Helios

Search this site

Helios v1 and v2 Verification Specs

Introduction

Helios is an open-audit voting system, which means that:

- Alice can verify that her vote was correctly captured,
- all captured votes are displayed (in encrypted form) for all to see, and
- anyone can verify that the captured votes were correctly tallied.

This document specifies all data formats and the exact verification protocols and algorithms. Using this document, it should be possible for an able programmer to build a complete verification program in any modern programming language. For the sake of concreteness, instead of pseudo-code, we use Python (2.3 or above.)

Accessing Data

All data for an election is easily accessible using simple HTTP GET requests. Consider an election with election id `<ELECTION_ID>`. The election data structure, including the election public key, questions, etc., can be obtained by requesting the following URL:

```
    http://www.heliosvoting.org/elections/<ELECTI
```

The list of voters, denoted `<VOTER_LIST>`, is available at:

```
    http://www.heliosvoting.org/elections/<ELECTI
```

Given this list, it is possible to extract individual voter identifiers, denoted `<VOTER_ID>` (the data structure will be explained later in this document.) Once obtained, a complete voter data structure, including encrypted vote, can be downloaded at:

```
    http://www.heliosvoting.org/elections/<ELECTI
```

When downloading multiple ballots at the same time, it is possible to request a list of voters *with* their encrypted vote:

```
    http://www.heliosvoting.org/elections/<ELECTI
```

It is recommended that lists of voters with their votes be downloaded in batches of no more than 50, which can be done with an additional URL argument:

```
    http://www.heliosvoting.org/elections/<ELECTI
```

And the next batch can be requested using the `after` parameter:

```
    http://www.heliosvoting.org/elections/<ELECTI
```

The result of an election is available at:

```
    http://www.heliosvoting.org/elections/<ELECTI
```

While the proof of this result is available at:

```
    http://www.heliosvoting.org/elections/<ELECTI
```

For testing purposes, the following election identifier can be used:

```
    agxoZWxpb3N2b3RpbmdyDwsSCEVsZWN0aW9uGOAQDA
```

which means that its election information is at http://www.heliosvoting.org/elections/agxoZWxpb3N2b3RpbmdyDwsSCEVsZWN0aW9uGOAQDA.

All data for this election was generated using the Helios system, and a verification program, built to the guidelines that follow, should be able to check all of the results.

If one wants to check the audit trail of a ballot, a new ballot audit trail can be generated at any time using the voting booth interface for this election. For convenience, one such ballot audit trail is available here, with choices 2 (Drummond) and 3 (Axel). The ballot fingerprint is `6PkosgVAY011FbVvqHGBeRo1SNs`.

Data Formats

We begin with a description of the data types and their representations. All data made available by Helios is in JavaScript Object Notation (JSON) format, with keys in alphabetical order and no extraneous whitespace other than that specified by JSON. These two conditions are particularly important, as hashing data structures will only yield the same hash if the conditions are respected. An example of a JSON data structure with these two conditions is:

Example (not an actual Helios data structure)
{"email": ["ben@adida.net", "ben@mit.edu"], "first_name": "Ben", "last_name": "Adida"}

Basic Cryptographic Datatypes

All large integers are represented in decimal form as strings, rather than integers. The reason is that some languages do not support big integers natively, and thus cannot properly parse large integers in JSON integer form. An El-Gamal public-key is then a dictionary including the prime p, the primer-order q of its intended subgroup, the generator g, and the public-key value y (with keys in alphabetical order):

<ELGAMAL_PUBLIC_KEY>
{"g": "68111451286792593845145063691 65", "p": "1699897197819409959350 3959", "q": "849948598909704979675 1", "y": "746466870347987716425372037325 9704"}

An El-Gamal ciphertext is a JSON structure containing properties `alpha` and `beta`, the two components modulo p.

<ELGAMAL_CIPHERTEXT>
{"alpha": "72348234202340223423", "beta": "123498235234234234324" }

In Helios, all ciphertexts are *Exponential ElGamal*, so `alpha = g^r mod p`, and `beta = g^m y^r mod p`.

