Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

# A privacy preserving Ethereum-based E-Voting System

Yunxuan Li

**Course of Study:** Informatik

**Examiner:** Prof. Dr. Ralf Küsters

**Supervisor:** Dipl.-Math. Mike Simon

**Commenced:** July 2, 2018

**Completed:** January 2, 2019

## Abstract

Both E-Voting and blockchain are popular topics these years. Although a blockchain seems to be a promising surrounding for E-Voting systems, there are only a few E-Voting systems based on Blockchain nowadays. Especially Ethereum, which is famous for its decentralized smart contract execution, provides interesting possibilities if one wants to build up a (partly) decentralized E-Voting system. In this thesis, we propose a new Privacy-preserving Ethereum-based E-Voting System (PEES) that provides accuracy, verifiability and privacy. Not like many other Blockchain based E-Voting systems that only saves encrypted votes on-chain to protect Voter's privacy only against the public, PEES uses Zero-Knowledge Proofs to break the connection between a Voter and his vote, to achieve a stronger privacy protection. With a small precondition, PEES protects Voter's privacy not only against the public but also against potential malicious election organizers — a group of people who has the decryption keys for the election.

# Contents


**1 Motivation** **13**

**2 Cryptographic Primitives** **15**
2.1 Cryptographic Hash Function . . . 15
2.2 Public-Key Encryption . . . 15
2.3 Digital Signature . . . 16
2.4 Commitment Scheme . . . 16
2.5 Merkle Tree and Merkle Proof . . . 17
2.6 Secret Sharing . . . 18
2.7 Zero-Knowledge Proof . . . 19

**3 Blockchain** **23**
3.1 Bitcoin . . . 23
3.2 Ethereum . . . 28
3.3 Zcash . . . 30

**4 Privacy Preserving Ethereum-based E-Voting System** **37**
4.1 PEES Overview . . . 37
4.2 Security Goals . . . 39
4.3 PEES in Detail . . . 40
4.4 PEES Protocol . . . 47

**5 Implementation** **51**
5.1 Program Languages and Dependency . . . 51
5.2 Zero-Knowledge Proof Generator . . . 53
5.3 Smart Contract . . . 67
5.4 Web Servers . . . 74

**6 Security Analysis** **89**

**7 Conclusion and Further works** **93**

**Bibliography** **95**

# List of Figures


2.1 A binary Merkle Tree [But15] . . . . . 17
2.2 A sample Merkle Proof [But15] . . . . . 18
2.3 A satisfying R1CS instance [Vit] . . . . . 21

3.1 An example of Blockchain [ZXD+17] . . . . . 25
3.2 An example of fork in a Blockchain [ZXD+17] . . . . . 26
3.3 Mint [SCG+14] . . . . . 32

4.1 Privacy preserving Ethereum-based E-voting System (PEES) overview . . . . . 37
4.2 Vote Coin (VC) - simplified *zerocoin* . . . . . 41
4.3 Variables and functions in Registration Smart Contract . . . . . 44
4.4 Variables and functions in Voting Smart Contract . . . . . 46
4.5 The entire process of an election in PEES . . . . . 49

5.1 Login button . . . . . 75
5.2 Admin Interface . . . . . 75
5.3 When decryption key has been combined . . . . . 80
5.4 Result of an election . . . . . 81
5.5 An election waiting to be opened . . . . . 82
5.6 After registration phase of an election is initialized . . . . . 82
5.7 After voting phase of an election is initialized . . . . . 85
5.8 Candidate list . . . . . 85
5.9 Result is ready . . . . . 88

6.1 Views of the public and of Election Authority (EA) during an election . . . . . 90

# List of Listings


5.1 Format of vote_witness.json . . . 54
5.2 Format of coin_witness and auth_path . . . 54
5.3 Constructor of coin_gadget . . . 55
5.4 Other functions of coin_gadget . . . 56
5.5 Constructor of complete_coin_gadget . . . 57
5.6 Constructor of coin_wrapper_gadget . . . 58
5.7 main function for $\pi_{Coin}$ . . . 59
5.8 adjustment for RSAEncryptionV1_5_Gadget . . . 61
5.9 Format of vote_witness . . . 61
5.10 buildCircuit from class RSAV1_5_Enc . . . 62
5.11 generateSampleInput from class RSAV1_5_Enc . . . 63
5.12 main function for $\pi_{Vote}$ . . . 64
5.13 buildCircuit function for Result_zkp_generator . . . 66
5.14 Registration Smart Contract: Variables . . . 68
5.15 Registration Smart Contract: function endReg . . . 68
5.16 Registration Smart Contract: function Mint . . . 69
5.17 Registration Smart Contract: function GenerateMT . . . 70
5.18 Voting Smart Contract: Variables . . . 72
5.19 Voting Smart Contract: function checkVote . . . 72
5.20 Voting Smart Contract: function Pour . . . 73
5.21 Admin Server: Set up . . . 75
5.22 Admin Server: phase 1 . . . 77
5.23 Admin Server: phase 2 - generate MT . . . 78
5.24 Admin Server: phase 2 - EventListener for treeGenerated . . . 78
5.25 Admin Server: phase 3 - Combine Decryption Key . . . 79
5.26 Admin Server: phase 3 - Compute Result . . . 80
5.27 Client Server: phase 1 - Sign on Token . . . 83
5.28 Voter's Browser: mint VC . . . 83
5.29 Voter's Browser: calculate Merkle Authentication Path . . . 86
5.30 Voter's Browser: submit vote . . . 87

# List of Abbreviations

**Dapp** Distributed Application. 28

**EOA** Externally Owned Account. 29

**E-Voting** Electronic Voting. 13

**EA** Election Authority. 7

**MT** Merkle Tree. 17

**PEES** Privacy preserving Ethereum-based E-voting System. 7

**PKE** Public Key Encryption. 15

**PoS** Proof of Stake. 27

**PoW** Proof of Work. 27

**PRF** Pseudorandom function. 31

**R1CS** Rank-1 Constraint Systems. 20

**VC** Vote Coin. 7

**ZK** Zero-Knowledge. 42

**zk-SNARK** Zero-Knowledge Succinct Non-Interactive Argument of Knowledge. 20

**ZK-Proof** Zero-Knowledge Proof. 19

# 1 Motivation

A free and fair election has always been the cornerstone of democracy. Nowadays, more and more government try to bring digitalization into traditional voting systems. For example, Estonia has begun to hold nation-wide elections via internet since 2005 [Est]. In our daily lives, there are also more and more websites or apps such like Doodle [Doo] or VoxVote [Vox] which is helping people to hold their own voting. Electronic Voting (E-Voting) refers to vote using a standalone electronic voting machines or to cast a ballot via the internet remotely like what Estonia does. In this thesis, E-Voting refers to the second scenario, that is, an E-Voting is a process that allows voters to cast their ballots via the internet from any location in the world.

Although E-Voting has brought us many conveniences, it is hard to keep an E-Voting system as secure as a traditional voting system. Generally, the most desired properties for any voting systems are 1) accuracy: all ballots should be correctly recorded and counted, 2) verifiability: any ballots shouldn't be manipulated without any detection and 3) privacy and anonymity of voters [MAAS13]. However, attacks like hardware Trojan which can totally tamper a voting results [ZAAA+14] and clash attack that allows the voting authorities to replace ballots without being detected [KTV12] are violating those secure requirements of an E-Voting system.

For these attacks, the newly introduced technology Blockchain seems to be a very promising solution. Known as the public ledger, everything that is recorded on a Blockchain is usually public and basically unchangeable, which makes it a perfect tool to record ballots in an E-Voting system. However, as a Blockchain keeps everything public, it badly hurts the voter's privacy, since everyone can easily get the choice of a voter from the Blockchain. One may want to solve this problem by first encrypting the ballot and then save it on a Blockchain, yet the party who has the decryption key (usually the election organizers) can still get access to the choices of every voter, which violates the voter's privacy as well.

Inspired by Zcash, a Blockchain based digital currency that keeps user's personal and transaction data confidential [SCG+14], we propose PEES as a solution to that problem above. PEES is an E-Voting system that uses Blockchain as the ballots recording tool and protects Voter's privacy against both public and election organizers. As an E-Voting system, PEES is expected to provide accuracy, verifiability and privacy (for more explanation please see Section 4.2).

In PEES, all participants can be divided into two parties, namely **Voters** and **EA**. Voters is a group of people who are eligible to vote for an election and EA is a group of Admins who are responsible to organize an election. In an election, each eligible Voter will first get a VC, which is similar to a *zerocoin* in Zcash, from EA. By showing that he has such a VC through a zero-knowledge Proof, each Voter can cast his ballot on the Blockchain privately. After the election has ended, EA is responsible to calculate and publish the result on the Blockchain. Along with the result, a valid proof that shows EA has correctly calculated the result should be published as well, without that, the result should be considered as invalid.

We use Ethereum as the underlying Blockchain since the Smart Contract it provides not only allow us to save each ballot on Blockchain but also enable us to check zero-knowledge Proofs on-chain. In the following, I'll first explain the relevant cryptographic primitives in Chapter 2. An introduction to Blockchain related knowledges, Smart Contract from Ethereum and the creation and usage of *zerocoins* from Zcash can be found in Chapter 3. Then, I'll explain the details of PEES in Chapter 4. After that I'll introduce the implementation of a PEES demo in Chapter 5. At the end, I'll give an informal security analysis in Chapter 6 and conclude this thesis with Chapter 7.

# 2 Cryptographic Primitives

For better understanding the details of PEES, some cryptographic primitives and schemes should be explained first. In the following, I'll shortly explain the cryptographic hash function, public key encryption, signature scheme, commitment scheme, secret sharing and zero-knowledge proof. Some of these primitives are building blocks of other schemes, together, they made it possible to protect Voter's privacy on Blockchain. A concrete algorithm or protocol won't be provided in this thesis, for those detailed information, please refer to [KL07] and [IOSGI06].

## 2.1 Cryptographic Hash Function

A hash function is a function that transform an input data $x$ of arbitrary size to a hash value $h_x$ of fixed size. The output hash value is usually interpreted as a hexadecimal string or digest. A cryptographic hash function is a special kind of hash functions that is **collision resistant**.

A hash function $h$ is collision resistant if no one should be able to find a pair of input data $x, x'$ with $x \neq x'$ such that the hashes of these two inputs are the same, i. e., $h(x) = h(x')$. Notice that collision resistance implicitly indicates that the hash function $h$ is also pre-image resistant, which means, given a hash value $h_x$, it is not possible to find an input data $x$ such that $h(x) = h_x$.

The output of a hash function is usually patternless. A small change in the input date may result in a huge difference in the output hash value. For example, the hash value of "I have a dog" is `19d7bd46fcdf09f7faaf34dbb7a865bc875709953879c19c61dd55bcc4d2d7ab`. With the change of only one letter in the previews input, the result hash may change completely, e. g., if the letter "d" is changed to "c", the hash value of "I have a **c**og" will be changed to `019e58fe7618631c120b2d97a97c432679189e040a06ae853e2f0d845dbdc07a`.

In practice, a cryptographic hash function can be used to shorten the input, protect data integrity or as a building block for some other schemes or structures such as commitment scheme in Section 2.4 or Merkle Tree in Section 2.5.

## 2.2 Public-Key Encryption

An Encryption scheme is aimed to provide data confidentiality. After a message is encrypted, only the party who has a corresponding key can decrypt the message. Public Key Encryption (PKE), also known as asymmetric encryption, is a type of encryption scheme that uses a pair of keys $(pk, sk)$ for encryption and decryption respectively. Unlike private key encryption, or symmetric encryption, where one uses the same key for both encryption and decryption, PKE demands people to use different keys for encrypt and decrypt.

In PKE, each party or participant has its own public/secret key pair. As the key's name indicates, the public key can be widely spread while the secret key should be kept only to the owner himself. When sending a message from one party to another, the sender must encrypt the message under the receiver's public key so that the receiver can decrypt the message using his secret key.

As any encryption schemes, PKE is expected to ensure data confidentiality and the correctness of the decryption. Besides that, since the public key is usually widely spread or even known to everyone, PKE is also expected to prevent one to derive the secret key from the public key, otherwise the encryption is useless. Hence, PKE are usually based on some well-known mathematical problems, such as factoring the product of two large primes for RSA [RSA78] or computing discrete logarithms for ElGamal [ElG85].

In PEES, we use RSAES-OAEP [KS98] to encrypt a Voter's vote under the public key from EA so that no one can read the votes directly from the Blockchain.

## 2.3 Digital Signature

Like a handwritten signature, a digital signature is aimed to provide authenticity of digital messages. A digital signature scheme is also a asymmetric scheme like PKE, whereas a sender $S$ uses his secret key $S_{sk}$ to sign on a message and the receiver verifies the signature using $S$'s public key $S_{pk}$.

A secure digital signature is expected to be unforgeable, that means, any party should not be able to sign on messages in others name, as long as they don't have other party's secret key. Besides that, a valid digital signature is also expected to insure the receiver that (1) the sender cannot deny that he has sent the message (non-repudiation), and (2) the message hasn't been manipulated during the transmission (integrity). However, a digital signature itself doesn't provide data confidentiality.

In PEES, a digital signature scheme is used to issue tokens that are signed by EA to eligible Voters. Later, when the Voter uses this token to generate VC, the Blockchain just needs to check the validity of the signature but doesn't need to interact with EA.

## 2.4 Commitment Scheme

A commitment scheme allows a user committing a certain value or statement without revealing it to the public until he later opens the commitment himself. Additionally, a commitment scheme guarantees that it is impossible for a user to commit to one value but opens it to another one. Typically, a commitment scheme consists of a commit phase and an open phase.

In commit phase, the sender first chooses a randomness $r$ for value $v$ that he wants to commit. Then he calculates the commitment $c$ for $v$ under the randomness $r$. At the end, he sends only the commitment $c$ to the receiver. Later in the open phase, the sender opens his commitment by simply sending the committed value $v$ and the randomness $r$ to the receiver. Upon receiving $v$ and $r$, the receiver first recalculates the commitment $c'$ himself and then checks the equality of $c$ and $c'$.

As mentioned above, a commitment scheme is expected to keep the committed value secret before it is opened, and it should prevent the sender committing to one value but opening it to another one. To that end, the *hiding* and *binding* properties must be achieved. More formally speaking, the

hiding property requires that upon receiving a commitment $c$, the receiver shouldn't be able to find a pair of $v$ and $r$ such that the commitment of $v$ under $r$ equals to $c$. In the meantime, the binding property demands that after the sender commits a value $v$ to $c$ under the randomness $r$, he shouldn't be able to open it to a different value-randomness pair.

Actually, a cryptographic hash function has already met these two requirements. The pre-image resistance guarantees that the receiver shouldn't be able to recover the value $x$ from its hash value $h(x)$, which is exactly what hiding property requires. For binding property, the collision resistance insures that the sender should not be able to find two different value-randomness pairs such that the hash of these two pairs are the same, i. e., the sender could not commit to one value but open to a different one.

Therefore, in PEES, a cryptographic hash function is used as a commitment scheme. That means, every time one wants to commit a value $v$ under randomness $r$, he calculates the hash of $v$ concatenate with $r$, i. e., $c := h(v||r)$, and sends the commitment $c$ to the receiver.

## 2.5 Merkle Tree and Merkle Proof

In cryptography, a Merkle Tree (MT) is a hash-based data structure. It is usually implemented as a binary tree, but it is also possible to be generalized to a $n$-nary tree. A MT is usually built upon a certain data set. As shown in Figure 2.1, the value of each node is the hash value of its two children whereas the value of leaf node is the hash of a certain data-block. In the end, we have just one hash value in the root node, which is called root hash. In practice, a MT, especially the root hash $r$, of a data set is usually publicly known and trusted.

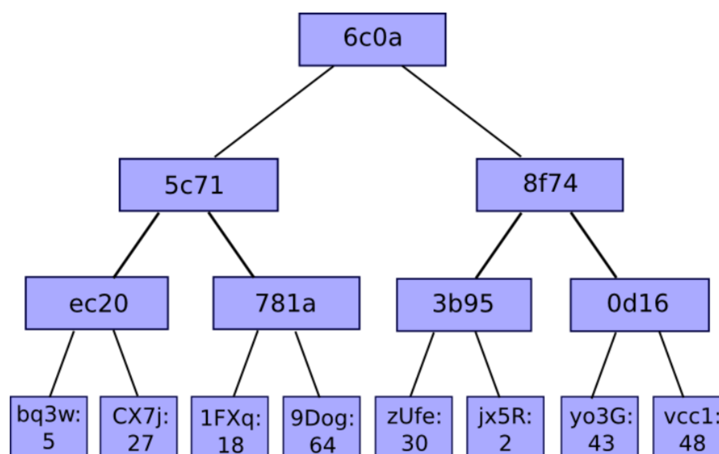



**Figure 2.1:** A binary Merkle Tree [But15]

A typical usage of MT is to verify whether a given data-block has been included in a particular data set or not. Thanks to MT, we don't have to download the whole data set and go through it to see whether the given data-block is in there or not. To do that, we just need a MT over that particular data set and a so-called Merkle Proof.

A Merkle proof consists of a hash value $h$ of a given data-block and a Merkle Authentication Path, which consists of all the hashes going up along the path from $h$ to the root [But15]. Assume there are $m$ hashes in the Authentication Path *path*, to check the validity of the given proof, we first compute the hash of $h$ concatenated with the first hash from *path* and save the new hash in $h_1$, i. e., $h_1 := hash(\, h \,||\, path[1]\,)$. Then we compute the hash of $h_1$ concatenated with the second hash from *path* and save it in $h_2$ and so on, till we get the final hash: $h_m := hash(\, h_{m-1} \,||\, path[m]\,)$. At last, we compare $h_m$ with the root hash of that particular data set.

