How to Vote Privately Using Bitcoin

Zhichao Zhao[*]
zczhao@cs.hku.hk

T-H. Hubert Chan[*]
hubert@cs.hku.hk

ABSTRACT

Bitcoin is the first decentralized crypto-currency that is currently by far the most popular one in use. The bitcoin transaction syntax is expressive enough to setup digital contracts whose fund transfer can be enforced automatically.

In this paper, we design protocols for the bitcoin voting problem, in which there are n voters, each of which wishes to fund exactly one of two candidates A and B. The winning candidate is determined by majority voting, while the privacy of individual vote is preserved. Moreover, the decision is irrevocable in the sense that once the outcome is revealed, the winning candidate is guaranteed to have the funding from all n voters.

As in previous works, each voter is incentivized to follow the protocol by being required to put a deposit in the system, which will be used as compensation if he deviates from the protocol. Our solution is similar to previous protocols used for lottery, but needs an additional phase to distribute secret random numbers via zero-knowledge-proofs. Moreover, we have resolved a security issue in previous protocols that could prevent compensation from being paid.

1. INTRODUCTION

Private e-voting is a special case of secure multi-party (MPC) computation [13] which allows a group of people to jointly make a decision such that individual opinion can be kept private. Researchers have worked on designing voting protocols that do not rely on trusted third parties, while still provide, among other features, anonymity and verifiability [26].

However, the MPC framework can only guarantee that the outcome is received by everyone, whose privacy is also protected. However, when the decision is financially related, it is not obvious how to ensure that the outcome is respected. For instance, a dishonest party may simply run away with his money.

Bitcoin [23] provides new tools for tackling this problem. Although it was originally intended for money transfer, surprisingly it can also be used to enforce a contract such that money transfer is guaranteed once the outcome is known, without the need of a trusted third party.

Developing protocols based on bitcoin has been considered by many researchers. For instance, a lottery protocol [7] using bitcoin was proposed, in which a group of gamblers transfer all their money to a randomly selected user among them. General secure multi-party computation protocols [11, 20] were considered, in which bitcoin provides a way to penalize dishonest users.

Our Problem. We study the *bitcoin voting problem.* There are n voters $P_1, P_2, \ldots, P_n$, each of which wishes to fund exactly one of two candidates A and B with 1Ƀ[1]. The winning candidate is determined by majority voting (assuming n is odd) and receives the total prize nɃ. The voting protocol should satisfy the following basic properties:

- *Privacy and Verifiability.* Only the number of votes received by each candidate is known, while individual votes are kept private. However, each voter can still prove that he follows the protocol. For instance, no voter can vote for the same candidate twice.
- *Irrevocability.* Once the final outcome of the voting is revealed, the winner is guaranteed to receive the total sum nɃ. No voter can withdraw his funding even if the candidate he voted for does not win.

In order to incentivize voters to follow the protocol, each voter needs to put extra bitcoins in the system as deposit, which will be refunded if he follows the protocol, but will be used as compensation (for other voters and/or the candidates) if he deviates from the protocol. The candidates A and B also need to participate in the bitcoin system such that the winner can collect the prize, but each of them just needs a bitcoin address without the need to own bitcoins initially.

1.1 Our Contributions

In this paper, we design protocols for the bitcoin voting problem described above. The design of our protocols have similar aspects with the lottery protocols [7, 11], in which each party generate a random integer $R_i \pmod{n}$, and the winner is simply $\sum_i R_i \pmod{n}$ (whose distribution is uniform, as long as at least one party generate his randomness correctly). In the voting problem, each voter P_i has a private vote $O_i \in \{0, 1\}$ (where 0 means candidate A and 1 means B), and the sum $\sum_i O_i$ reveals the winning candidate. However, we cannot directly use the lottery protocols to reveal the sum of votes, because the random integer R_i will be finally revealed to everyone. In the voting problem, we wish to protect the privacy of individual voters. Our first contribution is to design a protocol for the voters to commit to their masked votes.

Vote Commitment. This part of the protocol does not use the bitcoin network. On a high level, the n voters participate in a distributed protocol (without a trusted third party) to generate n random numbers R_i's summing to 0 (mod N, for some large enough $N > n$) such that for each i, R_i is known only by voter P_i, who commits to both R_i and the masked vote $\widehat{O}_i := O_i + R_i$. Zero-knowledge-proofs [10] are used to guarantee that each voter follows the protocol without revealing R_i and $\widehat{O}_i$. In particular, each P_i proves to everyone that $\widehat{O}_i - R_i \in \{0, 1\}$ to prevent double voting. The protocol ensures that as long as at least one voter gen-

[*]The University of Hong Kong

[1]We use the latex code from [6] to generate the bitcoin symbol Ƀ.

erates his randomness properly, the R_i's will be uniformly random numbers summing to 0. Hence, revealing all the voters' masked votes $\widehat{O}_i$ can determine the number of votes for each candidate without compromising the privacy of individual voters.

The next part of the protocol utilizes the existing bitcoin system for the voters to reveal their $\widehat{O}_i$'s. Moreover, the bitcoin system also guarantees that the winner can receive the prize and any voter that does not reveal his masked vote is penalized.

Vote Casting. In Section 3.2, as a warm up, we use the *claim-or-refund* technique as in [11] to design a protocol in which the voters reveal their masked votes sequentially in the order $P_1, P_2, \ldots, P_n$. If every voter reveals his masked vote, the winner is determined, and can collect his prize. Moreover, the first voter that does not reveal his masked vote before some specified deadline will cause the protocol to be terminated, but needs to pay a compensation to each voter that has already revealed his masked vote. However, this protocol takes $\Theta(n)$ rounds in the bitcoin system, and all the transactions can be described in $\Theta(n^2)$ bytes.

In Section 3.3, we use the idea of a *joint transaction* as in [7, 20] to let voters reveal their masked votes together. Hence, only a constant number of bitcoin rounds are needed, and all the transactions can be described in $O(n)$ bytes. Moreover, the *timed-commitment* technique also allows more flexible compensation rules. For instance, if a voter P_i does not reveal his masked vote, his deposit can be used to compensate both candidates and/or all other voters.

Previous Security Issue. However, as pointed out in [7], the way timed-commitment is used presents a serious security flaw because the compensation is paid with a transaction whose creation depends on the hash of a joint transaction that has not been confirmed yet. In the protocol described in [7], an adversarial party could create an alternate joint transaction with a different hash by re-signing it with a different (valid) signature. If this alternate joint transaction is confirmed on the blockchain instead of the original version, the transaction used for compensation will become invalid.

Resolving Issue with Threshold Signature Scheme. The previous issue is that the joint transaction is signed by each voter individually, and hence any voter can produce another joint transaction with a different hash by re-signing with a different signature. Another major contribution of the paper is that we resolve this issue by using a threshold signature scheme [15] in which a valid signature can only be produced by all n voters together. In particular, a different signature cannot be efficiently produced for a previously signed transaction, unless all n voters agree to re-sign it. Hence, an adversary can no longer perform the previous attack.

