Efficient Maximal Privacy in Boardroom Voting and Anonymous Broadcast

Jens Groth[1,2]

[1] BRICS*, University of Aarhus, Ny Munkegade bd. 540, 8000 Århus C, Denmark
[2] Cryptomathic A/S**, Jægergårdsgade 118, 8000 Århus C, Denmark
jg@brics.dk

Abstract. Most voting schemes rely on a number of authorities. If too many of these authorities are dishonest then voter privacy may be violated. To give stronger guarantees of voter privacy Kiayias and Yung [1] introduced the concept of elections with perfect ballot secrecy. In this type of election scheme it is guaranteed that the only thing revealed about voters' choices is the result of the election, no matter how many parties are corrupt. Our first contribution is to suggest a simple voting scheme with perfect ballot secrecy that is more efficient than [1].

Considering the question of achieving maximal privacy in other protocols, we look at anonymous broadcast. We suggest the notion of perfect message secrecy; meaning that nothing is revealed about who sent which message, no matter how many parties are corrupt. Our second contribution is an anonymous broadcast channel with perfect message secrecy built on top of a broadcast channel.

1 Introduction

Voting schemes are legion in the cryptographic literature. Common for most of them is that they rely on some authorities to conduct the election. Furthermore, if a large group of authorities is dishonest then individual votes may be revealed. To some extend this is unavoidable, some degree of privacy violation is inherent in any election; a group of voters may subtract their own votes from the result and thereby obtain some information about the remaining voters' choice. In terms of privacy, the best we can hope for is to ensure that nobody can deduce more about the distribution of honest voters' votes than what can be deduced from the result and knowledge of dishonest voters' choices. We call this type of security perfect ballot secrecy.

Kiayias and Yung [1] introduced the notion of perfect ballot secrecy together with self-tallying and dispute-freeness. Self-tallying means there is no need for authorities to tally the votes. Once all votes have been cast, the result can be tallied and verified by anybody. Dispute-freeness says that anybody may

* Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

** www.cryptomathic.com

verify that indeed the parties do follow the protocol. In other words, it is public knowledge whether a party performed correctly or tried to cheat.

Kiayias and Yung [1] presented a self-tallying dispute-free voting scheme with perfect ballot secrecy with security based on the Decisional Diffie-Hellman (DDH) assumption. Later Damgård and Jurik [2] suggested a somewhat similar scheme based on the Decisional Composite Residuosity (DCR) assumption [3]. Both schemes work in the random oracle model and assume an authenticated broadcast channel; in the present paper, we use this model too.

Kiayias and Yung [1, 4, 5] rely on a method they call zero-sharing for achieving maximal privacy. Not only do they build a voting protocol from this, but they also suggest protocols for anonymous vetoing and simultaneous disclosure of secrets.

Our contributions. Our first contribution is a new voting scheme that has the same security properties as [1, 2] but is simpler and more efficient. We base our scheme on the DDH assumption, i.e., ElGamal encryption, but the same ideas can be used in combination with the DCR assumption. The reason for this choice is that it is easy to generate in a distributed manner suitable groups where the DDH assumption is well founded. Distributed generation of a suitable group for the DCR assumption is more complicated [6].

Our second contribution is to construct an anonymous broadcast channel with perfect message secrecy, i.e., no matter which parties are dishonest, they are not able to tell among the honest senders who sent a particular message. This scheme is related to voting in the sense that using this anonymous channel to cast votes gives us a self-tallying voting scheme with perfect ballot secrecy, but it may of course also have many other applications.

1.1 Model

Throughout the paper, we assume all parties have access to an authenticated broadcast channel with memory. We imagine this in the form of a message board that all parties can access. Each party has a special designated area where he, and nobody else, can write. No party can delete any messages from the message board. One way of implementing such a message board would be to have a central server on the Internet handling the messages. We discuss this further in Section 4.

When considering security of the protocols we imagine that there is an active polynomial time adversary $\mathcal{A}$ trying to break them. $\mathcal{A}$ is static, i.e., from the beginning of the protocol it has control over a fixed set of parties.

The parties in the protocol work semi-synchronously; the protocol proceeds in phases and in each phase parties may act in random order. We let the adversary decide when to change to the next phase. Since the protocols we design are intended for use with a small number of participants, we find this to be a reasonable assumption. Should several parties by accident happen to execute their action at the same time anyway, then it is quite easy to recover.

2 Self-tallying Voting Scheme with Perfect Ballot Secrecy

2.1 Security Definitions

The requirements we want the voting scheme to satisfy are the following.

Perfect ballot secrecy: This is an extension of the usual privacy requirement. In a voting scheme with perfect ballot secrecy the partial tally of a group of voters is only accessible to a coalition consisting of *all* remaining voters. This is the best type of anonymity we can hope for in elections where we publish the result, since a coalition of voters may of course always subtract their own votes.

Self-tallying: After all votes have been cast, it is possible for anybody, both voters and third parties, to compute the result.

Fairness: Nobody has access to a partial tally before the deadline. We will interpret this demand in a relaxed way such that it is guaranteed by a hopefully honest authority.

Dispute-freeness: This notion extends universal verifiability. A scheme is dispute-free if everybody can check whether voters act according to the protocol or not. In particular, this means that the result is publicly verifiable.

2.2 The voting protocol

The basic idea. To quickly describe our idea let us use an analogue with the physical world. Assume a group of people want to vote yes or no to a proposal. To do this the voters take a box with a small slot and each voter puts a padlock on the box. Taking turns the voters one by one drop a white (yes) stone or a black (no) stone into the box and remove their padlock. When the last voter has removed his padlock, they may open the box and see the result of the election. The protocol has perfect ballot secrecy since the box cannot be opened before all honest voters have cast their vote, and thus any honest voter's vote is mixed in with the rest of the honest voters' votes.

