Remote E-Voting System

Crypto2-Spring 2013

Benjamin Kaiser
Jacob Shedd
Jeremy White
<table>
<thead>
<tr>
<th>Phases</th>
<th>Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Trusted Authority (TA) distributes 4 keys to Registrar, Notary, Vote Server, and Verifier. TA uses Shamir secret sharing scheme with ( k,n = 3,4 ). TA publishes public key as voting public key (vpk).</td>
</tr>
<tr>
<td>Registration</td>
<td>Voter goes to Registrar and presents drivers license, passport, birth certificate, etc... to obtain public and private keys (pk, sk), using Paillier scheme. Registrar publishes pk</td>
</tr>
<tr>
<td>Voting</td>
<td>Voter chooses vote ( v ) and has it signed by the notary, then sends in ciphervote and signed ciphervote</td>
</tr>
<tr>
<td></td>
<td>Vote server actively tallys ciphervote homomorphically</td>
</tr>
<tr>
<td>Verifying</td>
<td>After Voting phase, ciphervotes and ( \text{sum}(\text{ciphervotes}) ) are sent to verifier. Verifier collects ( k ) or ( n ) shared secrets to build private key, then uses it to decrypt votes and verify results. Verifier then publishes plaintext votes, along with ciphervotes.</td>
</tr>
<tr>
<td></td>
<td>Voters can verify their vote was tallied correctly by looking up their votes by their ciphervote, and may also verify entire summation of votes.</td>
</tr>
</tbody>
</table>
Conceptual Model

(E-voting minus transport layer security)

1. Trusted Authority
   - Generates voting public key and voting secret key, vpk, vsk
   - vsk is split k,n ways using Shamir Secret Sharing with k=3,n=4
   - n shares are distributed to Registrar, Notary, Vote Server, and Verifier

2. Registrar
   - Authenticates people in-person by means of drivers license, passport, etc..
   - Generates public/private keys for voter: pk, sk
   - Publishes pk

3a. Voter
   - Decide vote, v
   - Encrypt vote with Paillier and voting public key: c=E(v, vpk)
   - Blind ciphervote with RSA: 
     \[ c' = c^r \mod n, \] where \( r \) is a random number and e,n are the public key of the Notary
   - Connect to Notary
   - Receive random bits b from Notary
   - Send \( E(b, sk) \), c’ to Notary

3b. Notary
   - Generate random bits b, and send to Voter
   - Receive b signed by Voter
   - Notary decrypts bits with Voter public key
   - Checks if Voter has already committed vote.
   - Receive blinded ciphervote c’
   - Send back signed blinded ciphervote, s’

3c. Verifier
   - Receive \( S_i, c_i \) for \( i=0 \rightarrow v \)
   - Receive shared keys from k of n parties
   - Decrypt and publish \( S, c, v, c_{total} \)

4. Vote Server
   - Receive \( S, c \)
   - Check \( c = D(S, npk) \)
   - Record \( S, c \)
   - Tally homomorphically \( c_{total} \)

Voting Ends

-------------------------

Voter has personal verification in published votes by check \( c \leftrightarrow v \)

Publish \( S_i, c_i \) for \( i=0 \rightarrow v \)

Trusted Authority

Generate random bits b, and send to Voter
Receive b signed by Voter
Notary decrypts bits with Voter public key
Checks if Voter has already committed vote.
Receive blinded ciphervote c’
Send back signed blinded ciphervote, s’
Secure Channel

• Client generates 1024-bit AES key in CBC mode
• Encrypts key using server’s RSA public key, sends across channel.
• Server decrypts using private key, then AES key is used as a session key for future communications.
• Different session keys are used for registrar/notary communications.
Eligibility: Only eligible voters can cast votes
Privacy: No coalition of participants can gain information about a voter’s vote
Individual Verifiability: Each eligible voter can verify their vote was really counted
Fairness: No participant can gain knowledge about the tally before the counting stage
Robustness: Faulty behavior of certain sized coalition of participants can be tolerated.
Receipt-freeness, incoercibility: No voter can prove to another how they voted.
Untappable channel: Channel between parties is confidential.

Not Achieved

Untraceable anonymous channel: Messages on channel cannot be traced to sender.
Untappable anonymous channel: Channel guaranteeing both anonymity and confidentiality.

*http://people.ksp.sk/~zuzka/elevote.pdf*
### Attacks – Single Party

<table>
<thead>
<tr>
<th>Rogue Voter (Rog):</th>
<th>Balancing Party:Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>forges vote from client</td>
<td>Notary: verifies authentication</td>
</tr>
</tbody>
</table>
# Attacks – Single Party

<table>
<thead>
<tr>
<th>Eaves Dropper (Eve)</th>
<th>Balancing Party: Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>reveals vote on channel</td>
<td>Voter, Vote Server, Notary: Channels use AES-CBC</td>
</tr>
<tr>
<td>sends fake votes</td>
<td>Voter, Vote Server, Notary: Channel uses RSA private keys for authentication</td>
</tr>
<tr>
<td>replays legitimate votes</td>
<td>Vote Server, Published Results: Doubled up votes will share same signatures and ciphertext, easily caught.</td>
</tr>
<tr>
<td>Gains information from packet transfer</td>
<td>Voter, Vote Server, Notary: Channel uses cryptographically strong padding</td>
</tr>
<tr>
<td>Flips bits</td>
<td>Voter, Vote Server, Notary: Channel uses AES-CBC, CBC guarantees integrity</td>
</tr>
</tbody>
</table>
## Attacks – Single Party