Suppose we want to proof that the fourth data-block is in a data set, whose MT is shown in Figure 2.1, a sample Merkle proof is then illustrated in Figure 2.2. The green block represents the hash value of the given data-block, the yellow blocks (from bottom to top) builds up the Merkle Authentication Path and the brown blocks (from bottom to top) are all intermediate hashes $h_1$ to $h_3$ respectively.

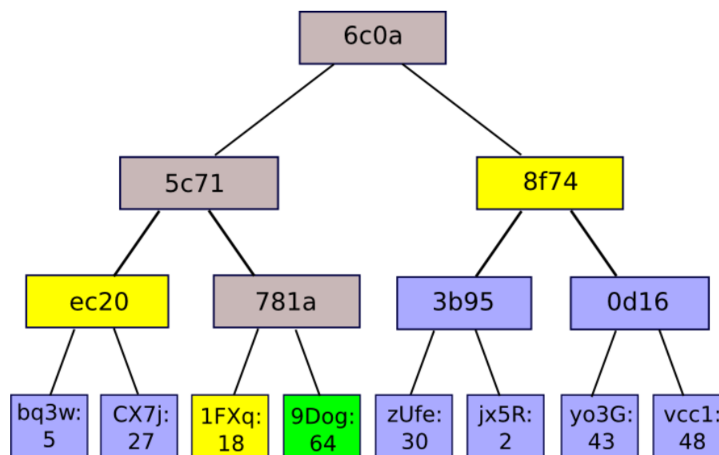



**Figure 2.2:** A sample Merkle Proof [But15]

With $h_m$ equaling to the root hash of the data set, the probability that the given data-block isn't a member of the data set is the same as the probability to find a collision in the cryptographic hash function that is used in MT. Hence, as long as the using cryptographic hash function is secure, the Merkle proof is unforgeable.

In PEES, we adapt the usage of MT and Merkle proof from Zcash. After Registration has ended and all VCs are generated, a MT will be constructed over all VCs and the root hash of that MT will be saved on the Blockchain. Later, when a Voter wants to vote using his VC, a Merkle proof will be given to prove that the given VC is one of the valid VCs.

## 2.6 Secret Sharing

Secret sharing is a method that can be used to split a given secret into several parts. Each part is called a **share** and they will be given to different participants so that each participant just has his own unique share. To reconstruct the secret, sufficient shares are required. The minimum number of shares that are needed to reconstruct a secret is called **threshold**. In practice, the threshold is usually bigger than 2 so that each share itself are of no use.

A secret sharing scheme is secure, if no one can recover the secret with a number of shares, that is smaller than the threshold and the secret will always be successfully reconstructed with enough shares. Notice that a secret sharing scheme cannot prevent the case when a share-holder is corrupted and gives his share away. Hence, it is not recommended to use a very small number as the threshold.

In PEES, Shamir's Secret Sharing Scheme [Sha79] is used to split the decryption key of EA to a group of admins, so that each admin himself cannot decrypt votes that have been recorded on Blockchain while the election is still running. The concrete protocol of Shamir's Secret Sharing Scheme is not part of this thesis, for those details please refer to [Sha79].

## 2.7 Zero-Knowledge Proof

A Zero-Knowledge Proof (ZK-Proof) of a statement is a proof that does not reveal anything except for showing that the statement is true [HL10]. It is not like a mathematical proof that convinces someone by establishing the validity of a statement but a proof of the possession of some specific values that fulfil some certain conditions. In general, any NP-statement like the following can be proven in zero-knowledge.

- "given a $n^2 \cdot n^2$ Sudoku $S$, there is a solution to $S$"
- "given a Cryptographic hash function $h$ and a hash value $h_x$, there is a input data $x$ such that $h(x) = h_x$"
- "given a root hash $r$ of a MT and a hash value $h$, there exist a Merkle Authentication Path that shows $h$ is a leaf value of that MT"

The satisfying input or assignment, e. g., the solution to $S$ or the input data $x$, are called **witness**. Knowing the witness to a statement, a prover can exchange messages with a verifier to convince the verifier that he possesses the very witness to the statement but without revealing it to the verifier. The method that the prover and the verifier followed to exchange messages is called Zero-Knowledge Proof or Zero-Knowledge protocol. The term "Zero-Knowledge" means, on verifying the proof, the only thing that a verifier can learn is that the statement is true, any other information, especially the witness of the statement is still remain secret to the verifier. More formally speaking, the process of the message exchange between a prover and a verifier is zero-knowledge, if there exists a simulator, which has no knowledge of the witness, but can still simulate the message exchange process such that the probability, that the verifier is convinced in the simulator is the same as the probability in real message exchange process.

If a malicious prover wants to prove a statement to be true while the statement is actually false, ZK-Proof should guarantee that, the verifier should only accept the proof with a very small probability $P_f$. On the other hand, when a prover proves a true statement, the verifier should accept the proof with a much higher probability $P_t$. In general, we only demand a clear difference between $P_t$ and $P_f$, i. e., $P_t > P_f$. The probability can be increased (for $P_t$) / decreased (for $P_f$) by repeatedly proving.

A Zero-Knowledge proof of Knowledge is a special case of ZK-Proof. It proves the statement that only consist of the fact that the prover possesses some secret knowledge but not the knowledge itself. In addition, ZK-Proof can be divided into two types, namely non-interactive and interactive. While an interactive ZK-Proof requires interaction between a prover and a verifier during a prove, non-interactive ZK-Proof doesn't need any. Therefore, if a prover wants to prove the same statement to different verifiers, he has to prove multiple times with an interactive ZK-Proof but he just needs to generate the proof once and sends it to all verifiers in a non-interactive ZK-Proof. Due to the simplicity and efficiency, zk-SNARK, an example of non-interactive ZK-Proof, is used in PEES.

### 2.7.1 zk-Snark

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) is an efficient variant of a zero-knowledge proof of knowledge. The meaning of "Zero-Knowledge" and "Non-Interactive" is as described above, "Succinct" means that the generated Proof is small (usually a few hundred bytes) and is easy to verify. As mentioned above, a zero-knowledge proof of knowledge only proofs the statement which states, the prover possesses some certain knowledge or secret information. While pure ZK-Proof proofs statements about properties of some secret, zero-knowledge proof of knowledge proofs the possession of that very secret of provers. As the name indicates, this kind of proof is also zero-knowledge, i. e., the verifier won't learn anything about the secret but that the prover possesses such a secret. Argument of knowledge is similar to Proof of knowledge, but it only protects the verifier against a computationally limited prover. That means, if a prover has unbounded computational power, he can fake proofs for any false statement without being noticed [Rei16]. As most of the computer in practice has only limited computational power, it is enough to use Argument of knowledge in PEES.

zk-SNARK is used for some NP-complete languages. That means, any statement that a prover wants to proof must be first converted to the specific NP-complete languages that zk-SNARK supports. In the implementation of PEES, `libsnark`, a C++ implementation of zk-SNARK schemes, is used. This library supports several NP-complete languages such as R1CS, BACS or USCS [Laba]. For the demo of PEES implemented in Chapter 5, we use Rank-1 Constraint Systems (R1CS) as the underlying NP-complete language for `libsnark`. An instance of R1CS is a set of equations over a prime field $F$ in the following form: $< \mathbf{A}, \mathbf{x} > * < \mathbf{B}, \mathbf{x} > = < \mathbf{C}, \mathbf{x} >$, where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are three vectors in field $F$, $\mathbf{x}$ represents a vector of variables and $<,>$ donates the dot product of two vectors [Vit].

| A | | B | | C | |
|---|---|---|---|---|---|
| 1 | 5 | 1 | 1 | 1 | 0 |
| 3 | 0 | 3 | 0 | 3 | 0 |
| 35 | 0 | 35 | 0 | 35 | 1 |
| 9 | 0 | 9 | 0 | 9 | 0 |
| 27 | 0 | 27 | 0 | 27 | 0 |
| 30 | 1 | 30 | 0 | 30 | 0 |

35 * 1 - 35 = 0

**Figure 2.3:** A satisfying R1CS instance [Vit]

Figure 2.3 shows a satisfying R1CS instance. The triple $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ builds a constrain for the variable vector $x$. In `libsnark`, to check whether a statement is fulfilled is equals to check whether every constrains in the corresponding constrain system is satisfied for the given solution (witness and public inputs). Further technical details of how zk-SNARK works is out of scope for this thesis, for those information please see [BCCT11], [BCI+12] and the references therein, for more information about `libsnark` please see Section 5.1.

In PEES, zk-SNARK is used to let Voter prove that he possesses a specific VC, which shows his eligibility to vote for a certain election. As the VC is related to a certain Voter, revealing the detail of a VC equals to expose the identity of a Voter to the public, especially the election organizer. Thus, to protect Voter's privacy, the possession of VC must be proven in zero-knowledge. Apart from this proof, zk-SNARK is also used in the result computation part to show that, EA possesses the correct decryption key for the election and it has decrypted all recorded votes and calculated the result correctly without revealing the decryption key. We don't want the decryption key to be revealed to the public even after the election is because that if an attacker somehow gets to know the identity of a Voter, there still is this decryption key that prevents the attacker of knowing that Voter's exact vote, hence, this key must be well protected.

# 3 Blockchain

Before getting into details of PEES, one more fundamental building block, Blockchain, should be explained. In this chapter, I'll first introduce Bitcoin, where Blockchain has first been implemented and widely used, in Section 3.1. In Section 3.2, I'll explain the Smart Contract from Ethereum, which enables Turing-complete programming on Blockchain. Finally, I'll introduce Zcash and its privacy providing methods `Mint` and `Pour` in Section 3.3.

## 3.1 Bitcoin

Blockchain was first conceptualized by a person known as Satoshi Nakamoto for the cryptocurrency, Bitcoin, in 2008. Bitcoin is a decentralized peer-to-peer electronic cash system [NBF+16]. The underlying cryptocurrency, i. e., the electronic cash, is also called Bitcoin. Decentralization indicates that there is no single administrator such like a central bank that has to be trusted or relied on in this system. In this case, all transactions are directly sent from one user to another user in the peer-to-peer network. If a transaction is verified to be valid, i. e., the sender is the owner of the Bitcoins that he tries to send, and he has enough money on his account, then this transaction will be recorded on a public distributed ledger that is also known as the Blockchain. The public ledger is designed in such a way that any recorded transactions are unchangeable, and it is public to everyone in the network. To get a current account balance of a user, one just need to go through every record on the public ledger which are related to the desired user and then calculate the current account balance.

In following, I'll explain some basic concepts such like user, miner, and some fundamental building blocks of Bitcoin system. This section is aimed to explain the concept and ideas of Bitcoin system, technical details are not included, for those details, please see [Nak08] and [NBF+16].

### 3.1.1 User and Miner

In Bitcoin, a user is represented by a pair of signing keys $(pk, sk)$ as described in Section 2.3. The public key $pk$ is like a bank accounts in real life, if a user wants to send a transaction to another one, the receiver's public key must be first known to the sender. The secret key $sk$ is like the password to the bank account, i. e., only with the corresponding secret key, a user can send money from that account. Since there is no central party that records which account belongs to which user, it is important that a user should never reveal his secret key to any other people, otherwise he could be easily impersonalized.

Miner is a special kind of people that group pending transactions and build up blocks from those transactions (this action is also called mining) based on the consensus algorithm of Bitcoin system (for more explanation please see Section 3.1.4). For every valid Block that the miner has added to the current Blockchain (public ledger), he will be rewarded with all transaction fees from transactions that he has grouped in that Block and a block reward.

Miner and user are two different roles in the Bitcoin system and they should not be confused. A participant in the Bitcoin system can be a miner and a user at the same time, if he mines the blocks and sends transactions as well. A participant can also just be a user who only uses this system but not making any contribution to the public ledger.

### 3.1.2 Transaction

On a high level, each Bitcoin can be considered as a tuple of a value $v$ and a public key $pk$ that indicates the ownership of the Bitcoin. Assume a user Alice wants to send a Bitcoin with value 3 to another user Bob, a transaction as described in the next paragraph will be constructed and then broadcasted to all nodes in the peer-to-peer network, so that everyone gets this update of her account. A node in peer-to-peer network represents a participant in the network, in the following, these two words will be used interchangeably.

A transaction in Bitcoin system consist of a unique identifier, input Bitcoins, output Bitcoins and a digital signature of the sender, in the example above, a digital signature of Alice. The input Bitcoins were got from some earlier transactions where other users sent to Alice. One output Bitcoin has a value of 3 and the public key of the receiver, Bob. This is the Bitcoin that Alice wants to send to Bob. Usually, there will be another output Bitcoin which has a value equals to the amount of all input Bitcoins minus 3 and the transaction fee, and a public key of Alice herself. This Bitcoin is generated when the sum of all input Bitcoins has a much higher value of the desired output Bitcoin. As Bitcoins cannot be split, to refund herself of unspent Bitcoins, Alice has to generate this new Bitcoin as a output Bitcoin as well. At the end, to authorize this transaction, Alice has to use her secret key to sign on this transaction. With this signature verified as valid, everyone in the Bitcoin system will be convinced that Alice has indeed authorized this transaction, otherwise this transaction will be discarded and never be added to a block.

A transaction must be first verified and then be added to a block. Only after a transaction is added to a block, it will be considered as executed. To verify a transaction, the following requirements must be checked:

- the input Bitcoins are valid, i. e., Signatures on transactions where input Bitcoins are generated, are valid, and
- the values of all input Bitcoins are bigger than the values of all output Bitcoins, and
- the signature on the transaction is valid

The difference between the values of all input Bitcoins and the values of all output Bitcoins is the transaction fee. Due to internet delay, when transactions are being broadcasted to all nodes in the network, they usually won't arrive at the same order as they have been broadcasted. Hence, transactions that will be grouped in the new blocks by miners are not based on transaction orders but miner's preference. Thus, a transaction with high transaction fee are always preferred.

### 3.1.3 Blockchain

In the Bitcoin system, the public ledger is implemented as a Blockchain. As its name indicated, a Blockchain is a sequence (chain) of blocks as shown in Figure 3.1. Each block consists of a block-header and a list of valid transactions. The block-header includes different variables, the most important ones are the *parent block hash*, the *current block hash* and a nonce *Nonce*. The *current block hash* is simply the hash of all data, i. e., the block header and all transactions that grouped in the current block. As shown in Figure 3.1, the *parent block hash*, which is the *current block hash* of the previews block, is contained in the current block-header. That means, when calculating the *current block hash*, the *parent block hash* will be included as the hash input as well. Notice that the the *parent block hash* is the only thing that connects different blocks to a chain.

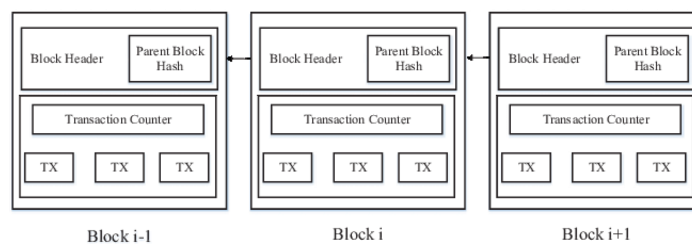



**Figure 3.1:** An example of Blockchain [ZXD+17]

In Bitcoin system, a Blockchain is valid iff all blocks on the chain are valid. A block is called valid, if its block-header is valid and all transactions that have been grouped in that block are valid. When broadcasting a newly generated block to all nodes in a peer-to-peer network, the message may not arrive at each block on the same time. Thus, it is normal that two (or more) miners have mined a new block almost simultaneously but contain different transactions. As shown in Figure 3.2, assume the two new blocks based on $B3$ are block $B4$ and block $U4$. Since miners choose transactions based on their preference, it is highly possible that the transactions included in $B4$ and $U4$ are different. This situation is known as a fork of a Blockchain. In this case, both new blocks will be currently considered as valid till one chain has been extended longer than the other, then only the longest chain will be considered as valid. This is known as the “longest chain” rule.

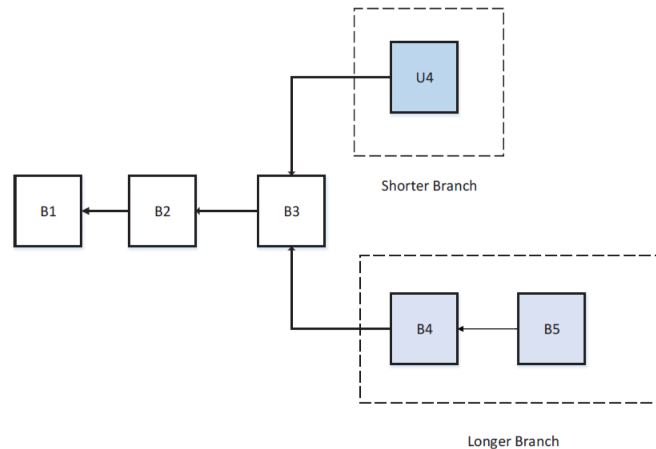



**Figure 3.2:** An example of fork in a Blockchain [ZXD+17]

For example, after $B4$ and $U4$ has been broadcast in the network, miners will randomly choose a block as the previews block to mine new blocks. After some time, assume a new block $B5$ has been mined based on $B4$ and there are still no new blocks mined based on $U4$, as shown in Figure 3.2, then the chain with block $U4$ will be considered as invalid and all miners will switch to the valid chain with block $B5$. As long as most miners stick to the longest chain rule, the Blockchain won't diverse.