1.2 Related Work

Researchers have designed e-voting protocols [26, 21], which make use of cryptographic tools. Efficiency, verifiability and privacy are the most important concerns in such protocols. Other features have also been considered, such as deniablility and receipt-freeness, which prevents vote-buying.

The bitcoin protocol [23] has inspired many lines of research since its introduction in 2008. Some researchers have worked on identifying the protocol's weakness [8]. Other researchers have designed new crypto-currency with more features. For instance, zero-knowledge proofs and accumulators have been used to improve the currency's anonymity [22, 9]. In [9], a new system is designed such that the *proof-of-work* to mine new coins is achieved by storage consumption, as opposed to computation power in bitcoin.

Another line of research, including this paper, is to design protocols that are compatible with the existing bitcoin system. Since bitcoin is still by far the most popular crypto-currency, protocols that can be deployed in the current bitcoin system have the most practical impact.

Basic functionalities can be implemented in the bitcoin system that can serve as building blocks for more complicated protocols. In [11], the *claim-or-refund* mechanism is designed to enforce the promise made by party P that that he will pay party Q a certain amount, provided that Q (publicly) reveals a certain secret before a certain time. As utilized in this paper, this mechanism allows a protocol in which parties reveal their secrets sequentially such that the first party that deviates from the protocol has to compensate the parties that have already revealed their secrets.

In [7], the *timed-commitment* mechanism is designed to enforce the promise made by party P that he will pay party Q a certain amount, unless he (publicly) reveals a certain secret before a certain time. In [20], the timed-commitment mechanism is extended for multiple-parties to reveal their secrets together. However, as mentioned in [7], their implementation of the timed-commitment mechanism has a security issue such that an adversary could prevent the compensation from being paid even when a party does not reveal his secret before the deadline. This issue and the way we resolve it are explained in details in Section 3.3.

In this paper, we also use zero-knowledge-proofs which, loosely speaking, allows one party to prove the correctness of a statement to another party, without revealing additional information. It is known [18] that any statement that can be represented by a boolean circuit can be proved efficiently by zero-knowledge-proof. Moreover, zero-knowledge-proof can be made non-interactive [24, 12, 10].

2. PRELIMINARIES

2.1 Bitcoin

Bitcoin [23] is a peer-to-peer digital currency system proposed in 2008. We summarize features of bitcoin as in [7]. The system consists of *transactions*, which are stored in *blocks*. Each peer maintains a chain of blocks known as the *blockchain*. The longest chain is accepted by all peers, which we assume to be publicly known without any delay. The process of extending the blockchain is called *mining*, which requires an enormous amount of computing power. The *miner* is incentivized by receiving profit from the special *coinbase* transaction.

Transaction. A bitcoin transaction consists of (possibly) multiple *inputs*, *outputs* and an optional *locktime*. The transaction id txid is the hash of all the transaction's contents. Each input contains a reference to a previous transaction in the form of its hash and *index* (indicating which output of the previous transaction), and an *input-script*. Each output contains a value (indicating the amount of bitcoin) and an *output-script*. Input and output scripts are used for validation of transactions. An output script serves as a validation program. An input-script serves as parameters (e.g., signature of the current transaction) for the program of the output script in the previous transaction. The optional locktime specifies the earliest time at which the transaction can be put on the blockchain. A *simplified transaction* is a transaction excluding the input script. For a transaction T, we denote its simplified transaction by $[T]$.

Validation. The output of a transaction can serve as only one input in a future transaction on the blockchain. If an output is referred (as input) in another transaction on the blockchain, we call it *spent*, otherwise we call it *unspent.* To validate a transaction T_x, a miner checks that all its inputs are unspent, and the total input value is at least the total output value. Any difference is collected by the miner as transaction fee, after he performs intensive computation to include the transaction on the blockchain. However, for simplicity, we assume in this paper that all transaction fees are zero. Then, for each input-script σ and the corresponding referred output script π, the miner runs the validation program $\pi(\sigma)$. In the simplest case σ is a signature on $\mathsf{hash}([T_x])$, while π verifies its validity. For a detailed description of bitcoin script and its power, please refer to the bitcoin wiki on contracts [3]. For simplicity of description, we treat π as a function that returns boolean values $\{0, 1\}$.

When we say a user owns a bitcoin, we mean that he knows a private key corresponding to an unspent output. When we talk about *address*, we mean some (hash of a) public key. We pay to an address by making a transaction with an output-script requiring the corresponding private key.

Example. We use the diagram notation from [7] to denote bitcoin transactions. In Figure 1, there is some user A associated with the public-private key pair $(\mathsf{pk}, \mathsf{sk})$. Transaction X essentially transfers 5Ƀ to user A. In the output-script of X, $\mathsf{Verify}_{\mathsf{pk}}(\sigma)$ means that the following transaction referring to X needs to be verified with a signature corresponding to the public key pk. Note that $\mathsf{Verify}_{\mathsf{pk}}$ implicitly takes another input which is the hash of the following transaction to be verified, e.g., $\mathsf{hash}([Y])$. This is done automatically by the bitcoin script system, and hence we omit it from the parameters.

Suppose user A wishes to use the 5Ƀ from Transaction X and another 5Ƀ from a different source. User A creates Transaction Y with two input transactions, each of which contributes 5Ƀ. In order for transaction Y to be accepted in the blockchain, in the input-script referring to Transaction X, user A needs to provide a signature for Transaction Y signing with the private key sk. The ownership of the 10Ƀ is specified in the output-script, and the locktime t means that this transaction cannot be placed on the blockchain before time t. If user A spends the 5Ƀ from Transaction X with another transaction that appears on the blockchain before time t, Transaction Y will not be accepted afterwards.

We use arrows with solid lines to indicate the normal flow of bitcoin between transactions when parties follow the planned protocol. We use dashed lines to indicate the flow of bitcoin between transactions when some party deviates from the planned protocol, in which case the transactions could represent penalty payment.

2.2 Security Model

In this paper, we assume that the blockchain is unique (no branching) and one block is grown at a fixed time interval known as *round.* The only ways to interact with the blockchain are submitting transactions and reading transaction histories. No one can affect the blockchain by other

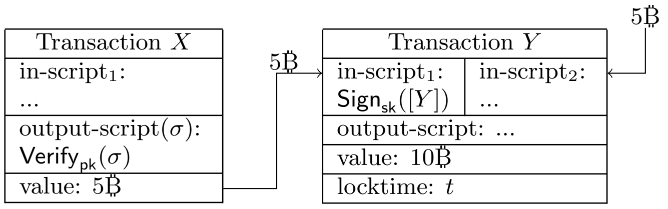


Figure 1: Example Transactions.

"nonstandard" ways. We assume that the blockchain is always publicly accessible. However, we do not assume such access is private. At the time a transaction is submitted, it is publicly known. A submitted transaction may or may not appear on the blockchain, depending on its validity. If a transaction is valid, it will be *confirmed* one round later, otherwise it will be *rejected.* If conflicting transactions are submitted in the same round, only one of them will be confirmed. Hence, if an adversary is able to create a different input-script for a (newly submitted) unconfirmed transaction. He could submit another transaction with the modified input-script within the same round. In such case, either transaction could appear in the final blockchain.

