On Electronic Voting Schemes

Ivan Damgård

CPT 2006

1 Introduction

This note describes protocols for electronic voting schemes, i.e., ways to hold an election by only exchanging digital messages. We do not give any formal definition of what secure voting is, this is postponed to later in the course. Intuitively, however, the scenario is that we have a number of voters $V_1, V_2, ..$ who wish to participate in an election. We want to find the election result in such a way that the correct result and *only* the correct result is made public, and such that only eligible voters can actually vote.

A very simplistic solution to this would be the following: each voter sends his vote to a central authority over a secure line, say an SSL connection. The central authority remembers who has voted and keeps a running total of the votes cast so far. When the vote is over, the authority publishes the result.

This solution requires complete trust in the central authority, and even if we believe that the organization behind it has honorable intentions, a successful hack on the machine receiving votes will destroy all privacy, and possibly correctness as well.

Hence, for elections of real importance, something better is required. To describe how we improve security, we will assume that communication takes place in the so-called Bulletin Board model, that is, we assume a public Bulletin Board BB, that can be seen by all players. Each player can post messages to BB, and it can always be seen on the BB who posted what. This can be implemented in practice by using digital signatures and possibly database replication techniques to ensure availability of the BB. Some systems might instead use a central machine with reliable logging mechanisms to implement the BB. This machine receives all messages that are posted and will return posted messages on request.

In any case, the important point is that the BB handles only public information, contrary to the central authority in the simplistic solution.

2 El-Gamal Encryption

El Gamal encryption is a central tool in our solutions. Suppose we have a group G_q of large prime order q, and a generator g of G_q. Here G_q could be a subgroup of Z_p^*, where p is a prime with $p = 2q + 1$.[1] Then the El-Gamal cryptosystem has secret key s chosen at random from Z_q, and public key $h = g^s$.

To encrypt a message $m \in G_q$, compute

$$(c, d) = (g^r, mh^r) \ ,$$

where r is random in Z_q. To decrypt, compute $c^{-s}d = m$. We will assume in the following that this cryptosystem is semantically secure, i.e., from a ciphertext, no information on m can be efficiently computed.

A central observation is that this system can also be used to encrypt numbers in Z_q, say 0 or 1. To encrypt such a number b, compute

$$E_h(b, r) = (g^r, g^b h^r) \ .$$

That is, we El-Gamal encrypt g^b. We can decrypt by using the El-Gamal decryption, which will give us g^b. If b could be arbitrary, we would have to solve discrete logarithms to find b, but if we know in advance that b is 0 or 1, we just test if g^b equals g^1 or g^0. The same idea will also work if we know that b is among a small set of possibilities.

This new system has a very nice property, it is *homomorphic* in the following sense: define a product on ciphertexts as $(c, d) \cdot (c', d') = (cc', dd')$. Then it is easy to see that

$$E_h(b, r) \cdot E_h(b', r') = E_h(b + b' \bmod q, r + r' \bmod q) \ .$$

That is, by multiplying together two ciphertexts, we get an encryption of the sum of the two plaintexts b, b', *but without having to decrypt* b, b', and without having to know the secret key.

3 A Simple-Minded Voting Protocol

The modified El-Gamal encryption immediately leads to a simple voting protocol that will work, assuming a large (and not too realistic) amount of trust in participants. We later remove the need for such trust.

[1] Any sub-group of Z_p^* has an order dividing $p - 1 = 2q$ and therefore has order 1, 2, q or $2q$. Therefore the sub-group Q_p of quadratic residues can be seen to have order q. Furthermore, $4 = 2^2$ can be seen to generate Q_p. We can therefore set $G_q = Q_q$ and set $g = 4$ in this case.

Assume a trusted third party, TTP, who first generates and posts an El-Gamal public key h and keeps the secret key s for himself.

Let us assume that we want to execute a yes/no vote, and fix the rule that the number 0 means NO and 1 means YES. Thus every voter V_i has a number v_i that represents how he wants to vote. Now, each V_i posts $E_h(v_i, r_i)$. The TTP computes

$$\prod_i E_h(v_i, r_i) = E_h(\sum_i v_i \bmod q, \sum_i r_i \bmod q) \ ,$$

where the product is over contributions from eligible voters, and decrypts the result. This result will be $g^{\sum_i v_i \bmod q}$. Say we have m voters. Now m will be much smaller than q, typical values might be $m =$ a few thousand and $q \simeq 2^{160}$. So hence $\sum_i v_i \bmod q = \sum_i v_i$. Moreover, we can find $\sum_i v_i$ from $g^{\sum_i v_i}$, just by trying all possibilities. Clearly, this sum will be exactly the number of yes votes.

