Vote selling resistant voting

Colin Boyd[1], Thomas Haines[1], and Peter Roenne[2]

[1] Norwegian University of Science and Technology, Trondheim, Norway
{colin.boyd,thomas.haines}@ntnu.no
[2] SnT, Université du Luxembourg, Luxembourg, Luxembourg
peter.roenne@uni.lu

Abstract. Creating a secure (purely) remote voting scheme which prevents active vote selling is an open problem. Observing that vote selling has a primarily economic motivation, we propose a novel approach to the problem which prevents a vote seller from profiting by allowing a different party to choose the seller's vote. As a proof of concept, we propose a concrete protocol which involves carefully restricting the ways the voter can prove how they voted and then penalising them for revealing it. With the assumption that the vote seller and vote buyer are mutually distrustful, we show that our protocol admits no situation where the buyer and seller can achieve a mutually agreeable selling price. We include a sample instantiation of our protocol demonstrating that it can be practically implemented including the outlay of a smart contract in Solidity.

Keywords: Vote buying resistance · Disincentives · Blockchain

1 Introduction

Coercion is a major security threat against electronic voting schemes, particularly for *remote* schemes where the voter is not isolated in a polling booth but can vote using any internet connection. A related, but in reality quite different, threat is *vote selling*. As with coercion, there is a malicious party whose goal is to influence or dictate the vote submitted by a valid voter. However, in contrast with a coercion scenario, the voter is willing to cooperate and indeed wants to vote in a way chosen by the malicious party, as long as the voter is paid (enough) for doing so. In this paper we address the vote selling threat.

A possible mechanism to mitigate the vote selling threat is to ensure that the voting scheme is *receipt free*, i.e. to ensure that voters are unable to prove to others how they voted, as suggested in the seminal paper by Benaloh and Tuinstra [4]. Denying the voter a receipt to prove how the vote was cast removes an instrument that the voter could use to negotiate or claim a payment from the buyer. However, we observe that this is not sufficient in many situations. In a remote setting the vote seller can perform the voting protocol in the presence of the buyer or can even hand over the voting credential required to vote and allow the buyer to vote on the seller's behalf. Indeed the scenario in which the voter

hands over the voting credential for cash may at first seem impossible to avoid in the remote setting without some assumptions about what the voter is willing to sacrifice, or without a trusted setup phase such as is present in the JCJ scheme [11]. Further, even assuming a trusted setup-phase, a scheme like JCJ lacks good usability and intuitive individual verifiability [10] and the construction makes it harder to achieve eligibility verifiability without trust assumptions [15].

In this paper we propose a novel remote voting scheme which discourages voter selling **without** providing some form of receipt-freeness but instead by monetarily penalising the voter for revealing their vote. We believe this is interesting due to the general unavailability of useable end-to-end verifiable coercion-resistant e-voting schemes and the inherent assumption in the remote receipt-freeness definitions that the voter does not deviate from the honest vote casting protocol. We also hope to convince the reader that the problem is technically challenging from a cryptographic perspective.

We achieve vote selling resistance by having each voter deposit some amount of money which is automatically returned to them after a certain period of time has elapsed. However, anyone can claim half the money and donate the other half to a predetermined charity if they know a certain secret. This secret happens to be leaked whenever the voter proves how they voted. Strictly speaking the voter could always prove how they voted with general zero-knowledge since the question of which vote the ballot encodes is in NP, but we carefully choose the encryption scheme so that this is infeasible in practice. We note in passing that donating the money to charity is but one of a number of options. Alternatively, the money could be given to the government as a tax or transferred to a dead address effectively burning it.

The idea of using economic incentives to prevent security threats is certainly not new. Indeed, the Workshop on the Economics of Information Security[3] has been pointing out the connection between security and economics for over 15 years. Furthermore, there are a number of cryptographic protocols whose security relies on economic incentives for adversaries, see e.g. [6, 8]. However, to our knowledge an economic approach to prevent vote buying has not been proposed before and seems natural given that the seller goal is to make a monetary profit.

Contribution

The overall contributions of this paper are to:

- introduce the idea of using economic incentives to mitigate the threat of vote buying;
- propose a concrete voting protocol incorporating an economic disincentive to vote selling and buying;
- demonstrate the feasibility of our approach by outlining an implementation in Solidity.

