Helios

Search this site

Helios v3 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.)

This document covers Helios 3.0, due out for release in Fall 2009. The single biggest change is the introduction of modularity of both services (election preparation, ballot preparation, ballot casting) and algorithms, and the additional verbosity required in the data formats to indicate which module is used for each component of an election.

Components

Helios is split into 4 major components:

- **election builder**: a web-based tool to create an election.
- **voting booth**: a web-based voting booth where ballots are filled out and encrypted.
- **ballot casting server**: the server where filled-out ballots are submitted.
- **audit server**: the place where all data is posted at the end of an election.

Audit Data

All data for an election is easily accessible using simple HTTP GET requests. The HTTP interface for accessing all data from a given election is built so as to enable static storage of this data in a simple filesystem made available over the web, to simplify long-term robustness. Consider an election with election id

`<ELECTION_ID>`. Assuming a host and prefix that we denote {HELIOS_HOST}, the election data structure, including the election public key, questions, etc., can be obtained by requesting the following URL:

```
{HELIOS_HOST}/elections/<ELECTION_ID>
```

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

```
{HELIOS_HOST}/elections/<ELECTION_ID>/voters/
```

Given this list, it is possible to extract individual voter identifiers, denoted `<VOTER_ID>`

The list of cast ballots is available at, with each ballot including the <VOTER_ID> that it corresponds to:

```
{HELIOS_HOST}/elections/<ELECTION_ID>/ballots
```

This call will return ballots in chronological (oldest to newest) order, and takes optional parameters `limit` and `after`.

The list of all ballots cast by a voter is:

```
{HELIOS_HOST}/elections/<ELECTION_ID>/ballots
```

For convenience, the last cast ballot is:

```
{HELIOS_HOST}/elections/<ELECTION_ID>/ballots
```

The result of an election is available at:

```
{HELIOS_HOST}/elections/<ELECTION_ID>/result
```

Every election has trustees (sometimes just one), and the list of trustees, including each trustee's public key and PoK, decryption factor and proof is at:

```
{HELIOS_HOST}/elections/<ELECTION_ID>/trustee
```

NOT YET IMPLEMENTED:

While the trustee's robustness information (e.g. Lagrange coeff) is at:

```
{HELIOS_HOST}/elections/<ELECTION_ID>/trustee
```

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": "6811145128679259384514506369165",
"p": "16998971978194099593503959", "q":
"8499485989097049796751", "y":
"7464668703479877164253720373259704"}
```

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`.

In Helios, all hash values are base-64 encoded, prefixed with the algorithm used for hashing, e.g:

```
Hash value example
{"election_hash":
"sha256:c0D1TVR7vcIvQxuwfLXJHa5EtTHZGHpDKdul
```

When there is no hash algorithm specified, the default hash is SHA256.

Voter

A single voter in Helios is represented using a few fields that identify the voter. *This data structure has changed from prior versions of Helios* in order to accommodate multiple types of users, not just users identified by email address.

```
<VOTER>
{"name": "Ben Adida", "uuid":
"60435862-65e3-11de-8c90-001b63948875",
"voter_id": "benadida@gmail.com",
"voter_type": "email"}
```

Together, the `type` and `id` identify the voter via some external authentication mechanism. In the example above, this is a user whose email address is `benadida@gmail.com`. Another example might be:

```
<VOTER>
{"name": "Ben Adida", "uuid":
"4e8674e2-65e3-11de-8c90-001b63948875",
"voter_id": "ben@adida.net", "voter_type":
"email"}
```

where this is a voter identified by the email address ben@adida.net.

The `uuid` field is used as a reference key within Helios.

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

```
<VOTER>
{"name": "Ben Adida", "uuid":
"4e8674e2-65e3-11de-8c90-001b63948875",
 "voter_id":
"http://benadida.myopenid.com",
"voter_type": "openid"}
```

Other fields may be present in a <VOTER> data structure, e.g. `category`. These do not affect the cryptographic processing, but if present, they become part of the hash of the voter list.

Protecting Voter Privacy

In order to protect voter privacy, Helios can obfuscate the voter_id, especially when that voter_id is an email address. This protection is not meant to resist a powerful attacker with other knowledge about the voter, but mostly to prevent activities such as email-address crawlers for the purpose of spamming. In this case, a voter can be represented with the field voter_id_hash replacing voter_id. The hash is SHA256 by default, or specified as a prefix when it is a different hash:

```
<VOTER>
{"name": "Ben Adida", "uuid":
"60435862-65e3-11de-8c90-001b63948875",
"voter_id_hash":
"47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU
"voter_type": "email"}
```

Voter Aliases

In some elections, it may be preferable to never reveal the identity of the voters. This is particularly applicable when organizers are worried about votes being decryptable in 20-30 years, when cryptographic advances make today's algorithms weaker. An election may thus publish only an ALIASED_VOTER:

```
<ALIASED_VOTER>
{"alias": "voter_123", "uuid": "b7dbd90a-
65e3-11de-8c90-001b63948875"}
```

An aliased voter still has a UUID, so it can still be referred appropriately in the rest of the system.

Casting a Vote

Once a voter has cast a ballot, the encrypted vote representation is then:

```
<CAST_VOTE>
{"cast_at": "2009-07-15 12:23:46", "vote":
<VOTE>, "vote_hash":
"sha256:8bncn23nsfsdk234234",
 "voter_hash": "sha256:2bxksdlkxnsdf",
"voter_uuid": "b7dbd90a-65e3-11de-
8c90-001b63948875"}
```

`cast_at` is the timestamp of the cast vote in UTC.

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

`vote_hash` is available to enable a shorter version of this data structure:

```
<SHORT_CAST_VOTE>
{"cast_at": "2009-07-15 12:23:46",
"vote_hash":
"sha256:c0D1TVR7vcIvQxuwfLXJHa5EtTHZGHpDKdul
 "voter_hash": "sha256:2bxksdlkxnsdf",
"voter_uuid": "b7dbd90a-65e3-11de-
8c90-001b63948875"}
```

where only the hash and not the full vote is listed.

Election

An election is represented as:

```
<ELECTION>
{"cast_url": "https://heliosvoting.org
/cast/",
 "description": "... blah blah blah ...
info about the election",
 "frozen_at": null,
 "name": "Student President Election at
Foo University 2010",
 "openreg": false, "public_key":
<ELGAMAL_PUBLIC_KEY>,
 "questions": <QUESTION_LIST>,
 "short_name": "fooprez2010",
 "use_voter_aliases": false,
 "uuid": "1882f79c-65e5-11de-
8c90-001b63948875",
 "voters_hash":
"G6yS/dAZm5hKnCn5cRgBGdw3yGo"}
```

`short_name`, `name`, and `description` describe the election. The short name must be a few characters without a space (almost like a database key), the name can be a long string, and the description is an even longer description of the election.

REMOVED 2009-11-19 `ballot_type` indicates the kind of ballot that this election expects. Helios only supports "homomorphic" at this time. And this is a question-level issue anyways, since each question is different.

`cast_url` indicates the URL where ballots for this election should be cast.

`frozen_at` indicates the timestamp at which this election was frozen. `It remains null until the election is frozen.`

`openreg` indicates whether voters can be added to the list after the election has started.

REMOVED 2009-11-19 `tally_type` indicates how tallying of the ballots should occur. Helios only supports "homomorphic". This is a separate field from `ballot_type` because in some homomorphic elections, ballots may be weighted differently depending on the voter category (e.g. students vs. faculty). tally_type also only pertains to each question, not the whole election.

`use_voter_aliases` indicates whether this election aliases its voters.

`uuid` 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"], "choice_type": "approval", "max":
1, "min": 0,
 "result_type": "absolute", "question":
"Who Should be President?", "short_name":
"President", "tally_type": "homomorphic"}
```

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. The parameter `min` indicates the minimum number of options that a voter must select, most often 0. Note how, given that `max` and `min` should be small integers, they are in fact serialized as integers, not as strings. `choice_type` indicates the kind of question, for now just approval (possibly with a maximum number of choices). If max is null, then it's approval voting for as many candidates as desired. `tally_type` indicates how the question is tallied, e.g. `homomorphic` or `mixnet`.