If all V_i and TTP are completely honest, this is secure: the correct result is computed, and nothing else is revealed as long as the encryption is secure.

Note that, despite the large amount of trust involved, this system is still a large improvement over the simplistic system from the intro: all information sent during the period where votes are cast is public and so does not need to be protected for privacy. The TTP could be a secure hardware box that is not even connected to the system before the time comes to decrypt the result.

4 Removing Trust in Voters

Clearly, a dishonest voter can mess up the result by sending an encryption of some number other than 0 or 1. To remove this problem, we require each voter to prove that he posts a correctly constructed encryption. For this, observe that if the ciphertext is (c, d), then if $(c, d) = E_h(0, r)$, we have $(c, d) = (g^r, h^r)$, put another way, the discrete log of c base g equals the discrete log of d base h. We write this as $\log_g(c) = \log_h(d)$. On the other hand, if $(c, d) = E_h(1, r)$, we would have $(c, d) = (g^r, gh^r)$, i.e., $\log_g(c) = \log_h(g^{-1}d)$.

In the note on Σ-protocols, we have seen a Σ-protocol for proving equality of discrete logs, which is exactly a statement of form $\log_g(c) = \log_h(d)$. Hence, from the standard or-construction, we get immediately a Σ-protocol for proving that either $\log_g(c) = \log_h(d)$ or $\log_g(c) =$

$\log_h(g^{-1}d)$. By the arguments above, this is equivalent to proving that the encryption contains 0 or 1.

This Σ-protocol can be made non-interactive using the Fiat-Shamir heuristic as we have seen before in the course, so this means that a voter can simply post his encrypted vote together with a proof that it was correctly formed.

More concretely, that way this works is as follows: Assume the ciphertext in question is (c, d) and the first message in the Σ-protocol is a. Then the voter computes $e = H(c, d, a, ID)$, where H is a hash function and ID is the voters own name. This e is used as the challenge in the Σ-protocol, so the voter computes the correct answer z to challenge e. He can then post (c, d) together with the proof a, z. To check this, a verifier will look up the identity ID of the voter who posted ciphertext and proof. This is always possible by assumption on the bulletin board. He can then compute the hash value e and check that (a, e, z) is an accepting conversation in the Σ-protocol. As mentioned in the note on Σ protocols this can be shown to be sound and zero-knowledge if we model H by a random oracle. This is not a full proof of security in the real world, but a good indication that the construction is OK in practice.

Why do we need to include the identity of the voter in the hashing? if we didn't, it would be possible for a voter V_1 to wait until someone else V_2 posts his ciphertext and proof and then post exactly the same information. This makes it possible for V_1 to always vote in the same way as V_2. This is not a desirable property, in particular it is not something one could do in an ordinary paper based election. Including the ID of the voter in the hashing prevents this problem.

5 Removing trust in the TTP

Clearly, the TTP forms a single point of attack, and if it would misbehave, it could start decrypting single votes. To avoid this, we can replace the TTP by a set of authorities $A_1, ..., A_n$.

We will equip the A_i's with some pieces of public and private information which will allow them to securely decrypt ciphertexts assuming that at most t of them will misbehave, where we let $t = \lfloor (n-1)/2 \rfloor$, i.e., t is maximal such that $t < n/2$. Here, securely means that if the honest players agree to execute a decryption, this will always succeed and no other information than the desired plaintext will be released. Furthermore, decryption will require participation of at least $t + 1$ players, so no ciphertext can be decrypted by the dishonest players alone.

The information we need is as follows: Each authority A_i knows a private value $s_i \in Z_q$, where $s_i = f(i)$ and where $f()$ is a random polynomial over Z_q of degree at most t chosen such that $f(0) = s$. Finally, for each A_i, the value $h_i = g^{s_i}$ is publicly known. For now, we will assume that there is a trusted dealer TD who initially generates and distributes this information privately and reliably. Note that this includes generating the public and private El-Gamal keys. Of course, this means we have not yet completely solved the problem with a single point of attack, but we have achieved that this single point only needs to exist for a limited amount of time when the system is set up. We remove also the TD later.