Voter

A single voter in Helios is represented using a few fields that identify the voter:

<VOTER>
{"email": "benadida@gmail.com", "name": "Ben Adida", "vote_hash": null, "voter_id": "agxoZWxpb3N2b3RpbmdyCwsSBVZvdGVyGCcM"}

The `voter_id` is unique across all elections hosted at this particular Helios server. The `vote_hash` is the SHA1 hash of the voter's encrypted ballot. Until the voter casts a ballot, `vote_hash` is null.

Voters may be identified by OpenID URL rather than email address, in which case their JSON representation is:

<VOTER> (2)
{"name": "Ben Adida", "openid": "http://benadida.myopenid.com", "vote_hash": null, "voter_id": "agxoZWxpb3N2b3RpbmdyCwsSBVZvdGVyGCcM"}

In elections where voters are placed in categories (e.g. precincts), an additional `category` field is present:

<VOTER> (3)
{"category": "02139", "email": "benadida@gmail.com", "name": "Ben Adida", "vote_hash": null, "voter_id": "agxoZWxpb3N2b3RpbmdyCwsSBVZvdGVyGCcM"}

Once a voter has cast a ballot, their JSON representation is augmented with their encrypted vote:

<VOTER_WITH_VOTE>
{"category": "02139", "name": "Ben Adida", "openid": "http://benadida.myopenid.com", "vote" : <VOTE>, "vote_hash":

"f1d2d2f924e986ac86fdf7b36c94bcdf32beec15", "voter_id": "agxoZWxpb3N2b3RpbmdyCwsSBVZvdGVyGCcM"}

We describe the details of the <VOTE> data structure later in this document, once we have described all of the required components.

It may be confusing to note that `vote_hash` and `vote` are both present within the data structure, when the hash is clearly derived from the vote. The reason for the presence of both is that, when voters are listed in a <VOTER_LIST>, only the `vote_hash` is listed, not the complete `vote`, for efficiency purposes.

Election

An election is represented as:

<ELECTION>
{"election_id": "agxoZWxpb3N2b3RpbmdyDgsSCEVsZWN0aW9uGAYM", "name": "foo234234", "pk": <ELGAMAL_PUBLIC_KEY>, "questions": <QUESTION_LIST>, "voters_hash": "G6yS\/dAZm5hKnCn5cRgBGdw3yGo", "voting_ends_at": null, "voting_starts_at": null}

`election_id` is a unique identifier for the election, and `name` is the election's name.

<`ELGAMAL_PUBLIC_KEY`> is, as detailed earlier, the JSON data structure that represents an El-Gamal public key.

<`QUESTION_LIST`> is a data structure that represents the list of questions and available answers to those questions.

<QUESTION_LIST>
[<QUESTION>, <QUESTION>, ...]

and a single question is a JSON object:

<QUESTION>
{"answer_urls": ["http://example.com/alice", null], "answers": ["alice", "bob"], "max": 1, "question": "Who Should be President?", "short_name": "President"}

which, in this case, contains two possible answers (alice and bob), URLs that describe these answers in greater detail, the text of the question, and a short name for the question. The parameter `max` indicates the maximum number of options that a voter can select, most often 1. Note how, given that this `max` value should be a small integer, it is in fact serialized as an integer, not as a string.

`voters_hash` is the base64 encoding of the SHA1 hash of the list of voters for the election. The list of voters is a JSON array, where each voter is represented *without* a cast ballot and *without* the `vote_hash`, of course (otherwise `voters_hash` changes as voters cast their ballot). For example, a list of voters might be:

<VOTER_LIST> (example)
[{"email": "ben@adida.net", "name": "Ben Adida", "voter_id": "453"}, {"name": "Ella", "openid": "http://ella.example.com", "voter_id": "834"}]

Note: we strictly follow the JSON specification, so the forward-slash character "/" is escaped as "\/". Not all JSON toolkits do this correctly, given that escaping the forward-slash is not necessary given JavaScript specifications, but for interoperability, we choose the strict interpretation. Python's `simplejson` does the right thing.