[3] `https://econinfosec.org/`

2 Aims

The core aim is easy to state; have a secure remote scheme which prevents vote selling. However we relax this, instead desiring a secure scheme which disincentives the voters for selling their votes. Ideally, this would be possible even if the vote buyers and sellers trusted each other completely and the vote buyers were election authorities. However, it seems enormously difficult to construct such a scheme. We settle, in this paper, for a restricted setting in which the vote buyers and vote sellers are mutual distrusting and disincentivising vote selling during the election suffices.

In this section we will first define our assumptions about the parties and then talk informally about the security proprieties, we would like to achieve.

2.1 Authorities

We assume the election is run by a set of N_a authorities. As a basic requirement for any secure voting scheme, we require that *verifiability* holds even if all authorities are corrupt. This means that anyone can verify that only authorised voters took part, on the assumption that there is a valid registration process. However, as is usual in voting schemes, we assume that at least one voting authority is honest in order to maintain voter privacy and also will not collude with other authorities to steal the voters' deposits.

2.2 Voters and Vote Buyers

We assume that voters do not trust the vote buyers and are only willing to change their vote if they receive the payment with a time delay less than P_d. We, further, assume they want to be paid at least C_s.

We assume the vote buyers are interested in buying votes but are unwilling to buy them above, and including, the price point C_b. We assume they are unwilling to trust the voter and require evidence of what the vote was before releasing the payment.

We do not explicitly model the negotiation of the price P which the buyer will pay to the seller, which can happen with any protocol that the parties choose. However, we do assume that there is some P which both parties are willing to accept. This can be summarised in the following equation and, in particular note that the existence of an agreed price implies $C_b \geq C_s$.

$$P \in [C_s, C_b] \tag{1}$$

Note that the mutual distrust of the voters and vote buyers means it suffices to disincentivise vote selling during the election period. We further assume that there does not exist any escrow process which would allow the voters and vote buyers to overcome their mutual distrust. We claim the assumption of no escrow is reasonable because voting selling is illegal and hence any escrow must be privately conducted with a third party whom the voter already trusts and is

willing to be engage in illegal activity. The assumption of no escrow also excludes the possibility of the voters and vote buyers creating a counter smart contract to overcome their distrust.

3 Preliminaries

In this section we will detail the building blocks and cryptographic primitives we need to build our scheme.

3.1 Smart Contract enabled Blockchain

We use a blockchain, such as Ethereum [1], as our public bulletin board which allows us to integrate the deposit mechanism with the election. We rely on the universal verifiability of the blockchain to ensure that the steps of the election can be observed and checked by any party. The inbuilt payment mechanism available in typical blockchains, such at Ether in Ethereum, will be used to provide the economic incentives. Using similar techniques to those used in hash-locked contracts [9], voters will be required to provide a stake, or deposit, at the time of voting. The deposit will be recovered in full by honest voters a short time after the election is complete. As we will explain later in Section 5, voters trying to sell their votes will end up losing some or all of their deposit.

The required functionality can be achieved using a standard smart contract in Ethereum, which includes mechanisms for automatic payments and timing. The election authorities will construct the voting contract and post it to the blockchain. This will allow any party to verify its functionality.

3.2 Encryption algorithm

Since voters will post their votes onto the blockchain, they must be encrypted first. This is not only in order to provide usual privacy of votes, but also so that the vote buyer cannot simply read the chosen vote. The encryption scheme chosen must also support certain proofs (see below for details of the proofs needed) and so a natural choice is ElGamal encryption [7] since it has the algebraic structure to support these proofs (and for that reason is often chosen for voting schemes). However, standard ElGamal encryption is not sufficient for our purposes since it allows an easy zero knowledge proof that the ciphertext takes on a certain value which could be used by the vote seller to convince the buyer that the desired choice has been made. We want instead to force the seller to release a specific value in order to convince the buyer.

Fortunately, there is a suitable solution already existing which is to use the OAEP 3-round (or simply OAEP3) transformation of Phan and Pointcheval [14]. Using similar principles to the well known OAEP transformation for trapdoor permutations (such as RSA), OAEP3 also works with ElGamal encryption. OAEP3 transformed ElGamal satisfies RCCA security [5] as proven by Phan and Pointcheval [14].