Some basic facts about this secret sharing of s: from any set of at most t of the s_i's, no information at all is revealed about s. However, from any set of at least $t + 1$ values, it is easy to compute s by taking a linear combination of the values. For instance, if we are given $s_1, ..., s_{t+1}$, there exist (easily computed) coefficients λ_i such that $s = \sum_i \lambda_i s_i \bmod q$.[2]

This can be used to decrypt a ciphertext (c, d) as follows: to decrypt, it is enough to compute c^s. To this end, each A_i posts $c_i = c^{s_i}$. He furthermore posts a proof that this is correct, namely a proof that $\log_c(c_i) = \log_g(h_i)$, constructed in a similar way as for the voters above.

Now, anyone can compute c^s: contributions with bad proofs are rejected, but we know that at least $t+1$ correct contributions will be present, since $t < n/2$. Say $c_1, ..., c_{t+1}$ were OK. Then we compute

$$\prod_i c_i^{\lambda_i} = c^{\sum_i s_i \lambda_i \bmod q} = c^s$$

which was what we needed.

Note that this is secure even if up to t of the A_i would go together to try to cheat the rest: they cannot find s so they will not be able to decrypt anything. One might object that there could still be a problem: the cheating players know t of the s_i's and each time a ciphertext is decrypted, they learn the c_i's posted by the honest players. Maybe all this together will reveal information that could be useful for an adversary? However, it can be shown that this is not the case: the information seen by the adversary can be *simulated* based on public information already known by the cheating players. This can be done using techniques similar to those in the final section of this note.

[2] Knowing $s_i = f(i)$ for $t+1$ distinct values of i we know $t+1$ points on a polynomial of degree at most t, which uniquely determines f, and thereby uniquely determines $s = f(0)$. The formula for computing s from the s_i follows from the theory of Lagrange interpolation, and this formula turns out to be linear. For more details, see the document on secret sharing, found on the CPT home-page.

6 Removing the Trusted Dealer

In this last part we give a way to generate keys for the El Gamal cryptosystem that can be executed by several players in such a way that afterwards, the players share the secret key, but no minority of them has enough information to compute the key. Furthermore, we establish a situation such that any honest majority of the players can afterwards use the secret key to decrypt a ciphertext.

6.1 The Goal

The previous sections start from the following scenario: we have authorities $A_1, ..., A_n$, and a large group G_q of prime order q generated by element g (for instance a subgroup of Z_p^*, where q divides $p-1$). We let $t = \lfloor (n-1)/2 \rfloor$, i.e., t is maximal such that $t < n/2$.

Furthermore $h = g^s$ is publicly known, where h is the public El-Gamal key.

As mentioned, this scenario can be set up by assuming that a trusted party does it once and for all by simply choosing s and $f()$ at random and compute the s_i, h_i's and sending s_i to A_i in private. This may be acceptable in some cases, but nevertheless creates (for a limited amount of time) a single entity that must be trusted completely, since the trusted party knows the secret key. Here, we look at a way to have the players create the keys from scratch.

6.2 A Key Generation Protocol

We note that the proposal we give here is not complete in the sense that more steps and complications in the protocol would be needed to construct a protocol that could be formally proved to be "as good" as when a single trusted party generates the keys. What we show here is only intended as an example to demonstrate some basic ideas. We will assume that the t dishonest players may try to misbehave and do as much damage as possible by sharing their information and coordinating their actions. An easy equivalent way to think of this is to assume a single adversary who has corrupted up to t of the players, has seen all their information and now control completely their actions.

The first basic step is the following:

– Each A_i chooses a random polynomial $f_i()$ of degree at most t, $f_i(x) = v_i + a_{i1}x + ... + a_{it}x^t$. (we give the degree 0 coefficient a special name

v_i as it plays a special role in the following). He publishes $g^{v_i}, ..., g^{a_{it}}$. He sends in private $f_i(j)$ to each A_j, $j = 1..n$.