Overview of the protocol. For simplicity, we first describe the protocol in the honest-but-curious setting, i.e., corrupted voters may leak information but follow the protocol. For simplicity, we also assume there are just two candidates that the voters can choose between.

Initialization: First, the voters agree on a group G_q of order q where the DDH problem is hard. Let g be a generator for G_q.
All voters now select at random an element in $\mathbb{Z}_q$. Each voter j keeps his element x_j secret but publishes $h_j = g^{x_j}$.

Casting votes: Voters may vote in any adaptively chosen order, however, for simplicity we assume in this example that they vote in the order $1, 2, \ldots, n$. Let their choices be $v_1, v_2, \ldots, v_n \in \{0, 1\}$.
The election now proceeds like this:

1. Voter 1 selects at random $r_1 \in \mathbb{Z}_q$ and publishes $(g^{r_1}, (\prod_{i=2}^{n} h_i)^{r_1} g^{v_1})$.

2. Voter 2 selects at random $r_2 \in \mathbb{Z}_q$ and computes $(g^{r_2}, (\prod_{i=2}^n h_i)^{r_2} g^{v_2})$. Multiplying this to the first vote, he gets $(g^{r_1+r_2}, (\prod_{i=2}^n h_i)^{r_1+r_2} g^{v_1+v_2})$. With his knowledge of the secret key x_2, he may peel of a layer of this ElGamal encryption of the partial result. In other words, he computes $(g^{r_1+r_2}, (\prod_{i=3}^n h_i)^{r_1+r_2} g^{v_1+v_2})$. He publishes this on the message board.
3. Voter 3 performs the same type of operations as voter 2. He ends up publishing $(g^{r_1+r_2+r_3}, (\prod_{i=4}^n h_i)^{r_1+r_2+r_3} g^{v_1+v_2+v_3})$ on the message board.

$\vdots$

n. Voter n performs the same type of operations as the previous voters. When he is done, his output is $(g^{\sum_{i=1}^n r_i}, g^{\sum_{i=1}^n v_i})$.

Tallying: From the last voter's output we can read off $g^{\sum_{i=1}^n v_i}$. We compute the discrete logarithm, this is possible since the exponent is at most n, to get $\sum_{i=1}^n v_i$. This is the number of 1-votes in the election.

The full protocol. The protocol as described is not fair, it is possible for the last voter to know the result before casting his own vote. As in [1] we deal with this by saying that a special election authority must act like a voter and cast a zero-vote in the end. Since it is a zero-vote, it does not affect the result. On the other hand, the perfect ballot secrecy of the voting scheme ensures that up to this point nobody but the authority can know any partial tally. Therefore, if the authority is honest then the voting scheme is fair.

To go beyond the honest-but-curious assumption and deal with all kinds of adversaries all we have to do is to add zero-knowledge proofs of knowledge of correctness. These proofs will be the typical 3-move honest verifier proofs (Σ-protocols [7]), where using the Fiat-Shamir heuristic we can make very efficient non-interactive zero-knowledge proofs. Security of the protocol will be proved in the random oracle model [8].

We wish to support a set W of possible votes. Let us write the c candidates in W as candidates as $0, \ldots, c-1$. We do this by encoding candidate number i as $(n+1)^i$. From a sum $\sum_{i=1}^n v_i$ of votes with this encoding we can read off the number of votes on each candidate. To compute the result, we have to compute the discrete logarithm of $g^{\sum_{i=1}^n v_i}$. With n voters and c candidates, the number of possible results is $\binom{c+n-1}{c-1}$. With a small number of voters or a small number of candidates, it is possible to compute the discrete logarithm. If we have a larger number of voters and candidates, we may use a cryptosystem similar to the one in [2]. This allows computing discrete logarithms efficiently, but on the other hand the key generation becomes much more complicated. Alternatively, we may use the anonymous broadcast protocol we present in the next section.

The full protocol can be seen in Figure 1.

Performance. Let n be the number of voters, c be the number of candidates, and k be the security parameter. We assume that $n^c \leq q$.

For each voter it takes $\mathcal{O}(1)$ exponentiations to compute the key h_i and the associated proof. The size of the key is $\mathcal{O}(k)$. Verification of the n keys takes $\mathcal{O}(n)$ exponentiations.

Voting Protocol

Setup: The voters agree on a suitable group G_q of order q where the DDH assumption holds. Let g be a generator for G_q.

Key Registration: Voter i selects at random $x_i \in \mathbb{Z}_q$ and sets $h_i = g^{x_i}$. He publishes h_i and makes a proof of knowledge of x_i, $\mathrm{PK}[x_i : h_i = g^{x_i}]$.
Any voters who did not supply a key are removed from the list of eligible voters. Set the current state of the election to be $(1, 1)$.

Voting: Voter i wishing to cast a vote $v_i \in W$ downloads the current state of the election (u, v), and verifies the correctness of keys and all votes cast up to now. Then he selects r_i at random from $\mathbb{Z}_q$, sets $U = ug^{r_i}$ and
$V = vu^{-x_i}(\prod_{j \in T} h_j)^{r_i} g^{v_i}$, where T is the set of remaining voters. He broadcasts (U, V) as the new state of the election together with a zero knowledge proof of knowledge
$\mathrm{PK}[(r_i, v_i, x_i) : h_i = g^{x_i} \wedge U = ug^{r_i} \wedge V = vu^{-x_i}(\prod_{j \in T} h_j)^{r_i} g^{v_i} \wedge v_i \in W]$, i.e., a proof that he knows r_i, v_i, x_i making his vote correct.