### 3.1.4 Consensus Algorithm

The longest chain rule has prevented a Blockchain from diverging, however, the longest chain rule has also made Blockchain vulnerable to **Double Spending Attack**s. In a double spending attack, an attacker is trying to spend one Bitcoin to two or more users. With the longest chain rule, this attack can be easily done as following:

Consider the Blockchain shown in Figure 3.2, assume blocks $B1$ to $B3$ are already mined

1. An Attacker sends one Bitcoin to Alice as payment for some item in real life in a transaction *Tran*
2. The attacker mines block $U4$ that contains *Tran*
3. Alice sees that *Tran* is in a block on the chain, she then sends an item to the attacker in real life, as they earlier agreed
4. After that the attacker gets the item, he sends the same Bitcoin to Bob in *Tran'*
5. The attacker mines block $B4$ with *Tran'* in it, and he keeps mining new blocks based on $B4$

As soon as the lower chain in Figure 3.2 is longer than the upper chain, the transaction *Tran'* will be considered as executed and *Tran* will be considered as not executed, i. e., Alice didn't get the payment, but the attacker already got the item from Alice.

In this case, a consensus algorithm will be needed to prevent any nodes in the network to attack the network. A consensus algorithm is aimed to help untrustworthy nodes to reach a consensus. In a decentralized network like Bitcoin system, a consensus algorithm is used to decide which node could propose a new block. In Bitcoin, the consensus algorithm **Proof of Work (PoW)** is used. The basic idea of PoW is that each block can only be proposed if a lot of work has been put to generate that block. The term “work” here means computer calculations, more precisely hash calculations [ZXD+17].

In Section 3.1.3, we mentioned that in each block, there is a variable called “*current block hash*” in the block header. PoW then requires that this hash value should be smaller than a certain value otherwise the mined block is invalid. To do that, the variable *Nonce* in the block header should be changed, so that the current block hash meets the requirements. As a cryptographic hash function is pre-image resistance (Section 2.1), the best way to find a satisfying hash value is to brute-force, i. e., constantly change the value of *Nonce*, till the result hash meets the requirement. Hence, with PoW, every node in the network is theoretically possible to propose a new block. However, the probability is based on the hash power it has.

Apart from PoW, there are other consensus algorithms such like Proof of Stake (PoS) where the participant with high account balance will be selected to propose a new block. It is believed that participants with more currencies are less likely to attack the network. For more consensus algorithms, please see [ZXD+17].

### 3.1.5 Security Analysis

In this section, I’ll informally explain why transactions on a Blockchain are unchangeable and how it prevents double spending.

Assume we have a Blockchain of length $n$, and all honest miners are currently mining new blocks based on $B_n$. An attacker wants to change a transaction in block $B_i$ with $i < n$. Then, after the attacker has changed the transaction in $B_i$ (we use $B_i'$ to donate the changed block), he has to recalculate the current block hash of $B_i'$ and resolve the PoW problem for $B_i'$ to make this block valid. Since the data in $B_i$ and $B_i'$ are different, the *current block hash* of these two blocks won’t be the same. Therefore, the block $B_{i+1}$ and all blocks behind will still be considered as linked to block $B_i$ while $B_i'$ will be considered as a fork of the original Blockchain as shown in Figure 3.2. Because of the longest chain rule, this fork will not be considered as valid as long as it is shorter than the original Blockchain. The attacker may try to mine more blocks based on $B_i'$ so that this fork is longer. However, PoW guarantees that if the hash power of all honest miner is bigger than the attacker, it’s almost impossible for the attacker to catch up to the original chain. Thus, transactions that has been recorded on a Blockchain is unchangeable.

As for the double spending attack that described in Section 3.1.4, it is almost the same with the setting above. However, in the double spending attack, if Alice sends the item to the attacker as soon as she sees the transaction has been recorded in a block, the attacker may have only few blocks to catch up with. In this case, if an attacker is lucky, i. e., solve a PoW problem with just few tries, he may actually make the fork chain longer than the original one. Hence, it is recommended to trust a transaction to be actually executed when there are 6 more blocks that have been mined based on

the block that recorded that transaction. This is the so called "6-block" rule. With 6 blocks ahead, it is almost impossible for the attacker to generate a fork for double spending and catch up with the original Blockchain.

However, if an attacker has more than half hash powers of the whole network, PoW can no longer protect the network from any attacks that have been described above. Since the attacker has more hash power than the rest of the network, he could always catch up the original Blockchain, the only variants here is when he will catch up and exceed the original Blockchain. This is known as the 51% attack [**51a**]. Theoretically this attack works, but in practice, it takes a lot of time and money to have more than half hash power of the whole network. Hence, in general, especially for PEES, we will consider the Blockchain as a secure data structure to keep recorded data unchangeable and prevent of double spending attacks.

## 3.2 Ethereum

Ethereum is also a Blockchain based decentralized system. Unlike Bitcoin system which is aimed to provide an alternative electronic payment system, Ethereum attempts to establish a decentralized software platform that enables Smart Contract and Distributed Application (Dapp). Although the technical details of the Blockchain that used in Bitcoin and Ethereum differs from many aspects, the basic ideas and concepts remain the same.

In the following, I will focus on the unique feature, Smart Contracts, that provided by Ethereum. I'll first explain what a Smart Contract is and then describe how it should be used in Ethereum.

### 3.2.1 Smart Contract

In general, a Smart Contract is a digitization of a legal contract whereas in Ethereum, a Smart Contract is a piece of executable code that is implemented, deployed and executed within Ethereum environment [Mod18].

After a Smart Contract is deployed in Ethereum, i. e., been recorded on the Blockchain, anyone in the network can use the functionality of this Smart Contract. A function in a Smart Contract can be called either by a user in the network or by another deployed Smart Contract. To do that, a trigger such like a transaction from a user or a message (will be explained in Section 3.2.2) from another Smart Contract is needed. After successfully executing the function, the state and the storage of the Smart Contract will be updated and be recorded on the Blockchain.

Ensured by the Blockchain, any deployed Smart Contracts are also unchangeable. On the one hand, this is good for Dapp developers, since they don't have to worry about any other party manipulating their applications. However, if they found any bugs in their Dapp after they deployed the Smart Contract, they could not simply fix it in the original Smart Contract but deploy a new one.

### 3.2.2 Externally Owned Account and Contracts Account

Different from the Bitcoin system, there are two kinds of accounts in Ethereum, namely Externally Owned Account (EOA) and Contract Account [BS18].

An EOA is similar to a user account in Bitcoin. Each EOA is represented by a pair of keys $(pk, sk)$. Like Bitcoin system, an EOA can send transactions that transfer Ether, the underlying cryptocurrency in Ethereum, to another EOA or Smart Contract. Apart from that, an EOA can also deploy or interact with Smart Contracts through transactions.

In Ethereum, each deployed Smart Contract is also considered to have an account, namely the Contract Account. A contract account is identified by an unique contract id that is generated when the Smart Contract is first deployed to the Blockchain. There is no public/secret key pair for a contract account, since these keys have to be stored in the corresponding Smart Contract on the Blockchain and hence is public to everyone in the network. A contract account also keeps the Ether balance as an EOA, besides that, the associated code, current state and the storage of the Smart Contract is stored in the contract account as well.

However, as a contract account doesn't have a private key, it is not possible for this kind of account to sign on any transactions, thus a contract account is not able to send any transactions as an EOA. As an alternative, contract accounts send messages to other Smart Contracts to invoke functions they provide. A message between Smart Contracts is similar to a function call, but unlike a transaction, they exist only in the Ethereum execution environment and won't be recorded on the Blockchain [BS18].

### 3.2.3 Transactions

As mentioned in Section 3.2.2, a transaction can only be sent from an EOA. Based on the receiver and the purpose, transactions can be divided in 3 different kinds.

The first kind of transaction is the Payment transaction. This kind of transaction is used to transfer Ether from one EOA to another. A Payment transaction is similar to a transaction in Bitcoin system, it consists of a receiver, the amount of the Ether that sender wants to transfer, a transaction fee and a signature of the sender.

The second kind of transaction is called Contract Creation transaction. This kind of transaction is sent to an empty address with the code of a Smart Contract as data. Apart from that, the sender also has to pay for the execution and sign on this transaction to show that he really intended to create this Smart Contract [JW17].

The last kind of transaction is used to invoke a function of a Smart Contract. This kind of transaction is sent from an EOA to a Smart Contract. Instead of using the public key to identify the receiver, the Smart Contract's id is used. In addition, the function name and input parameters should be included in the transaction data as well. It is also possible to send Ether to a Smart Contract, and like a Contract Creation transaction, the sender has to pay for the execution and sign on the transaction.

The execution in the last two kinds of transactions are executed when miners mining new blocks. Similar to the Bitcoin system, before a transaction is record to a block, it first needs to be verified. The verification for the Payment transaction is the same as the verification for transactions in Bitcoin

system. However, the verification for the last two kinds of transactions is a little bit different. Except from verifying the signature, miners have to either execute the contract creation function provided by Ethereum to initialize a Smart Contract [Woo14] or execute the called function with given input parameters and update Smart Contract's state and storage. Only after the signature has been verified as valid and the execution ended without errors, the transaction will be recorded into a block. Notice that the function call might be verified on different computers, but the result needs to be the same for all different runs, therefore, all functions in Smart Contract should be deterministic, i. e., no true randomness can be invoked in Smart Contract.

### 3.2.4 Concept of Gas

Ethereum provides a Turing-complete program language, known as solidity, for Smart Contracts. As solidity is a Turing-complete language, it is possible that a function will never end. Then, when miners executing this function call, they will never halt and hence no blocks will be further generated. Since the halting problem is undecidable, it is not possible to tell whether a function will halt or not when creating the Smart Contract or before a function call. To solve this problem, Ethereum introduced the concept of Gas.

In Ethereum, Gas refers to fees that the sender pays for the execution in a transaction [BS18]. Each execution step will be broken down to fundamental operators and each of these operators have a fixed price. When sending a transaction which invokes a Smart Contract, the sender has to specify the amount of Gas he's willing to pay for the execution and a gasPrice in Ether. Then, when verifying such a transaction, miners will use the Gas provided by the sender to execute each operation. As soon as the Gas is used out the execution is stopped. With this mechanism, every function in a Smart Contract will stop after finite steps.

During an execution, if a transaction uses less Gas than the sender provided, the sender will get a refund of remaining Gas in Ether. However, if executing a transaction needs more Gas than the sender provided, the execution will be stopped when the Gas ran out and all state or storage changes during this execution will be reverted as this execution has never been started. Although the execution is reverted, the transaction itself is still valid and the fee will also be collected by the miner. Thus, it is recommended to estimate the Gas cost before sending a transaction.

## 3.3 Zcash

Zerocash protocol is aimed to bring privacy in digital currencies like Bitcoin [SCG+14]. The full-fledged digital currency of this protocol is called Zcash [BACa]. Unlike Bitcoin where every user's payment history is recorded directly on the Blockchain, Zerocash hides the amount of the payment and the identity of the receiver under encryption and proof the correctness of a transaction using ZK-Proof. To reach this goal, Zerocash protocol introduced a new anonymous coin, *zerocoin*, to distinguish with non-anonymous coins (also called *basecoin*) such as Bitcoin. When using the Zcash, a user first has to convert some of his *basecoin* to the same amount of *zerocoin* using the function **Mint**. After that, he can make private payment using *zerocoin* and function **Pour** so that everyone in the network doesn't know to whom and how much he has paid. The function pour also allows a user to split his *zerocoin* or to convert it back to *basecoin*.

In The following, I'll describe how Zcash represents a user in Section 3.3.1, and then explain how mint and pour function works in Section 3.3.2 and Section 3.3.3. At the end, I'll give a security analysis of privacy in Section 3.3.4.

### 3.3.1 User Account

Unlike a user in Blitcoin system or in Ethereum, a user in Zcash is represented by two pairs of keys. The first pair of keys $(a_{pk}, a_{sk})$ is called address key pair. Key $a_{pk}$ represents the identity of a user and it is also associated to all *zerocoin* that belongs to this user. To spend a *zerocoin*, the user has to show that he has the knowledge of the corresponding $a_{sk}$. $a_{pk}$ and $a_{sk}$ are not chosen randomly but generated using a so called Pseudorandom function (PRF).

A PRF is a function with values seemingly chosen randomly within its range in cryptography [GGM84]. Zcash uses collision resistant PRFs, similar to a cryptographic hash function, it is infeasible to find two different inputs such that they result in the same output of a PRF.

To generate an address key pair, a user first sample a random seed $a_{sk}$, and use it to select a PRF from the Pseudorandom function family *addr*. The value of the output of this PRF on input 0 is $a_{pk}$, i. e., $a_{pk} := PRF^{addr}_{a_{sk}}(0)$

The other key pair $(pk_{enc}, sk_{enc})$ is the encryption key pair. It should be generated according to the underlying PKE. It is used to encrypt transaction data, so that they are not public on the Blockchain. Together, the key pair $(a_{pk}, pk_{enc})$ is the public key pair of a user and the key pair $(a_{sk}, sk_{enc})$ is the secret key pair.

### 3.3.2 Mint

In Zcash, the Mint function is used to convert *basecoins* to *zerocoins*. To convert $v$ *basecoins*, the user first has to deposit that much *basecoin* to a backing escrow pool [SCG+14], after that, the user should generate a *zerocoin* with the value $v$ and send the transaction $\texttt{tx}_{MINT}$ to the Blockchain to indicate that he has minted a new *zerocoin* $c$.

As shown in Figure 3.3 ($b$), a *zerocoin* consists of a user's public key pair $(a_{pk}, pk_{enc})$, a value $v$, three randomness $\rho$, $r$, $s$ and the coin commitment $cm$. After a Mint transaction $\texttt{tx}_{MINT}$ has been verified as valid, the coin commitment $cm$ of the new coin will be saved in a MT as illustrated in Figure 3.3 ($a$).

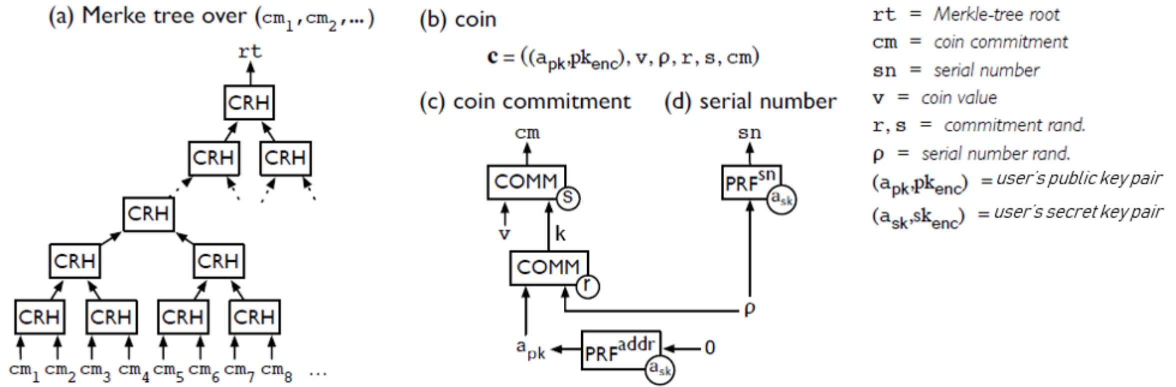



**Figure 3.3:** Mint [SCG+14]

To generate a new *zerocoin* $c$ with value $v$, a user with public key pair $(a_{pk}, pk_{enc})$ should execute the Mint function off-chain. On receiving $v$ and $a_{sk}$, the Mint function does the following steps [SCG+14]:

Mint$(v, a_{sk})$ :

1. calculate $a_{pk}$ from $a_{sk}$ as described in Section 3.3.1.
2. sample the serial number randomness $\rho$.
3. sample a commitment randomness $r$, then compute the first commitment $k := Comm_r(a_{pk} \,||\, \rho)$.
4. sample another commitment randomness $s$, then compute the coin commitment $cm := Comm_s(v \,||\, k)$.
5. compute the serial number of $c$ using a PRF chosen from Pseudorandom function family $sn$ based on $a_{sk}$, i. e., $sn := PRF^{sn}_{a_{sk}}(\rho)$.
6. broadcast Mint transaction $\texttt{tx}_{MINT} := (v, k, s, cm)$ to Blockchain.
7. return $c := ((a_{pk}, pk_{enc}), v, \rho, r, s, cm)$ and $sn$.

The relation between $a_{sk}$, $a_{pk}$, $\rho$, $r$, $s$, $v$ and $cm$ is illustrated in Figure 3.3 $(c)$ and $(d)$. The first commitment is aimed to bind the coin with its owner's address public key $a_{pk}$ and to connect the coin commitment $cm$ and the corresponding serial number $sn$ through the serial number randomness $\rho$. The second commitment, i. e., the coin commitment, is a commitment over the value of the coin $v$ and the result of the first commitment $k$. Instead of using just one commitment over all values $v$, $a_{pk}$ and $\rho$, Zcash uses two hierarchies of commitment to hide $\rho$ from the public, otherwise when including all committed value in $\texttt{tx}_{MINT}$ the public will know which *zerocoin* has been used by which user (for more explanation see Section 3.3.4).

To verify the validity of a Mint transaction, a miner first needs to check whether the sender has deposited $v$ *basecoins*, if he hasn't deposit that many *basecoins*, this Mint transaction will be considered as invalid and discarded. If the same amount of *basecoins* have been deposited to

a backing escrow pool, the miner will continue checking the transaction by computing the coin commitment himself using $v$, $k$, $s$ and comparing the result with the given value $cm$. If this check passes, the Mint transaction will be grouped to a block.

Notice that, in $\texttt{tx}_{MINT}$, only the value that related to the coin commitment $cm$ is revealed to the public, the sender should still keep other randomness $r$, especially $\rho$ and the serial number $sn$ secret, otherwise the privacy might be leaked.

### 3.3.3 Pour

The Pour function is used to spend *zerocoins*. In general, a Pour function is expected to consume two *zerocoins* $c_1$, $c_2$ and generate two new *zerocoins* $c'_1$, $c'_2$ for the receiver with the total value of $c_1$ and $c_2$ equals to the total value of $c'_1$, $c'_2$. If a user wants to consume more *zerocoins* or transfer to more receivers, he can call Pour function multiple times and if he only wants to spend one *zerocoin* to one receiver, he can set $c_2$ and $c'_2$ to NULL. Thus, we will only consider the Pour function with two in/output coins in the following.