The values $g^{v_i}, g^{a_{i1}}..., g^{a_{it}}$ serve as a sort of commitment to the coefficients of $f_i()$. Note that $g^{v_i} = g^{f_i(0)}$. Note also that having seen these values, everyone can compute $g^{f_i(j)}$ for any i, j, as follows:

$$g^{f_i(j)} = g^{v_i + a_{i1} j + ... + a_{it} j^t} = g^{v_i} \cdot (g^{a_{i1}})^j \cdots (g^{a_{it}})^{j^t}$$

Therefore, we can do the next step:

– Each A_j looks at the value u_i he received from A_i (which hopefully equals $f_i(j)$). He computes $g^{f_i(j)}$ from the public information as above and compares this to g^{u_i}. If there is a mismatch, he posts a public complaint, and now A_i must make $f_i(j)$ public. If this value is correct, A_j uses this value in the following, otherwise A_i has obviously misbehaved and is disqualified.

It may seem strange that the protocol may require a value to be made public that should otherwise be secret. But the point is the following: if A_j complains in the above step, then clearly either A_i or A_j (or both) is corrupt. If A_j is corrupt, then the adversary already knows $f_i(j)$ from Step 1 and and it doesn't hurt to make it public. If A_i is corrupt, the adversary has chosen $f_i(j)$ itself, and again he learns nothing new. Furthermore, if A_j complains and A_i is honest, then clearly A_i is not disqualified. I.e. only corrupt parties are disqualified.

What we have obtained is that now every player A_j knows correct values $f_i(j)$ for all i, except those that have been disqualified and can now be ignored. For simplicity, we will assume that no one was disqualified.[3] Then we do the following:

– Each A_j computes $s_j = f_1(j) + f_2(j) + ... + f_n(j)$. We set the public key to be $h = g^{f_1(0)} \cdot g^{f_2(0)} \cdots g^{f_n(0)} = g^{v_1 + ... + v_n}$. And we define $h_j = g^{f_1(j)} \cdots g^{f_n(j)}$. Note that everyone can compute h and the h_j's from the information made public earlier.

Define the polynomial $f()$ by $f() = f_1() + ... + f_n()$, and $s = f(0)$, then $s = f_1(0) + ... + f_n(0) = v_1 + ... + v_n$, and in general $f(j) = f_1(j) + ... + f_n(j)$. We then have the situation we wanted: we have a public key h and a corresponding secret key s such that $h = g^s$. We have a polynomial $f()$

[3] If some party A_i was disqualified, the remaining parties can just define the values that A_i should have contributed to be some dummy, fixed values of the correct form.

such that $f(0) = s$, and each A_j knows $s_j = f_1(j) + f_2(j) + ... + f_n(j) = f(j)$. Finally we have $h_j = g^{f_1(j)} \cdots g^{f_n(j)} = g^{f(j)}$.

How secure is this protocol? We argued underway that it is guaranteed to produce values s_j, g^{s_j} for each A_j that are *correct*, i.e., there really is a polynomial $f()$ of degree at most t that is consistent with all these values. Therefore, decryption using this sharing of the secret key is guaranteed to work correctly. The remaining question is: how much can the adversary learn from the protocol about the secret key? If he can find out too much, encryption using the public key we created may not be secure. In the following, we argue that the protocol is good enough in practice in this respect, but that it cannot be formally proved to be as good as in the case where a single trusted party chooses the keys. Reaching this goal requires more complex constructions. The analysis below of these questions is not mandatory reading, but is included for those interested.

How secret is the secret key? Consider the situation where an honest A_i executes Step 1, and suppose the adversary has corrupted/broken into a subset S consisting of t A_j's. The adversary sees $g^{v_i}, g^{a_{i1}}, ..., g^{a_{it}}$ plus the values $\{f_i(j)|\ j \in S\}$. Suppose for a minute that we also knew v_i. Then it is a consequence of Lagrange interpolation that one can compute any of the coefficients a_{il} of $f_i()$ by a linear combination, i.e., there exist ν_0 and $\{\nu_j|\ j \in S\}$ such that

$$a_{il} = \nu_0 v_i + \sum_{j \in S} \nu_j f_i(j) \ .$$

The adversary can of course not do this computation, since he doesn't know $v_i = f_i(0)$, but he *can* compute ν_0 and the ν_j's. This means that knowing g^{v_i} and $f_i(j), j \in S$ he can in fact compute the following:

$$g^{a_{il}} = (g^{v_i})^{\nu_0} \cdot \prod_{j \in S} (g^{f_i(j)})^{\nu_j} \ .$$