Open Registration

Helios supports "open registration elections", when the election administrator so desires. In those elections, the voter list is not set ahead of time. In that case, an election data structure does not contain `voters_hash`, but does contain a new field, `openreg`:

<ELECTION>
{"election_id": "agxoZWxpb3N2b3RpbmdyDgsSCEVsZWN0aW9uGAYM", "name": "foo234234", "openreg": True, "pk": <ELGAMAL_PUBLIC_KEY>, "questions": <QUESTION_LIST>, "voting_ends_at": null, "voting_starts_at": null}

Election Fingerprint

Once an election is ready to be used for voting, the administrator *freezes* the election, at which point Helios prevents changing any of the question parameters and voter registration settings: an open election remains an open election, and a closed election remains closed with a fixed voter list.

Such a frozen election can be designated by its **Helios Election Fingerprint**, which is the base-64-string-encoded SHA1 of the election data structure serialized as JSON (with properly alphabetized field names). Note how this fingerprint depends on the list of voters if the election registration status is closed, but not if it is open. In any case, this fingerprint does *not* depend on any cast vote or cast-vote hash.

Vote

A vote contains a list of encrypted answers, and a reference to the election, both by ID (for convenience) and by hash (for integrity.) The hash is the election fingerprint just described.

<VOTE>
{"answers": [<ENCRYPTED_ANSWER>, <ENCRYPTED_ANSWER>, ...], "election_hash": "Nz1fWLvVLH3eY3Ox7u5hxfLZPdw", "election_id": "agxoZWxpb3N2b3RpbmdyDgsSCEVsZWN0aW9uGAYM"}

Each "encrypted answer" corresponds to one election question: it contains a list of ciphertexts (one for each possible choice for that question), a list of corresponding proofs that the ciphertext is correctly formed, and an overall proof that all of the ciphertexts for that election question, taken together, are correctly formed.

<ENCRYPTED_ANSWER>
{"choices": [<ELGAMAL_CIPHERTEXT>, <ELGAMAL_CIPHERTEXT>, ...], "individual_proofs": [<ZK_PROOF_0..1>, <ZK_PROOF_0..1>, ...], "overall_proof": <ZK_PROOF_0..max>}

The value of `max` in `overall_proof` matches the value of `max` in the election's question definition.

When a voter generates a ballot, Helios provides the ballot fingerprint, which is the base64-encoding of the SHA1 hash of the <`VOTE`> data structure defined above.

Proofs

A zero-knowledge proof, denoted <`ZK_PROOF_0..max`>, is a transcript of a non-interactive proof that the corresponding ciphertext encodes an integer value between 0 and `max`. For the overall proof, the ciphertext whose value is being proven to be between 0 and `max` is the homomorphic sum (element-wise product) of the `choices` ciphertexts.

In Helios, all `0..max` proofs are disjunctive proofs (CDS & CP), meaning that the transcript includes `max+1` proofs, one for each possible value of the plaintext, 0 through `max`. The `max+1` individual challenges must sum up to the single actual protocol challenge, which ensures that one of the proofs is real (while the others are simulated.)

<ZK_PROOF_0..max>
[<ZK_PROOF(0)>, <ZK_PROOF(1)>, ..., <ZK_PROOF(max)>]

A single ZK proof is then composed of three messages: the commitment, the challenge, and the response. Since the proof is a Chaum-Pedersen proof of a DDH tuple, the commitment is composed of two values, A and B. Thus, a ZK proof is:

<ZK_PROOF(plaintext)>
{"challenge": "2342342", "commitment": {"A": "28838", "B": "9823723"}, "response": "970234234"}

Ballot Audit Trail

When a voter chooses to audit their ballot, each encrypted answer contains additional information concerning the actual selected choice and the randomness used to encrypt each choice's ciphertext. Specifically, the JSON structure for <`VOTE_WITH_PLAINTEXTS`> is as follows.