Tallying: After all voters have cast their votes the state (u, v) has $v = g^{\sum_{i=1}^{n} v_i}$. The discrete logarithm $\sum_{i=1}^{n} v_i$ can be computed if there are not too many voters and candidates, and from this the result can be extracted.

Fault-correction: If some voters abstained from voting it is possible that the remaining voters still want to carry out the election. In that case, they can repeat the voting step with the now reduced set of voters. They may gain a factor $\log c$ in efficiency by proving that they cast the same vote as in the first voting phase instead of proving from scratch that the vote belongs to W.

Fig. 1. The voting protocol

In the voting phase, it takes $\mathcal{O}(\log c)$ exponentiations to compute the vote and the proof associated with it.[3] The vote has size $\mathcal{O}(k \log c)$. It takes $\mathcal{O}(n \log c)$ exponentiations to verify all the voters' proofs.

In comparison, the protocol in [1] lets the voter do $\mathcal{O}(n)$ exponentiations in the key registration phase, the key has size $\mathcal{O}(nk)$, and verification of the keys takes $\mathcal{O}(n^2)$ exponentiations. In the voting phase, the voter must do $\mathcal{O}(\log c)$ exponentiations, the vote has size $\mathcal{O}(k \log c)$, and it takes $\mathcal{O}(n \log c)$ exponentiations to verify all the votes.

The Kiayias and Yung protocol does have the advantage that many voters can vote at the same time, whereas we demand that they download the current

[3] Let us sketch where the $\log c$ factor comes from. In the proof of correctness of a vote the voter has to argue that the encrypted vote is on the form $(1+n)^i$ for $i \in \{0, \ldots, c-1\}$. Let $\{b_1, \ldots, b_{\lceil \log c \rceil}\}$ be a set of positive integers with the following property: for any number $1, \ldots, c-1$ there is a subset where the numbers have this sum, and for no number larger than $c-1$ is there a subset with elements having this sum. Write the vote as $(1+n)^v = (1+n)^{\sum_{i=1}^{\lceil \log c \rceil} a_i} = \prod_{i=1}^{\lceil \log c \rceil} (1+n)^{a_i}$, where $a_1 = b_1 \vee a_1 = 0, \ldots, a_{\lceil \log c \rceil} = b_{\lceil \log c \rceil} \vee a_{\lceil \log c \rceil} = 0$. This shows that the vote can be built as a product of $\lceil \log c \rceil$ elements. It is possible to prove correctness of such elements and make proofs of products in $\mathcal{O}(1)$ exponentiations, giving a total of $\mathcal{O}(\log c)$ exponentiations.

state and use that in making their vote. Since the voting protocols are designed for self-tallying and demand that all voters participate we can only see them as being realistic in settings with few voters though. With few voters, we believe it is reasonable to assume that voters act one at a time; and even if they occasionally do not it is easy to correct.

2.3 Security.

To argue perfect ballot secrecy of the voting protocol in Figure 1 we will show that a real-life execution of the protocol can be simulated with knowledge of the sum of the honest voters' votes only. To do so we define two experiments, a real-life experiment, and a simulation experiment.

Real-life experiment. In the real-life experiment the voters $V_1, \ldots, V_n$ have votes $v_1, \ldots, v_n$ that they want to cast. An adversary $\mathcal{A}$ tries to break the protocol. $\mathcal{A}$ has full control over a fixed set of corrupt voters and gets as input a string z. $\mathcal{A}$ controls the flow of the protocol, i.e., it decides when to shift to the next phase, and within each phase it can adaptively activate voters. Upon activation, a voter reads the contents of the message board, computes its input according to the voting protocol, and posts it on the message board. After an honest voter has been activated control is passed back to $\mathcal{A}$. Please note that $\mathcal{A}$ may choose not to activate a voter, in that case the voter does not get to submit a vote. Once the election is over $\mathcal{A}$ computes an output s and halts. The output of the experiment is $(s, \text{cont}, \text{result})$, where cont is the contents of the message board and result is the outcome of the election if this can be computed from cont.

We write $\mathbf{Exp}^{\text{real}}_{V_1,\ldots,V_n,\mathcal{A}}(v_1, \ldots, v_n, z)$ to denote the distribution of $(s, \text{cont}, \text{result})$ from the real-life experiment.

Simulation. In this experiment, a simulator $\mathcal{S}$ has to simulate the election. $\mathcal{S}$ gets as input a string z, including a list of corrupt voters. $\mathcal{S}$ controls the random oracle; this enables it to simulate zero-knowledge proofs. In the simulation, we let a trusted party $\mathcal{T}$ handle the message board as well as computation of the result. $\mathcal{T}$ learns the votes $v_1, \ldots, v_n$ and which voters are corrupt. In the key registration phase, the voting phase and the fault correction phase, $\mathcal{T}$ expects to receive also the witnesses when $\mathcal{S}$ submits a valid key or a valid vote on behalf of a corrupt voter. In particular, this means that $\mathcal{T}$ learns the plaintext vote whenever a corrupt voter tries to cast a vote. Due to the self-tallying property of the voting scheme, the honest voters' partial tally may be revealed at some point. We formulate the following rule for letting $\mathcal{T}$ reveal this partial tally to $\mathcal{S}$. First, $\mathcal{T}$ notes which honest voters did not participate in the setup phase or the key-registration phase. In the voting phase, if $\mathcal{S}$ is about to activate the last remaining honest voter then it may query $\mathcal{T}$ for the partial tally of the honest voters. Afterwards, we demand that $\mathcal{S}$ posts a vote on behalf of this simulated voter. After the election, $\mathcal{S}$ halts with output s. $\mathcal{T}$ computes the result using the plaintext votes and the honest voters votes, and outputs the contents of the message board and the result.