<table>
<thead>
<tr>
<th>Voter (Vin)</th>
<th>Balancing Party: Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>votes twice</td>
<td>Notary: won’t sign second vote</td>
</tr>
<tr>
<td>votes for fake candidate</td>
<td>Verifier: discard invalid vote</td>
</tr>
<tr>
<td>claims vote is wrong</td>
<td>??Registrar, Notary, Vote Server??:</td>
</tr>
</tbody>
</table>
## Attacks – Single Party

<table>
<thead>
<tr>
<th>Vote Server (Mal)</th>
<th>Balancing Party: Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>inserts fake, unsigned votes</td>
<td>Notary: invalidates unsigned votes</td>
</tr>
<tr>
<td>discards legitimate, signed votes</td>
<td>Notary: raises suspicion, signed votes were not sent in</td>
</tr>
<tr>
<td></td>
<td>Voter: complains vote was not counted</td>
</tr>
<tr>
<td>inserts votes for registered voters that did not vote</td>
<td></td>
</tr>
<tr>
<td>decrypts votes early</td>
<td>Registrar, Notary, Verifier: need 3 out of 4 shares to create voting secret key to decrypt vote</td>
</tr>
</tbody>
</table>
Purpose of the Registrar

• Each voter needs to, of course, register with the Registrar.

• When you show up at the Registrar they would check your identification and have you set up a password while they set up the software for you.

• Later on, now that your software is active, you can log on using the password you provided.
• The software would then be the one to generate your public/private key pair and publish your public key.
  – We had the key generated in the Registrar simulating the software generating your key
  – You are prompted twice in a row for your password since there is no wait time between interacting with the Registrar and then signing on to the voting software
Purpose of the Notary

- Each voter who has registered with the Registrar can sign on to their voting software and create a legitimate vote. They then send a blinded copy to the Notary for a signature.
- The Notary validates the signature with the voter’s key, to make sure that they are valid, then signs their blinded vote.
Possible attack on Hash Protocol

• In the protocol we received, the message packet that is sent to the server from the voter once the connection has been established is: \( m = E(\text{sign}(\text{hash}(\text{voter info, including vote, and including nonces}))) \)

• Part of the verification to make sure the server hasn’t tampered with the vote is posting the signed hash.
• This would lead to a potential attack were the server could go through and generate a list of nonces coupled with the vote for the candidate they want to win until they get a hash collision.

• They can then post the signed hash they received with their corresponding vote.

• The reason this attack was un-successful was because along side this message was also sent another signed copy of the message which the server would only be able to spoof if they got the voters private key.
Crypto2 Attacks
Transport-level Security

- The transport level security is very strong.
- RSA-2048 encrypts the handshake, which is when a 256-bit AES session key is established.
- Messages are given nonces and padding, then hashed with SHA-512 and encrypted with that session key.
Server.py

votestxt = open("votes.txt", "w")
if len(temp) == 0:
    votestxt.write(str(long(msg)))
else:
    d = paillier.add(long(temp), long(msg), paillier.PAILLIER_Private("keys/private/homomorphic.private"))
votestxt.write(str(d))
votestxt.close()
def add(c1, c2, priv):
    n1 = priv.p * priv.q
    return (c1 * c2) % (n1 ** 2)
SIS - insecurity

Example question 4: Could I require everyone to use their RIN# as part of their authentication?
-Yes, but RIN’s could be easily guessed or forged.

- SIS is used as the registrar in this protocol, which means it maintains the valid voter list.
- A lot of trust is placed in SIS, as it stores a full list of public keys and their associated voters.
- Since public keys are published along with votes after voting, if SIS was compromised, all votes could be linked to the voters.
- SIS has extremely poor single-factor authentication based on student RIN and a six-digit PIN.
- RINs can be easily phished and PINs can be reset to the user’s birthday: mmddyy, also easily phished or publicly accessed.
Client.py

- Voting candidate is secured on client side

```python
custom = int(raw_input("custom:"))

#message = pack_message(paillier.encrypt(crypto_dictionary["candidates"],
message = pack_message(paillier.encrypt(custom, crypto_dictionary["home"])
if message[0]:
    try:
        # code
```
Anonymity

• No anonymous channel
• Public keys<->votes<->IP address
Pycrypto RSA

- **Encrypt() Attention**: this function performs the plain, primitive RSA encryption (*textbook*). In real applications, you always need to use proper cryptographic padding, and you should not directly encrypt data with this method. Failure to do so may lead to security vulnerabilities. It is recommended to use modules `Crypto.Cipher.PKCS1_OAEP` or `Crypto.Cipher.PKCS1_v1_5` instead.
Textbook RSA

- Deterministic

```python
# Padding nonced_message to length of 128
while len(nonced_message) < padding_length:
    nonced_message += "."
```