One major threat to the bitcoin network is caused by signature *malleability* [5]. It means that one (users and miners) has the ability to alter a transaction's signature in a way such that: (1) it is still valid, and (2) the hash of the transaction changes. Note that malleability is not a serious problem for normal users but a problem for (1) bitcoin trading companies; and (2) designing bitcoin contracts. The former may use hash of transactions (txid) to trace the status of a transaction, using a computer program. Failing to do so may result in incorrect accounting. In the design of bitcoin transactions [7, 11], one might rely on the hash of an unconfirmed transaction.

There are two major sources of malleability: (1) adding an extra operator in the input-script. (2) changing the signature of a message directly. The former is a design weakness of the original bitcoin protocol, and is being fixed by BIP 62 [25]. While the latter weakness is due to the particular signature scheme. E.g., given a valid ECDSA signature (r, s), $(r, -s)$ is also a valid signature. BIP 62 also fixed this kind of malleability by having a rule to accept only one of them. These fixes are gradually being implemented in the bitcoin network, and will be forced with the block version 4 [1]. Hence, we assume that without the corresponding private key, one cannot produce modify a signed transaction with a different input-script. Note that with the correct private key, one can always sign the message again and get a different signature.

Another way to deal with malleability is to assume that the hash of a transaction is calculated from its simplified transaction only, instead of the complete signed transaction. For example, [20] made this assumption. However, they argue that their protocol can still be implemented in the current bitcoin network by using secure multi-party computation. However, as pointed out in [7], it has the weakness that, even with BIP 62 deployed, a participant can change the hash of a joint transaction (with multiple inputs and outputs) to broadcast a different version of the transaction, and as a result, all parties might lose their coins. Such method is not secure and should not be used. We propose new protocols that are realizable on the current bitcoin network without any modification. The details are in Section 3.1.

Peers (voters and candidates) need to communicate in the protocol. We assume there exists a secure private channel between any pair of participants. We also assume there exists a public broadcast channel among all participants.

2.3 Cryptographic Primitives

Zero-knowledge proof. We utilize the zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARKs) [10]. On a high level, it proves a statement without revealing the corresponding witness such as "I know an x such that $\mathsf{sha256}(x) = y$". The use of zk-SNARKs is to guarantee that the voters cannot deviate from the protocol. We use the definition for zk-SNARKS from [10].

Given a boolean circuit $C : \{0,1\}^n \times \{0,1\}^h \to \{0,1\}$, a binary relation is defined as $\mathcal{R}_C = \{(x,w) \in \{0,1\}^n \times \{0,1\}^h : C(x,w) = 1\}$, and its language is $\mathcal{L}_C = \{x \in \{0,1\}^n : \exists w, C(x,w) = 1\}$. Here, for every $x \in \mathcal{L}_C$, a *witness* for x is a w such that $C(x,w) = 1$. Zero-knowledge proof allows a party to convince others that he knows a secret witness w such that $C(x,w) = 1$, where x is known by all parties. Note that this is also a proof that $x \in \mathcal{L}_C$.

Informally, zk-SNARKs is a triple of (randomized) algorithms (G, P, V):

- **Generator** $G : (1^\lambda, C) \mapsto (\mathsf{pk}, \mathsf{vk})$. Taking as inputs a security parameter 1^λ and a boolean circuit C, it outputs a proving key pk and a verification key vk. Both keys are publicly known and can be used any number of times.
- **Prover** $P : (pk, x, w) \mapsto \eta$. Taking as inputs a proving key pk, $x \in \mathcal{L}_C$ and a corresponding witness w, it outputs a proof η.
- **Verifier** $V : (vk, x, \eta) \mapsto \{0,1\}$. Taking as inputs a verification key vk, x and a proof η, it outputs 1 if it is convinced that $x \in \mathcal{L}_C$.

We informally summarize the properties satisfied by zk-SNARKs. For the formal description, please refer to [10].

- *Completeness.* Given $(x,w) \in \mathcal{R}_C$, the prover P can produce a proof η such that the verifier V accepts (x,η) with probability 1.
- *Soundness.* No polynomial-time adversary can generate a proof η for $x \notin \mathcal{L}_C$ such that the verifier V accepts (x,η) with non-negligible probability.
- *Efficiency.* The randomized algorithms G, P and V all run in time polynomial in the sizes of their inputs and some given security parameter.
- *Zero-knowledge.* There exists a (randomized) polynomial simulator S, who first generates key pairs $(\mathsf{pk}, \mathsf{vk})$, such that for any $x \in \mathcal{L}_C$ chosen by a polynomial adversary, S generates a proof for x. The proof (and key pairs) generated by S is computationally indistinguishable from honestly generated ones.
- *Proof of knowledge.* There exists a polynomial-time extractor E such that if a polynomial-time prover $\widetilde{P}$ convinces the verifier V to accept some $x \in \mathcal{L}_C$ with non-negligible probability, then given oracle access to $\widetilde{P}$, the extractor can produce a witness w such that $(x,w) \in \mathcal{R}_C$ with non-negligible probability.

Commitment Scheme. A commitment scheme [16] is a two-party protocol between a *sender* and a *receiver*. A commitment scheme consists of (randomized) algorithms Commit and Open, and involves two phases, a *committing phase* and an *opening phase*.

- In the committing phase, the sender commits to a secret value m by generating $(c,k) \leftarrow \mathsf{Commit}(m)$, where c is the *commitment* and k is the *opening* key. The commitment c is sent to the receiver, while m and k are secret.
- In the opening phase, the sender reveals k to the receiver, who gets the original secret value $m \leftarrow \mathsf{Open}(c,k)$. $\mathsf{Open}(c,k)$ returns $\perp$, if the pair (c,k) is considered to be invalid.

We informally summarize the properties of a commitment scheme. Please refer to [14] and [16] for formal description.

- *Correctness.* An honestly generated $(c,k) \leftarrow \mathsf{Commit}(m)$ can always reveal the secret $m \leftarrow \mathsf{Open}(c,k)$.
- *Binding.* No polynomial-time adversary can generate a commitment c and two opening keys k and k' such that both (c,k) and (c,k') are opened by Open to different valid messages with non-negligible probability.
- *Hiding.* For any messages m and m', the distributions of the corresponding commitments c and c' generated by Commit are computationally indistinguishable.

Compatibility with Bitcoin Protocol. As mentioned in [7], the current implementation of bitcoin protocol forbids some functions in the scripting language. In particular, calculating hashes, verifying signatures and simple arithmetic operations are the only supported operations. We follow their method and define commitment schemes using hash functions. Suppose λ is a publicly known security parameter. To commit to a non-negative integer m, $\mathsf{Commit}(m) = (\mathsf{hash}(y), y)$, where y is a random string of length $(\lambda + m)$. Here, hash can be any hash function supported by the bitcoin protocol; in this paper, we use $\mathsf{sha256}$. Moreover, $\mathsf{hash}(y)$ is the commitment and y is the opening key. Then, $\mathsf{Open}(h, y)$ returns $|y| - \lambda$ if $h = \mathsf{hash}(y)$, and $\perp$ otherwise.