<VOTE_WITH_PLAINTEXTS>

{"answers": [<ENCRYPTED_ANSWER_WITH_PLAINTEXT>, <ENCRYPTED_ANSWER_WITH_PLAINTEXT>, ...], "election_hash": <B64_HASH>, "election_id": <ELECTION_ID>}

And the contained `<ENCRYPTED_ANSWER_WITH_PLAINTEXT>` is as follows.

<ENCRYPTED_ANSWER_WITH_PLAINTEXT>
{"answer": 1, "choices": [<ELGAMAL_CIPHERTEXT>, <ELGAMAL_CIPHERTEXT>, ...], "individual_proofs": [<ZK_PROOF_0..1>, <ZK_PROOF_0..1>, ...], "overall_proof": <ZK_PROOF_0..max>, "randomness": [<BIGINT>, <BIGINT>, <BIGINT>]}

Result

The result of an election is represented using two structures: `<RESULT>` and `<RESULT_PROOF>`. The result simply displays the count of votes for each candidate within each question, in an array of arrays format.

<RESULT>
[[<QUESTION_1_CANDIDATE_1_COUNT>, <QUESTION_1_CANDIDATE_2_COUNT>, <QUESTION_1_CANDIDATE_3_COUNT>], [<QUESTION_2_CANDIDATE_1_COUNT>, <QUESTION_2_CANDIDATE_2_COUNT>]]

The `<RESULT_PROOF>` data structure is formatted exactly the same way, with a Chaum-Pedersen proof of proper decryption for each candidate within each question:

<RESULT_PROOF>
[[<QUESTION_1_CANDIDATE_1_PROOF>, <QUESTION_1_CANDIDATE_2_PROOF>, <QUESTION_1_CANDIDATE_3_PROOF>], [<QUESTION_2_CANDIDATE_1_PROOF>, <QUESTION_2_CANDIDATE_2_PROOF>]]

A Note on the Source Code in this Specification

In the rest of this document, we show how to verify various aspects of a Helios election using Python code for concreteness and legibility. We assume that certain data structures have been defined: `election`, `vote`, `proof`, `disjunctive_proof`, and a few others, all of which correspond to collections of fields that directly map to the JSON data structures described above. However, we note that a verification program need not necessarily parse these JSON strings into custom Python objects. It is perfectly acceptable to extract individual fields when necessary.

In particular, in a number of cases, our sample code will call `election.toJSON()`, or `vote.toJSON()` in order to re-convert the data structure to JSON so that it can be hashed and checked for integrity. A verification program that handles JSON strings directly without de-serializing them to Python objects would obviously not need to re-serialize to JSON, either. The original JSON provided by the Helios server hashes appropriately to the intended values.

Verifying a Single Ballot

Recall the Chaum-Pedersen proof that a ciphertext `(alpha,beta)` under public key `(y, (g,p,q))` is proven to encode the value `m` by proving knowledge of `r`, the randomness used to create the ciphertext, specifically that `g, y, alpha, beta/g^m` is a DDH tuple, noting that `alpha = g^r` and `beta/g^m = y^r`.

- Prover sends `A = g^w mod p` and `B = y^w mod p` for a random `w`.
- Verifier sends `challenge`, a random challenge `mod q`.
- Prover sends `response = w + challenge * r`.
- Verifier checks that:
    - `g^response = A * alpha^challenge`
    - `y^response = B * (beta/g^m)^challenge`

```
verify_proof(ciphertext, plaintext, proof, public_key):
if pow(public_key.g, proof.response, public_key.p) !=
((proof.commitment.A * pow(ciphertext.alpha, proof.challenge,
public_key.p)) % public_key.p): return False beta_over_m =
modinverse(pow(public_key.g, plaintext, public_key.p),
public_key.p) * ciphertext.beta beta_over_m_mod_p =
beta_over_m % public_key.p if pow(public_key.y,
proof.response, public_key.p) != ((proof.commitment.B *
pow(beta_over_m_mod_p, proof.challenge, public_key.p)) %
public_key.p): return False return True
```

In a disjunctive proof that the ciphertext is the encryption of one value between 0 and `max`, all `max+1` proof transcripts are checked, and the sum of the challenges is checked against the expected challenge value. Since we use this proof in non-interactive Fiat-Shamir form, we generate the expected challenge value as `SHA1(A0 + "," + B0 + "," + A1 + "," + B1 + ... + "Amax" + "," + Bmax)` with `A0, B0, A1, B1, ... ,Amax, Bmax` in decimal form. (`Ai` and `Bi` are the components of the commitment for the `i`'th proof.)