We write $\mathbf{Exp}^{\mathrm{sim}}_{\mathcal{T},\mathcal{S}}(v_1,\ldots,v_n,z)$ to denote the distribution of $(s,\mathrm{cont},\mathrm{result})$ in the simulation.

The simulator $\mathcal{S}$. $\mathcal{S}$ runs a copy of $\mathcal{A}$ and simulates everything that $\mathcal{A}$ sees, including the behavior of the honest voters. When $\mathcal{A}$ changes phase in the protocol so does $\mathcal{S}$. If $\mathcal{A}$ lets a corrupt voter post something on the message board, $\mathcal{S}$ verifies the proof. If the proof is valid, $\mathcal{S}$ uses rewinding techniques to extract the witness. It then submits the entire thing to $\mathcal{T}$. In particular, this means that the vote is submitted in plaintext to $\mathcal{T}$. If $\mathcal{A}$ activates an honest party in the key registration phase, $\mathcal{S}$ selects h_i at random and simulates the proof of knowledge of x_i. It submits h_i and the simulated proof to $\mathcal{T}$. If $\mathcal{A}$ activates an honest voter in the voting phase, and this is not the last remaining honest voter to vote, $\mathcal{S}$ picks (U,V) at random and simulates a proof of knowledge of the corresponding x_i, r_i, v_i. If the activated honest voter is the last honest voter to submit a vote, then $\mathcal{S}$ queries $\mathcal{T}$ for the partial tally of the honest voters. Knowing the witnesses for the corrupt voters' submissions it can then compute the partial tally of voters that have voted so far. Let S be the set of voters that have voted, including the voter to vote right now. Let T be the set of remaining eligible voters; all of them are corrupt. $\mathcal{S}$ picks U at random and computes $V = U^{\sum_{j\in T} x_j} g^{\sum_{i\in S} v_i}$. It then simulates the proof for having computed (U,V) correctly and gives it to $\mathcal{T}$. At some point the simulated $\mathcal{A}$ halts with output s. $\mathcal{S}$ outputs s and halts.

Lemma 1. *For any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that the distributions $\mathbf{Exp}^{\mathrm{real}}_{V_1,\ldots,V_n,\mathcal{A}}(v_1,\ldots,v_n,z)$ and $\mathbf{Exp}^{\mathrm{sim}}_{\mathcal{T},\mathcal{S}}(v_1,\ldots,v_n,z)$ are indistinguishable for all $v_1,\ldots,v_n,z$.*

Proof. We use the simulator $\mathcal{S}$ described above. To show indistinguishability we will go through a series of intermediate experiments $Exp_1,\ldots,Exp_3$. We then show that $\mathbf{Exp}^{\mathrm{real}}_{V_1,\ldots,V_n,\mathcal{A}}(v_1,\ldots,v_n,z) \approx Exp_1(v_1,\ldots,v_n,z) \approx Exp_2(v_1,\ldots,v_n,z) \approx Exp_3(v_1,\ldots,v_n,z) \approx \mathbf{Exp}^{\mathrm{sim}}_{\mathcal{T},\mathcal{S}}(v_1,\ldots,v_n,z)$.

Exp_1 works like $\mathbf{Exp}^{\mathrm{real}}_{V_1,\ldots,V_n,\mathcal{A}}$ except whenever $\mathcal{A}$ submits a valid input on behalf of a corrupt voter. In these cases, we use rewinding techniques to extract the corresponding witnesses in expected polynomial time. This way for each key registration from a corrupt voter we know the corresponding exponent x_i, and for each vote we know the vote v_i as well as the randomness r_i and x_i. Having knowledge of the witnesses, we may now run the entire protocol using the trusted party $\mathcal{T}$ from the simulation experiment to control the message board. The outputs of the two experiments are the same, so indistinguishability is obvious.

Exp_2 works like Exp_1 except we simulate all proofs made by honest voters. Typically, these proofs are statistical zero-knowledge and then we get statistical indistinguishability between Exp_1 and Exp_2.

Let us consider Exp_2 a little further. Define $g_i = g^{r_i}$ and $h_{ij} = h_j^{r_i}$, where r_i is the randomness used by voter i. Consider the voting phase, denote at a given time S to be the voters that have cast votes already and T to be the voters that have not yet acted in this phase. The state at this time is

$$(u,v) = (\prod_{i\in S} g_i, (\prod_{i\in S}\prod_{j\in T} h_{ij})g^{\sum_{i\in S} v_i}).$$

Since we are simulating the proofs, we do not need knowledge of x_i, r_i for honest voters. Therefore, to carry out Exp_2 we can first compute a table of the g_i's,h_j's and h_{ij}'s for the honest voters and then use these values.

Define Exp_3 to be Exp_2 where we choose the g_i's,h_j's and h_{ij}'s randomly from G_q. By a hybrid argument using the DDH assumption, the tables of these elements in Exp_2 and Exp_3 are indistinguishable. Therefore, the two experiments Exp_2 and Exp_3 are indistinguishable.

Remaining is the fact that we still use individual votes v_i from honest voters to perform the experiment. However, note that in the voting phase when an honest voter V_i updates from (u,v) to (U,V) he sets $U = ug_i$ and $V = v(\prod_{j\in S} h_{ji}^{-1})(\prod_{j\in T} h_{ij})g^{v_i}$. The elements $\{h_{ij}\}_{j\in T}$ contain new randomness and therefore the vote v_i is perfectly hidden unless T has no honest voters, i.e., V_i is the last honest voter to vote.