A Pour function is also executed off-chain. It takes two generated *zerocoins* $c_1^{old}$, $c_2^{old}$ and their corresponding serial number $sn_1^{old}$, $sn_2^{old}$, the value of two new *zerocoins* $v_1^{new}$ and $v_2^{new}$, the public key pair of two receiver $(a_{pk,1}^{new}, pk_{enc,1}^{new})$ and $(a_{pk,2}^{new}, pk_{enc,2}^{new})$, and the root hash of the MT that saves all coin commitment $rt$ as its input and does the following [SCG+14]:

$\text{Pour}(c_1^{old}, c_2^{old}, sn_1^{old}, sn_2^{old}, v_1^{new}, v_2^{new}, a_{pk,1}^{new}, a_{pk,2}^{new}, pk_{enc,1}^{new}, pk_{enc,2}^{new}, rt)$ :

1. for $i \in \{1,2\}$ : sample randomness $\rho_i^{new}$, $r_i^{new}$, $s_i^{new}$ separately.
2. for $i \in \{1,2\}$ : compute $k_i^{new} := Comm_{r_i^{new}}(a_{pk,i}^{new} \,||\, \rho_i^{new})$.
3. for $i \in \{1,2\}$ : compute $cm_i^{new} := Comm_{s_i^{new}}(v_i^{new} \,||\, k_i^{new})$.
4. for $i \in \{1,2\}$ : assign $c_i^{new} := ((a_{pk,i}^{new}, pk_{enc,i}^{new}), v_i^{new}, \rho_i^{new}, r_i^{new}, s_i^{new}, cm_i^{new})$.
5. for $i \in \{1,2\}$ : encrypt the new generated *zerocoin* under receiver's encryption public key, i. e., $C_i = Enc_{pk_{enc,i}^{new}}(c_i^{new})$.
6. generate a ZK-Proof $\pi_{POUR}$ for the following statement:
   "for $i \in \{1,2\}$ : Given the MT root hash $rt$, the serial number $sn_i^{old}$, and the coin commitment $cm_i^{new}$, I (sender) know *zerocoins* $c_i^{old}$, $c_i^{new}$ and address secret keys $a_{sk,i}^{old}$ such that:
   - for $i \in \{1,2\}$ : the *zerocoins* $c_i^{old}$, $c_i^{new}$ are well formed, i. e., for each *zerocoin*, the two commitments that described in Section 3.3.2 is valid.
   - for $i \in \{1,2\}$ : the address secret key matches the address public key, i. e., $a_{pk,i}^{old} := PRF_{a_{sk,i}^{old}}^{addr}(0)$.
   - for $i \in \{1,2\}$ : the correctness of serial number computation of consumed *zerocoins*, i. e., $sn_i^{old} := PRF_{a_{sk,i}^{old}}^{sn}(\rho_i^{old})$.
   - for $i \in \{1,2\}$ : the coin commitment $cm_i^{old}$ is a leaf of a MT with root hash $rt$.
   - The values add up: $v_1^{new} + v_2^{new} = v_1^{old} + v_2^{old}$."

7. broadcast $\texttt{tx}_{POUR} := (rt, sn_1^{old}, sn_2^{old}, cm_1^{new}, cm_2^{new}, \pi_{POUR}, C_1, C_2)$ to the Blockchain.

The steps 1 to 4 is used to generate new *zerocoins*, the step 5 encrypts the detail of new generated *zerocoins* under the corresponding receiver's encryption public key, so that only the receivers of these *zerocoins* are allowed to know the associated randomness. Thus, these *zerocoins* can only be spent by their owner. Since only the serial number of consumed *zerocoins* and the coin commitment of the new generated *zerocoins* is revealed to the public, a ZK-Proof (generated by step 6) is needed to show the correctness of this transaction.

To verify a Pour transaction, a miner first checks whether the given serial numbers $sn_1^{old}$ and $sn_2^{old}$ have been published before or not. If a serial number has occurred in an earlier transaction, which means the current transaction is double spending a used *zerocoin*, thus the current transaction will be considered as invalid and be abandoned. On the other hand, if a serial number has not occurred before, the miner will continue to check the validity of the ZK-Proof $\pi_{POUR}$ with the given values $rt$, $sn_1^{old}$, $sn_2^{old}$, $cm_1^{new}$, $cm_2^{new}$. If this proof has been checked to be correct, then the corresponding Pour transaction is valid and will be eventually grouped in a block. The coin commitment of new *zerocoins* $cm_1^{new}$, $cm_2^{new}$ will be saved in the MT as well, so that the receiver can generate a valid ZK-Proof later, when they use the new *zerocoins*.

As mentioned in the beginning of this chapter, the Pour function can be used to split *zerocoins* to *zerocoins* with new values and convert *zerocoins* back to *basecoins*. To split *zerocoins*, the sender just needs to specify the receiver as himself, and select values for new *zerocoins* as he wishes. However, to convert *zerocoins* back to *basecoins*, the Pour function needs a little modification.

To enable the conversion, new public outputs $v_{pub}$ and *info* are needed. The value $v_{pub}$ indicates how many *zerocoins* the sender wants to convert back to *basecoins* and the the sender can specify the target of these *basecoins* in variable *info*. Along with these changes, the last requirement in the ZK-Proof statement should be changed to $v_1^{new} + v_2^{new} + v_{pub} = v_1^{old} + v_2^{old}$ as well. The two public outputs $v_{pub}$ and *info* should be added to $\texttt{tx}_{POUR}$, since the *basecoin* doesn't provide privacy, any changes of *basecoins* should be recorded on the Blockchain.

### 3.3.4 Security Analysis

Zcash is aimed to hide the original coins that have been spent and the new coins that have been generated in a transaction from both the public and the sender. That means, the public should only learn the serial number $sn$ of a spent *zerocoin* but not the very coin and especially the coin value $v$ that has been spent, i. e., the public shouldn't be able to connect the corresponding coin commitment $cm$ to $sn$. As for the sender, although he has generated the new *zerocoin* for the receiver, he shouldn't be able to tell that if a spent coin is generated from him or not, otherwise, the sender will know the value of that spent *zerocoin*. Apart from these privacy requirements, Zcash should also guarantee that only the owner of a *zerocoin* can spend it, otherwise the fundamental requirement of a cryptocurrency system is broken.

In the following, I'll explain how Zcash guaranteed these requirements using Mint and Pour functions.

Start with Pour transaction $\texttt{tx}_{POUR} := (rt, sn_1^{old}, sn_2^{old}, cm_1^{new}, cm_2^{new}, \pi_{POUR}, C_1, C_2)$. For an attacker from the network, if the used commitment scheme, PKE and the ZK-Proof is secure, then it is infeasible for him to retrieve the value of new generated *zerocoins* from $cm_i^{new}$, or any

details of new generated *zerocoins* from $C_i$, for $i \in \{1, 2\}$. As for the consumed *zerocoins*, if the underlying PRF $PRF^{sn}$ is secure, then it is also impossible for an attacker to retrieve the serial number randomness $\rho$ from the revealed $sn$. Hence, an attacker cannot connect a serial number $sn$ to its corresponding coin commitment $cm$ and therefore he cannot know the value of the spent *zerocoin*. Analogous, if the using ZK-Proof is secure, then the attacker cannot get any information about both consumed *zerocoins* and newly generated *zerocoins* from $\pi_{POUR}$.

As for the creator of new *zerocoins* in a Pour function, although he knows all randomness $\rho$, $r$ and $s$ for new *zerocoins*, as long as he doesn't know the address secret key $a_{sk}$ of the receiver, he can't know which serial number $sn$ corresponding to the coin commitment $cm$ that he has computed for new *zerocoins*. Thus, the creator has no idea when the receiver has used these *zerocoins*. For the same reason, since the creator (and also the attacker from the network) doesn't know the address secret key $a_{sk}$ of the owner of a *zerocoin*, he could not generate a valid ZK-Proof as described in Pour function step 6. Thus, no one except the owner of a *zerocoin* can use it.

However, any transactions regarding to *basecoins* are public and to create *zerocoins* one first needs to deposit the same amount of *basecoins* to backing escrow pool, it is not hard to find out the value of the first *zerocoin* that a user spent. Therefore, it is recommended to split a newly minted *zerocoin* into new *zerocoins* using Pour function instead of directly using it, so that the value of split *zerocoins* are hidden from the public.

# 4 Privacy Preserving Ethereum-based E-Voting System

PEES is an Ethereum based, privacy preserving E-Voting system. It uses Smart Contracts to record ballots, so that no one can manipulate them once they are saved on the Blockchain. PEES also uses ZK-Proof to protect Voter's privacy against both public and EA. In the following, I'll first give an overview of PEES in Section 4.1 and then explain the security goals of PEES on an informal level in Section 4.2. In Section 4.3, I will explain the adopted mint and pour functions, ZK-Proofs and Smart Contracts that are used in PEES in detail. Finally, I will describe the full protocol of PEES in Section 4.4.

## 4.1 PEES Overview

Figure 4.1 shows all components and the relations between these components in PEES. As shown in Figure 4.1, PEES runs the two web servers: Admin Server and Client Server. These two servers are configured by PEES and the source code should be published, so that everyone can check that these two servers do the exact thing that they are expected to do. Although all ballots are saved on Blockchain, PEES still uses a shared database to save information such like election configurations, so that each web server has access to it. During an election, EA can interact with Admin Server to configure and control the pace of an election, and Voters are able to interact with the Client Server to get needed information for voting. The two Smart Contract: Registration Smart Contract and Voting Smart Contract are generated during an election by Admin Server. In the following, I will explain each component with more details.

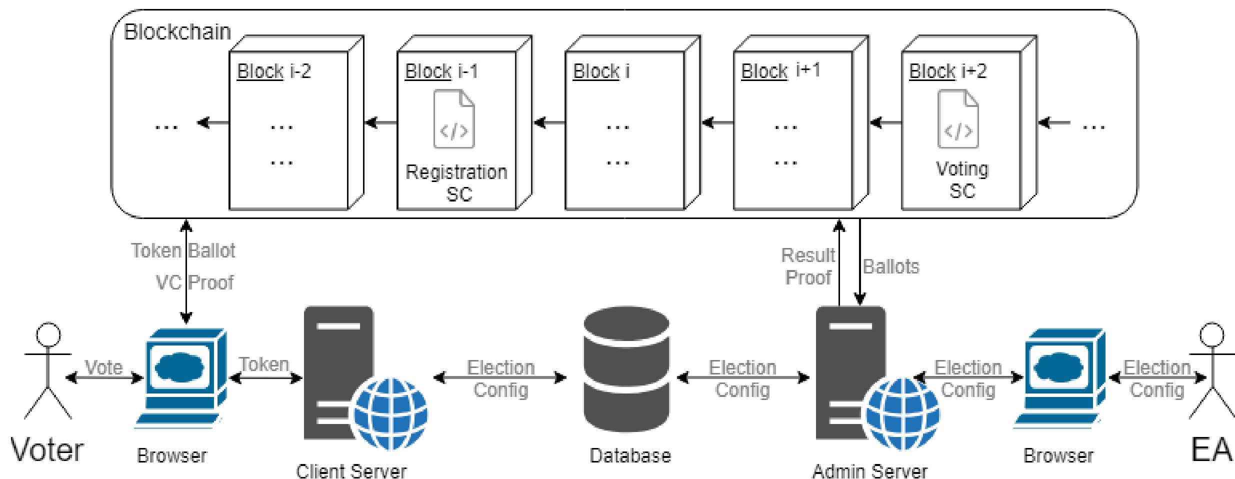



**Figure 4.1:** PEES overview

**Admin Server**
: a web server that interacts with EA through their browser. It enables EA to configure an election and authorize result computation. The configuration of an election will be saved in a shared database, so that the Client Server also has access to it. The result computation is completely done on the server-side. After EA authorized the result computation, this server will collect all ballots from the Blockchain, calculate the result, prepare a ZK-Proof of the correctness of this result and publish the result and the proof on the Blockchain.

**Client Server**
: a web server that Voters can interact with through their browser. It checks the eligibility of Voters and provides a voting interface as well. This server will provide each eligible Voter a unique token to mint the VC that will be consumed when the Voter casts a ballot. As for the voting interface, this server only provides the candidates list to the Voter's browser, the generation and minting of VCs and ballots casting are all done locally on Voter's browser but not through this server.

The connection between these two servers is a shared database. In this shared database, only the configuration of elections is saved, all ballots are still saved in Smart Contract on a Blockchain. During an election, two Smart Contracts will be generated by Admin Server and will be deployed on the Blockchain. These Smart Contracts are election related and are used to save VCs and record ballots. The following description explains the functionalities of these two Smart Contracts.

**Registration Smart Contract**
: is used to check the validity of minted VCs. Only valid VCs will be saved in this Smart Contract. When the registration time is over, a MT of all VCs will be generated, and the root hash of that MT will be saved in this Smart Contract as well.

**Voting Smart Contract**
: is used to check the format of cast ballots. Similar to the Registration Smart Contract, only the valid ballots should be saved in this Smart Contract. In addition, the final result of the election and its related ZK-Proof is also saved here, so that everyone can check it later.

There are two types of participants in PEES, namely Voters and EA. As an election organizer, EA has the full control of one election, i. e., it provides candidates list, authentication information for all eligible Voters and determines the time period of registration and voting phases. In addition, EA has the ability to manually end any phase of an election earlier than the expected time. However, both EA and Admin Server don't have to be trusted. The result published by the Admin Server should only be considered as valid, when the ZK-Proof of the result is checked to be valid.

To hold an election using PEES, EA first needs to configure it through the Admin Server and then open it for Voters. While opening an election, the encryption key and the decryption key of that election will be generated. The encryption key is then saved in the shared database, but the description key will be split using a secret sharing scheme to a subgroup of Admins from EA. During the election, Admin Server will deploy the Smart Contracts for this election. Then, when the election is ended, the subgroup of Admins which contains a share of the decryption key is asked to upload their shares to Admin Server and combine the decryption key within the Admin Server. After that, EA needs to authorize the result computation.

As for Voters, after the election is open, they first need to register to this election and then cast their vote. As mentioned above, if a person is eligible to vote for an election, he will be given a token from the Client Server that shows his eligibility to the election. Then, he needs to mint a VC locally and send a transaction along with his token to Registration Smart Contract. Later in the voting phase, each Voter can cast his ballot to Voting Smart Contract by consuming a valid VC. The VCs are consumed in private using ZK-Proof. This is aimed to protect Voter's privacy. The created ZK-Proof is also sent to Voting Smart Contract, so that the validity of the consumed VC can be checked.

After the result of an election is published in Voting Smart Contract, everyone can check it through the Client Server or directly from Blockchain. By receiving a request for the result, Client Server will return both the result and the validity of its ZK-Proof to the sender. As mentioned before, this result should only be trusted when its ZK-Proof is valid.

## 4.2 Security Goals

PEES assumes that every messages in the internet and on the Blockchain that is not encrypted are public. That means, the transactions Voters sent to Smart Contracts or VCs and ballots that saved on Blockchain is public to everyone especially to any attacker. In addition, if a party has the decryption key for the encrypted messages, then these messages are considered to be public to that party as well. This means, although the encrypted vote on the Blockchain is not public to everyone in the internet, but they are public to the malicious EA, who tries to combine the decryption key outside of the Admin Server. Then for honest Voters who don't deviate from the protocol, PEES is expected to achieve following security goals:

**accuracy**
: in PEES, accuracy demands that every **recorded** ballot should be correctly counted. That means, during the result computation, no one should be able to add, delete or manipulate any recorded ballots and the computation itself is also correct. As there are attacks on Blockchain that blocks a user from sending any transactions to Blockchain, PEES can't guarantee that every **cast** ballot will be correctly counted.

**verifiability**
: verifiability ensures that no one should be able to manipulate any ballots without being noticed. This property also enables Voters to check whether their ballots have been manipulated during the election or not.

**privacy against public**
: in PEES, privacy against public means that the vote of a Voter is kept secret from the public (i. e., from anyone that is neither a member of Voters nor a member of EA), unless the Voter himself voluntarily discloses his vote. Notice that the Voter's identity is not protected by this property as for many elections, it won't be a problem, if the identity of a Voter is revealed, as long as others don't know the exact vote of this Voter.

**conditional privacy against EA**
: Analogously, privacy against EA means that the vote of a Voter is kept secret from the EA unless the Voter himself voluntarily disclose his vote. The condition here is that a Voter

must send his mint transaction and his ballot cast transaction from two different Ethereum accounts. As EA knows both the identity of the Voter and the decryption key to the ballot, EA can easily connect the decrypted vote to the original Voter, if the Voter uses the same account for both transactions.

## 4.3 PEES in Detail

In this section, I will explain the mint and pour functions adopted from Zcash, ZK-Proofs and Smart Contracts in detail, the whole protocol of PEES is listed in Section 4.4.

### 4.3.1 Mint and Pour in PEES

Before goes into details of mint and pour function in PEES, I will first explain the token and VCs from PEES.

The token is a digital signature that is sent from the Client Server to a Voter after his authentication. If a person is eligible to vote for an election, the Client Server will compute an election related token and then send it back to him. To compute this token, the Client Server first hashes the Voter's Id *id* concatenated with the election Id *eid*, and then signs on this message. The message and the signature on this message are the token that the Client Server sends back, i. e., $token := (Hash(id \,||\, eid), Sign(Hash(id \,||\, eid)))$.

