the Start Voting Key (SVK) that is printed on the voting material. The voter can thus only vote if they type in the correct SVK.

Additionally to the private key, the keystore also carries a hash of all cryptographic parameters and of the definition of the ballot. The voting client verifies that the hash matches a hash calculated with the parameters that were actually received by the client. The list of parameters that are fed into the hash are given in the GetKey algorithm of the protocol specification as seen below:

Operation: ▷ For all algorithms see the crypto primitives specification

- 1: $\mathbf{h}_{\text{aux}} \leftarrow (\text{"Context"}, p, q, g, \mathbf{ee}, \mathbf{vcs}, \text{"ELpk"}, \text{EL}_{\text{pk},0}, \dots, \text{EL}_{\text{pk},\delta-1})$
- 2: $\mathbf{h}_{\text{aux}} \leftarrow (\mathbf{h}_{\text{aux}}, \text{"pkCCR"}, \text{pk}_{\text{CCR},0}, \dots, \text{pk}_{\text{CCR},\varphi-1}, \text{"EncodedVotingOptions"}, \tilde{p}_0, \dots, \tilde{p}_{n-1})$
- 3: $\mathbf{h}_{\text{aux}} \leftarrow (\mathbf{h}_{\text{aux}}, \text{"VotingOptions"}, v_0, \dots, v_{n-1})$
- 4: $\mathbf{h}_{\text{aux}} \leftarrow (\mathbf{h}_{\text{aux}}, \text{"ciSelections"}, \text{ciSelection}_0, \dots, \text{ciSelection}_{\psi-1})$
- 5: $\mathbf{h}_{\text{aux}} \leftarrow (\mathbf{h}_{\text{aux}}, \text{"ciVotingOptions"}, \text{ciVotingOption}_0, \dots, \text{ciVotingOption}_{n-1})$

Figure 4.2 List of parameters that are included in the hash which is verified when decrypting the private key of the voter (GetKey algorithm in the System Specification)

If the hashes do not match, the voting client refuses to vote. We verified this behaviour in the source code of the client and during our online test.

If the voter thus verifies the integrity of the voting client downloaded from the home page, they will have certainty that the voting client will only proceed if none of the parameters received by the server have been tampered. The more inclined voters might even be able to calculate the hash of the parameters and verify it themselves.



This authenticated hash has only been introduced in the latest versions of the protocol (1.0.0). In our opinion, this way of verifying that the parameters are authentic is superior to what is achieved by the signature scheme described in the previous section. This satisfies section 2.7.3 of the OEV that requires that voters must be able to verify that they received the correct code of the client and parameters.

5 Technical security tests

5.1 Analysis of Open Ports

All Java applications, voting server with its microservices as well as the control components, have an open port for debugging. The port gives access to the Java Debug Wire Protocol (JDWP). This protocol can be abused to execute arbitrary command on the machine.

The debugging port is activated from the command line when starting the application with the parameter:

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*<port>
```

This parameter is set in the common configuration file of all docker containers of the end-to-end test system, `docker-compose.common.yml`.

We assume that this parameter is not set in production environments.



Other than JDWP, which is valid in a test environment, we did not discover any unnecessary open ports.

5.2 Analysis of exposed REST endpoints

The different components of the back-end each offer a set of REST endpoints that are used to interact with them. For example, the extended authentication microservice on the voting server has an endpoint called