These considerations lead us to modify Exp_3 the following way. An honest voter who is not the last honest voter to act in the voting phase computes the new state (U,V) by picking it at random in $G_q \times G_q$. An honest voter V_i who is the last honest voter to vote computes $\sum_{i\in S} v_i$, picks U at random from G_q and sets $V = U^{\sum_{i\in T} x_i} g^{\sum_{i\in S} v_i}$.

This modifies Exp_3 into $\mathbf{Exp}^{\mathrm{sim}}_{\mathcal{T},\mathcal{S}}$, so these two experiments are perfectly indistinguishable. □

Lemma 1 says that the election can be simulated without knowledge of the honest voters' individual votes. Moreover, it forces the simulator to submit plaintext votes on behalf of corrupt voters, so their votes cannot be related to the honest voters' votes.

Theorem 1. *The voting protocol described in Figure 1 is self-tallying, dispute-free, and has perfect ballot secrecy. If the last voter is an honest authority that submits a zero-vote then the protocol is fair.*

Proof. It is easy to see that the protocol is self-tallying if all parties act according to the protocol, and the zero-knowledge proofs force the parties to act according to the protocol. Likewise, since the zero-knowledge proofs force parties to act according to the protocol it follows that the protocol is dispute-free. Perfect ballot secrecy follows from Lemma 1. Fairness follows from perfect ballot secrecy, since perfect ballot secrecy implies that we cannot compute any partial result before the authority submits its vote, and if honest the authority does not submit its vote before the end of the election. □

2.4 A Veto Protocol

Kiayias and Yung suggested a veto-protocol in [5]. By this, we mean that any party may veto a proposal, however, it should not be possible to learn who vetoed the proposal or how many vetoed a proposal.

It is easy to implement such a veto protocol with the voting scheme we have suggested. We let acceptance of the proposal correspond to a 0-vote. On the other hand, a veto is a vote on a random element from $\mathbb{Z}_q$. This way, if nobody vetoed we get a tally, which is 0. On the other hand, if anybody vetoed, then we get a tally, which is a random number from $\mathbb{Z}_q$. Discrete logarithms are difficult to compute, however, we do not have to do that, all we need to do is to verify that $g^{\text{result}} \neq 1$.

One problem, which also pertains to the scheme in [5], remains with this scheme, since any vetoer knows his own random element and therefore he may check whether he is the only one who vetoed. To guard against that we may rely on the authority disclosing the result to raise (u, v) to a random exponent from $\mathbb{Z}_q^*$ before decrypting. This way it is impossible for any cheating vetoer to see whether he is the only one to veto the proposal.

3 Self-disclosing Anonymous Broadcast with Perfect Message Secrecy

3.1 Security Definitions

In this section, we deal with the possibility of building an anonymous broadcast channel on top of an authenticated broadcast channel. We want some strict security requirements to be satisfied. The security requirements are quite similar to those for self-tallying elections with perfect ballot secrecy but we rename the latter notion to stress that anonymous broadcast has many other applications than voting.

Perfect message secrecy: Knowledge of the set of messages to be broadcast is only accessible to a coalition of all remaining senders, and this knowledge does not include the connection between senders and messages. This means that a sender is hidden completely among the group of honest senders.

Self-disclosing: Once the last sender has submitted his message, anybody may see which messages were broadcast.

Fairness: Until the deadline is reached it is impossible to know what messages will be broadcast. Again, we will only demand fairness in a restricted sense, namely it will be ensured by a hopefully honest authority.

Dispute-freeness: It is publicly verifiable whether senders follow the protocol or not.

3.2 The Anonymous Broadcast Protocol

Physical analogue. The senders one after another enter a room alone. Bringing with them they take a box, all boxes look alike, and a padlock for each of the remaining senders. In the room, they write down their message, put it in the box, and lock the box with the padlocks corresponding to the remaining senders. Then they shuffle around the boxes so nobody can tell them apart. In the presence of the remaining senders, they now remove one lock from each box, namely the locks that fit their key. As the last sender removes the locks, the messages are revealed.

Idea in the protocol. We use similar ideas as we did in the voting protocol. Each voter encrypts his message with the keys of the remaining senders. This means that the message will not be revealed until all honest voters have been involved in the protocol and peeled off the layer of encryption corresponding to their secret key. The sender will rely on this last honest sender to anonymize his message with respect to all the honest senders.

Since the sender cannot know whether he is the last honest sender, he must also ensure himself that his message is mixed with the messages of the previous senders. Since ElGamal encryption is homomorphic, it is easy to permute and rerandomize (shuffle) all the ciphertexts made up to this point. Furthermore, efficient proofs of a correct shuffle exist, see [9–11].

Summarizing the protocol the method is as follows. The senders all register public keys just as in the voting protocol. When a sender wants to add his message to the pool, he encrypts it with the public keys of the remaining senders including his own key. Then he shuffles all the ciphertexts in a random way. Finally he peels of a layer of the encryption, namely he decrypts all the ciphertexts with respect to his own key. He proves in zero-knowledge that all these steps have been performed correctly.

The full protocol can be seen in Figure 2.

Performance evaluation. Key registration takes $\mathcal{O}(1)$ exponentiations for each sender, and each key has size $\mathcal{O}(k)$. To verify the correctness of the keys we use $\mathcal{O}(n)$ exponentiations.