The OAEP3 transformation uses three hash function H_1, H_2, H_3. Its inputs are randomness r and the message m to be encrypted. Then the following values are computed:

$$s = m \oplus H_1(r) \quad t = r \oplus H_2(s) \quad u = s \oplus H_3(t) \quad c = (t, u).$$

To encrypt message m, we first compute the OAEP3 transform to derive c and then encrypt c using standard ElGamal.

The trick here is to observe that when the hash functions H_1, H_2, H_3 are ideal hash functions (thus using the Random Oracle Model) the OAEP3 output c, which is used as the input to the normal ElGamal encryption, is indistinguishable from a random string. This means that a normal ZK proof that the ciphertext contains a particular choice of vote cannot be used unless the randomness r is revealed so that $c = (t, u)$ can be reconstructed. We will use this observation to force a vote seller to give up the randomness r in order to convince the vote buyer.

Note that the ElGamal encryption uses its own randomness distinct from the r used in the OAEP3 transform. When we say we release the deposit to anyone who knows the randomness we mean the r value uses in OAEP3 transform not the randomness used in the ElGamal encryption.

3.3 Zero Knowledge Proofs

We need to use some standard proofs to provide verifiable evidence of correct working of the voting scheme. Although these proofs are usually applied to standard ElGamal, they can also be used with the OAEP3 transformed variant as pointed out by Pereira and Rivest [13].

NIZKP of correct encryption We use a knowledge of discrete log zero knowledge proof to allow the voter to show they know the ballot inside their encrypted vote, these proofs should be inherently tied to the voter using signatures which will be submitted with the encrypted ballot.

Verifiable proof of shuffle We use a zero knowledge proof of correct shuffle for ElGamal. This can be instantiated with known techniques [2, 16].

NIZKP of correct decryption We use a zero knowledge proof of correct decryption of ElGamal.

4 A Vote-Buying Resistant Scheme

We will now describe the phases of the scheme. We envision a smart contract on the bulletin board which enforces the public facing elements of the scheme. For simplicity we assume a pre-existing voter PKI though this can be removed with the expected loss of eligibility verification.

The contract is posted by the election authority to the blockchain (which may either be public or private). The contract describes the stages of voting analogous to the phases described below. In addition the contract describes

what can/should occur in these stages. Essentially the contract functions as the bulletin board, while imposing certain constraints on who can do what when. Since verifiability occurs based on the data posted on the contract, the contract itself does not need to be correct for universal variability to hold. On the other hand, the eligibility verification is enforced by the contract and this component should be checked for correctness.

We have included a sketch of the contract in appendix A.

Setup. In the setup phase the authorities jointly generate the public key pk to be used to encrypt votes [12]. The joint generation includes a shared set of decryption keys so that all authorities (possibly up to a threshold) need to cooperate to decrypt votes. The authorities also construct the contract and submit it to the blockchain. The voter credentials $\{pk_i\}_{i\in[1,n]}$ from the existing PKI are assumed to be known or can be included explicitly in the contract. The authorities also specify the deposit amount D.[4]

Submission. In the submission phase each voter may submit a ballot $\mathbf{Enc}_{pk}(m)$ by encrypting with OAEP3 and ElGamal and producing a zero knowledge proof of knowledge of the encrypted value (after OAEP3 transformation). In addition the voter creates a deposit of value D. All of these values are signed by the voter. The contract checks that the submission is well-formed by checking the proof and that the signature comes from an eligible voter who has not previously voted. If so, it accepts the ballot and marks the voter as having voted.

Claim. The claim phase runs from submission until the beginning of the proving phase. During this phase any party can submit an encrypted claim with respect to any vote. The contract will then perform a plaintext equivalence test and, if the test succeeds, the deposit is paid out half to the claimed party and half to the predetermined charity. This payout can only happen once according to the blockchain validity rules.

Note that since the messages in the ElGamal ciphertext are OAEP3 transformed, they are close to uniformly random in the message space and hence just guessing the message is infeasible. If denial of service attacks are a threat this claim process can be modified to require a small payment to submit. This payment should be small enough so that the money claimed if successful is higher than the fee.

Tallying. The authorities take turns to re-encrypt and shuffle the ciphertext using the verifiable shuffle to prove correctness. The authorities then jointly decrypt the ballots and publish the result. At this point they do not make public the proofs of correct decryption or reveal the randomness used in the OAEP3 transform.[5].