VCs are similar to *zerocoins* (see Section 3.3). Analogy to Zcash, a VC is generated using mint function and is consumed using pour function. Recall that with mint and pour, only the serial number *sn* of a *zerocoin* is revealed when consuming that *zerocoin*, the very identity of that *zerocoin*, especially the coin commitment *cm* remains in secret. PEES adapts this feature to break the connection of a Voter and its vote from EA's point of view. According to the security assumption, the token that sent from the Client Server to a Voter is known to EA. Hence, if a Voter uses this token to show his eligibility when he votes, a malicious EA will know exactly, to whom this Voter has voted. In order to hide this connection, the token must be first converted into a VC.

In PEES, a VC can be considered as an anonymous token that shows the eligibility of a Voter, therefore, each VC should be only used once, when the Voter casting his ballot. Unlike *zerocoins*, which are still a type of digital currency, VCs are not used in any payment, thus, the value of VCs is useless and there is no need to create new VCs for any receiver when one is consumed. Under these observations, we simplified *zerocoin* to the form that showed in Figure 4.2.

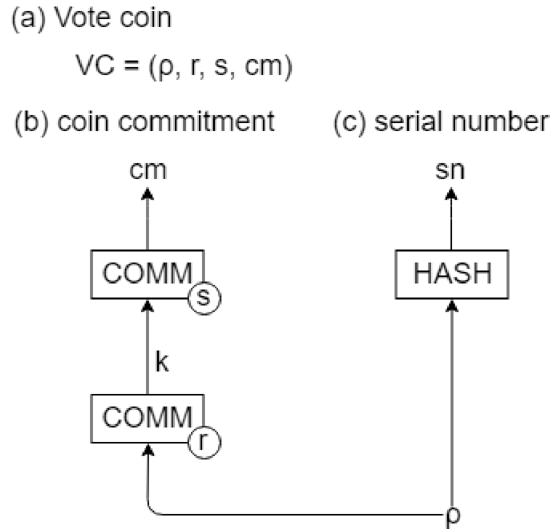



**Figure 4.2:** VC - simplified *zerocoin*

Compared to the format of a *zerocoin* in Figure 3.3, we delete the value $v$ of the coin. In addition, as a sender does not have to create new coins for the receiver, we don't need the secret key $a_{sk}$ for the serial number calculation as well (recall that in *zerocoin* the secret key of a coin owner is used in serial number calculation to prevent the coin creator to use this coin). Then, the serial number calculation is simplified to the hash of the serial number randomness $\rho$. Furthermore, recall that the commitment of the public key $a_{pk}$ in a *zerocoin* is used to guarantee that only the actual owner of the secret / public key pair can generate a valid ZK-Proof when using this coin. Since we don't need the secret key $a_{sk}$ for the serial number calculation of a VC, we don't need to commit the corresponding public key $a_{pk}$ in the first commitment as well.

The new form of a coin ($\rho$, $r$, $s$, $cm$) is the VC that is used in PEES. As the form of the coin has changed, the mint and pour function must be adapted as well. In PEES, Mint function is defined as following:

Mint(*token*, *eid*):

1. sample the serial number randomness $\rho$.
2. sample a commitment randomness $r$, then compute the first commitment $k := Comm_r(\rho)$.
3. sample another commitment randomness $s$, then compute the coin commitment $cm := Comm_s(k)$.
4. compute the serial number $sn := Hash(\rho)$.
5. send Mint transaction $\texttt{tx}_{MINT} := (eid, token, k, s, cm)$ to Registration Smart Contract.
6. return $vc := (\rho, r, s, cm)$ and $sn$.

The token here is like the *basecoin* in Zcash. Only with a valid token, a new VC can be created. While the mint transaction contains the *token* in plaintext and is directly sent to the network, it might happen that an attacker may steal the *token* from other Voters to generate his own VC. To prevent this from happening, a eligible Voters' account list can be saved in the Registration Smart Contract, so that the Registration Smart Contract will only accept mint transactions from these account. On the other hand, one can hide this *token* using a ZK-Proof as well.

As for the Pour function in PEES, it gets a VC $vc$, a serial number $sn$, a Voter's vote $v$, an election encryption key $key$ and a MT root hash $rt$ as input, and then creates a ballot which contains the encrypted vote and a corresponding ZK-Proof. At last, this function sends the ballot to Voting Smart Contract:

Pour($eid$, $vc$, $sn$, $v$, $key$, $rt$) :

1. encrypt $v$ under encryption key $key$: $V_E := Enc_{key}(v)$
2. generate a ZK-Proof $\pi_{POUR}$ for the following statement:
   " Given the MT root hash $rt$, the serial number $sn$, the encrypted vote $V_E$ and an encryption key $key$, I (Voter) know a valid VC $vc$ and the vote $v$ is well formed. "
3. assign $V_E$ and $\pi_{POUR}$ to ballot $b$, i. e., $b := (V_E, \pi_{POUR})$.
4. send Pour transaction $\texttt{tx}_{POUR} := (eid, rt, sn, b)$ to Voting Smart Contract.

As mentioned above, the Pour function in PEES does not create new VCs but create a ballot instead. As the ballot is directly sent to the Blockchain, it must not contain a vote $v$ in plaintext, otherwise the vote is leaked to the public. In the description above, we only give an idea what the ZK-Proof $\pi_{POUR}$ needs to prove, the concrete constriction such as the definition of a valid VC can be found in next section.

### 4.3.2 ZK-Proofs in PEES

There are two different ZK-Proofs in PEES. The first one is the ZK-Proof of Pour that was already mentioned in Section 4.3.1. This ZK-Proof is created when a Voter consumes his VC and casts his ballot. It proves in Zero-Knowledge (ZK) that the Voter knows the detail of a valid VC that he minted before, i. e., the Voter knows the corresponding randomness's $\rho$, $r$, $s$ and its coin commitment $cm$ and serial number $sn$. In addition, it proves that the Voter's vote is well-formed, i. e., he didn't vote for anyone who is not on the candidates list and didn't vote for more than one candidate. Formally speaking, the ZK-Proof of Pour $\pi_{POUR}$ is a ZK-Proof for the following statement:

" Given the MT root hash $rt$, the serial number $sn$, the encrypted vote $V_E$ and an encryption key $key$, I (Voter) know VC $vc := (\rho, r, s, cm)$, vote $v$, such that:

- the VC $vc$ is well formed, i. e., the two commitments $k := Comm_r(\rho)$ and $cm := Comm_s(k)$ are valid.
- the serial number calculation of consumed VC is correct, i. e., $sn := Hash(\rho)$.
- the coin commitment $cm$ is a leaf of a MT with root hash $rt$.
- let $m$ be the number of all candidates, the vote $v$ is well formed, i. e., $v$ contains only one choice and $1 \leq v \leq m$.

- the encryption of vote $v$ is correct, i. e., $V_E := Enc_{key}(v)$ ”

The root hash $rt$ refers to the root hash of the MT over all VCs. This is saved in Registration Smart Contract, hence, when verifying the proof, the public input $rt$ should be get from the Registration Smart Contract. If this input is not directly got from the Registration Smart Contract, one should first check the equality of these two root hashes. Analogue, the other public input $key$ should also be the same as the encryption key saved in the election configuration. With these inputs equal to their required values, the first three requirements guaranteed that the Voter knows the very detail of the VC he's about to spend and this VC is a VC that saved in the Registration Smart Contract which indicates the eligibility of the Voter. The last two requirements are aimed to ensure that the vote $v$ is valid and $V_E$ is indeed the ciphertext of this vote.

The other ZK-Proof in PEES is the ZK-Proof of result computation $\pi_{result}$. This ZK-Proof is generated after the Admin Server has computed the result of an election and it's aimed to show the correctness of the result computation without revealing the decryption key to the public. To show the correctness of the result computation, Admin Server has to show that it has decrypt every vote correctly and every vote has been counted correctly, i. e., $\pi_{result}$ is a ZK-Proof for the following statement:

“ Given ciphertext $V_1, V_2, \ldots, V_n$ and result $(r_1, r_2, \ldots, r_m)$ with $r_i$ represents number of votes that candidate $i$ received, I (Admin Server) know decryption key $key$ and plaintext $v_1, v_2, \ldots, v_k$, such that:

- all ciphertexts are decrypted, i. e., $k = n$.
- each ciphertext is decrypted correctly, i. e., $\forall i \in \{1, 2, \ldots, n\} : v_i := Dec_{key}(V_i)$.
- each vote has been counted correctly, i. e., $\forall i \in \{1, 2, \ldots, m\}$ : there are exact $r_i$ votes voted for candidate $i$ and $k = \sum_{i=1}^{m} r_i$ ”

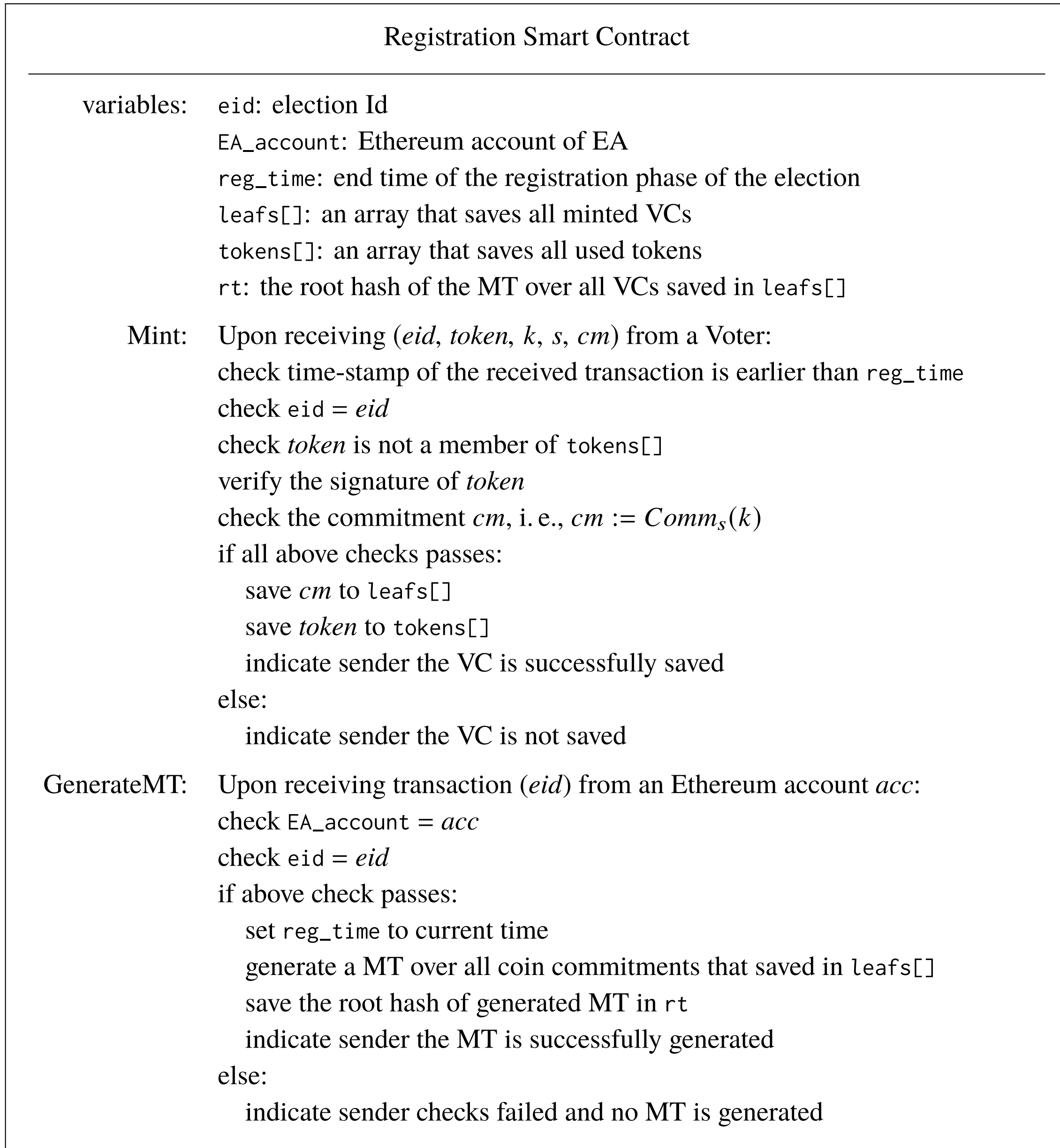
Similar to the ZK-Proof of Pour, all ciphertexts are encrypted votes and they should be retrieved from Voting Smart Contract or at least be checked to be the same as all the encrypted votes that were saved in Voting Smart Contract. Without this guarantee, one may generate a valid proof with all votes being manipulated during the result computation.

### 4.3.3 Smart Contracts in PEES

During an election, two election related Smart Contracts will be generated, namely Registration and Voting Smart Contract. The Registration Smart Contract can also be seen as a VC pool where all VCs of eligible Voters are saved. This Smart Contract provides two main functions, `Mint` and `GenerateMT` as shown in Figure 4.3.

When generating Registration Smart Contract, three variables should be hard-coded, namely `eid`, `EA_account` and `reg_time`. As mentioned before, each Smart Contract is election related, therefore it needs the variable `eid`, which indicates to which election this Smart Contract belongs to. The `reg_time` is a variable, that is specified by EA and indicates the registration end time. Every mint-transaction that arrives after this time will be considered as invalid and the coin commitment in this transaction will not be saved in this Smart Contract. The last hard-coded variable is EA's Ethereum account. This variable is used to guarantee that only EA can authorize the MT generation

over all saved VCs. Apart from these hard-coded variables, there are three other variables that save coin commitments, used tokens and the root hash of the MT over all saved coin commitments respectively. The states of these variables may change when one of the functions from this Smart Contract is successfully executed.



Registration Smart Contract

variables: `eid`: election Id
`EA_account`: Ethereum account of EA
`reg_time`: end time of the registration phase of the election
`leafs[]`: an array that saves all minted VCs
`tokens[]`: an array that saves all used tokens
`rt`: the root hash of the MT over all VCs saved in `leafs[]`

Mint: Upon receiving (*eid*, *token*, *k*, *s*, *cm*) from a Voter:
check time-stamp of the received transaction is earlier than `reg_time`
check `eid` = *eid*
check *token* is not a member of `tokens[]`
verify the signature of *token*
check the commitment *cm*, i. e., $cm := Comm_s(k)$
if all above checks passes:
  save *cm* to `leafs[]`
  save *token* to `tokens[]`
  indicate sender the VC is successfully saved
else:
  indicate sender the VC is not saved

GenerateMT: Upon receiving transaction (*eid*) from an Ethereum account *acc*:
check `EA_account` = *acc*
check `eid` = *eid*
if above check passes:
  set `reg_time` to current time
  generate a MT over all coin commitments that saved in `leafs[]`
  save the root hash of generated MT in `rt`
  indicate sender the MT is successfully generated
else:
  indicate sender checks failed and no MT is generated

**Figure 4.3:** Variables and functions in Registration Smart Contract

`Mint` function is used to convert a valid token to an anonymous VC. This function receives a mint transaction as described in Section 4.3.1, it then checks the arrival time and the election Id of this mint transaction to make sure that this transaction arrives within the registration time period and is targeting the right Smart Contract. After that, `Mint` function checks the validity of the digital

signature from *token* and makes sure that this *token* has not been used before. At last it checks the correctness of the coin commitment *cm* by recalculation the commitment *cm'* using *k* and *s* and check the equality of *cm'* and *cm*. If all those checks pass, `Mint` function will save the coin commitment *cm* in `leafs[]`, the corresponding *token* in `tokens[]` and indicates the sender that he has successful minted a VC, otherwise it will indicate the sender that one of the checks is failed and the VC is not saved.

Recall that in Zcash, each minted *zerocoin* should have some deposit *basecoins* with the same value, in PEES, the Client Server signed *token* can be considered as the basecoin for VC. However, as VC doesn't have value, the only "value" that has been converted from the basecoin *token* to the anonymous VC is the eligibility of the Voter. Thus, the successful save of the coin commitment of a VC in Registration Smart Contract with hard-coded `eid` means that the owner of the VC is eligible to vote for the election with Id equals to `eid`. Therefore, when the Voter casts his ballot, he will use the anonymous VC instead of the *token* to show his eligibility.

The other function `GenerateMT` in Registration Smart Contract provides the interface for EA to generate the MT over all saved coin commitments in `leafs[]`. As this function directly ends the registration time, it should only accept transactions from EA, thus, the sender's account must be compared with the hard-coded `EA_account` at the beginning. When calling this function, an election Id should also be given as the input and it should be checked with the hard-coded `eid` to avoid the case that EA has send the transaction to a wrong Smart Contract.