With respect to message submission, we may use the efficient shuffle proofs of [9–11]. This way it takes $\mathcal{O}(n)$ exponentiations to compute the new batch of ciphertexts and the proofs, and such a batch has size $\mathcal{O}(nk)$. It takes $\mathcal{O}(n^2)$ exponentiations to verify all the senders' proofs.

Simultaneous disclosure. If we remove the shuffling part of our anonymous broadcast protocol, we get a simultaneous disclosure protocol. We can therefore compare our performance with the simultaneous disclosure protocol of [5], which uses $\mathcal{O}(n^2)$ exponentiations for each voter in the registration phase, and $\mathcal{O}(n)$ exponentiations for each voter in the message submission phase.

3.3 Security

To argue perfect message secrecy we show that the broadcast protocol can be simulated without knowledge of the individual messages. Very similar to the case of the voting protocol we therefore define a real-life experiment and a simulation experiment.

Anonymous Broadcast Protocol

Setup: The senders agree on a suitable group G_q of order q, where the DDH problem is hard. Let g be a generator for G_q.
The senders also set up suitable keys for commitment schemes that will be used in the zero-knowledge proofs to follow.

Key Registration: Sender i selects at random $x_i \in \mathbb{Z}_q$ and sets $h_i = g^{x_i}$. He publishes this public key together with a proof of knowledge of x_i.
Any senders who did not supply a public key are removed from the list of senders.

Message submission: Sender i wishing to send message $m_i \in G_q$.
Let S be the set of senders who already sent a message, including i, and let T be the set of senders who did not send a message. Let $\{(u_j, v_j)\}_{j \in S \setminus \{i\}}$ be the ciphertexts constituting the state.
Sender i first checks that all proofs of the previous senders are correct. Then he encrypts his message as $(u_i, v_i) = (g^{r_i}, (\prod_{j \in T \cup \{i\}} h_j)^{r_i} m_i)$. He picks at random a permutation π_i over S, permutes all ciphertexts $\{(u_j, v_j)\}_{j \in S}$ according to this permutation, and rerandomizes them into $\{(U_j, V_j')\}_{j \in S}$.
Finally, he removes the layer of encryption corresponding to his own private key. I.e., he computes $\{(U_j, V_j) = (U_j, V_j' U_j^{-x_i})\}_{j \in S}$.
He broadcasts this list of ciphertexts together with a proof of knowledge of having done all this correctly.

Broadcasting: The last senders' output contains V_j's that are the messages permuted according to the permutations selected by the senders.

Fault-correction: We do not have a clever fault correction algorithm; we simply start the protocol over again. Depending on the user requirements, we may now demand that the senders prove in zero-knowledge that they are submitting the same message as before.

Fig. 2. The anonymous broadcast protocol

Real-life experiment. We have parties $P_1, \ldots, P_n$ with messages $m_1, \ldots, m_n$ that they want to broadcast anonymously. An adversary $\mathcal{A}$ with input z controls a fixed set of these parties. $\mathcal{A}$ also controls the scheduling in the protocol, in other words, $\mathcal{A}$ decides when to proceed to the next phase, and within each phase $\mathcal{A}$ activates parties adaptively. When activated a party receives the contents of the message board, computes its input according to the protocol, and posts it on the message board. Control then passes back to $\mathcal{A}$. In the end, $\mathcal{A}$ outputs some string s and halts.

We denote by $\mathbf{Exp}^{\mathrm{real}}_{P_1,\ldots,P_n,\mathcal{A}}(m_1, \ldots, m_n, z)$ the distribution of outputs $(s, \mathrm{cont}, \mathrm{messages})$ from the experiment, where cont is the content of the message board, and messages is a sorted list of messages from cont.

Simulation. Again, we have a trusted party $\mathcal{T}$ and a simulator $\mathcal{S}$. $\mathcal{T}$ controls the message board and has as input $m_1, \ldots, m_n$ and a list of corrupted parties. During the execution of the protocol it expects $\mathcal{S}$ to provide witnesses for correctness of the actions performed by corrupted parties. When only one honest party remains in the broadcast phase, $\mathcal{S}$ can query $\mathcal{T}$ for the set of messages $m_1, \ldots, m_k$ submitted by honest parties. After this $\mathcal{S}$ must then submit this honest party's broadcast to $\mathcal{T}$. In the end, $\mathcal{S}$ halts with output s, and $\mathcal{T}$ outputs the contents of the message board and the set of messages submitted in lexicographic order.

We write $\mathbf{Exp}^{\mathrm{sim}}_{\mathcal{T},\mathcal{S}}(m_1, \ldots, m_n, z)$ for the distribution of $(s, \mathrm{cont}, \mathrm{messages})$.