3. OUR PROTOCOLS

We present our protocol for bitcoin voting. Apart from a one-time setup using zk-SNARKs [10], our protocol works in a peer-to-peer fashion without a centralized server. The setup is run only once, and can be used for multiple votings. Suppose N is the least power of 2 that is greater than the number n of voters. We use $\mathbb{Z}_N$ to denote the group of integers modulo N equipped with modulo addition. We choose N to be a power of 2 to simplify the implementation of modulo arithmetic.

On a high level, our protocol consists of two components.

- **Vote Commitment.** In this phase, each voter P_i has a private vote $O_i \in \{0,1\}$, where 0 indicates candidate A and 1 indicates candidate B. Each voter P_i receives a secret random number R_i, which is constructed in a distributed fashion such that $\sum_j R_j = 0$.

  At the end of this phase, each voter P_i makes commitment C_i to R_i, and commitment $\widehat{C}_i$ to his masked vote $\widehat{O}_i := O_i + R_i$. The commitments C_i and $\widehat{C}_i$ are broadcast publicly, while the underlying values and opening keys remain secret.

  Every participant convinces others that he follows the protocol using zero-knowledge proofs. In particular, everyone is convinced that for each i, the underlying value of the commitment $\widehat{C}_i$ minus that of C_i is either 0 or 1, and the sum of the underlying values of C_i over i is 0. Hence, the sum of the underlying values of $\widehat{C}_i$

over i is the number of votes candidate B receives. The details are in Section 3.1.

- **Vote Casting.** In this phase, the votes are cast using transactions in the bitcoin protocol, which are responsible for revealing the outcome and guaranteeing money transfer to the winning candidate. After each voter P_i reveals his masked vote $\widehat{O}_i$, the outcome $\sum_i \widehat{O}_i$ (the number votes supporting candidate B) is known, and the winning candidate is guaranteed to receive nɃ.
  Moreover, parties that deviate from the protocol are penalized. We have two versions for vote casting, which have different consequences for the penalty and the funding outcome for the candidates when a voter deviates from the protocol.
  (a) The first version is based on the lottery protocol in [11] using a *claim-or-refund* functionality. The voters reveal their masked votes in the order: $P_1, P_2, \ldots, P_n$. If voter P_i is the first to deviate from the protocol, he pays a penalty to each voter that has already revealed his masked vote. Everyone else gets his deposit back, and the protocol terminates while neither candidate A nor B gets any money.
  (b) The second version is an improvement over other protocols [7, 20] using *joint transaction* to incentivize fair computation via the bitcoin system. Each voter P_i places $(1+d)$Ƀ into the bitcoin system, where 1Ƀ is for paying the winning candidate and dɃ is for deposit. If a voter reveals his masked vote $\widehat{O}_i$, he can get back the deposit dɃ. For each voter that does not reveal his masked vote within some time period, his deposit dɃ will be used as compensation. For instance, with $d = 2n$, the deposit can be shared between the candidates A and B. Observe that if at least one voter does not reveal his masked vote, the nɃ for the supposedly winning candidate will be locked into the bitcoin system.
  The details for the two versions are in Sections 3.2 and 3.3.

3.1 Vote Commitment

Recall that we wish to uniformly generate random numbers R_i's that sum to 0 such that for each i, only voter P_i receives R_i. Moreover, the commitments to R_i and $\widehat{O}_i = (R_i + O_i)$ are public. We first give a high level idea of the procedure.

We imagine that there is an $n \times n$ matrix $[r_{ij}]$ whose entries contain elements from $\mathbb{Z}_N$. The protocol can be described in terms of the matrix as follows.

1. For each i, voter P_i generates the i-th row whose sum $\sum_j r_{ij}$ is zero. This can be done by generating $n-1$ random numbers in $\mathbb{Z}_n$ and setting the last one to be the additive inverse of the sum of the first $n-1$ numbers.
2. For each i and j, voter P_i sends the number r_{ij} to P_j via the secret channel.
3. For each i, voter P_i knows the i-th column of the matrix. Hence, he computes and commits both $R_i := \sum_j r_{ji}$ and the masked vote $\widehat{O}_i := O_i + R_i$.

This idea is standard in the literature (for instance, used recently in [19][Section IV.A]). The twist here is that commitment schemes and zero-knowledge proofs are used to ensure that every party follows the protocol, while maintaining the secrecy of the random numbers. The details are given in Figure 2.

Vote Commitment Protocol

This protocol runs among n voters, where for $i \in [n]$, party P_i has secret vote $O_i \in \{0,1\}$. We assume the proving and verification keys for zk-SNARKs are already generated and distributed to all voters. For each $i \in [n]$, the procedure for P_i is as follows.

1. Generate n secret random numbers $r_{ij} \in \mathbb{Z}_N$, for $j \in [n]$, such that they sum to 0.

   For $j \in [n]$, commit $(c_{ij}, k_{ij}) \leftarrow \mathsf{Commit}(r_{ij})$, where k_{ij} is the opening key to the commitment c_{ij}.

2. Generate zero-knowledge proofs that shows $\sum_j r_{i,j} = 0$. Specifically, the circuit C takes two components. The input component is the n commitments, while the witness component is the n corresponding opening keys. The circuit C evaluates to 1 if the opened values sum to 0.

   Broadcast the commitments and zero-knowledge proofs to all voters.

3. Receive commitments and verify the zero-knowledge proofs from all other parties generated in Step 2.

4. For all $j \in [n] \setminus \{i\}$, send to P_j the opening key k_{ij}.

   For $j \in [n] \setminus \{i\}$, wait for the opening key k_{ji} from P_j, and check that $r_{ji} = \mathsf{Open}(c_{ji}, k_{ji}) \neq \perp$.

5. Compute $R_i \leftarrow \sum_j r_{ji}$ and $\widehat{O}_i \leftarrow R_i + O_i$, and commit $(C_i, K_i) \leftarrow \mathsf{Commit}(R_i)$ and $(\widehat{C}_i, \widehat{K}_i) \leftarrow \mathsf{Commit}(\widehat{O}_i)$, where K_i,$\widehat{K}_i$ are the opening keys.

   Broadcast the commitment C_i and $\widehat{C}_i$ publicly.

6. Generate and broadcast publicly the zero-knowledge proofs for the following:
   (a) "$R_i = \sum_j r_{ji}$". This is similar to Step 2.
   (b) "The committed value in $\widehat{C}_i$ minus that in C_i is either 0 or 1." The input part of the circuit is the two commitments C_i and $\widehat{C}_i$, and the witness part is their opening keys. The circuit evaluates to 1 if the opened values differ by 0 or 1 as required.

