**Neff’s Cryptographic Electronic Voting Machine Protocol**

**Introduction**

Electronic voting promises convenient and efficient voting that we come to expect from our modern world, unfortunately security pays a price for this convenience. In a democracy, fair elections should be a number one priority for the government, which is why we can’t move from a more secure system to a less secure one even if its more convenient to do so. Neff’s cryptographic voting scheme attempts to solve security issues by encrypting voter’s ballots in the memory of the direct-recording electronic (DRE) voting machine.

**Neff’s Scheme**

**Assumptions**

Neff’s scheme assumes that we can generate a number at random using a cryptographically secure pseudorandom number generator. We also assume that every ballot has a unique number called a ballot sequence number (BSN).

**Protocol**

Prior to the election, a distributed key generation protocol will be used in order to generate the secret ‘a’ value for an elgamal encryption scheme. This will also generate the public key [*P* (chosen prime), *G* (chosen primitive root), *B* ()] for ElGamal encryption that will later be used by the DRE to encrypt the ballots. The point of using the distributed key generation protocol is that it will take a select chosen number of people from to retrieve the secret *‘a’* value used to decrypt the ballots.

Assume we have candidates. Once the voter selects to vote for candidate , the DRE constructs a verifiable choice (VC). A VC is essentially an encrypted vote, which is an table where *L* is a security parameter that is determined prior to the election. The *L* parameter is significant for the zero-knowledge proof that will be used for the ballot verification, where Neff suggests 10 <= *L* <= 15. Each cell in the VC contains a ballot marked pair (BMP), where a BMP consists of two random bits *B1, B2*. Theirencrypted values [*B1], [B2]* are encrypted with the master public key with a random K value. Although *B1, B2* are selected at random they must follow certain conditions determined by the row that they fall in. In the row that candidate represents each BMP will have and every other unchosen candidate will have a row of BMP’s that consist of , this is what separates a chosen candidate from the unchosen one’s. Figure 1 represents a VC where we have three candidates, is chosen, and L = 3.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | 2 | 3 |
| Candidate 1 | ([0],[1]) | ([1],[0]) | ([1],[0]) |
| Candidate 2 | ([0],[0]) | ([1],[1]) | ([1],[1]) |
| Candidate 3 | ([1],[0]) | ([0],[1]) | ([1],[0]) |

*Figure 1: Example of a* Verifiable *Choice where the number of candidates = 3, and L = 3*

Once the VC is created the DRE outputs the voters receipt. The receipt contains the BSN, Hash(VM), and *L* pledge bits. These pledge bits represent the value used to encrypt *B1, B2* in each BMP in the row *Ci*. These bits are used for the interactive zero-knowledge protocol, that confirms with a high probability that the vote was indeed submitted correctly.

When the voter collects the receipt, he has a choice to verify his vote. To confirm his vote was cast correctly, he would provide the DRE with *L* number of challenges. The challenges consist of the voter picking the left or right bit to verify, this is done to all the BMP’s of the chosen candidate’s row. Because the DRE was the one to do the encryption it knows what K value was used to generate each ciphertext. For example, using the same VC in figure 1 we can see that voter has selected candidate 2. The DRE would then print out the pledge bits 0,1,1. Let’s say the voter submits left, left, right as his challenges, the DRE will provide the *K* value used to encrypt the left bit of BMP1, left bit of BMP2, and the right bit of BMP3. Knowing the *K* value allows you to encrypt the pledge bits yourself to ensure that you get the same ciphertext, thus confirming that the pledge bit is indeed the bit encrypted for that part of the BMP. If the DRE was malicious, or changing the vote in any way, the user would have a ½ chance to detect that every time they do a challenge against the DRE for their vote. Thus, the user can be certain with a probability of that the ballot was generated correctly.

After the election, at least the minimum number of trustees come together and reveal the private ElGamal key that was generated prior to the election. The secret ElGamal key is used to decrypt all the ballots and count the votes.

**Possible Attacks**

A couple attacks show flaws in Neff’s scheme. One such attack involves a malicious DRE submitting fake ballots under valid BSN’s. This type of attack could be detected because you would have two votes under the same BSN, however, officials would be unable to determine the fake vote from the real vote.

Another possible attack involves the malicious DRE attempting to trick the voter to give away the secret challenges. Suppose a voter just finished selecting his challenges, the DRE can then reboot pretending to be affected by an error. After the reboot the DRE can forge a valid ballot voting for another candidate, in the forgery ballot the bits for the voters intended candidate would match up to his previous challenge selections thus defeating the zero-knowledge proof. If the voter selects a new challenge that is different than his previous one, then the DRE could once again reboot and act normally.

**Conclusion**

Neff’s scheme is very interesting as it implements distributed key generation and a zero-knowledge proof. But perhaps the most thought-provoking idea is how the scheme highlights the semantic security property of ElGamal, the scheme would have been impossible if it was done with RSA. RSA has deterministic encryption, so it would be easy to tell in the VC which bits decrypt to 0 and to 1 thus defeating the purpose of the encryption.

Overall Neff’s scheme shows us many of the challenges presented when voting in an electronic environment.

**Comments**

We have submitted this document with a ruby command line implementation that will highlight the main functionality of this scheme. It will allow a user to run a mock election for different candidates, encrypt the ballots, allow the user to verify the ballot was cast correctly, and then decrypt the votes and count them in the end. Installation is quite straight forward for ruby and the required library, just follow the readme.