Thus, to verify a <ZK_PROOF_0..max> on a <ELGAMAL_CIPHERTEXT>, the following steps are taken.

```
verify_disjunctive_0..max_proof(ciphertext, max,
disjunctive_proof, public_key):
for i in range(max+1): # the proof for plaintext "i" if not
verify_proof(ciphertext, i, disjunctive_proof[i], public_key): return
False # the overall challenge computed_challenge =
sum([proof.challenge for proof in disjunctive_proof]) %
public_key.q # concatenate the arrays of A,B values
list_of_values_to_hash = sum([[p.commitment.A,
```

p.commitment.B] for p in disjunctive_proof], []) # concatenate as strings str_to_hash = ",".join(list_of_values_to_hash) # hash expected_challenge = int_sha(str_to_hash) # last check return computed_challenge == expected_challenge

Thus, given `<ELECTION>` and a `<VOTE>`, the verification steps are as follows:

verify_vote(election, vote):
# check hash (remove the last character which is a useless '=') computed_hash = base64.b64encode(hash.new(election.toJSON()).digest())[:-1] if computed_hash != vote.election_hash: return False # go through each encrypted answer by index, because we need the index # into the question array, too for figuring out election information for question_num in range(len(vote.answers)): encrypted_answer = vote.answers[question_num] question = election.questions[question_num] # initialize homomorphic sum (assume operator overload on __add__ with 0 special case.) homomorphic_sum = 0 # go through each choice for the question (loop by integer because two arrays) for choice_num in range(len(encrypted_answer.choices)): ciphertext = encrypted_answer.choices[choice_num] disjunctive_proof = encrypted_answer.individual_proofs[choice_num] # check the individual proof (disjunctive max is 1) if not verify_disjunctive_0..max_proof(ciphertext, 1, disjunctive_proof, election.public_key): return False # keep track of homomorphic sum homomorphic_sum = ciphertext + homomorphic_sum # check the overall proof if not verify_disjunctive_0..max_proof(homomorphic_sum, question.max, encrypted_answer.overall_proof, election.public_key): return False # done, we succeeded return True

Auditing/Spoiling a Single Ballot

Given a `<VOTE_WITH_PLAINTEXTS>` and a claimed vote fingerprint, verification entails checking the fingerprint, checking all of the proofs to make sure the ballot is well-formed, and finally ensuring that the ballot actually encodes the claimed choices.

verify_ballot_audit(vote_with_plaintexts, election, vote_fingerprint)
# check the proofs if not verify_vote(election, vote_with_plaintexts): return False # check the proper encryption of each choice within each question # go through each encrypted answer for encrypted_answer in vote_with_plaintexts.answers: # loop through each choice by integer (multiple arrays) for choice_num in range(len(encrypted_answer.choices)): # the ciphertext and randomness used to encrypt it ciphertext = encrypted_answer.choices[choice_num] randomness = encrypted_answer.randomness[choice_num] # the plaintext we expect, g^1 if selected, or g^0 if not selected if choice_num == encrypted_answer.answer: plaintext = public_key.g else:

```
plaintext = 1 # check alpha if pow(public_key.g, randomness,
public_key.p) != ciphertext.alpha: return False # check beta
expected_beta = (pow(public_key.y, randomness, public_key.p)
* plaintext) % public_key.p if expected_beta != ciphertext.beta:
return False # check the fingerprint vote_without_plaintexts =
vote_with_plaintexts.remove_plaintexts() computed_fingerprint =
base64.b64encode(hash.new(vote_without_plaintexts.toJSON()).d
return computed_fingerprint == vote_fingerprint
```

Verifying a Complete Election Tally

To verify a complete election tally, one should:

- display the computed election fingerprint.
- ensure that the list of voters matches the election voter-list hash.
- display the fingerprint of each cast ballot.
- check that each cast ballot is correctly formed by verifying the proofs.
- homomorphically compute the encrypted tallies and verify, using the result proof, that they correctly decrypt to the claimed results. Display these results.