7. Receive and verify all proofs from other parties generated in Step 6. The protocol terminates.

Figure 2: Vote Commitment Protocol

Security analysis. The security of the vote commitment protocol follows readily from the security of commitment schemes [16] and zero-knowledge proofs from zk-SNARKs [10]. Observe that our zero-knowledge-proofs ensure that each P_i

generate n numbers that sum to 0, but do not ensure that they are generated uniformly at random. However, as long as at least one party generate his random numbers uniformly at random, the resulting R_i's will still be n uniformly random numbers summing to 0. From the definition of commitment schemes and zero-knowledge-proofs, each P_i commits to both R_i and the masked vote $\widehat{O}_i := R_i + O_i$, and convinces everyone that $\widehat{O}_i - R_i \in \{0, 1\}$, while keeping both values secret.

3.2 Vote Casting via Claim-or-Refund

This version of the vote casting protocol is based on the lottery protocol in [11] that makes use of bitcoin transactions to guarantee money transfer. The protocol is not symmetric among the voters, and the voters are supposed to reveal their masked votes in the order: $P_1, P_2, \ldots, P_n$. In order to participate in the protocol, for $i \in [n-1]$, voter P_i needs to place $(i+1)$Ƀ into the system, and voter P_n needs to place $(3n-1)$Ƀ into the system. This protocol requires a linear number of bitcoin rounds, as opposed to constant number of rounds in the protocol that is given in Section 3.3. The protocol in this section guarantees the following:

- If every voter reveals his masked vote, the net effect is that each voter pays 1Ƀ to the winning candidate.
- If voter P_i is the first that does not reveal his masked vote, the net effect is that he pays 1Ƀ to each voter that has already revealed his masked vote. Observe that neither candidate receives any money in this case.

Claim-or-refund. As in [11], we use the claim-or-refund (COR) functionality. Suppose π is some boolean function. Informally, COR consists of several bitcoin transactions, whose effect is to allow a sender P to guarantee that if the receiver Q reveals some secret w such that $\pi(w) = 1$ before some specified time τ, P will transfer a certain amount qɃ to Q. Note that the revealed secret is publicly known afterwards.

The claim-or-refund functionality consists of (potentially) three phases.

(a) *Deposit Phase.* Sender P creates a *deposit transaction* whose output value is qɃ. In the output script, it requires either (i) signatures from both P and Q, or (ii) Q's signature and a value w such that $\pi(w) = 1$. The deposit transaction is kept secret for the time being.
Sender P also creates a (simplified) *refund transaction* with a timelock τ whose input is the deposit transaction just created. The simplified refund transaction is sent to receiver Q, who provides his signature. Sender P also appends his own signature to complete the creation of the refund transaction.
The deposit phase is completed, when sender P broadcast the deposit transaction, and it appears on the blockchain.

(b) *Claim Phase.* Receiver Q can create a *claim transaction* to claim the amount qɃ from the deposit transaction by providing his signature and some w such that $\pi(w) = 1$ in the input-script before time τ.

(c) *Refund Phase.* If Q does not broadcast a valid claim transaction by time τ, P can get back the amount qɃ from the deposit transaction by broadcasting the refund transaction created in (a).

We use following notation to indicate the claim-or-refund protocol as described above: $P \xrightarrow[q,\tau]{\pi} Q$.

Example. In our application, we need a claim-or-refund protocol such that P sends Q an amount of qɃ, if Q reveals the opening key $\widehat{K}$ to the commitment $\widehat{C}$ of some secret $\widehat{O}$ before time τ. We use the notation $P \xrightarrow[q,\tau]{\widehat{O}} Q$ to indicate such a protocol. Recall that we use hashing as the commitment scheme in Section 2.3. When we say Q reveals the secret $\widehat{O}$, we mean Q submits to the bitcoin network a claim transaction whose input-script contains the opening key $\widehat{K}$. The function π in the deposit transaction takes the opening key $\widehat{K}$ and accepts if $\mathsf{hash}(\widehat{K}) = \widehat{C}$. Observe that the secret is revealed as $\widehat{O} = |\widehat{K}| - \lambda$, where $|\widehat{K}|$ is the length of the key and λ is some publicly known security parameter.

We can also have more complicated claiming condition such as Q needs to provide the opening keys to reveal $\widehat{O}_1$ and $\widehat{O}_2$ such that $\widehat{O}_1 + \widehat{O}_2 = 0$. This is denoted as:
$P \xrightarrow[q,\tau]{\widehat{O}_1, \widehat{O}_2 : \widehat{O}_1 + \widehat{O}_2 = 0} Q$.

Vote Casting via COR. We show how the vote casting protocol can be implemented with a sequence of COR instances in Figures 3 and 4. The idea is similar to that in [20]. For n voters, there are $2n$ COR instances. The deposit transactions of the COR instances are placed in the reversed order as the claim transactions are broadcast to reveal the masked votes.

Correctness. The correctness of the protocol is analyzed in the same way as [20]. The voters are supposed to reveal their masked votes in the order: $P_1, P_2, \ldots, P_n$. At the moment just after voters $P_1, P_2, \ldots, P_i$ have revealed their masked votes in the corresponding claim transactions, the net effect is that P_{i+1} has paid each of the previous voters 1Ƀ. Hence, if P_{i+1} does not reveal his masked vote, eventually all outstanding COR instances will expire and the protocol terminates.

On the other hand, if everyone follows the protocol, then at the end all the masked votes can be summed up to determine the winner, who can collect nɃ from P_n.

3.3 Vote Casting via Joint Transaction

The protocol in Section 3.2 requires a linear number of bitcoin rounds. In this section, we give an alternative protocol that only needs constant number of bitcoin rounds. Also, our protocol has the advantage of small total transaction size. The total size of the transactions in the protocol is $\Theta(n)$ bytes, instead of $\Theta(n^2)$ from previous protocol. Loosely speaking, we achieve this by locking all bitcoins involved in a transaction that is jointly signed by all voters. The protocol is symmetric among the n voters. To participate in the protocol, each voter P_i needs $(1+d)$Ƀ, of which 1Ƀ is to be paid to the winning candidate if everyone reveals his masked vote and the remaining dɃ is for deposit that will be used for compensation if P_i does not reveal his masked vote. The *timed-commitment* technique in [7] can be used to handle the deposit and compensation.

The protocol guarantees the following:

- If a voter reveals his masked vote, he can get back the deposit dɃ.
- If every voter reveals his masked vote, the sum $\sum_i \widehat{O}_i$ determines the winner who receives nɃ.
- If at least one voter does not reveal his masked vote, the nɃ originally intended for the winner will be locked. For each voter that does not reveal his masked vote, his deposit will be used for compensation. Here are several options:

Vote Casting: Deposit Phase

Assume that the commitments $\widehat{C}_i$'s to the masked votes $\widehat{O}_i$'s are publicly known, and each P_i knows the opening key $\widehat{K}_i$ for $\widehat{C}_i$. Assuming that n is odd, the winner is B if $\sum_i \widehat{O}_i > \frac{n}{2}$.
Assume that the times $\tau_1 < \tau_2 < \ldots < \tau_n < \tau_{n+1}$ are spaced sufficiently wide apart, for they will be used as locktimes.
The protocol runs as follows.