[4] It is possible to adjust the deposit amount during the election. The authorities may wish to do this if a significant number of successful claims are being made.

[5] It is significantly simpler if the tallying and proving phases are made into one phase. If they are separate, we either need to assume the authorities won't steal the deposits or develop someway for them to undo the OAEP3 transform using some distributed technique like MPC

Proving. During the proving phase, which occurs at least P_d after the election closes, the authorities post the proofs of correct decryption and reveal the randomness used in the OAEP3 transform.

Verification. In the verification phase any party can check the publicly available evidence.

5 Security

In many ways the core of the scheme is a very similar to a standard e-voting scheme template based on mixnets. The small, but important, differences are designed to allow the disincentive mechanism to function. The basic voting security properties follow the standard pattern.

5.1 Verifiability

The argument for the universal verifiability of the election scheme is entirely standard. All ballots are signed by their respective voters and the bulletin board will accept only one ballot per eligible voter which ensures that the list of collected ballots is correct up to denial of service. The ballots are then verifiably mixed and verifiably decrypted which ensures they are counted as collected.

We do not specify any specific method for cast as intended verification. However, the standard methods like Benaloh challenges [3] can clearly be applied.

5.2 Privacy

The general privacy of the scheme follows for the ballot independence of the submitted ballots, the mixing of the ballots and the IND-CPA security of the encryption scheme.

5.3 Vote buying resistance

The scheme has vote buying resistance for the following reasons. First recall that the voter and vote buyer are mutually distrustful. In particular, both are economically incentivised, so the voter wants to maximise the price P paid while the buyer will try to minimise it. The strategy to show that our protocol provides vote buying resistance is to show that Equation 1 cannot be satisfied, so that the buyer and seller are unable to agree on any suitable value for P.

We consider two mutually exclusive cases. Either (1) the voter produces the vote and later tries to convince the buyer of the choice of vote or (2) the buyer produces the vote with the help of the seller. For the first strategy, due to the usage of the OAEP3 transformation, the voter must reveal the OAEP3 transformed message, or equivalently the randomness r, to show how they voted. But this is precisely the information needed to claim the voter's deposit.

We need to consider two possible scenarios, which differ depending on what happens with the deposit. Note that the seller will not wait until the voting

protocol finishes to reclaim the deposit because the seller knows that the buyer will try to claim it before then. However, the voter can also try to reclaim half the deposit in the **Claim** phase, so there will be a race condition between buyer and seller to try to get the deposit.

Scenario 1 The buyer pays P to the seller and the seller successfully executes a **Claim**.
Outcome: Seller loses half deposit and gains P. Buyer pays P

Scenario 2 The buyer pays P to the seller and the buyer successfully executes a **Claim**.
Outcome: Seller loses whole deposit and gains P. Buyer pays P but recovers half the deposit

Table 1 shows the payoff for each party in each of these scenarios.

	Seller Payoff	Buyer Payoff
Scenario 1	$P - D/2 - C_s$	$C_b - P$
Scenario 2	$P - D - C_s$	$C_b - P + D/2$

Table 1. Payoff Matrix

Note that the vote seller and buyer will only proceed with the deal if the expected return is greater than 0, in other words their behaviour is determined only by their economic incentives. The question becomes: does there exist a price P such that both parties will have gained something?

Let $\Pr(\text{Buyer wins race}) = p$ so that $\Pr(\text{Seller wins race}) = 1 - p$. The analysis, and crucially the deposit amount, is independent of the expected success rates of the parties provided they are consistent, however, note that if both parties believe they will win the race condition all the time then no deposit amount suffices. The seller loses half the deposit in scenario 1 and all the deposit in scenario 2. Thus the seller is happy only if:

$$P - D/2 - p \cdot D/2 > C_s. \tag{2}$$

Consider then the buyer who gains half the deposit in scenario 2 and nothing in scenario 1. The buyer is happy only if:

$$P - p \cdot D/2 < C_b. \tag{3}$$

The event that the buyer and seller proceed will only occur if both inequalities (2) and (3) are satisfied. Therefore the vote selling attack occurs only if:

$$C_s + P - p \cdot D/2 < P - D/2 - p \cdot D/2 + C_b.$$

or $D < 2(C_b - C_s)$. To prevent the attack we therefore choose $D > 2(C_b - C_s)$.

