E-Voting Protocol (version 1.1h)

Preliminaries

The protocol assumes that an administrator creates an election and appoints two collectors, C_1 and C_2, who are assumed to be conflict-of-interest. The administrator and both collectors need public keys for secure communication and authentication. These three parties are identified by their hosts, ports, and public keys. The protocol also assumes M candidates and N voters. The administrator acts as a server for the voters and a client for the collectors. The collectors act as servers for the administrator and collector 1 is a server to collector 2. Voters are just clients.

Before performing the protocol, all parties must agree on the prime A and generator g. You can choose $A = 2^{256} + 230191$ and $g = 5$. This should allow the system to handle any $L \leq 64$.

Integers are written to messages in big-endian byte order. In some parts of the protocol, a "large integer" must be sent through a message. In all such cases, the field of the message is variable length and is prefixed by a 4-byte length field. The integer is included in big-endian byte order in two's complement with enough bytes to include the entire integer as well as at least one sign bit.

Public Key Cryptography

Each entity participating in an election needs an RSA key pair, and must know the public keys of the administrator and the two collectors. The administrator must know every entity's public key. Every message sent in our protocol is signed, and some of them are encrypted as well. The formats are specified below.

If a message is signed, the message is packaged as follows (here, payload is the unsigned, unencrypted message):

```
payload_length   4 bytes
payload          var. length
random_bits      32 bytes
signature_length 4 bytes
signature        var. length
```

`signature` is a value that represents an RSA signature. The value is equal to $m^d \pmod{n}$, where m is the nonnegative integer represented in big-endian by the BLAKE2b hash of the concatenation of `payload` and `random_bits`. The purpose of `random_bits` is to prevent an adversary from having any control over the hash that is signed with RSA.

To check the signature, the recipient recomputes the hash and compares the result against $(m^d)^e \pmod{n}$.

If a message is signed and encrypted, a random 256-bit symmetric key is generated just for this message. Then, the message is packaged as follows:

`payload_length`	`4 bytes`
`payload_encrypted`	`var. length`
`key_encrypted_length`	`4 bytes`
`key_encrypted`	`var. length`
`signature_length`	`4 bytes`
`signature`	`var. length`
`knowledge_proof_length`	`4 bytes`
`knowledge_proof`	`var. length`

`payload_encrypted` is the original message encrypted with Salsa20 using the symmetric key generated. The nonce is set to 0, which is safe since keys are never reused. `key_encrypted` is a value that represents RSA ciphertext. The value is equal to $m^e \pmod{n}$ (using the public key of the recipient), where m is the nonnegative integer represented in big-endian by the symmetric key generated. `signature` is a value that represents an RSA signature. The value is equal to $m^d \pmod{n}$ (using the private key of the sender), where m is the nonnegative integer represented in big-endian by the BLAKE2b hash of the concatentation of `payload_length`, `payload_encrypted`, and `key_encrypted`. The purpose of `knowledge_proof` is to prove that the sender knows the symmetric key. It is the RSA encryption of the symmetric key (big-endian, nonnegative) under the sender's public key.

When such a message is received, first the signature is checked. To do this, the recipient recomputes the hash and compares the result against $(m^d)^e \pmod{n}$. Assuming this check passes, the encrypted key is decrypted and `knowledge_proof` is checked by re-encrypting the symmetric key, this time under the sender's public key, and comparing the result against `knowledge_proof`. Assuming the check passes, this symmetric key is then used to decrypt the payload.

Election ID and Key Hash

Note that all messages include the election ID, and some messages include the hash of the administrator's public key (`key_hash`). These should be checked once a message is received to ensure the message is intended for the correct election. `key_hash` is the BLAKE2b hash of the administrator's public key, and similarly, `collector_key_hash` is the BLAKE2b hash of the collector's key.

Paillier Cryptosystem Details

(Note: the notation in this section differs from other sections)

The Paillier cryptosystem is used essentially as described on Wikipedia (link) for primes of equal length. These primes should be significantly longer than A (e.g. twice as many bits) to minimize the probability of verification of sub-protocol 1 failing for correct information. To initialize the Paillier cryptosystem, collector 1 generates two distinct primes p and q of equal length. Then, $n = pq$, $\lambda = (p-1)(q-1)$, and $g = n+1$.

To encrypt a value x where $0 \leq x < n$, first a random value $r \in \mathbb{Z}_n^*$ is selected. The encrypted message is then $g^x r^n \mod n^2$. Encryption is notated as $c = E(x)$.

To decrypt a value c where $c \in \mathbb{Z}_{n^2}^*$, collector 1 performs $x = \lambda^{-1} \lfloor \frac{(c^\lambda \mod n^2) - 1}{n} \rfloor \mod n$. Decryption is notated as $x = D(c)$.

Sending and Receiving Messages

Messages are sent through TCP. To allow the receiver to easily determine where one message ends and another begins, the sender prefixes each message (signed or signed and encrypted) with the length of the message, represented by 4 bytes in big-endian order.