1. P_n submits the deposit transactions of the following COR instances to the bitcoin network:
$$P_n \xrightarrow[n,\tau_{n+1}]{\widehat{O}_1,\ldots,\widehat{O}_n:A \text{ wins}} A$$
$$P_n \xrightarrow[n,\tau_{n+1}]{\widehat{O}_1,\ldots,\widehat{O}_n:B \text{ wins}} B$$
2. Simultaneously for each $i \neq n$, P_i verifies that the deposit transactions broadcast in the previous step are on the block chain, and broadcasts the deposit transaction of the following COR instance to the bitcoin system:
$$P_i \xrightarrow[2,\tau_n]{\widehat{O}_1,\ldots,\widehat{O}_n} P_n$$
3. Sequentially for i from n down to 2:

   P_i verifies that all deposit transactions broadcast previously have appeared in the blockchain, and broadcasts the deposit transaction of the following COR instance to the bitcoin system:
$$P_i \xrightarrow[i-1,\tau_{i-1}]{\widehat{O}_1,\ldots,\widehat{O}_{i-1}} P_{i-1}$$

Figure 3: Deposit Phase of Vote Casting

(a) For $d = 2n$, the deposit can be shared between candidates A and B. We shall concentrate on this option in this section.

(b) For $d = n$, the deposit can be distributed among all voters in a similar way.

Problems with Joint Transaction in Previous Approach. In [7, 20], a joint transaction is used to temporarily lock down deposits from n parties such that if a party does not reveal his secret before some time, others can get compensated from his deposit. The idea is to use a joint transaction T, whose n inputs are the sources of the n parties' deposits. Each output i of T can be redeemed by one of two ways: (1) a transaction signed by party i revealing his secret, or (2) a transaction signed by everyone.

However, before transaction T is broadcast to the bitcoin network, for each i, a transaction PAY_i signed by everyone that can redeem the i-th output (as compensation after some locktime) needs to be created first. The issue is that to create the transaction PAY_i, $\mathsf{hash}(T)$ of the signed transaction T is required in the current bitcoin protocol. One way to resolve the issue is to modify the current bitcoin protocol

Vote Casting: Claim/Refund Phase

- For $i \neq n$, if before time τ_i, all previous secrets $\widehat{O}_1, \ldots, \widehat{O}_{i-1}$ are revealed, then P_i reveals his secret $\widehat{O}_i$ and use the claim transaction to receive iɃ from P_{i+1}.
- If before time τ_n, all secrets $\widehat{O}_i$ for $i \neq n$ are revealed, P_n reveals his secret $\widehat{O}_n$ and use the claim transactions to receive 2Ƀ from each P_i for $i \neq n$.
- If before time τ_{n+1} all secrets are revealed, the winner is determined and he can use the corresponding claim transaction to receive nɃ from P_n.
- At any time when the locktime of a COR instance has passed, the sender can immediately use the corresponding refund transaction to get his amount back.

Figure 4: Vote Casting: Claim/Refund Phase

such that $\mathsf{hash}([T])$ of the simplified (unsigned) transaction T is used instead.

Another way to resolve the issue is that $\mathsf{hash}(T)$ is computed via secure multi-party computation. After all the transactions PAY_i's are created, then all parties can complete the creation of the transaction T by signing it (in exactly the same way as they compute $\mathsf{hash}(T)$ in the secure multi-party computation). However, in [7], it is mentioned that there is a serious security issue with this approach. An adversarial party can replace his signature with a different valid one to create another version T' of the transaction, whose hash is different from that of T. If he broadcasts this alternate version T' and it appears on the blockchain before the original T, then all the transactions PAY_i's become useless. Hence, the compensation will not be paid even if a party does not reveal his secret before the deadline, but could still claim back his deposit by revealing his secret eventually after a very long time.

We resolve this issue by using an (n,n)-threshold signature scheme [15] for a group of parties to sign a message together, such that no adversary can re-sign the same message again without all parties' permissions.

Description of Vote Casting Protocol.
Key Setup. We use the threshold signature scheme [15] for Elliptic Curve Digital Signature Algorithm (ECDSA). The n voters jointly generate a group address such that voter P_i learns the group public key $\widehat{\mathsf{pk}}$ and his share $\widehat{\mathsf{sk}}_i$ of the private key. Observe that no party knows the underlying secret key $\widehat{\mathsf{sk}}$, which could be reconstructed from all parties' secret shares. We use $(\mathsf{pk}_i, \mathsf{sk}_i)$ to denote the address of P_i in the bitcoin system.
Coin Lock. Eventually, the n voters will sign some transaction JOIN together, whose inputs are contributed by the n voters. We introduce a protocol that locks the contribution from each voter in a state such that only with all voters' permission can it be redeemed. This ensures that only one version of the JOIN transaction can use these coins later. On the other hand, if the protocol ends prematurely and the JOIN transaction is not created successfully, we wish to

let each voter P_i get back his contribution with the transaction BACK_i. The coin lock protocol is described in Figure 5.

Coin Lock Protocol

Each voter locks $(1+d)$₿ into the system, where 1₿ is to fund the winner, and d₿ is for deposit; here, we set $d := 2n$. Each voter P_i does the following:

1. P_i creates a (secret) transaction LOCK_i. Its input is $(1+d)$₿ owned by P_i, and its output is the address of the group public key $\widehat{\mathsf{pk}}$.

   P_i also creates a simplified transaction BACK_i that transfers the money from LOCK_i back to an address pk_i owned by P_i. Note that $\mathsf{hash}(\mathsf{LOCK}_i)$ is embedded in BACK_i, but LOCK_i remains secret.

   P_i broadcasts (simplified) BACK_i to all other voters.

2. On receiving BACK_j for $j \in [n] \setminus \{i\}$, P_i checks that the hash value referred to by its input is not $\mathsf{hash}(\mathsf{LOCK}_i)$. At this point, P_i has only contributed coins to $\widehat{\mathsf{pk}}$ through the transaction LOCK_i, and hence, he can sign anything else using $\widehat{\mathsf{sk}}_i$ without losing money.

3. For each $j \in [n]$, P_i participates in the threshold signature scheme to sign BACK_j using his secret key share $\widehat{\mathsf{sk}}_i$.

4. On receiving the correct signature for BACK_i, P_i is ready to submit LOCK_i to the bitcoin network later.

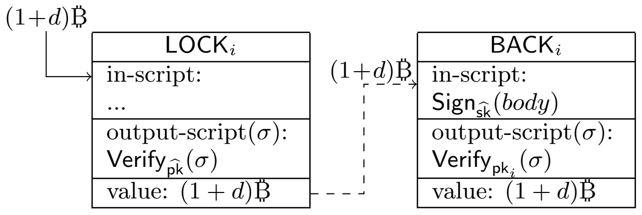


Figure 5: Coin Lock Protocol

Joint Transaction. In the next step, the n voters shall jointly sign a transaction JOIN using the threshold signature scheme, each with his private key share $\widehat{\mathsf{sk}}_i$. The JOIN transaction has n inputs referring to the LOCK_i's, each of which contributes $(1+d)$₿. It has $(n+1)$ outputs, of which $\mathsf{out\text{-}prize}$ delivers n₿ to the winning candidate, while each $\mathsf{out\text{-}deposit}_i$ of the remaining n outputs handles the deposit d₿ of each voter.