ADDED 2010-08-06: result_type indicates how the result is computed, i.e. absolute or relative, and other result computations. The result_type depends on the choice_type and the max.

`voters_hash` is the hash of the list of voters for the election. The list of voters is a JSON array of <VOTER> data structures. For example, a list of voters might be:

```
<VOTER_LIST> (example)
[{"id": "benadida@gmail.com", "name": "Ben
Adida", "type": "email", "uuid":
"60435862-65e3-11de-8c90-001b63948875"},
{"id": "ben@adida.net", "name": "Ben2
Adida", "type": "email", "uuid":
"4e8674e2-65e3-11de-8c90-001b63948875"}]
```

Note: where Helios v1.0 and v2.0 escaped the "/" character as "\/", Helios v3.0 does not. We believe this escaping was in error.

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 contains a null `voters_hash`, and sets `openreg` to true.

Note: in Helios v1.0 and v2.0, the `openreg` field was present only for open registration, and `voters_hash` was absent when openreg was true. For serialization consistency, Helios v3.0 always contains all fields.

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. The `frozen_at` field then indicates the timestamp at which the election was frozen.

Such a frozen election can be designated by its **Helios Election Fingerprint**, which is the base-64-string-encoded SHA256 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":
"sha256:Nz1fWLvVLH3eY3Ox7u5hxfLZPdw",
"election_uuid": "1882f79c-65e5-11de-
8c90-001b63948875"}
```

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.

For approval voting questions, the overall_proof is absent.

When a voter generates a ballot, Helios provides the ballot fingerprint, which is the base64-encoding of the SHA256 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_uuid": <ELECTION_UUID>}
```

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 the `<RESULT>` data structure. The proofs of the decryption are done at the Trustee level. 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>]]
```

Trustee

Even if there is only one keypair in the case of a simple election, Helios v3 (in a departure from previous versions), represents every election as having trustees. If there is only one trustee, that's fine, but the data structure remains the same:

```
<TRUSTEE>
{"decryption_factors":
<LIST_OF_LISTS_OF_DEC_FACTORS>,
 "decryption_proofs":
<LIST_OF_LISTS_OF_DEC_PROOFS>,
 "pok": <POK_OF_SECRET_KEY>,
 "public_key": <PUBLIC_KEY>,
 "public_key_hash": <PUBLIC_KEY_HASH>,
 "uuid": <UUID_OF_TRUSTEE>}
```

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_
    ((proof.commitment.A * pow(ciphertext.al
       return False

beta_over_m = modinverse(pow(public_key.g, p
beta_over_m_mod_p = beta_over_m % public_key

if pow(public_key.y, proof.response, public_
   ((proof.commitment.B * pow(beta_over_m_mo
      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, disjunc
    return False

# the overall challenge
computed_challenge = sum([proof.challenge fo

# concatenate the arrays of A,B values
list_of_values_to_hash = sum([[p.commitment.

# concatenate as strings
str_to_hash = ",".join(list_of_values_to_has

# hash
expected_challenge = int_sha(str_to_hash)

# last check
return computed_challenge == expected_challe
```