Therefore, once the adversary has seen g^{v_i} and $\{f_i(j)|\ j \in S\}$, it makes no difference for him whether any of the $g^{a_{il}}$ are made public, since he could compute them himself anyway. Furthermore, since $f_i()$ has been randomly chosen of degree at most t, the values $\{f_i(j)|\ j \in S\}$ are uniformly random and independent values in Z_q. This is because there are only t of them, and it is a basic property of Shamir's secret sharing scheme that as long as only t shares are given, these contain no information about the secret (the value of the polynomial in 0), so in particular, they are independent of g^{v_i}. Hence, it makes no difference from the adversary's point of view whether the values $\{f_i(j)|\ j \in S\}$ are made public by A_i or not. He might as well choose t random values himself and this would result in a distribution perfectly indistinguishable from what he sees in real life.

This means that from the point of view of what the adversary learns, we can replace Step 1 by:

- Each A_i chooses a random polynomial $f_i()$ of degree at most t, $f_i(x) = v_i + a_{i1}x + ... + a_{it}x^t$. He publishes g^{v_i}.

We conclude that the way our protocol chooses the public and secret key is equivalent to the following:

- Each honest player A_i chooses at random a value v_i and publishes g^{v_i}. Having seen these values, the adversary must choose values v_j, g^{v_j} on behalf of the corrupted players (he may also choose to have some of them disqualified). Finally, the secret key becomes $s = v_1 + ...v_n$ and the public key is $h = g^{v_1} \cdots g^{v_n}$.

The goal of the adversary is of course to choose his own values v_j in a malicious way such that the sum s becomes easier for him to guess. Of course, if he knew the contributions v_i from the honest players, he could force s to be whatever value he wanted. But in fact the v_i's are random and he only knows the g^{v_i}'s. If discrete log is a hard problem (and we must of course assume this anyway), the adversary cannot compute any of the v_i's and it is therefore reasonable to assume that he cannot choose the v_j's such that they depend on the v_i's in a way that would be useful for him. So it would not be unreasonable to use the protocol in practice.

However, since g^{v_i} does determine v_i uniquely, we *cannot* prove that the v_j's are independent of the v_i's and so it is not clear that s will be a uniformly random value - as it would have been if a trusted party had chosen s. Solving this technical problem requires addition of more steps to the protocol. For details on this, see the paper by Gennaro et al. from EuroCrypt 2000: *Secure Distributed Key Generation for Discrete Log based Cryptosystems*, Springer Verlag LNCS series 1592.

7 Exercises

Exercise 1. The above sections described how to hold a simple yes/no election, revealing only the number of yes-votes. This is appropriate for holding e.g. a referendum. Assume, however, that instead we want to hold an election for parliament. Assume in particular that C candidates are given and that each voter is allowed to vote for exactly one candidate, and assume that we want to reveal the number of votes for each candidate — and nothing else. The straight-forward way to do this is to hold C yes/no elections in parallel. A voter who wants to vote for candidate i then votes yes in election i and votes no in all other elections. The number of yes-votes in election i is then the number of votes for candidate i. There is however an obvious way to cheat in this system. A dedicated voter can cast a vote on *all* the candidates that she likes. We need a way to ensure that each voter votes yes in exactly one of the C elections.

Assume in particular that $(c_i, d_i) = E_h(v_i, r_i) = (g^{r_i}, g^{v_i} h^{r_i})$ is the encryption of the vote v_i cast by the voter in election i. We then need a way to ensure that among the votes $v_1, \ldots, v_C$, exactly one equals 1 (this is sufficient as the individual elections already ensures that each $v_i \in \{0, 1\}$).

Find a way for the voter to prove that exactly one of the v_i's is 1. [Hint: consider the values $c = \prod_{i=1}^{C} c_i \pmod{G_q}$, $d = \prod_{i=1}^{C} d_i \pmod{G_q}$, $v = \sum_{i=1}^{C} v_i$, $r = \sum_{i=1}^{C} r_i \bmod q$ and observe that $(c, d) = E_h(v, r)$.]

Exercise 2. In the above voting system each voter can vote for exactly one candidate. How do you hold an election where each voter is allowed to vote for exactly V different candidates? [Hint: (c, dg^{-V}).]

Exercise 3. The above voting system forces the voter to vote for exactly V candidates. How do you hold an election where each voter is allowed to vote for *at most* V different candidates? I.e. the voter is allow to vote for either no candidate, or one candidate, or two different candidates, and so on, but for at most V different candidates.