In other words, the complete results of a verified election includes: the election fingerprint, the list of ballot fingerprints, and the actual count. Any party who verifies the election should re-publish all of these items, as they are meaningless without one another. This is effectively a "re-tally".

Part of this re-tally requires checking a decryption proof, which is almost the same, but not quite the same, as checking an encryption proof with given randomness. First, we document the verification of a decryption proof.

```
verify_decryption_proof(ciphertext, plaintext, proof, public_key):
# Here, we prove that (g, y, alpha, beta/g^m) is a DDH tuple. #
Before we were working with (g, alpha, y, beta/g^m) if
pow(public_key.g, proof.response, public_key.p) !=
((proof.commitment.A * pow(public_key.y, proof.challenge,
public_key.p)) % public_key.p): return False beta_over_m =
modinverse(pow(public_key.g, plaintext, public_key.p),
public_key.p) * ciphertext.beta beta_over_m_mod_p =
beta_over_m % public_key.p if pow(ciphertext.alpha,
proof.response, public_key.p) != ((proof.commitment.B *
pow(beta_over_m_mod_p, proof.challenge, public_key.p)) %
public_key.p): return False # compute the challenge generation,
Fiat-Shamir style str_to_hash = str(proof.commitment.A) + "," +
str(proof.commitment.B) computed_challenge =
int_sha(str_to_hash) # check that the challenge matches return
computed_challenge == proof.challenge
```

Then, the re-tally proceeds as follows.

```
retally_election(election, voters, result, result_proof):
# compute the election fingerprint election_fingerprint =
b64_sha(election.toJSON()) # compute the voter list hash on
```

just the voter identities voters_without_votes = votes.remove_votes() voters_hash = b64_sha(voters_without_votes.toJSON()) # verify, no need to continue if we fail here if voters_hash != election.voters_hash: return False # keep track of voter fingerprints vote_fingerprints = [] # keep track of running tallies, initialize at 0 # again, assuming operator overloading for homomorphic addition tallies = [[0 for a in question.answers] for question in election.questions] # go through each voter, check it for voter in voters: if not verify_vote(election, voter.vote): return False # compute fingerprint vote_fingerprints.append(b64_sha(voter.vote.toJSON())) # update tallies, looping through questions and answers within them for question_num in range(len(election.questions)): for choice_num in range(len(election.questions[question_num].answers)): tallies[question_num][choice_num] = voter.vote.answers[question_num].choices[choice_num] + tallies[question_num][choice_num] # now we have tallied everything in ciphertexts, we must verify proofs for question_num in range(len(election.questions)): for choice_num in range(len(election.questions[question_num].answers)): # verify the tally for that choice within that question # check that it decrypts to the claimed result with the claimed proof if not verify_decryption_proof(tallies[question_num][choice_num], pow(election.public_key.g, result[question_num][choice_num], election.public_key.p), result_proof[question_num][choice_num], election.public_key): return False # return the complete tally, now that it is confirmed return { 'election_fingerprint': election_fingerprint, 'vote_fingerprints' : vote_fingerprints, 'verified_tally' : result }

Election with Multiple Trustees

A Helios election can be configured to have multiple trustees, each of which holds a share of the election secret key.