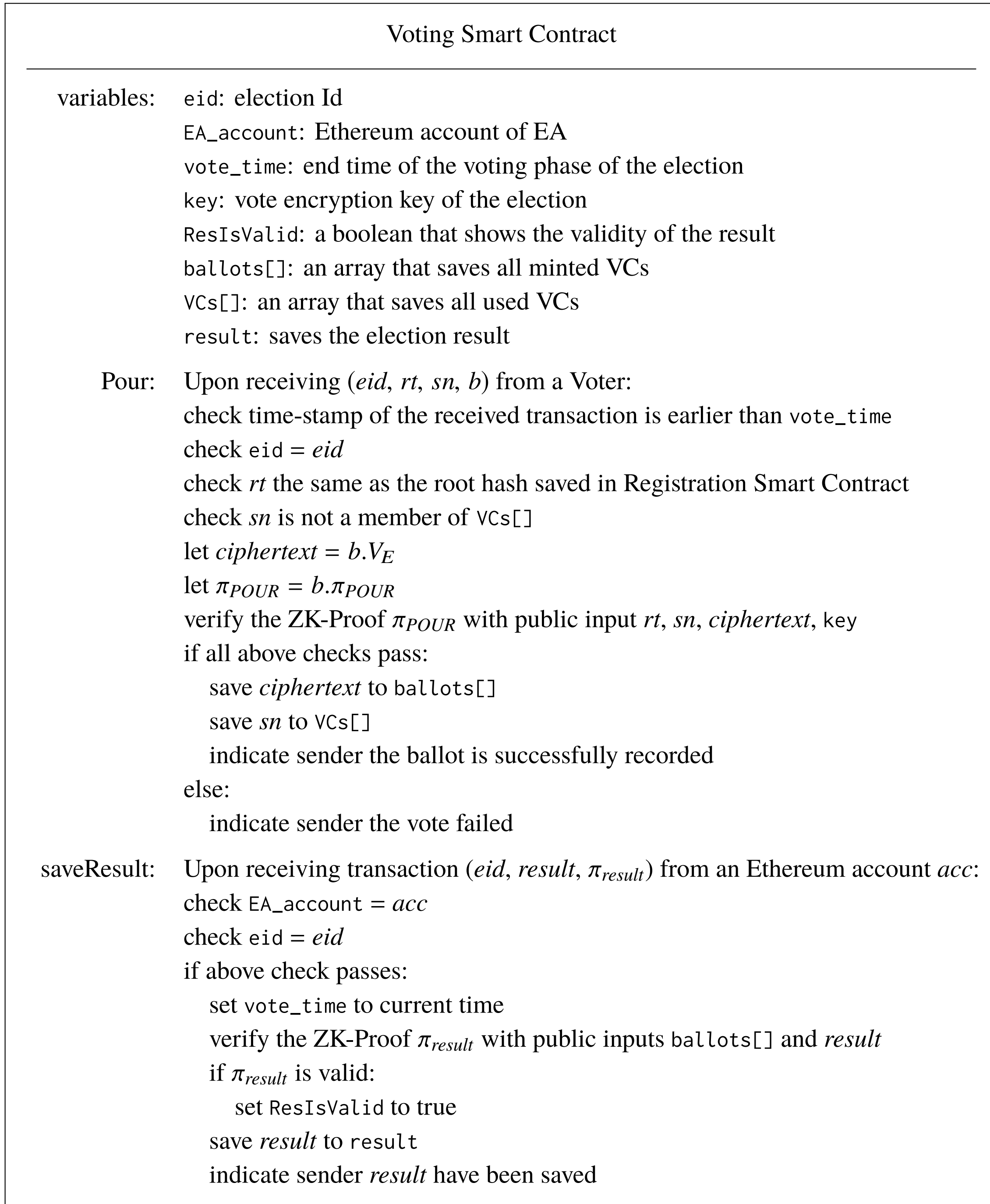
If the above checks pass, the registration phase will be ended, and the MT will be generated as well. Due to the costly write operations, the full MT will not be saved in the Registration Smart Contract, but only its root hash will be saved in *rt*. Recall that the root hash of a MT is basically the hash value of all its leafs, it is enough to only save the root hash of the generated MT to prevent anyone to alter the saved coin commitments. However, as EA is the only party who can call this function, it should be implemented such that the EA can only call it once. Otherwise, a malicious EA could just manipulate some coin commitments and then regenerate the MT.

The other Smart Contract that will be generated during an election is the Voting Smart Contract. This Smart Contract is used to verify the Pour ZK-Proof as described in Section 4.3.2, and record ballots. In addition, the final result and the ZK-Proof of result computation is also saved or verified in this Smart Contract.

Similar to the Registration Smart Contract, Voting Smart Contract also has hard-coded `eid`, `EA_account` and `vote_time`. Additionally, it hard codes the encryption key in `key`, so that this Smart Contract doesn't need to interact with any party to get this key when verifying the ZK-Proof $\pi_{POUR}$ as described in Section 4.3.2. Apart from these four hard-coded variables, there are two arrays `ballots[]` and `VCs[]` that save the cast ballots (only the ciphertext of a vote) and the serial numbers of used VCs respectively. The variable `result` is used to save the final result of an election. The variable `ResIsValid` is a boolean that indicates the validity of the ZK-Proof $\pi_{result}$. This variable is default set to `false` and it can be only set to `true` iff the ZK-Proof $\pi_{result}$ is verified to be valid. The variables and functions from Voting Smart Contract is listed in Figure 4.4.

Recall that to cast his ballots, a Voter needs to send a Pour transaction (*eid*, *rt*, *sn*, *b*) to Voting Smart Contract. More precisely, this transaction is targeting the Pour function in Voting Smart Contract. When receiving such a transaction, the Pour function first checks the transaction arrive time, election Id and the double usage of *sn* similar to what Mint function does in Registration Smart Contract. It also checks the equality of received *rt* and the root hash `rt` saved in Registration Smart

Contract as mentioned in Section 4.3.2. Then, it phases the encrypted vote $V_E$ and the ZK-Proof $\pi_{POUR}$ from ballots $b$ and verifies the $\pi_{POUR}$. If all these checks and verification pass, the Pour function will save $V_E$ to `ballots[]` and the serial number $sn$ to `VCs[]`, otherwise, it indicates the sender, his vote failed.



Voting Smart Contract

variables:
`eid`: election Id
`EA_account`: Ethereum account of EA
`vote_time`: end time of the voting phase of the election
`key`: vote encryption key of the election
`ResIsValid`: a boolean that shows the validity of the result
`ballots[]`: an array that saves all minted VCs
`VCs[]`: an array that saves all used VCs
`result`: saves the election result

Pour:
Upon receiving ($eid$, $rt$, $sn$, $b$) from a Voter:
check time-stamp of the received transaction is earlier than `vote_time`
check `eid` = $eid$
check $rt$ the same as the root hash saved in Registration Smart Contract
check $sn$ is not a member of `VCs[]`
let $ciphertext = b.V_E$
let $\pi_{POUR} = b.\pi_{POUR}$
verify the ZK-Proof $\pi_{POUR}$ with public input $rt$, $sn$, $ciphertext$, `key`
if all above checks pass:
  save $ciphertext$ to `ballots[]`
  save $sn$ to `VCs[]`
  indicate sender the ballot is successfully recorded
else:
  indicate sender the vote failed

saveResult:
Upon receiving transaction ($eid$, $result$, $\pi_{result}$) from an Ethereum account $acc$:
check `EA_account` = $acc$
check `eid` = $eid$
if above check passes:
  set `vote_time` to current time
  verify the ZK-Proof $\pi_{result}$ with public inputs `ballots[]` and $result$
  if $\pi_{result}$ is valid:
    set `ResIsValid` to true
  save $result$ to `result`
  indicate sender $result$ have been saved

**Figure 4.4:** Variables and functions in Voting Smart Contract

Notice that, the ZK-Proof $\pi_{POUR}$ only proves that the sender of the transaction (Voter) has a valid VC but cannot prove that this VC hasn't been used, thus, the revealed serial number *sn* must be saved after spending the VC so that it will be detected when someone is trying to use the same VC to vote again.

Similar to `GenerateMT` function in Registration Smart Contract, `saveResult` can only be called from EA as well, since it also ends the voting phase immediately. After the regular EA account and election Id checks, the `saveResult` function will end the current voting phase (if it is not ended) and begin to verify the ZK-Proof $\pi_{result}$. Notice that all encrypted votes which are a set of the public inputs for $\pi_{result}$ are directly obtained from the variable `ballots[]`. As these data are saved on Blockchain, it prevents EA from manipulating the votes and providing a false proof by giving altered encrypted votes as public inputs.

The `saveResult` function will always save the *result* in `result` no matter the ZK-Proof $\pi_{result}$ is valid or not, however the variable `ResIsValid` can only be set true iff the ZK-Proof is valid. The purpose of this construction is to provide public a result no matter EA is malicious or not. However, one should only trust this result if the variable `ResIsValid` is set to `true`.

Unlike `GenerateMT` function from Registration Smart Contract, the `saveResult` can be called multiple times from EA as long as the variable `ResIsValid` remain private to Voting Smart Contract. This means, a malicious EA could try to generate as many false ZK-Proofs as he can and send it to `saveResult`, but as long as EA can't change the value of `ResIsValid` directly, the probability that EA convinces the public, that it has calculated the result correctly, should be the same as the probability that anyone could forge a valid ZK-Proof with invalid inputs.

## 4.4 PEES Protocol

PEES is an E-Voting system that can be used for any elections. The protocol that described in this section is the protocol of an election using PEES. The process of an election can be divided into 4 phases, apart from phase 0, each phase requires the participants of both Voters and EA. The detail of each phases is listed below and a sequence diagram of the process of one election can be found in Figure 4.5.

- **Phase 0: set-up**
  This phase requires only the participants of EA. In set-up phase, the party EA is asked to initialize and open an election. That is, prepare the questions and candidates, decide a list of eligible Voters and generate the encryption / decryption key pairs for the election. This information will be saved in the shared data bank through Admin Server but the decryption key will be split in shares and be given to a subgroup of EA.
- **Phase 1: registration**
  To begin phase 1, EA has to first determine a time period for registration phase. After receiving the time period, the Admin Server will generate a Registration Smart Contract as described in Section 4.3.3 and deploy it on Blockchain. In addition, the registration time period will also be saved in the election configuration in the shared database so that Client Server knows in which phase the current election is in.

Within the registration time period, each Voter is asked to authenticate himself to the Client Server. As mentioned before, if a Voter is eligible to vote for the election, he will receive a *token* from Client Server. After receiving the *token*, it is recommended that each Voter mints his VC immediately using the `Mint` function that has been introduced in Section 4.3.1. Voters can decide to mint their VCs afterwards, in that case, they need to face the risk that their `mint`-transactions may arrive later than the registration time period and therefore not be saved in Registration Smart Contract.

- **Phase 2: voting**
  Similar to phase 1, EA has to determine a time period for voting phase to start phase 2. Notice that the start time of voting phase doesn't have to be later than the end time of the registration end time, i. e., EA can end phase 1 earlier than it is scheduled, but this is not recommended.

  On receiving the time period for voting phase, the Admin Server first sends a transaction to Registration Smart Contract to issue the MT generation. After that, the Admin Server generates a Voting Smart Contract as described in Section 4.3.3 and then deploys it on the Blockchain. At last, the Admin Server saves the voting time period in the shared Database.

  When the voting time periods begins, each Voter is able to get the candidates list from Client Server. After a Voter has selected his preferred candidate, he then prepare the public inputs $(rt, sn, V_E, key)$ and corresponding witness $(vc := (\rho, r, s, cm), v)$ for the ZK-Proof $\pi_{POUR}$ and using the `pour` function that has been described in Section 4.3.1 to generate this ZK-Proof and send the `Pour` transaction to Voting Smart Contract.

  Notice that all these computations are done locally on Voter's computer and are directly sent to the Smart Contract. That means, the Client Server will not learn any information, especially the witness, from the Voter.

- **Phase 3: compute result**
  When EA wants to close an election (same as in phase 2, EA could close an election earlier but is not recommended), each Admin is asked to upload his share of the decryption key to the Admin Server. When there are enough shares, the decryption key will be combined, and EA can then authorize the result computation.

  On receiving the authorization to calculate the result, the Admin Server will first get all saved ballots from the Voting Smart Contract and then compute the result. After the result computation, the Admin Server needs to create a ZK-Proof $\pi_{result}$ as described in Section 4.3.2 and publish the result along with the ZK-Proof to the Voting Smart Contract. At last, the Admin Server will change the election state in the shared database so that the Client Server knows that the result is already calculated.

  After the result of an election is published, there are two ways to check it. On the one hand, one can get the result through the Client Server, which provides an interface to show both the election result and the validity of the corresponding ZK-Proof. On the other hand, one can also directly interact with the Blockchain to retrieve the result and the validity of the proof. There is no big difference between these two methods, no matter which method one uses to get the result, he should always keep in mind, that the result should only be trusted when the corresponding ZK-Proof $\pi_{result}$ is valid.

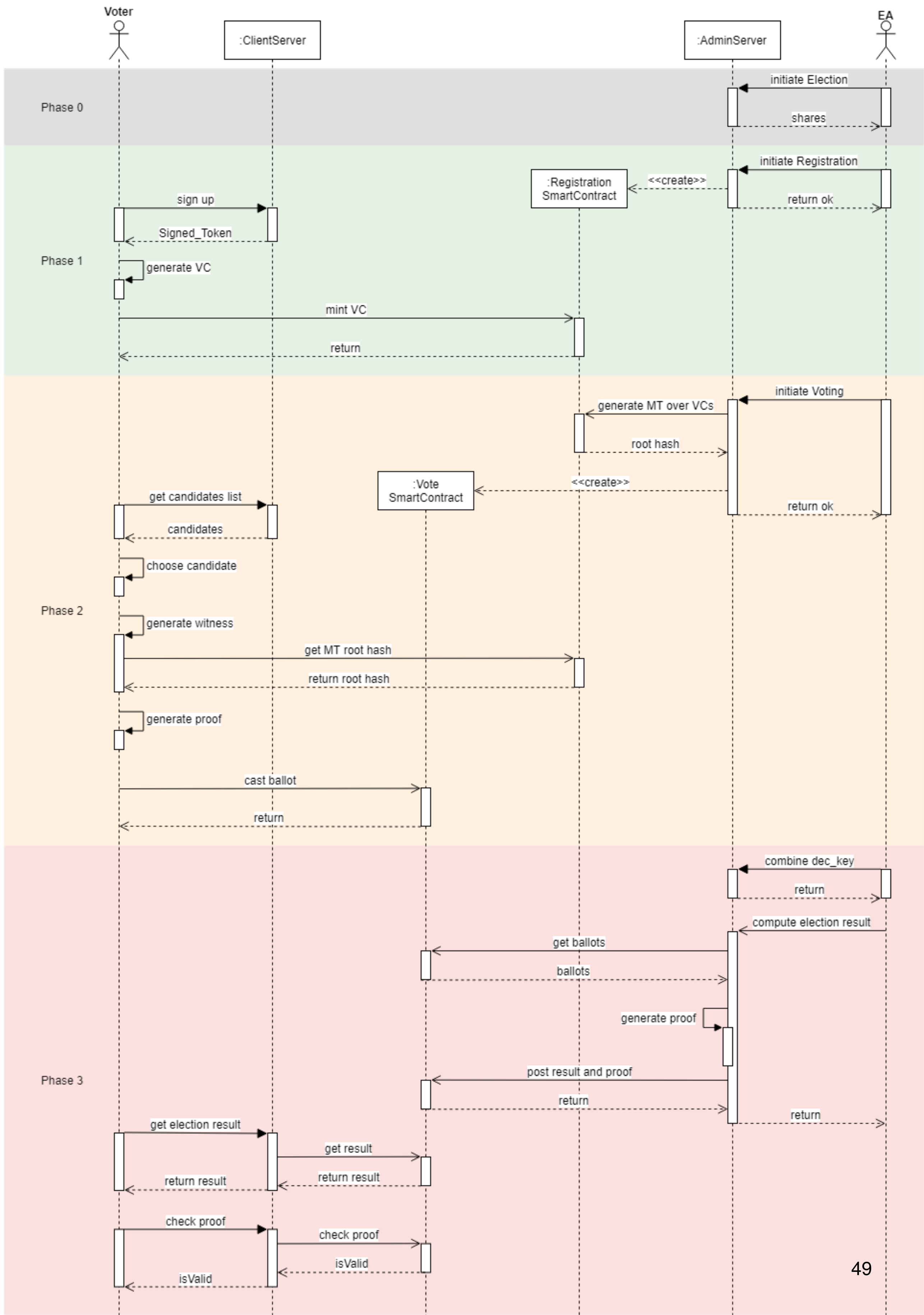





**Figure 4.5:** The entire process of an election in PEES

# 5 Implementation

In this section, I will introduce an implementation of the demo of PEES. This demo contains an implementation of two web servers, two Smart Contracts and ZK-Proof generators for $\pi_{POUR}$ and $\pi_{Result}$. The connection of the web servers and Smart Contracts are the same as described in Section 4.3. For the ZK-Proof generators, they expected a file that contains the public inputs and prover witnesses as the input and output the generated proof in another file. This file will be later sent to Smart Contract for a verification. In the following, I will first introduce the program languages that are used to implement this demo and the dependency of this implementation in Section 5.1. Then, I will give the concrete implementations of ZK-Proof generators, Smart Contracts and web servers from Section 5.3 to Section 5.4.

## 5.1 Program Languages and Dependency

### 5.1.1 Program Languages

#### JavaScript

JavaScript is a well-known script language for Web pages [Kam18]. It is mainly used in browsers, but many non-browser environments also use it, such as Node.js [Fou]. PEES uses JavaScript to implement front-end functions and uses the Express Framework [Str] provided by Node.js to develop Admin Server and Client Server. In addition, PEES uses Web3 [Eth], a JavaScript API provided by Ethereum to interact with Smart Contracts from both front-end and back-end.

#### Solidity

Solidity is a contract-oriented language for implementing Smart Contracts on Ethereum [CAa]. It was influenced by C++, Python and JavaScript. Each Smart Contract in Solidity is like a `Class` in Java, it has its own state variables and functions. Recall that functions of Smart Contracts are executed when miners start mining the block that contains these functions. Therefore, the function call of functions in Smart Contract is usually asynchronous. To get the result of the function call, Ethereum provides a mechanism called `Event`. An `Event` is trigged when the function that contains this `Event` is executed and when it is triggered, it will deliver the desired values back to the transaction sender. For more information about Solidity please refer to [CAa].

### 5.1.2 Dependency

#### Libsnark

libsnark is a C++ implementation of zk-SNARK schemes [Laba]. This library implements a preprocessing zkSNARK (ppzkSNARK) scheme. The term "preprocessing" means that before a proof of a NP-statement can be created and verified, one needs to first represent this NP-statement as a constrain system or circuit and then run a generator algorithm to create corresponding public parameters. Briefly speaking, using libsnark includes following steps: express the desired NP-statement as R1CS (or any other languages that libsnark supports), run libsnark's generator algorithm to create public parameters, run libsnark's prover algorithm to create proofs of satisfying input for the desired NP-statement and finally, run libsnark's verifier algorithm to check the proofs.