Finally we consider case (2) where the buyer forms the ciphertext and the seller cooperates by signing the vote ciphertext and proofs, or the seller can even

give the signing credential to the vote buyer. In this case only the buyer is able to make a successful claim. Either the buyer or seller must stake the deposit, assuming that the signing credential also allows payments on the blockchain. If the seller pays the deposit then Scenario 2 above applies since only the Buyer can make the claim. If the buyer makes the deposit then the roles are reversed and buyer will lose since the buyer will only be satisfied when $P \leq C_s$.

6 Conclusion

We have introduced a novel approach to vote buying resistance which utilises economic incentives to remove the economic motivation to buy voters rather than making it impossible to do so. We have, then, provided a reasonable instantiation of this scheme for distrusting vote buyers and sellers.

We believe that there are opportunities to optimise and refine our proposals in a number of different ways.

- A more formal analysis with formal definitions for security and a more comprehensive economic model may yield interesting insights.
- It may be possible to develop better protocols which remain secure against stronger adversaries. For example it would be good to allow the vote buyer and seller to collude rather than assuming they are mutually distrusting.
- We have assumed that values of C_s and C_b exist and are known to the implementor. This may not be reasonable in all situations. Increasing the value of the deposit allows a looser estimate of these values, but there is a limit to how large a deposit can reasonably be.
- Is it possible to construct an encryption system which allows efficient proofs of correct decryption without allowing efficient proofs of encryption to a particular message? Such a construction could be usefully applied in our protocol.
- It would be interesting to construct a variant of the scheme where each voter holds a long-term credential to authenticate their ballot, and it is this credential which is leaked to the vote buyer by a proof of the cast vote. In this case the deposit can be kept lower since the vote buyer can release the deposit in future elections, too.

Acknowledgements

PBR would like to thank Reto Koenig for discussions. The authors acknowledge support from the Luxembourg National Research Fund (FNR) and the Research Council of Norway for the joint project SURCVS.

References

1. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: building smart contracts and dapps. O'Reilly Media (2018)

2. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 7237, pp. 263–280. Springer (2012)
3. Benaloh, J.: Ballot casting assurance via voter-initiated poll station auditing. In: EVT. USENIX Association (2007)
4. Benaloh, J., Tuinstra, D.: Receipt-free secret-ballot elections. In: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing. pp. 544–553. ACM (1994)
5. Canetti, R., Krawczyk, H., Nielsen, J.B.: Relaxing chosen-ciphertext security. In: CRYPTO. Lecture Notes in Computer Science, vol. 2729, pp. 565–582. Springer (2003)
6. van Dijk, M., Juels, A., Oprea, A., Rivest, R.L., Stefanov, E., Triandopoulos, N.: Hourglass schemes: how to prove that cloud files are encrypted. In: Yu, T., et al. (eds.) ACM Conference on Computer and Communications Security, CCS'12. pp. 265–280. ACM (2012), https://doi.org/10.1145/2382196.2382227
7. Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Information Theory **31**(4), 469–472 (1985)
8. Halpern, J.Y., Teague, V.: Rational secret sharing and multiparty computation: extended abstract. In: Babai, L. (ed.) Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004. pp. 623–632. ACM (2004), https://doi.org/10.1145/1007352.1007447
9. Herlihy, M.: Atomic cross-chain swaps. In: Newport, C., Keidar, I. (eds.) ACM Symposium on Principles of Distributed Computing, PODC. pp. 245–254. ACM (2018), https://dl.acm.org/citation.cfm?id=3212736
10. Iovino, V., Rial, A., Rønne, P.B., Ryan, P.Y.: Using Selene to verify your vote in JCJ. In: International Conference on Financial Cryptography and Data Security. pp. 385–403. Springer (2017)
11. Juels, A., Catalano, D., Jakobsson, M.: Coercion-resistant electronic elections. In: Proceedings of the 2005 ACM workshop on Privacy in the electronic society. pp. 61–70. ACM (2005)
12. Pedersen, T.P.: A threshold cryptosystem without a trusted party. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 547, pp. 522–526. Springer (1991)
13. Pereira, O., Rivest, R.L.: Marked mix-nets. In: Brenner, M., et al. (eds.) Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA. Lecture Notes in Computer Science, vol. 10323, pp. 353–369. Springer (2017), https://doi.org/10.1007/978-3-319-70278-0_22
14. Phan, D.H., Pointcheval, D.: OAEP 3-round: A generic and secure asymmetric encryption padding. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 3329, pp. 63–77. Springer (2004)
15. Roenne, P.B.: JCJ with improved verifiability guarantees. In: The International Conference on Electronic Voting E-Vote-ID 2016 (2016)
16. Terelius, B., Wikström, D.: Proofs of restricted shuffles. In: AFRICACRYPT. Lecture Notes in Computer Science, vol. 6055, pp. 100–113. Springer (2010)