The simulator $\mathcal{S}$. $\mathcal{S}$ runs a copy of $\mathcal{A}$ simulating anything $\mathcal{A}$ would see in a real-life execution, including the actions of the honest parties. Whenever $\mathcal{A}$ changes phase, so will $\mathcal{S}$. If $\mathcal{A}$ lets a corrupt party submit something with a valid proof for the message board, $\mathcal{S}$ uses rewinding to extract the witness. This way, in the key registration phase $\mathcal{S}$ learns the exponent x_i, when corrupt party P_i registers key h_i. Likewise, when corrupt party P_i makes a broadcast, then $\mathcal{S}$ learns the randomizers used, the new message that was submitted, and the permutation π_i. After extracting the witness, $\mathcal{S}$ sends everything to the trusted party $\mathcal{T}$. If $\mathcal{A}$ activates an honest party P_i in the key registration phase then $\mathcal{S}$ picks h_i at random and simulates a proof that it knows the exponent x_i. If $\mathcal{A}$ activates an honest party P_i in the message submission phase, and this is not the last honest party to act, $\mathcal{S}$ selects (u_i, v_i) at random from $G_q \times G_q$. For each $k \in S$, where S is the set of senders that have been active in the protocol, including P_i, $\mathcal{S}$ selects at random (U_k, V_k') and (U_k, V_k). It then simulates proofs that it knows the message inside the (u_i, v_i) encryption, that it knows a permutation π_i and randomizers such that $\{(U_k, V_k')\}_{k\in S}$ is a shuffle of $\{(u_k, v_k)\}_{k\in S}$, and that for each $k \in S$, (U_k, V_k) is the decryption of (U_k, V_k') with key x_i used to form h_i. If the sender activated is the last remaining honest sender, $\mathcal{S}$ queries $\mathcal{T}$ for the list of messages for honest senders. Furthermore, it knows the messages submitted by corrupt parties. It labels in random order the messages $\{m_k\}_{k\in S}$. It picks (u_i, v_i) at random and picks at random (U_k, V_k') for $k \in S$. Then for $k \in S$ it sets $V_k = U_k^{\sum_{j\in T} x_j} m_k$, where T is the set of (corrupt) senders that have not yet been activated. $\mathcal{S}$ simulates the proofs of correctness and submits it all to $\mathcal{T}$. In the end the simulated $\mathcal{A}$ terminates with output s. $\mathcal{S}$ outputs s and halts.

Lemma 2. *For any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that the two distributions $\mathbf{Exp}^{\mathrm{real}}_{P_1,\ldots,P_n,\mathcal{A}}(m_1, \ldots, m_n, z)$ and $\mathbf{Exp}^{\mathrm{sim}}_{\mathcal{T},\mathcal{S}}(m_1, \ldots, m_n, z)$ are indistinguishable for all $m_1, \ldots, m_n, z$.*

Proof. The proof is similar to the proof for Lemma 1. We use the simulator described above. We define three intermediate experiments Exp_1, Exp_2 and Exp_3 and prove that $\mathbf{Exp}^{\mathrm{real}}_{P_1,\ldots,P_n,\mathcal{A}}(m_1, \ldots, m_n, z) \approx Exp_1 \approx Exp_2 \approx Exp_3 \approx \mathbf{Exp}^{\mathrm{sim}}_{\mathcal{T},\mathcal{S}}(m_1, \ldots, m_n, z)$.

Exp_1 is the real-life experiment where we use rewinding techniques to extract witnesses for valid actions that $\mathcal{A}$ lets corrupt parties make. Having the witnesses, we can then execute this experiment in the trusted message board model, giving $\mathcal{T}$ the witnesses to go along with the messages.

Exp_2 is a modification of Exp_1 where we simulate all proofs that honest parties make. Consider how an honest party P_i computes the new state $\{(U_j, V_j)\}_{j\in S}$, where S is the set of parties that have submitted their message.

Write T for the set of remaining parties that have not yet made a broadcast. P_i first selects r_i at random and sets $(u_i, v_i) = (g_i, (\prod_{j \in T \cup \{i\}} h_{ij}) m_i)$, where $g_i = g^{r_i}$ and $h_{ij} = h_j^{r_i}$. Then P_i selects π_i as a random permutation over S, and computes the pairs $(U_k, V_k') = (u_{\pi_i^{-1}(k)} g_{ik}, v_{\pi_i^{-1}(k)} \prod_{j \in T} h_{ijk}\}_{k \in S}$, where $g_{ik} = g^{r_{ik}}$ and $h_{ijk} = h_j^{r_{ijk}}$, with the r_{ik}'s and r_{ijk}'s chosen at random from $\mathbb{Z}_q$. Finally, for $k \in S$ it sets $(U_k, V_k) = (U_k, V_k U_k^{-x_i}) = (U_k, V_k \prod_{j \in S} h_{ji}^{-1})$. All this can be computed from a table of g_i's, h_j's, h_{ij}'s, g_{ik}'s, and h_{ijk}'s for the honest parties without knowing the underlying randomizers.

Exp_3 is a modification of Exp_2 where the g_i's, h_j's, h_{ij}'s, g_{ik}'s and h_{ijk}'s for honest parties are selected at random from G_q. By a hybrid argument using the DDH assumption, Exp_2 and Exp_3 are indistinguishable.

Looking at Exp_3, we notice that we might as well pick the elements u_i, v_i, U_k, V_k', V_k completely at random from G_q instead of bothering with picking a permutation π_i and inserting messages, as long as P_i is not the last honest party to broadcast a message. An honest party P_i that is the last honest party to broadcast a message chooses u_i, v_i, U_k, V_k' at random. It picks a permutation π at random and sets $V_k = U_k^{\sum_{j \in T} x_j} m_{\pi(k)}$ for $k \in S$. This last experiment is exactly what happens in the simulation so Exp_3 and $\mathbf{Exp}^{\mathrm{sim}}_{\mathcal{T},\mathcal{S}}$ are perfectly indistinguishable. □

Theorem 2. *The protocol described in Figure 2 is a self-disclosing, dispute-free anonymous broadcast protocol with perfect message secrecy. If the last sender is an honest authority (who does not submit a message himself) then the protocol is fair.*

Proof. It is easy to see that the protocol is self-disclosing. The zero-knowledge proofs entail dispute-freeness. Perfect message secrecy follows from Lemma 2. Finally, fairness follows from the perfect message secrecy. □