```
ea-ws-rest/extendedauthentication/tenant/{tenantId}/electionevent/{electionevent}/blocked
```

that can be used to get a list of cards that have been blocked because of too many failed authentication attempts.

The applications are written in Java. On startup all, endpoints and their corresponding Java method are listed in the logs of the application.



We analysed the lists of endpoints available for every service of the back-end to check if there was a service that would give access to protected information or functionality, maybe for debugging purposes. We did not identify any end-point that did not seem to be legitimate.

Note that the control components do not expose any service, they connect to a message broker, with which they can then exchange messages.

5.3 Scanning for vulnerabilities

The docker containers only contain the minimal set of programs needed for e-voting. We scanned the containers with a vulnerability scanner and also checked the version of some of the software manually.



All software seems to be up-to-date with no known vulnerabilities.

6 Documentation of the Architecture

Finally, we studied the E-Voting Architecture Document and checked if it mached the architecture found in the end-to-end testing system.

We found several differences with the document. Some of them may be due to the fact that the test system is not identical to the production environment.

The E-Voting Architecture Document makes abstraction of intermediate systems like reverse proxies and the load balancer. The information about these elements can be found in the Infrastructure whitepaper. This could be mentioned in the introduction of the architecture document.

7 Conclusions

We checked the back-end systems (Voting Server, Control Components and accessory services) for vulnerabilities or misconfigurations that would enable any attack on the cryptographic protocol.

Our tests did not reveal any weakness in the implementation of the back-end systems.

The fact that it is possible to download and run all elements necessary to cast a vote is a great way to see the code working and to interact with it. It allows carrying out tests without needing access to test systems of Swiss Post.

A Additional tests of version 1.3.2.1

In week 30 of 2023 we have reviewed some aspects of version 1.3.2.1 of the publicly available end-to-end test system. We verified the following items:

A.1 New authentication mechanisms and incorrect local time

The new authentication mechanism implemented since version 1.3 makes use of a time-based one time password (TOTP). The authentication is based on the system time and has a validity of one minute. This implies that voters with a local clock that is off by more than one minute would not be able to vote. To prevent this situation, the server adds its current system time in error messages related to authentication. This allows the client to display a specific error message if it finds out that the system time of the client is different than the one of the server.

Note that system time is always given in universal time, thus being in a different time-zone would still make it possible to authenticate.



We advanced the local clock of the client by 5 minutes. When sending the correct start voting key and birthday, the voting client displays an error message which states that the client's clock is wrong and recommends setting the clock to the correct time or using a different device.

A.2 Tally of invalid ballots

The cryptographic protocol prevents the submissions of almost all types of invalid votes. Without knowing what the content of a ballot is, the system makes sure that for each question there is the correct number of answers (e.g. yes, no or candidate names), that these answers belong to the list of possible answers and that no answer appears more than once.

Still, there is one specific type of invalid ballot that can be submitted. In a proportional election, a list with a party name must contain at least one candidate. This special case is not detected by the protocol and is accepted into the ballot box.

Note that the voting client does not allow to cast this type of invalid ballot. The voter has to wilfully manipulate the client – or use a different client – to submit such a ballot. The server correctly returns the verification codes for the party list and blank candidates.



We submitted an invalid ballot with our own voting client and ran the tally. The eCH-0110 output file correctly lists an unaccounted, invalid ballot:

```
<eCH-0110:countOfReceivedBallotsTotal>
  <eCH-0110:total>1</eCH-0110:total>
</eCH-0110:countOfReceivedBallotsTotal>
<eCH-0110:countOfAccountedBallots>
  <eCH-0110:total>0</eCH-0110:total>
</eCH-0110:countOfAccountedBallots>
<eCH-0110:countOfUnaccountedBallots>
  <eCH-0110:total>1</eCH-0110:total>
</eCH-0110:countOfUnaccountedBallots>
<eCH-0110:countOfUnaccountedBlankBallots>
  <eCH-0110:total>0</eCH-0110:total>
</eCH-0110:countOfUnaccountedBlankBallots>
<eCH-0110:countOfUnaccountedInvalidBallots>
  <eCH-0110:total>1</eCH-0110:total>
</eCH-0110:countOfUnaccountedInvalidBallots>
```

A.3 Modifying signed parameters

We intercepted the traffic between the control components and tried to modify the data in two occasions:

CreateLCC: This is the algorithm where the control components collaborate to create the long return code (LCC) by creating each a share of it. We intercepted and modified a CreateLCCShare request to one control component.



When we replaced one of the exponentiated Gammas by 0 we got a generic error, as 0 is not a valid element of the group Gq: /

```
java.lang.IllegalArgumentException:
  Cannot create a GroupElement with value 0 as it is not an element of
  group Group Gq
```

We then replaced the value by 3, which is an element of the group. The control component accepted the message but detected the manipulation when verifying the signature:

```
ch.post.it.evoting.domain.InvalidPayloadSignatureException:
  Signature of payload ControlComponentPartialDecryptPayload is invalid.
  [...]
  at ch.post.it.evoting.controlcomponent.voting.sendvote.LCCShareProcessor.verifySignature(LCCShareProcessor.java:191)
```

MixOnline: In the algorithm, the control components mix and partially decrypt the ballots. Each component generates a proof that the partial decryption was done correctly. The next control component must verify the proof of the previous component before proceeding with decryption.



We modified the value of the decryption proof and obtained the following error message:

```
ch.post.it.evoting.domain.InvalidPayloadSignatureException:
  Signature of payload ControlComponentShufflePayload is invalid
```

We removed the signature of the message, to check if the control component would then by-pass signature checking:

```
Cannot construct instance of `ch.post.it.evoting.cryptoprimitives.domain.signature.CryptoPrimitivesSignature`,
  problem: `java.lang.NullPointerException`
```