Protocol Description

Initialization

To create an election, an administrator first needs to appoint two collectors. These collectors should be chosen such that they are unlikely to collude. For example, they may be conflicting political parties. The administrator connects and sends a message as follows to each desired collector:

Signed

`message_type`	`TYPE_COLLECT_REQUEST`
`election_ID`	`16 bytes`
`collector_index`	`1 byte`
`pk_length`	`4 bytes`
`pk`	`var. length`
`collector_key_hash`	`64 bytes`

Each collector then responds as follows to indicate acceptance or denial:

Signed

`message_type`	`TYPE_COLLECT_STATUS`
`key_hash`	`64 bytes`
`election_ID`	`16 bytes`
`acceptance`	`0x00 or 0x01`

The administrator can then check whether the desired collectors have accepted or rejected the request. Once the administrator has chosen two collectors who have accepted the position, they can continue.

Next, the administrator needs to send election metadata to each collector. They do this with a message formatted as follows:

Signed

`message_type`	`TYPE_METADATA_COLL`
`election_ID`	`16 bytes`
`other_C_host_length`	`4 bytes`
`other_C_host`	`var. length`
`other_C_port`	`2 bytes`
`other_C_pk_length`	`4 bytes`
`other_C_pk`	`var. length`
`M`	`1 byte`

Collector 2 now has the information necessary to form a connection to collector 1.

Registration

The voter must prove their right to vote to the administrator before registering. How this is done is out of scope of this protocol. Once the voter is authorized to vote, they are given an ID (4 bytes) and they give the administrator a public key. Both are just for this election.

The voters now connect to the administrator and register, asking for each collector's host, port, and public key, along with a list of the candidates. First, the voter sends the following to the administrator:

Signed	
message_type	TYPE_REGISTER
key_hash	64 bytes
voter_ID	4 bytes

The administrator responds as follows:

Signed	
message_type	TYPE_METADATA_VOTER
election_ID	16 bytes
C1_host_length	4 bytes
C1_host	var. length
C1_port	2 bytes
C1_pk_length	4 bytes
C1_pk	var. length
C2_host_length	4 bytes
C2_host	var. length
C2_port	2 bytes
C2_pk_length	4 bytes
C2_pk	var. length
M	1 byte
name1_length	4 bytes
name1	var. length
...	...

Once all voters have registered (e.g. the registration period ends), the administrator sends the list of registered voters to each collector:

Signed	
message_type	TYPE_VOTERS
election_ID	16 bytes

```
N                  4 bytes
voter1_ID          4 bytes
voter1_pk_length   4 bytes
voter1_pk          var. length
...                ...
```

At this point, collector 1 initializes a Paillier cryptosystem, and then LAS is performed. To perform LAS, collector 1 sends collector 2 message `TYPE_LAS1`, and then collector 2 sends collector 1 message `TYPE_LAS2`. This also provides collector 2 with n for later use of the Paillier cryptosystem. See the document provided in the class for details on the contents of these messages.

Signed and Encrypted

```
message_type      TYPE_LAS1
...               ...
```

Signed and Encrypted

```
message_type      TYPE_LAS2
...               ...
```

Voting

Each voter must obtain shares from each collector, and also must find their location and obtain N. The voter first connects and sends a message to collector 2 as follows:

Signed

```
message_type  TYPE_SHARES_REQUEST
key_hash      64 bytes
election_ID   16 bytes
voter_ID      4 bytes
```

The collector then responds with:

Signed and Encrypted

```
message_type      TYPE_SHARES
key_hash          64 bytes
election_ID       16 bytes
N                 4 bytes
R_length          4 bytes
Rj,i              var. length
Si,Cj_length      4 bytes
Si,Cj             var. length
```

`S'i,Cj□length`	`4 bytes`
`S'i,Cj`	`var. length`
`~Si,Cj□length`	`4 bytes`
`~Si,Cj`	`var. length`
`~S'i,Cj□length`	`4 bytes`
`~S'i,Cj`	`var. length`

The voter then does the same with collector 1. They must wait until they obtain a message back from collector 2 before connecting to collector 1. Otherwise, collector 1 may not be ready.