A Contract

For simplicity at present, we have described some functionality with comments and proofs of correct mixing are excluded from the contract. We stress that it

is straightforward to extend this sketch to the full contract but we omit the fine details.

```
pragma solidity >=0.3.0;

contract controlled { ///A contract which remembers the initial creator and allows functions to be restricted to the initial creator
  address public electionCouncil;

  function controlled() public {
    electionCouncil = msg.sender;
  }

  modifier onlyElectionCouncil {
    require(electionCouncil == msg.sender);
    _;
  }
}

///This contract implements the voting scheme described previously
///in the paper
contract Election is controlled {

     uint256 public groupOrder
         =21888242871839275222246405745257275088548364400416034343698204186575808495617 ;


  function addEc (bytes32 point1x, bytes32 point1y, bytes32 point2x, bytes32 point2y) private returns (bytes32, bytes32) {
    bytes32 ret_1;
    bytes32 ret_2;

    assembly {
      let size := mload(0x40)
      mstore(size, point1x)
      mstore(add(size, 32), point1y)
      mstore(add(size, 64), point2x)
      mstore(add(size, 96), point2y)

      let res := call(1000, 6, 0, size, 128, size, 64)
      ret_1 := mload(size)
      ret_2 := mload(add(size, 32))
    }

    return (ret_1, ret_2);
```

```
41    }
42
43    function multiEc (bytes32 point1x, bytes32 point1y, bytes32
          scaler) private returns (bytes32, bytes32)  {
44      bytes32 ret_1;
45      bytes32 ret_2;
46
47      assembly {
48        let size := mload(0x40)
49        mstore(size, point1x)
50        mstore(add(size, 32), point1y)
51        mstore(add(size, 64), scaler)
52
53        let res := call(50000, 7, 0, size, 96, size, 64)
54        ret_1 := mload(size)
55        ret_2 := mload(add(size, 32))
56      }
57
58      return (ret_1, ret_2);
59    }
60
61    function negateScalaEc(bytes32 scala) private view returns
          (bytes32){
62      return bytes32(groupOrder-(uint256(scala)\%groupOrder));
63    }
64
65    function negatPointEc(bytes32 point1x, bytes32 point1y)
          private returns (bytes32,bytes32) {
66      uint256 con = groupOrder-1;
67      return multiEc(point1x, point1y, bytes32(con));
68    }
69
70    struct groupElement{
71      bytes32 x;
72      bytes32 y;
73    }
74
75    struct ElGamalCiphertext{
76    //ElGamal Ciphertext
77      bytes32 c1_x;   //g^r
78      bytes32 c1_y;
79      bytes32 c2_x;   //y^r v
80      bytes32 c2_y;
81    }
82
83    groupElement public pk;         //y = g^x
84
85    string public question;    //The question of the election
86    uint public totalVoted;
87
```

```
88    uint public maxNumVoters;
89    uint public numVoters;
90
91    uint public numTellers;
92
93    uint public deposit;
94    address public charity;
95
96    mapping(address => bool) public eligible;
97    mapping(address => bytes32[]) public vote;
98    mapping(uint => bytes32) public pkSharesStorage;
99
100   enum State { SETUP, VOTE, TALLYING, PROVING, FINISHED }
101   State public state;
102
103   modifier inState(State s) {
104     require(state == s);
105     _;
106   }
107
108   /// Create a new election
109   function Election(string elctionQuestion, bytes32[]
          pkShares, uint numberVoters, uint eldeposit, address
          elcharity) public  {
110     state = State.SETUP;
111     question = elctionQuestion;
112     deposit = eldeposit;
113     charity = elcharity;
114     require(pkShares.length %2 ==0 && pkShares.length >= 2);
115     var (temp1, temp2) = (pkShares[0], pkShares[1]);
116     pkSharesStorage[0] = pkShares[0];
117     pkSharesStorage[1] = pkShares[1];
118     for(uint i = 2; i < pkShares.length; i=i+2){
119       pkSharesStorage[i] = pkShares[i];
120       pkSharesStorage[i+1] = pkShares[i+1];
121       (temp1, temp2) = addEc(temp1, temp2, pkShares[i],
              pkShares[i+1]);
122     }
123     numTellers = pkShares.length/2;
124     pk = groupElement({x: temp1, y:temp2});