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

`verify_vote(election, vote):`

```
     # check hash (remove the last character whic
     computed_hash = base64.b64encode(hash.new(el
     if computed_hash != vote.election_hash:
         return False

     # go through each encrypted answer by index,
     # into the question array, too for figuring
     for question_num in range(len(vote.answers))
        encrypted_answer = vote.answers[question_
        question = election.questions[question_nu

        # initialize homomorphic sum (assume oper
        homomorphic_sum = 0

        # go through each choice for the question
        for choice_num in range(len(encrypted_ans
          ciphertext = encrypted_answer.choices[c
          disjunctive_proof = encrypted_answer.in

          # check the individual proof (disjuncti
          if not verify_disjunctive_0..max_proof(
             return False

          # keep track of homomorphic sum
          homomorphic_sum = ciphertext + homomorp

        # check the overall proof
        if not verify_disjunctive_0..max_proof(ho
                                              en
                                              el
            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_plain
         return False

     # check the proper encryption of each choice
```

```
    # go through each encrypted answer
    for encrypted_answer in vote_with_plaintexts
        # loop through each choice by integer (m
        for choice_num in range(len(encrypted_an
          # the ciphertext and randomness used t
          ciphertext = encrypted_answer.choices[
          randomness = encrypted_answer.randomne

          # the plaintext we expect, g^1 if sele
          if choice_num == encrypted_answer.answ
            plaintext = public_key.g
          else:
            plaintext = 1

          # check alpha
          if pow(public_key.g, randomness, publi
            return False

          # check beta
          expected_beta = (pow(public_key.y, ran
          if expected_beta != ciphertext.beta:
            return False

    # check the fingerprint
    vote_without_plaintexts = vote_with_plaintex
    computed_fingerprint = base64.b64encode(hash

    return computed_fingerprint == vote_fingerpr
```

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
- verify each trustee's partial decryption
- combine the partial decryptions and verify that those decryptions, the homomorphic encrypted tallies, and the claimed plaintext results are consistent.

In other words, the complete results of a verified election includes: the election fingerprint, the list of ballot fingerprints, the trustee decryption factors and proofs, and the final plaintext counts. 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 partial decryption proof, which is almost the same, but not quite the same, as checking an encryption proof with given randomness.

Given a ciphertext denoted `(alpha,beta)`, and a trustee's private key `x` corresponding to his public key `y`, a partial decryption is:

```
  dec_factor = alpha^x mod p.
```

The trustee then provides a proof that (g, y, alpha, dec_factor) is a proper DDH tuple, which yields a Chaum Pedersen proof of discrete log equality. Verification proceeds as follows:

verify_partial_decryption_proof(ciphertext, decryption_factor, proof, public_key):

```
     # Here, we prove that (g, y, ciphertext.alph
     # Before we were working with (g, alpha, y,
     if pow(public_key.g, proof.response, public_
        ((proof.commitment.A * pow(public_key.y,
           return False

     if pow(ciphertext.alpha, proof.response, pub
        ((proof.commitment.B * pow(decryption_fac
           return False

     # compute the challenge generation, Fiat-Sha
     str_to_hash = str(proof.commitment.A) + ","
     computed_challenge = int_sha(str_to_hash)

     # check that the challenge matches
     return computed_challenge == proof.challenge
```

Then, the decryption factors must be combined, and we check that:

```
  dec_factor_1 * dec_factor_2 * ... *
dec_factor_k * m = beta (mod p)
```

Then, the re-tally proceeds as follows.

retally_election(election, voters, result, result_proof):

```
     # compute the election fingerprint
     election_fingerprint = b64_sha(election.toJS

     # keep track of voter fingerprints
     vote_fingerprints = []

     # keep track of running tallies, initialize
     # again, assuming operator overloading for h
     tallies = [[0 for a in question.answers] for

     # 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.v

            # update tallies, looping through questi
            for question_num in range(len(election.q
                for choice_num in range(len(election
                    tallies[question_num][choice_num



        # now we have tallied everything in cipherte
        for question_num in range(len(election.quest
            for choice_num in range(len(election.que

                decryption_factor_combination = 1

                for trustee_num in range(len(electio

                    trustee = election.trustees[trus

                    # verify the tally for that choi
                    # check that it decrypts to the
                    if not verify_partial_decryption
                        trustee.decryption_factors[q
                        trustee.decryption_proof[que
                        trustee.public_key):
                      return False

                    # combine the decryption factors

                    decryption_factor_combination *=

                if (decryption_factor_combination *

                    != tallies[question_num][choice_

                        return False

        # return the complete tally, now that it is
        return {
            'election_fingerprint': election_fingerp
            'vote_fingerprints' : vote_fingerprints,
            'verified_tally' : result
        }
```

Comments

You do not have permission to add comments.