Before the trustees have submitted their public key shares, the election's public key is null, e.g.:

Election with Trustees, before shares
{"election_id": "agxoZWxpb3N2b3RpbmdyDgsSCEVsZWN0aW9uGAUM", "name": "test-trustees", "pk": null, "questions": [], "voters_hash": "l9Fw4VUO7kr8CvBlt4zaMCqXZ0w", "voting_ends_at": null, "voting_starts_at": null}

The list of trustees for an election can be obtained at

```
http://www.heliosvoting.org/elections/<ELECTI
```

which returns the <TRUSTEE_LIST> data structure as follows:

<TRUSTEE_LIST>
[<TRUSTEE>, <TRUSTEE>, ..., <TRUSTEE>]

where a single <TRUSTEE> data structure is:

<TRUSTEE>
{"decryption_factors": <DECRYPTION_FACTORS>, "decryption_proofs": <DECRYPTION_PROOFS>, "email": "trustee@election.com", "pk": <ELGAMAL_PUBLIC_KEY>, "pok": <ELGAMAL_KEY_POK>}

Key Share and Proof of Knowledge of Secret

The `<ELGAMAL_PUBLIC_KEY>` field is a normal ElGamal public key, as before. The <ELGAMAL_KEY_POK> is a non-interactive proof of knowledge of the secret key corresponding to the given public key. Helios uses the simple Schnorr proof of knowledge of discrete log, which is a simple three-round protocol proof as follows:

- Prover generates `w`, a random integer modulo `q`, and computes `commitment = g^w mod p`.
- Verifier provides `challenge` modulo `q`.
- Prover computes `response = w + x*challenge mod q`, where `x` is the secret key.

Then, the verifier checks that `g^response = commitment * y^challenge`, where `y` is the public key. In the non-interactive setting, the challenge is generated as the decimal representation of the SHA1 of the commitment.

The format for the resulting proof is as follows.

<ELGAMAL_KEY_POK>
{"challenge": "2342342", "commitment": "124235235", "response": "970234234"}

Freezing a Trustee Election

Once all trustees have submitted their public key shares,

Decryption Shares

In Helios, the trustees are all required to show up for decryption. Threshold decryption is *not implemented at this time*. Thus, come decryption time, each trustee provides a decryption factor and a proof that this decryption factor was correctly generated given the trustee's public key.

Thus, `<DECRYPTION_FACTORS>` is structured the same way as `<RESULT>`, an array of arrays, to provide one decryption factor for each choice of each question.

<DECRYPTION_FACTORS>
[[<Q1_CANDIDATE_1_DEC_FACTOR>, <Q1_CANDIDATE_2_DEC_FACTOR>, <Q1_CANDIDATE_3_DEC_FACTOR>], [<Q2_CANDIDATE_1_DEC_FACTOR>, <Q2_CANDIDATE_2_DEC_FACTOR>]]

Then, `<DECRYPTION_PROOFS>` is a similarly structured array of arrays, where each element is a proof of the corresponding decryption factor, much like `<RESULT_PROOF>` is an element-wise decryption proof of `<RESULT>`.

```
<DECRYPTION_PROOFS>
[[<Q1_CANDIDATE_1_PROOF>,
<Q1_CANDIDATE_2_PROOF>,
<Q1_CANDIDATE_3_PROOF>],
[<Q2_CANDIDATE_1_PROOF>,
<Q2_CANDIDATE_2_PROOF>]]
```

Each of these proofs is a DH-tuple proof, just like the original result decryption proof, with the fourth element of the DH tuple the corresponding decryption factor. So, for example, `<Q1_CANDIDATE_1_PROOF>` is a transcript of the proof that `g, Q1_C1_TALLY.alpha, y, <Q1_CANDIDATE_1_DEC_FACTOR>` is a proper DH tuple. The homomorphic tally, prior to decryption, is computed exactly as it was without trustees.

Putting it All Together

At verification time, the steps for a trustee election are only slightly different:

- Each trustee's public-key share is verified against the corresponding proof of knowledge.
- The election's single public key is indeed the product of the key shares.
- Each individual ballot is verified *just as before*.
- The encrypted tally for each candidate to each question is homomorphically computed, *just as before*.
- Each partial decryption for each candidate to each question is verified.
- The final tally for a given candidate is obtained by multiplying the partial decryption factors, and dividing it out of the corresponding encrypted tally's `beta`.

Comments

You do not have permission to add comments.