125
126     maxNumVoters = numberVoters;
127     }
128
129   function designateVoters(address[] voterRoll) public
                        onlyElectionCouncil inState(State.SETUP)
          {
130     require(maxNumVoters >= voterRoll.length + numVoters);
131     numVoters = numVoters + voterRoll.length;
132     for(uint i=0; i<voterRoll.length; i++) {
```

```
133     eligible[voterRoll[i]] = true;
134     }
135   }
136
137   function open() inState(State.SETUP) public
          onlyElectionCouncil  {
138     state = State.VOTE;
139   }
140
141   //cipher is expected to contain 4 elements and proofs is
          expected to contain n*2 (where n is the number of
          elements)
142   function Vote(bytes32[] cipher, bytes32[] proofs) public
          inState(State.VOTE)  {
143       require( //We should really change everthing else to
              require or we will take money from ineligable
              voters
144             msg.value >= deposit,
145             "In sufficent money deposited."
146         );
147
148     //Check the data is formatted as expected
149     require(cipher.length == 4);
150     require(proofs.length == 2);
151     //Prepare ciphertext
152     if(eligible[msg.sender]){ //Caller (voter) must be
            eligable and this must be there first time
153       if(Proof(cipher, proofs)){ //Check proof, and if it
              passes add the vote
154         vote[msg.sender] = cipher;
155         eligible[msg.sender] = false;
156         totalVoted++;
157       }
158     }
159   }
160
161   function Proof (ElGamalCiphertext cipher, bytes32 challenge
          , bytes32 response) private returns (bool) {
162     //We need to check the hash
163     var temp = [bytes32(hex"00"), bytes32(hex"00"),bytes32(
            hex"00")bytes32(hex"00")];
164     (temp[0], temp[1]) = multiEc(cipher.c1_x, cipher.c1_y,
            negateScalaEc(challenge));
165     (temp[2], temp[3]) = multiEc(hex"
            0000000000000000000000000000000
166     0000000000000000000000000000001", hex"
            0000000000000000000000000000000000000000
167     0000000000000000000000002", response);
168     (temp[0], temp[1]) = addEc(temp[0], temp[1], temp[2],
            temp[3]);
```

```
169
170     bytes32 expectChallenge = keccak256(hex"
            0000000000000000000000000000
171     0000000000000000000000000000000000000001", hex"
            0000000000000000000000000000000000000000
172     00000000000000000000000002", cipher.c1_x, cipher.c1_y,
173     cipher.c2_x, cipher.c2_y, temp[0], temp[1], temp[2], temp
            [3]);
174     return(challenge == expectChallenge);
175   }
176
177   function close() inState(State.VOTE) public
          onlyElectionCouncil  {
178     state = State.TALLYING;
179   }
180
181   bool claimUnderway = false;
182   address targetsAddress;
183   address claimerAddress;
184
185
186   //Claim
187   function claim(address target, ElGamalCiphertext cipher)
          public returns (bool) {
188         claimUnderway = true;
189         targetsAddress = target;
190         claimerAddress = msg.sender;
191   }
192
193   function processClaim(bool correct) public
          onlyElectionCouncil returns (bool) {
194         claimUnderway = false;
195         if(correct){ //For simplicity and cost we assume the
                PET occurs off chain
196             claimerAddress.transfer(deposit/2);
197             charity.transfer(deposit/2);
198         }
199   }
200
201   //Tally
202   function tally(bytes32[] result) inState(State.DECRYPT)
          public onlyElectionCouncil returns (bool)  {
203     state = State.PROVING; //Since the results are never used
            onchain inputting is sufficent
204   }
205
206   function prove(uint[] proofs) inState(State.PROVING)
          public onlyElectionCouncil returns (bool)  {
207     state = State.FINISHED; //Since the proofs are never used
            on chain inputting them is sufficent
```

```
208     }
209 }
```

Listing 1.1. Ethereum Contract