Using the timed-commitment technique as in [7], the output $\mathsf{out\text{-}deposit}_i$ can be redeemed by a transaction that either (1) reveals the masked vote $\widehat{O}_i$ and is signed with the key associated with P_i, or (2) is signed with the group signature. Hence, before JOIN takes effect (by appearing in the blockchain), a transaction PAY_i with some timelock that can redeem $\mathsf{out\text{-}deposit}_i$ needs to be created and signed under the threshold signature scheme, in case P_i does not reveal his masked vote and his deposit is used for compensation. The details are in Figure 6.

Joint Transaction Protocol

Assume that the Coin Lock Protocol has been run, and each P_i has created the (secret) transaction LOCK_i, whose hash is publicly known. Suppose $t_1 < t_2$ are times far enough in the future. Each voter runs the following protocol.

1. Each voter generates the same simplified transaction JOIN as follows.
   - It has n inputs, each of which refers to LOCK_i that contributes $(1+d)$₿.
   - It has $n+1$ outputs:
     $\mathsf{out\text{-}deposit}_i$, $i \in [n]$: each has value d₿, and requires either (1) the opening key $\widehat{K}_i$ (revealing $\widehat{O}_i$) and a signature verifiable with P_i's public key pk_i, or (2) a valid signature verifiable with the group's public key $\widehat{\mathsf{pk}}$.
     $\mathsf{out\text{-}prize}$: has value n₿, and requires all opening keys $\widehat{K}_i$'s (revealing the masked votes $\widehat{O}_i$'s) and a signature from the winning candidate (which can be determined from the sum $\sum_i \widehat{O}_i$).

2. The voters jointly sign JOIN using the threshold signature scheme, each with his private key share $\widehat{\mathsf{sk}}_i$. Observe that JOIN has n inputs, each of which requires its own group signature. (See [2] for details.) The signed JOIN is ready to be submitted.

3. Each voter generates, for each $i \in [n]$, the same simplified transaction PAY_i with timelock t_2 whose input refers to $\mathsf{out\text{-}deposit}_i$. The output handles the compensation d₿ if voter P_i does not reveal his masked vote by time t_2. For instance, with $d = 2n$, the compensation can be shared between candidates A and B. The n voters jointly sign PAY_i using the threshold signature scheme.

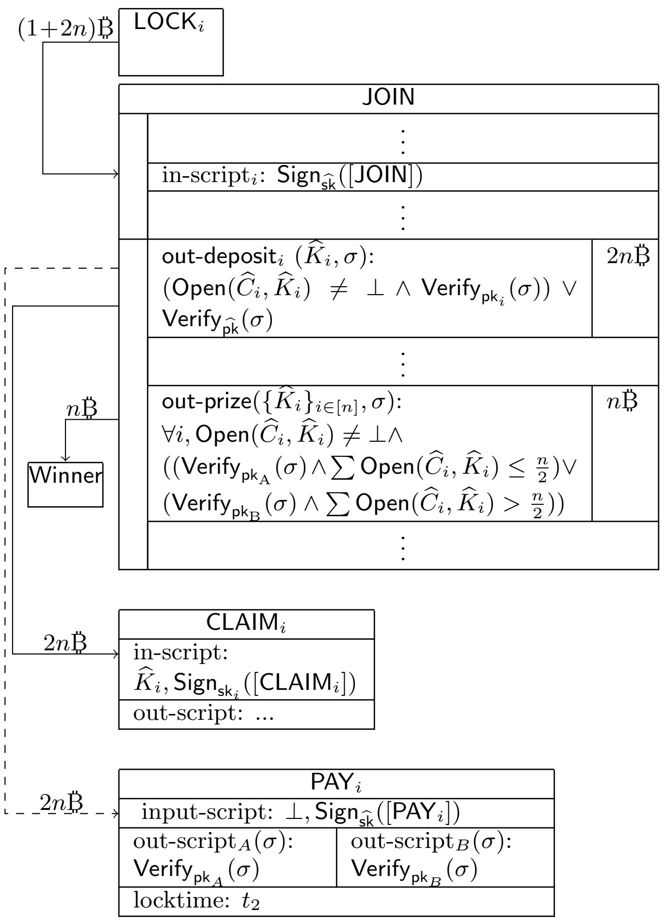
4. Each voter P_i verifies that the above steps have been completed, and submit LOCK_i to the bitcoin system.

5. After all LOCK_i's have appeared on the blockchain, JOIN is submitted to the blockchain.

6. As long as JOIN has not appeared on the blockchain, say by time t_1, any voter P_i can terminate the whole protocol by submitting BACK_i to get back $(1+d)$₿.

Figure 6: Joint Transaction

Outcome Revealing Phase. After JOIN appears on the blockchain, each voter P_i can collect his deposit d₿ (from the output $\mathsf{out\text{-}deposit}_i$ of JOIN) by submitting a CLAIM_i transaction that provides the opening key $\widehat{K}_i$ to reveal his masked vote $\widehat{O}_i$. If all voters have submitted their transactions CLAIM_i's, the winning candidate is determined and can redeem n₿ from $\mathsf{out\text{-}prize}$ with his signature.



$(1+2n)$Ƀ
LOCK_i
JOIN
⋮
in-script$_i$: $\mathsf{Sign}_{\widehat{\mathsf{sk}}}([\mathsf{JOIN}])$
⋮
$\mathsf{out\text{-}deposit}_i$ $(\widehat{K}_i, \sigma)$:
$(\mathsf{Open}(\widehat{C}_i, \widehat{K}_i) \neq \perp \wedge \mathsf{Verify}_{\mathsf{pk}_i}(\sigma)) \vee$
$\mathsf{Verify}_{\widehat{\mathsf{pk}}}(\sigma)$
$2n$Ƀ
⋮
$\mathsf{out\text{-}prize}(\{\widehat{K}_i\}_{i\in[n]}, \sigma)$:
$\forall i, \mathsf{Open}(\widehat{C}_i, \widehat{K}_i) \neq \perp \wedge$
$((\mathsf{Verify}_{\mathsf{pk}_A}(\sigma) \wedge \sum \mathsf{Open}(\widehat{C}_i, \widehat{K}_i) \leq \frac{n}{2}) \vee$
$(\mathsf{Verify}_{\mathsf{pk}_B}(\sigma) \wedge \sum \mathsf{Open}(\widehat{C}_i, \widehat{K}_i) > \frac{n}{2}))$
nɃ
⋮
nɃ
Winner
$2n$Ƀ
CLAIM_i
in-script:
$\widehat{K}_i, \mathsf{Sign}_{\mathsf{sk}_i}([\mathsf{CLAIM}_i])$
out-script: ...
$2n$Ƀ
PAY_i
input-script: $\perp, \mathsf{Sign}_{\widehat{\mathsf{sk}}}([\mathsf{PAY}_i])$
out-script$_A(\sigma)$:
$\mathsf{Verify}_{\mathsf{pk}_A}(\sigma)$
out-script$_B(\sigma)$:
$\mathsf{Verify}_{\mathsf{pk}_B}(\sigma)$
locktime: t_2

On the other hand, if some voter i does not reveal his masked vote, then the nɃ from out-prize cannot be accessed anymore. However, since PAY_i is publicly known, after time t_2, the dɃ from $\mathsf{out\text{-}deposit}_i$ can be redeemed by PAY_i as compensation.