To express NP-statements as R1CS, libsnark provides gadget library for gadget creating. Gadgets are sets of constrains that interprets some basic functions. The gadget library already provides some gadgets for basic functions such like hash function or Merkle Authentication Path calculation. Using these gadgets and other basic constrains, one can create complex gadget for NP-statements he wants to prove.

#### Jsnark

Jsnark is a circuit building library for preprocessing zkSNARK [Kos]. It is a Java library for building circuit, which is another NP-language that libsnark supports, according to pinocchio system [PHGR16]. Jsnark uses libsnark as the backend for the generated circuit. It provides a C++ interface that uses libsnark to read the generated circuit, generate the corresponding proof and verify the proof (if needed).

The reason that PEES not only uses libsnark but also Jsnark is that this library provides an implemented Modulus gadget, which is important for RSA encryption and decryption circuit generation. With RSA encryption and decryption being two important components in ZK-Proofs $\pi_{POUR}$ and $\pi_{Result}$, Jsnark enables an easier circuit generation when implementing the ZK-Proof generators for these two proofs.

#### Other Library

The main dependency of PEES are libsnark and Jsnark. Apart from that, it also uses following libraries:

**Jsoncpp [BC]**
: JSON (JavaScript Object Notation) is a lightweight data-interchange format that is wildly used in JavaScript. This library simplifies the reading and creating of JSON objects within C++. Using this library, one can create or read JSON objects similar to JavaScript.

**jsbn [Tom]**

`jsbn` library is a fast, portable implementation of large-number math in pure JavaScript. This library is designed for use in browser and it also provides a RSA interface for encryption / decryption.

**node-bignumber [Lau]**

This library is a packaging of the original code from `jsbn` for a use case in Node.js.

**ssss-js [Gab]**

This library is a JavaScript version of Shamir's Secret Sharing Scheme [Sha79]. It can be used with Node.js to split secrets in multiple shares and recombine the secret through a number of shares that are bigger than the pre-determined threshold. This library only supports splitting hex-strings, if a secret is not represented as a hex-string, this library provides a `str2hex` function to convert the secret into a hex-string.

## 5.2 Zero-Knowledge Proof Generator

This section contains the implementations of two ZK-Proof generators. Each generator takes the corresponding public inputs and prover witnesses as input and generates ZK-Proofs as described in Section 4.3.2. Using `libsnark` as the underlying library for zk-SNARK, each statement that needs to be proved must be first interpreted as an instance of a NP-language that `libsnark` supports. Then a set-up phase for each instance is needed. During the set-up phase, a proving key and a verifying key that related to the instance will be generated. The proving key is needed for provers to generate proofs and the verifying key is used by verifier to check the corresponding proof. As mentioned above, these two keys are related to the instance, i. e., related to the statement that needs to be proven. This means, for the same statement but with different inputs and witnesses, the prover can use the same proving key to generate different proofs and the verifier can using the same verifying key to verify these proofs.

In the following, I will introduce the implementation for ZK-Proof $\pi_{POUR}$ in Section 5.2.1 and the implementation for ZK-Proof $\pi_{Result}$ in Section 5.2.2.

### 5.2.1 Generator of Pour Proof $\pi_{POUR}$

Recall the statement of ZK-Proof of Pour $\pi_{POUR}$ from Section 4.3.2. In the implementation of the demo, this ZK-Proof is split into two parts, where the first part only proves the format of a VC and the second parts proves the correctness of the vote. The first parts of the $\pi_{POUR}$ is called Proof of Coin ($\pi_{Coin}$) and it proofs the following statement:

" Given the MT root hash $rt$ and the serial number $sn$, I (Voter) know VC $vc := (\rho, r, s, cm)$, such that:

- the VC $vc$ is well formed, i. e., the two commitments $k := Comm_r(\rho)$ and $cm := Comm_s(k)$ are valid.
- the serial number calculation of consumed VC is correct, i. e., $sn := Hash(\rho)$.
- the coin commitment $cm$ is a leaf of a MT with root hash $rt$. "

The second part of $\pi_{POUR}$ is called Proof of Vote Format ($\pi_{Vote}$) and it proofs the following statement:

“ Given the encrypted vote $V_E$ and an encryption key $key$, I (Voter) know vote $v$ such that:

- let $m$ be the number of all candidates, the vote $v$ is well formed, i. e., $v$ contains only one choice and $1 \leq v \leq m$.
- the encryption of vote $v$ is correct, i. e., $V_E := Enc_{key}(v)$ ”

The reason that $\pi_{POUR}$ is split into two proofs is that the library `libsnark` doesn't provide a RSA encryption gadget, which is crucial for $\pi_{Vote}$. Another library, `Jsnark`, provides the needed RSA encryption gadget, but the performance is not as good as `libsnark`. Thus, we have implemented $\pi_{Coin}$ in `C++` using `libsnark` and $\pi_{Vote}$ in `Java` using `Jsnark`.

Both generators reads the same JSON-file `vote_witness.json` as the input, but they generate different ZK-Proofs. Listing 5.1 shows the format of the input JSON-file. It contains an election Id `eid`, the ciphertext of the encrypted vote `vote_ct` and a set of witness. The first two sets of `witness`: `coin_witness` and `auth_path` are necessary for generating $\pi_{Coin}$ while the third set of `witness`: `vote_witness` and `vote_ct` are needed for generating $\pi_{Vote}$.

**Listing 5.1** Format of `vote_witness.json`

```
{
  "eid" : "...",
  "vote_ct" : "0x...",
  "witness" : {
               "coin_witness": {...},
               "auth_path": {...},
               "vote_witness": {...}
              }
}
```

Notice that although it is called “`witness`”, the corresponding public inputs are also contained in `coin_witness`, `auth_path` and `vote_witness` for better grouping. The concrete variables of this 3 witness sets will be introduced below.

#### Proof of Coin $\pi_{Coin}$

As mentioned above, the implementation of $\pi_{Coin}$ is in `C++` using `libsnark`. The program expected an `eid` and two JSON objects: `coin_witness` and a `auth_path` from `vote_witness.json` as input.

**Listing 5.2** Format of coin_witness and auth_path

```
  "coin_witness" : {
                    "cm": {...},
                    "k": {...},
                    "s": {...},
                    "rho": {...},
```

```
                    "r": {...},
                    "sn": {...}   //public input
                  }

"auth_path"  :  {
                    "tree_depth": {...},
                    "path": [...],
                    "address_bits": {...},
                    "address": {...},
                    "root": {...}   //public input
                 }
```

Listing 5.2 shows the format of these two JSON objects. Except the variables that are marked as public input, all other variables are prover witnesses to $\pi_{Coin}$. The meaning and relationship of variables `cm, k, s, rho, s, sn` are the same as all components in a VC as shown in Figure 4.2. For JSON object `auth_path`, it contains the necessary witnesses and public inputs for the third statement of $\pi_{Coin}$, which is a Merkle Proof as explained in Section 2.5. The variable `tree_depth` donates the depth of the MT with root hash `root`. The variable `path` is an array that contains a merkle authentication path as defined in Section 2.5. The `address_bits` is a binary representation of the integer value `address`. Each bit of `address_bits` indicates the position of a corresponding hash value from `path`. The bit `1` means the corresponding hash value in `path` is on the left (i. e., the hash value is from a node that is the leftchild with respect to its parent) when rebuilding the branch from a leaf to the root of a MT, while bit `0` means the hash is on the right.

As `libsnark` takes care of the verifying / proving key generation and proof generation, the main work to generate a ZK-Proof using `libsnark` is to interpret the NP-statement as a NP-language that `libsnark` supports. The NP-language used in this ZK-Proof generator is R1CS. To do that, `libsnark` provides a library `gadgetlib1`, which contains pre-implemented gadgets for some basic functions such like Merkle Proof, for programmers to implement their own constrain systems. In this ZK-Proof generator, to express the constrain system of $\pi_{Coin}$, two gadgets are needed. The first gadget is called `coin_gadget`, it builds up a constrain system that checks the format of a VC, i. e., the first two statements in $\pi_{Coin}$. The second gadget is called `complete_coin_gadget` which adds the constrains of the third statement in $\pi_{Coin}$ to the constrain system built by `coin_gadget`. In addition, a `coin_wrapper_gadget` is needed. This gadget wraps `complete_coin_gadget` so that it can be used more easier in the `main` function.

Each these gadgets are inherited from the class `gadget` provided by `gadgetlib1`. As the inherited class, two functions of class `gadget` must be implemented individually. The first function is called `generate_r1cs_constraints`, this function is used to generate the concrete constrain system of that defined by a gadget. The second function is called `generate_r1cs_witness`, this function assigns the inputs or witnesses to the corresponding variables, and if a variable does not have an assignment, a corresponding value will be evaluated based on the constrain system and the values of other variables.

**Listing 5.3** Constructor of coin_gadget

```
1 coin_gadget(protoboard<FieldT> &pb,
2             const digest_variable<FieldT> &sn, const digest_variable<FieldT> &rho,
3             const digest_variable<FieldT> &r, const digest_variable<FieldT> &k,
```

```
4             const digest_variable<FieldT> &s, const digest_variable<FieldT> &cm,
5             const std::string &annotation_prefi) :
6   gadget<FieldT>(pb, annotation_prefi), sn_var(sn), rho_var(rho),
7   r_var(r), k_var(k), s_var(s), cm_var(cm)
8   {
9     block1_digest.reset(new digest_variable<FieldT>(pb, sha256_512digest_len, "block1"));
10    block2_digest.reset(new digest_variable<FieldT>(pb, sha256_512digest_len, "block2"));
11
12    // Initialize the hash gadget
13    h1_gadget.reset(new sha256_256_bits_gadget<FieldT>(
14                                  this->pb, rho_var, sn_var, "sn == hash(rho)"));
15    h2_gadget.reset(new sha256_512_bits_gadget<FieldT>(
16                                  this->pb, *block1_digest, k_var,   "k == hash(rho || r)"));
17    h3_gadget.reset(new sha256_512_bits_gadget<FieldT>(
18                                  this->pb, *block2_digest, cm_var, "cm == hash(k || s)"));
19    }
```

Listing 5.3 shows the constructor of `coin_gadget`. It takes all 6 variables from `coin_witness` and a `protoboard` as input and checks the connection of these variables as illustrated in Figure 4.2. The `protoboard` is a class provided by `gadgetlib1` where all gadgets are connected to build one bigger constrain system. The `block1_digest` and `block2_digest` are two shared pointer that points to two `digest_variables`, which serves as temporary variables that save the concatenations `rho||r` and `k||s` respectively. The 3 hash function gadgets `h1_`, `h2_` and `h3_gadget` are used to check the correct hash calculation according to SHA256 [FIP95] with different input length. As the default hash gadget provided by `gadgetlib1` only takes care of hash compression function [MVO96] with input length 512 bits, we first implement two gadgets: `sha256_256_bits_gadget` and `sha256_512_bits_gadget` that deal with different input length and add the finalizing step according to SHA256 protocol to the hash gadget provided by `gadgetlib1`. Each self-implemented hash gadget builds up a group of constrains that checks the validity of a hash calculation. That means, for the given input `x` and hash value `h`, it checks whether $h = SHA256(x)$. The exact equation that is checked is annotated in Listing 5.3 as the annotation string for each hash gadget. Together, these gadgets build up a constrain system that checks all 3 hash calculations for each component of a VC.

**Listing 5.4** Other functions of coin_gadget

```
1  void generate_r1cs_constraints()
2  {
3      //ensure the hashes validate
4      h1_gadget->generate_r1cs_constraints();
5      h2_gadget->generate_r1cs_constraints();
6      h3_gadget->generate_r1cs_constraints();
7
8  }
9
10 void generate_r1cs_witness()
11 {
12     block1.reset(new block_variable<FieldT> (this->pb, {rho_var.bits, r_var.bits}, "rho||r"));
13     block2.reset(new block_variable<FieldT> (this->pb, {k_var.bits, s_var.bits}, "k||s"));
14
15     block1_digest->generate_r1cs_witness(block1->get_block());
```

```
16      block2_digest->generate_r1cs_witness(block2->get_block());
17
18      //generate r1cs witness for hash compression func
19      h1_gadget->generate_r1cs_witness();
20      h2_gadget->generate_r1cs_witness();
21      h3_gadget->generate_r1cs_witness();
22  }
```

To generate the constrain system that `coin_gadget` built, one needs to call the `generate_r1cs_constraints` function from an instance of `coin_gadget`. As shown in Listing 5.4, this function recursively calls the `generate_r1cs_constraints` function from all gadgets that is used in `coin_gadget`. As this function is called from an instance of `coin_gadget`, the generated constrain system is saved in the `protoboard` that is used to initialize this instance.

In the other function `generate_r1cs_witness`, the given witnesses and public input will be assigned to the corresponding variables. As shown in Listing 5.4, the concatenations of `rho||r` and `k||s` will be assigned to variables `block1_digest` and `block2_digest` respectively. Then the value of other variables that are needed for the hash validity check will be evaluated by line 19 to line 21. With every variable being assigned to a value, the `protoboard` can check whether the constrain system is fulfilled or not.

The second gadget `complete_coin_gadget` is similar to `coin_gadget`. Listing 5.5 shows the constructor of `complete_coin_gadget`. The `c_gadget` is an instance of `coin_gadget`, the `mtcr_gadget` is a `merkle_tree_check_read_gadget` that is provided by `gadgetlib1`, which checks whether a given value (here `cm_var`) is a leaf of the MT with root hash `root_var`. As this gadget is provided by `libsnark`, we refer readers to their documentation [Laba] for more details of this gadget. In `complete_coin_gadget`, the constrains of `coin_gadget` and the constrains of `merkle_tree_check_read_gadget` will be grouped together on the `protoboard` that is used to initialize the `complete_coin_gadget`. Together, these constrains represents all 3 statement in $\pi_{Coin}$.

The `generate_r1cs_constraints` and `generate_r1cs_witness` functions of `complete_coin_gadget` are similar to the same functions in `coin_gadget`, it recursively calls the `generate_r1cs_constraints` and `generate_r1cs_witness` of the gadgets that is used in `complete_coin_gadget`.

**Listing 5.5** Constructor of complete_coin_gadget

```
1  complete_coin_gadget(protoboard<FieldT> &pb, const digest_variable<FieldT> &sn,
2              const digest_variable<FieldT> &rho, const digest_variable<FieldT> &r,
3              const digest_variable<FieldT> &k, const digest_variable<FieldT> &s,
4              const digest_variable<FieldT> &cm, const digest_variable<FieldT> &root,
5              const merkle_authentication_path_variable<FieldT, HashT> &path,
6              const pb_variable_array<FieldT> &address_bits, const size_t tree_depth,
7              const pb_variable<FieldT> &flag, const std::string &annotation_prefi) :
8    gadget<FieldT>(pb, annotation_prefi), sn_var(sn), rho_var(rho), r_var(r),
9    k_var(k), s_var(s), cm_var(cm), root_var(root), path_var(path),
10   address_bits_var(address_bits), tree_depth(tree_depth), flag(flag)
11   {
12        // Initialize the coin_gadget
13        c_gadget.reset(new coin_gadget<FieldT>(this->pb, sn_var, rho_var, r_var,
14                                                  k_var, s_var, cm_var, "coin_gadget"));
```

```
15
16        // Initialize the merkle_tree_check_read_gadget
17        mtcr_gadget.reset(new merkle_tree_check_read_gadget<FieldT, HashT>(
18                                this->pb, tree_depth, address_bits_var, cm_var,
19                                root_var, path_var, flag, "merkle_tree_check_read_gadget"));
20  }
```

Although the `complete_coin_gadget` already builds up the whole constrain system for $\pi_{Coin}$, to generate a ZK-Proof from that, we still need to tell the program which variables are witnesses that need to be hide, and which variables are public inputs. This is done in the `coin_wrapper_gadget`.

**Listing 5.6** Constructor of coin_wrapper_gadget

```
1 coin_wrapper_gadget(protoboard<FieldT> &pb, const size_t &tree_depth) :
2 gadget<FieldT>(pb, "test_vote_gadget"), tree_depth(tree_depth)
3 {
4     // Allocate space for the verifier input.
5     const size_t input_size_in_bits = sha256_256digest_len * 2;
6
7     {
8       // We use a "multipacking" technique which allows us to constrain
9       //the input bits in as few field elements as possible.
10            const size_t input_size_in_field_element =
11                               div_ceil(input_size_in_bits, FieldT::capacity());
12            input_as_field_elements.allocate(pb, input_size_in_field_element,
13                                                                 "input_as_field_elements");
14            this->pb.set_input_sizes(input_size_in_field_element);
15        }
16
17    // Public inputs:
18    sn_var.reset(new digest_variable<FieldT>(this->pb, sha256_256digest_len, "sn_var"));
19    root_var.reset(new digest_variable<FieldT>(this->pb, sha256_256digest_len, "root_var"));
20
21    input_as_bits.insert(input_as_bits.end(), sn_var->bits.begin(), sn_var->bits.end());
22    input_as_bits.insert(input_as_bits.end(), root_var->bits.begin(), root_var->bits.end());
23
24    // Multipacking
25    assert(input_as_bits.size() == input_size_in_bits);
26    pack_inputs.reset(new multipacking_gadget<FieldT>(this->pb, input_as_bits,
27                                              input_as_field_elements, FieldT::capacity(),
28                                              FMT(this->annotation_prefix, "pack_inputs")));
29
30    // Prover inputs:
31    rho_var.reset(new digest_variable<FieldT>(this->pb, sha256_256digest_len, "rho_var"));
32    r_var.reset(new digest_variable<FieldT>(this->pb, sha256_256digest_len, "r_var"));
33    k_var.reset(new digest_variable<FieldT>(this->pb, sha256_256digest_len, "k_var"));
34    s_var.reset(new digest_variable<FieldT>(this->pb, sha256_256digest_len, "s_var"));
35    cm_var.reset(new digest_variable<FieldT>(this->pb, sha256_256digest_len, "cm_var"));
36    path_var.reset(new merkle_authentication_path_variable<FieldT, HashT>(this->pb,
37                                                                 tree_depth, "path_var"));
38    address_bits_var.allocate(this->pb, tree_depth, "address_bits");
```

```
39     flag.allocate(this->pb, "flag");
40
41     // initialize vote_gadget
42     cc_gadget.reset(new vote_gadget<FieldT, HashT>(this->pb, *sn_var, *rho_var, *r_var,
43                                         *k_var, *s_var, *cm_var, *root_var, *path_var,
44                                         address_bits_var, tree_depth, flag,
45                                         "complete_coin_gadget"));
46 }
```

As shown in Listing 5.6, `coin_wrapper_gadget` only gets `tree_depth` as input and implement other variables such like `sn` and `root` as attribute. In the constructor, it first allocates the storage place for public inputs (line 5 to line 15) and then initialize the pointers that points to the public input variables, in this case, variable `sn` and `root`. Then it packs these two variables into `input_as_bits` so that the program knows which variables are public inputs. Then `coin_wrapper_gadget` initializes the pointers that point to all other witnesses variables and at last, it initializes the `complete_coin_gadget`. Similar to the other 2 gadgets, the `generate_r1cs_constraints` of `coin_wrapper_gadget` uses the `generate_r1cs_constraints` of `complete_coin_gadget` to generate the constrain system. As for `generate_r1cs_witness` function, it takes exact values of all witnesses and public input variables, then assigns them to the corresponding variables in `coin_wrapper_gadget`, after that, it calls the `generate_r1cs_witness` function of `complete_coin_gadget` to assign the values to the variables in `complete_coin_gadget`.