The voter can verify that the N's are consistent, and compute their location by summing the two values received for $R_{j,i}$. The voter then creates their ballots and commitments ($p_i, p'_i, g^{s_{ii}}, g^{s'_{ii}}, g^{s_{ii}s'_{ii}}$). They send these to both collectors:

Signed

`message□type`	`TYPE□BALLOT`
`key□hash`	`64 bytes`
`election□ID`	`16 bytes`
`voter□ID`	`4 bytes`
`p□i□length`	`4 bytes`
`p□i`	`var. length`
`p'□i□length`	`4 bytes`
`p'□i`	`var. length`
`g^s□ii□length`	`4 bytes`
`g^s□ii`	`var. length`
`g^s'□ii□length`	`4 bytes`
`g^s'□ii`	`var. length`
`g^ss□length`	`4 bytes`
`g^(s□ii s'□ii)`	`var. length`

Sub-protocol 1

A collector initiates verification of a voter's vote by sending the corresponding voter ID to the other collector:

Signed

`message□type`	`TYPE□VERIFY1`
`key□hash`	`64 bytes`
`election□ID`	`16 bytes`
`voter□ID`	`4 bytes`

Verification in sub-protocol 1 involves two applications of secure two-party multiplication (STPM). For each application, the collector who did not initiate the check will send the other collector a value, and then the collector who initiated the check will respond with another value (see later sections for more details). This is done as follows:

Signed and Encrypted

`message_type`	`TYPE_VERIFY2`
`key_hash`	`64 bytes`
`election_ID`	`16 bytes`
`STPM_index`	`1 byte`
`value_length`	`4 bytes`
`value`	`var. length`

Signed and Encrypted

`message_type`	`TYPE_VERIFY3`
`key_hash`	`64 bytes`
`election_ID`	`16 bytes`
`STPM_index`	`1 byte`
`value_length`	`4 bytes`
`value`	`var. length`

Since the e-voting protocol requires that STPM is performed twice, these two messages are sent two times, first with `STPM_index` set to 0, then with `STPM_index` set to 1.

Then, each collector sends the other their product:

Signed and Encrypted

`message_type`	`TYPE_VERIFY4`
`key_hash`	`64 bytes`
`election_ID`	`16 bytes`
`product_length`	`4 bytes`
`product`	`var. length`

Next, both collectors complete sub-protocol 1.

Sub-protocol 2

Each collector sends their $g^{\tilde{S}_i}$ to the other:

Signed and Encrypted

`message_type`	`TYPE_VERIFY5`
`key_hash`	`64 bytes`

election_ID	16 bytes
g^~S_i_length	4 bytes
g^~S_i	var. length
g^~S'_i_length	4 bytes
g^~S'_i	var. length

They can then use the received values to perform sub-protocol 2.

As the collectors receive and verify ballots from the voters, they send them to the administrator to be published on the bulletin board:

Signed	
message_type	TYPE_PUBLISH
key_hash	64 bytes
election_ID	16 bytes
voter_ID	4 bytes
p_i_length	4 bytes
p_i	var. length
p'_i_length	4 bytes
p'_i	var. length

As the administrator receives ballots from the collectors, they sum them and publish them on the web-based bulletin board (see Zou et al. (2017)). No information about the vote totals is visible until all ballots are received.

Constant values:

Constant Name	Constant Value
TYPE_COLLECT_REQUEST	0x00
TYPE_COLLECT_STATUS	0x01
TYPE_METADATA_COLL	0x02
TYPE_REGISTER	0x03
TYPE_METADATA_VOTER	0x04
TYPE_VOTERS	0x05
TYPE_LAS1	0x06
TYPE_LAS2	0x07
TYPE_SHARES_REQUEST	0x08
TYPE_SHARES	0x09
TYPE_BALLOT	0x0a
TYPE_VERIFY1	0x0b
TYPE_VERIFY2	0x0c
TYPE_VERIFY3	0x0d

`TYPE_VERIFY4`	`0x0e`
`TYPE_VERIFY5`	`0x0f`
`TYPE_PUBLISH`	`0x10`

Sub-protocol 1 Details

Again, sub-protocol 1 is initiated with the `TYPE_VERIFY1` message. As described in Zou et al. (2017), the two applications of STPM are used to compute $S_{i,C_1}S'_{i,C_2}$ and $S'_{i,C_1}S_{i,C_2}$ (`TYPE_VERIFY2` and `TYPE_VERIFY3`). For this protocol, the first application is used to compute $S_{i,C_1}S'_{i,C_2}$, and the second application is used to compute $S'_{i,C_1}S_{i,C_2}$. Then, the products, computed according to Zou et al. (2017), are sent in the `TYPE_VERIFY4` message.

STPM is performed as described in Zou et al. (2017) using the Paillier cryptosystem as described above. Assume we are trying to compute x_1x_2, where collector A has x_1 and collector B has x_2. STPM should be done as if as follows: Collector 1 sends collector 2 $E(x_1)$ in message `TYPE_VERIFY2`. Collector 2 then sends $c = E(x_1)^{x_2}E(r_2)^{-1} \pmod{n^2}$ to collector 1 in message `TYPE_VERIFY3`, where r_2 is a random integer $0 \leq r_2 < n$. Then, Collector A computes $r_1 = D(c)$. Finally, for each collector, $R = r$ if $2r < n$, and $R = r - n$ otherwise. Then, it is very likely that $R_1 + R_2 = x_1x_2$ (not modular addition), assuming the primes were chosen to be large enough.