Correctness. After the coin lock protocol, all the transactions LOCK_i's remain secret, while their hashes and the BACK_i's are publicly known. Observe that before the transaction JOIN appears on the blockchain, any voter can terminate the whole protocol without losing any money by submitting BACK_i to the bitcoin system. On the other hand, once JOIN has appeared on the block chain, no voter can terminate the protocol without either revealing his masked vote or losing his deposit dɃ.

4. EXPERIMENT

We describe our implementation of the protocols in Section 3.

Vote Commitment. We have implemented the zero-knowledge-proofs described in Section 3.1. For zk-SNARKs [10], we choose snarkfront [17], which is available online. We use snarkfront to implement the required zero-knowledge proofs. We run the program using a computer with 4G RAM and Intel Core i5-3570 CPU. The key generator typically takes 5 times longer than the time to generate proofs. However, since it only needs to be run once globally, we omit its running time here as it is not a serious performance concern. In Figure 7, we report the time to generate proofs for different number of users. We consider three kinds of proofs: (1) to prove n numbers sum to 0, (2) to prove n numbers sum to the $(n+1)$-st number, (3) to prove the subtraction of a number from another is either 0 or 1.

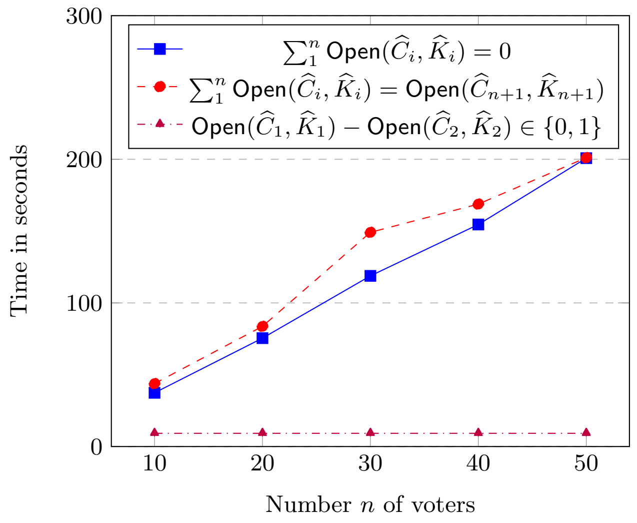


Figure 7: Performance of zk-SNARKS

In zk-SNARKS, the time for verification is only linear in the size of the input (and the security parameter). Typically, it takes less than 0.1 second.

Vote Casting. As a proof of concept, we have executed the protocols in bitcoin (testnet) network [4]. We use bitcoinj Java library to create and send the transactions.

Below we present txid of the transactions. There are 9 voters in our protocol. One may read the full transaction data on chain.so website.

For the protocol using claim-or-refund in Section 3.2, we first create a transaction with multiple outputs, each of which acts as the source address of each claim-or-refund transaction. The source of each claim-or-refund transaction can be found with index $0 - 18$ at:

d3f62d6dfd9722699938a3d7457e23ba786a3e8d14615d128847ad7ca56b7a1a.

All following transactions can be found following the outputs. Another execution in which an adversary terminated the protocol and was punished is here:

8d4031dfa71bf9b1a296b6f67c3cb1d801e899d4ff7d1ee6dd6751622032b60f.

For the protocol using joint transaction in Section 3.3, the JOIN transaction of a successful execution is here:

ca42f58d2a7eadc4360029ea31e6f2224c9b7f47c18ef985a9b477ac869822c7.

All claim transactions can be found by following the outputs.

A JOIN transaction of an unsuccessful (terminated by adversary) execution is here:

6506857a75b1f25b930a923e6bd8274cbccb42339425e21f29cf5ba2ce389738.

All CLAIM_i and PAY_i transactions can be found by following the outputs.

5. REFERENCES

[1] Bitcoin developer reference - block versions. `https://bitcoin.org/en/developer-reference#block-versions`. Accessed: 2015-05-10.

[2] Checksig - bitcoin wiki. `https://en.bitcoin.it/wiki/OP_CHECKSIG`. Accessed: 2015-05-10.

[3] Contracts - bitcoin wiki. `https://en.bitcoin.it/wiki/Contracts`. Accessed: 2015-05-10.

[4] Testnet - bitcoin wiki. `https://en.bitcoin.it/wiki/Testnet`. Accessed: 2015-05-10.

[5] Transaction malleability - bitcoin wiki. `https://en.bitcoin.it/wiki/Transaction_Malleability`. Accessed: 2015-05-10.

[6] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. How to deal with malleability of bitcoin transactions. *CoRR*, abs/1312.3230, 2013.

[7] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 443–458, 2014.

[8] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to better - how to make bitcoin a better currency. In *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, pages 399–414, 2012.

[9] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474, 2014.

[10] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 90–108, 2013.

[11] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 421–439, 2014.

[12] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Erratum: Succinct non-interactive arguments via linear interactive proofs. In *TCC*, 2013.

[13] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988.

[14] Ivan Damgård. Commitment schemes and zero-knowledge protocols. In *Lectures on Data Security, Modern Cryptology in Theory and Practice, Summer School, Aarhus, Denmark, July 1998*, pages 63–86, 1998.

[15] Steven Goldfeder, Rosario Gennaro, Harry Kalodner, Joseph Bonneau, Joshua A Kroll, Edward W Felten, and Arvind Narayanan. Securing bitcoin wallets via a new dsa/ecdsa threshold signature scheme. `http://www.cs.princeton.edu/~stevenag/threshold_sigs.pdf`, 2015.

[16] Oded Goldreich. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA, 2006.

[17] jancarlsson. snarkfront: a c++ embedded domain specific language for zero knowledge proofs. `https://github.com/jancarlsson/snarkfront`.

[18] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 955–966, 2013.

[19] Taeho Jung, Xiang-Yang Li, Zhiguo Wan, and Meng Wan. Privacy preserving cloud data access with multi-authorities. In *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, pages 2625–2633, 2013.

[20] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 30–41, 2014.

[21] Yehuda Lindell. Highly-efficient universally-composable commitments based on the DDH assumption. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, pages 446–466, 2011.

[22] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 397–411, 2013.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system,âĂİ http://bitcoin.org/bitcoin.pdf, 2008.

[24] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252, 2013.

[25] Pieter Wuille. Dealing with malleability - bip62. `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`, 2014.

[26] Xukai Zou, Huian Li, Yan Sui, Wei Peng, and Feng Li. Assurable, transparent, and mutual restraining e-voting involving multiple conflicting parties. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*, pages 136–144, 2014.