4 Various Comments

Reusing the public keys. In both the voting protocol and the anonymous broadcast protocol we may reuse the public keys in many instantiations of the protocols presented here, but some care must be taken. The reason to be careful is the fact we must be able to rewind and extract witnesses from proofs made by the adversary. In the simulation, however, we cannot rewind the trusted party $\mathcal{T}$, so we must be careful that we never have to rewind past a point where $\mathcal{T}$ gives us a partial tally or partial set of honest senders messages. When only a single protocol is running this is no problem since in the zero-knowledge proofs we query the random oracle with the current state. When a partial result is released we always let an honest party act right after it, and this honest party injects some new randomness into the state. For this reason an adversary can not predict what the state will be after the release of a partial result, and therefore cannot make queries *before* the release of the partial result that it uses *after* the release of the partial result. This means that we never have to rewind back before the release of a partial result. When running multiple protocols we have to query the random oracle with the states of all protocols to guarantee not having to rewind back past a point where a partial result was released. If we do this, we may use the same public keys to run many protocols.

Universal composability. The statement of our lemmas is somewhat inspired by the universal composability framework of Canetti [12, 13]. However, we have *not* proved the protocols to be universally composable. In particular, we do not include a party $\mathcal{Z}$ to model the exterior environment. It is possible to make the protocols universally composable against non-adaptive adversaries by generating a key for a public key cryptosystem in the setup phase. After this we can in the key registration phase encrypt the keys x_i and prove to have done so in zero-knowledge. We can set this up so the simulator knows the corresponding secret key for the cryptosystem, and therefore it can make a straight-line extraction of the x_i's. Knowing the x_i's it can then extract votes and messages, and carry on the simulation without ever having to rewind. Unfortunately, the technique above may make the protocols considerably less efficient and we have therefore not pursued this option in the paper.

Flexibility in participation. It is easy to set up an election where only a part of the participants is allowed to participate. In that case, we simply ignore the public keys of those not allowed to participate in this instance of the protocol.

In the voting protocol, it is easy to include new voters that may participate in future election. We can choose the group $G_q \leq \mathbb{Z}_q$ specified by p, q, g in a publicly verifiable manner, e.g., chosen at random from the binary expansion of π, or chosen from a string of hashes on some random value. Considering uniform adversaries it would seem reasonable that this gives us a suitably hard group.[4] Since the new voter can trust this group, he simply needs to register a public key himself in order to join.

In the anonymous broadcast protocol, we may also include new senders. However, here the new senders have to beware of the risk that the commitment scheme may be chosen with a trapdoor known to the senders already registered. Therefore, the new sender will have to update this commitment key in a publicly verifiable way.

The authenticated broadcast channel. We do not need something fancy to form this channel. We may for instance assume that a central server stores all the data, and this central server may act like the authority too.

To ensure correctness of the data we will assume that all communication is signed with a digital signature. We cannot rely on a certification authority to issue these digital signatures in the strict setting we are working in. Instead,

[4] While it varies from group to group how hard it is to compute discrete logarithms, we do not know of any groups where the DDH problem can be efficiently solved, provided the groups are some subgroup of $\mathbb{Z}_p^*$ where q, p are suitably large primes. See also [14] on this issue.

each participant must certify each other participants public key. Since we assume only a few voters or senders are participating in the protocol, this is a reasonable burden to put on the participants.

Imagine now that the central server fails. Since everything is digitally signed, the participants may restore the state of the message board from their own data. They may now simply set up a new server to run the protocol. It is easy to modify the votes in a publicly verifiable manner such that the data fits the public key of the new authority.

References

1. Kiayias, A., Yung, M.: Self-tallying elections and perfect ballot secrecy. In: proceedings of PKC '02, LNCS series, volume 2274. (2002) 141–158
2. Damgård, I., Jurik, M.J.: A length-flexible threshold cryptosystem with applications. In: proceedings of ACISP '03, LNCS series, volume 2727. (2003) 350–364
3. Paillier, P.: Public-key cryptosystems based on composite residuosity classes. In: proceedings of EUROCRYPT '99, LNCS series, volume 1592. (1999) 223–239
4. Kiayias, A., Yung, M.: Robust verifiable non-interactive zero-sharing. In Gritzalis, D., ed.: Secure Electronic Voting. Kluwer Academic Publishers (2003) 139–151
5. Kiayias, A., Yung, M.: Non-interactive zero-sharing with applications to private distributed decision making. In: proceedings of Financial Crypto, LNCS series, volume 2742. (2003) 303–320
6. Algesheimer, J., Camenisch, J., Shoup, V.: Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In: proceedings of CRYPTO '02, LNCS series, volume 2442. (2002) 417–432
7. Cramer, R., Damgård, I., Schoenmakers, B.: Proofs of partial knowledge and simplified design of witness hiding protocols. In: proceedings of CRYPTO '94, LNCS series, volume 893. (1994) 174–187
8. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: ACM Conference on Computer and Communications Security 1993. (1993) 62–73
9. Furukawa, J., Sako, K.: An efficient scheme for proving a shuffle. In: proceedings of CRYPTO '01, LNCS series, volume 2139. (2001) 368–387
10. Neff, A.C.: A verifiable secret shuffle and its application to e-voting. In: ACM CCS '01. (2001) 116–125
11. Groth, J.: A verifiable secret shuffle of homomorphic encryptions. In: proceedings of PKC '03, LNCS series, volume 2567. (2003) 145–160
12. Canetti, R.: Security and composition of multi-party cryptographic protocols. Journal of Cryptology **13** (2000) 143–202
13. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001. (2001) 136–145
14. Gordon, D.M.: Designing and detecting trapdoors for discrete log cryptosystems. In: proceedings of CRYPTO '92, LNCS series, volume 740. (1992) 66–75