**Listing 5.7** `main` function for $\pi_{Coin}$

```
1  int main(int argc, char* argv[]) {
2      string read_file, write_file;
3    if (argc != 3){
4      cout << "incorrect argument number, Programm will exit without doing anything" << endl;
5      return 0;
6    }else{
7      string default_path = "...";
8      read_file = default_path + argv[1];
9      write_file = default_path + argv[2];
10   }
11
12   default_r1cs_ppzksnark_pp::init_public_params();
13   typedef Fr<default_r1cs_ppzksnark_pp> FieldT;
14   protoboard<FieldT> pb;
15
16   string eid;
17   bit_vector sn, root;
18   bit_vector rho, r, k, s, cm;
19
20   std::vector<merkle_authentication_node> path;
21   bit_vector address_bits;
22   size_t address, tree_depth;
23
24   get_data(eid, sn, root, rho, r, k, s, cm, address,
25                              tree_depth, address_bits, path, read_file);
26
```

```
27    coin_wrapper_gadget<FieldT, sha256_merkle_hash_gadget<FieldT> > coin(pb, tree_depth);
28    coin.generate_r1cs_constraints();
29    const r1cs_constraint_system<FieldT> constraint_system = pb.get_constraint_system();
30
31    auto keypair = r1cs_ppzksnark_generator<default_r1cs_ppzksnark_pp>(constraint_system);
32    auto proof = generate_proof<default_r1cs_ppzksnark_pp, sha256_merkle_hash_gadget<FieldT>>
33    (keypair.pk, pb, vote, sn, root, rho, r, k, s, cm, address, tree_depth, address_bits, path);
34
35    const r1cs_ppzksnark_proof<default_r1cs_ppzksnark_pp> proof_data = *proof;
36    const r1cs_ppzksnark_verification_key<default_r1cs_ppzksnark_pp> vk = keypair.vk;
37    const r1cs_primary_input<FieldT> input_as_field_elements = l_input_map<FieldT>(sn,root);
38
39    generate_proof_file<default_r1cs_ppzksnark_pp, FieldT>(vk, proof_data,
40                        input_as_field_elements, eid,
41                        bit_string_to_hex_string(bit_vector_to_bit_string(sn)), write_file);
42
43    return 0;
44  }
```

As all constrains are implemented and wrapped by other gadgets, it is relatively easy to implement the `main` function. The `main` function is shown in Listing 5.7, it first read `vote_witness.json` file to get the data of `eid` and all values from `coin_witness` and an `auth_path`. Then it initializes the `coin_wrapper_gadget` and generate the constrain system. When the constrain system is generated, it uses the `r1cs_ppzksnark_generator` (line 31) to create the verifying and proving keys for this constrain system. After that it calls `generate_proof` on the proving key and all values read from `vote_witness.json` file to generate the ZK-Proof. Notice that this proof will only be generated, when the constrain system is fulfilled with the given input. At the end, it writes the verifying key, the generated proof and the public input for the proof to the file `coin_proof.json`, so that it can be later uploaded to Blockchain.

Notice that the constrain system for $\pi_{Coin}$ is actually the same for each Voter and each election. As long as VC doesn't change its format, each Voter (of any election) can generate his $\pi_{Coin}$ proof by just giving his witnesses and public inputs to the program. As mentioned at the beginning of this section, since the constrain system remains the same, the proving key and the constrain system can be hard-coded in the program / Smart Contract to increase performance, however, this is not implemented in this demo due to simplicity and testability.

### Proof of Vote Format $\pi_{Vote}$

The ZK-Proof generator of $\pi_{Vote}$ is implemented in `Java` using library `Jsnark`. `Jsnark` is an external circuit building library for `libsnark`. It is used to represent NP-statement as circuit that supported by `libsnark`. Similar to `libsnark`, `Jsnark` also has a class `Gadget`. Any class that extends `Gadget` needs to implement the `buildCircuit` function. Similar to `generate_r1cs_constraints`, this function is used to build the circuit of a gadget.

The wrapper gadget in `libsnark` is implemented as class `CircuitGenerator`. Similar to the `protoboard` in `libsnark`, this class is used to combine the circuit from different gadgets. For any class that extend `CircuitGenerator`, two function needs to be override. The first one is also called `buildCircuit`, in this function not only the circuit will be generated but also the public inputs and witnesses

variables will be identified. The second function is called `generateSampleInput` and it is similar to `generate_r1cs_witness` in `libsnark`. In this function, the values of public inputs and witnesses will be assigned to the corresponding variables.

In addition, `Jsnark` provides a `RSAEncryptionV1_5_Gadget` that implements the RSA PKCS# 1 v1.5 encryption scheme according to [Labb]. However, this implementation assumes a fixed public exponent of value $0x10001$. As we don't want to restrict our encryption key with a fixed encryption key, we first adjust it a little to suit for a more generalized purpose.

**Listing 5.8** adjustment for `RSAEncryptionV1_5_Gadget`

```
1   int l = this.exponent.length() - 1;
2   LongElement h = new LongElement(new BigInteger[] {BigInteger.ONE});
3   LongElement k = paddedMsg;
4   while( l >= 0) {
5     if(this.exponent.charAt(l) == '1') {
6       h = h.mul(k);
7       h = new LongIntegerModGadget(h, modulus, false).getRemainder();
8     }
9     k = k.mul(k);
10    k = new LongIntegerModGadget(k, modulus, false).getRemainder();
11    l--;
12  }
13
14  h = new LongIntegerModGadget(h, modulus, true).getRemainder();
15  // return the cipher text as byte array
16  ciphertext = h.getBits(rsaKeyBitLength).packBitsIntoWords(8);
```

Listing 5.8 shows the adjustment for a RSA encryption gadget with an arbitrary public exponent. This new gadget is called `RSAEncryptionV1_5_Gadget_Free_Exponent`. Unlike a fixed exponent, `this.exponent` is given as a binary string of an arbitrary public exponent. To calculate $x^e \bmod n$, with $x$ represents the padded message according to RSA PKCS# 1 v1.5, $e$ represents the public exponent and $n$ represent the RSA modulus, we implement the fast modular exponentiation algorithm according to [Eli]. The variable `h` and `k` are used to save interim result. The multiplication and modulo calculation are done using the function `mul` and gadget `LongIntegerModGadget` provided by `Jsnark`. Notice that the code segment listed in Listing 5.8 not actually calculate the ciphertext but generate the corresponding circuit for the calculation. When the input variables in this circuit are assigned, the ciphertext will be evaluated through this circuit. The other parts of `RSAEncryptionV1_5_Gadget_Free_Exponent` are exactly the same as `RSAEncryptionV1_5_Gadget` provided by `Jsnark`. As this gadget is provided by `Jsnark`, we refer readers to their documentation [Kos] for more details of this gadget.

**Listing 5.9** Format of vote_witness

```
"vote_witness":{
                  "pt":"0001000",
                  "r":"...",
                  "n":"...",    //public input
                  "e":"..."   //public input
                 }
```

To actually generate a ZK-Proof $\pi_{Vote}$, a class `RSAV1_5_Enc` that extends `CircuitGeneration` should be implemented. An instance of this class is expected to get `vote_ct` and all variables from `vote_witness` as input and generate the corresponding circuit that represents all statement in $\pi_{Vote}$.

Listing 5.9 shows the format of `vote_witness`. It is the third witness set from `vote_witness.json` file and contains two witnesses and two public inputs. The public inputs `n` and `e` are a RSA public key pair. The witness `pt` contains the plaintext of `vote_ct`. It is also the plaintext of a Voter's vote. The content of this variable is a sting representation of an unit vector. An unit vector is a vector that contains only 0s and 1s, and the sum of all element in an unit vector is always equal to 1. The dimension of the unit vector in `pt` (i. e., the length of `pt`) equals to the number of all candidates. The value of each element in `pt` corresponding to the vote for that candidate. For example, the plaintext `0001000` in Listing 5.9 means a Voter has chosen the 4-th candidate. The last witness `r` is the randomness that is used in RSA encryption. As RSA PKCS# 1 v1.5 standard is non-deterministic, to check the correctness of the encryption, one has to provide the randomness that is used during the encryption, otherwise it is impossible to reproduce the same result.

**Listing 5.10** `buildCircuit` from class `RSAV1_5_Enc`

```
1  @Override
2  protected void buildCircuit() {
3    inputMessage = createProverWitnessWireArray(plainTextLength); // in bytes
4    for(int i = 0; i < plainTextLength;i++){
5      inputMessage[i].restrictBitLength(8);
6    }
7
8    //check vote format => only 1 occurs (as ascii code)
9    Wire sum = zeroWire;
10   for(int i =0; i<inputMessage.length; i++) {
11     sum = sum.add(inputMessage[i]);
12   }
13   addAssertion(sum, oneWire, oneWire.mul(48*inputMessage.length).add(1));
14
15   randomness = createProverWitnessWireArray(RSAEncryptionV1_5_Gadget_Free_Exponent
16                         .getExpectedRandomnessLength(rsaKeyLength, plainTextLength));
17
18   rsaModulus = createLongElementInput(rsaKeyLength);
19
20   rsaEncryptionV1_5_Gadget = new RSAEncryptionV1_5_Gadget_Free_Exponent(rsaModulus,
21                           new BigInteger(this.exponent, 16).toString(2),
22                           inputMessage, randomness, rsaKeyLength);
23
24   rsaEncryptionV1_5_Gadget.checkRandomnessCompliance();
25
26   Wire[] cipherTextInBytes = rsaEncryptionV1_5_Gadget.getOutputWires(); // in bytes
27
28   //constrains to check the ciphertext is the expected ciphertext
29   correct_ciphertext_in_bytes = createInputWireArray(cipherTextInBytes.length);
30   for(int i = 0; i<cipherTextInBytes.length; i++ ) {
31     addAssertion(cipherTextInBytes[cipherTextInBytes.length-1-i], oneWire,
32                                               correct_ciphertext_in_bytes[i]);
33   }
```

```
34
35     // do some grouping to reduce VK Size
36     cipherText = new WireArray(cipherTextInBytes).packWordsIntoLargerWords(8, 30);
37     makeOutputArray(cipherText, "Output cipher text");
38   }
```

The `buildCircuit` function from class `RSAV1_5_Enc` builds a circuit to check the format of the vote and the correctness of the encryption. As shown in Listing 5.10, it first initializes `inputMessage` as a witness variable and builds the necessary constrains to check the byte-length. In line 9 to line 13, it checks the format of the plaintext, i. e., whether the Voter has chosen and only chosen one candidate. As the plaintext is given as a string of the unit vector, the value that saved in `inputMessage` are not 0 or 1 but the integer value of `char` '0' or '1' according to `ASCII`. Thus, instead of checking whether the sum of all elements equals to 1, we need to check whether the sum of all elements equal to `inputMessage.length`∗48 + 1, since the `ASCII` code for '0' or '1' are 48 and 49 respectively. After that, `buildCircuit` initializes the variables for randomness and RSA modulus, then it creates an instance of `RSAEncryptionV1_5_Gadget_Free_Exponent` to build the circuit for RSA encryption. The ciphertext evaluated by this circuit is saved in `cipherTextInBytes`. To compare the ciphertext provided by Voter and the evaluated ciphertext, we use the circuit segment built by line 29 to line 33. These lines build equation constrains for each byte between the Voter provided ciphertext and the evaluated ciphertext. If these constrains pass, one can believe that the Voter has encrypted his vote correctly. Together, `buildCircuit` builds a circuit for both statements in $\pi_{Vote}$.

In `generateSampleInput`, the given values will be assigned to the corresponding variables for the circuit that is generated by `buildCircuit`. As the Voter provided ciphertext and `r` are given as hex-string, it first needs to be converted to a byte array using `hexStringToByteArray` as shown in Listing 5.11 line 5 and line 6. The RSA modulus `n` is interpreted as a `BigInteger`, and each char in plaintext string `pt` will be assigned to the variable `inputMessage` as an integer according to its `ASCII` code value. The assignment is done through a `CircuitEvaluator`. It is a class provided by `Jsnark` that records all assignment and evaluates the circuit according to these assignments.

**Listing 5.11** `generateSampleInput` from class `RSAV1_5_Enc`

```
1    @Override
2    public void generateSampleInput(CircuitEvaluator evaluator) {
3
4      BigInteger modulus = new BigInteger(this.modulus, 16);
5      byte[] sampleRandomness = hexStringToByteArray(this.rand);
6      byte[] voter_given_ct = hexStringToByteArray(this.ciphertext);
7
8      for (int i = 0; i < inputMessage.length; i++) {
9        evaluator.setWireValue(inputMessage[i], this.plaintext.charAt(i));
10     }
11
12     evaluator.setWireValue(this.rsaModulus, modulus, LongElement.BITWIDTH_PER_CHUNK);
13
14     for (int i = 0; i < sampleRandomness.length; i++) {
15       evaluator.setWireValue(randomness[i], (sampleRandomness[i]+256)%256);
16     }
17
```

```
18     for (int i = 0; i < soll_ct.length; i++) {
19       evaluator.setWireValue(correct_ciphertext_in_bytes[i], (soll_ct[i]+256)%256);
20     }
21   }
```

As shown in Listing 5.12, the main function of $\pi_{Vote}$ is similar to the main function of $\pi_{Coin}$. It first reads all needed inputs from vote_witness.json file (line 7 to line 28) and uses them to create the instance generator of class RSAV1_5_Enc (line 33 - 34). After that, we only need to call 4 functions of generator in sequence to create the ZK-Proof $\pi_{Vote}$. The first function generateCircuit will call the buildCircuit function of generator to generate the whole circuit for $\pi_{Vote}$, the second function evalCircuit will use generateSampleInput to evaluate the whole circuit and checks whether the circuit is fulfilled or not. If it is fulfilled, the third function prepFiles will be called, which generates two files that contains the coded circuit and all inputs (public inputs and witnesses) in a way that libsnark understands. Otherwise, an exception will be thrown and the program will be stopped. At the end, the program calls the fourth function runlibsnark that runs libsnark on the circuit and all inputs from the newly generated files. Jsnark provides an interface for calling libsnark within the java program, we added the same code as in Listing 5.7 line 35 to line 41 to that interface, so that the generated ZK-Proof, the public input of the proof and the verifying key will be saved to vote_proof.json file for later use.

**Listing 5.12** main function for $\pi_{Vote}$

```
1   public static void main(String[] args) throws Exception {
2
3     String pt="", n="", e="", r="", ct="", eid="";
4     int msgLength = 0;
5     int keyLength = 1024;
6
7     JSONParser parser = new JSONParser();
8     try {
9       Object obj = parser.parse(new FileReader("path/to/vote_witness.json"));
10      JSONObject jsonObject = (JSONObject) obj;
11      eid = (String) jsonObject.get("eid");
12      ct = (String) jsonObject.get("ciphertext");
13      ct = ct.substring(2);
14      JSONObject witness = (JSONObject) jsonObject.get("witness");
15      JSONObject vote_witness = (JSONObject) witness.get("vote_witness");
16
17      pt = (String) vote_witness.get("pt");
18      n = (String) vote_witness.get("n");
19      e = (String) vote_witness.get("e");
20      r = (String) vote_witness.get("r");
21
22    } catch (FileNotFoundException e1) {
23          e1.printStackTrace();
24    } catch (IOException e1) {
25          e1.printStackTrace();
26    } catch (ParseException e1) {
27          e1.printStackTrace();
28    }
```

```
29
30      msgLength = pt.length();
31      keyLength = n.length() * 4;
32
33      TestRSAV1_5_Enc generator = new TestRSAV1_5_Enc(
34                  "Vote_zkp_generator", keyLength, msgLength, pt, n, e, r, ct);
35      generator.generateCircuit();
36      generator.evalCircuit();
37      generator.prepFiles();
38      generator.runLibsnark(eid, ct);
39    }
```

Notice that the code provided in Listing 5.8 enables Voter to validate a RSA encryption using arbitrary public exponent, however, this exponent is implemented as a part of the circuit but not given as a public input. That means, if the public exponent of the RSA encryption key is different in each election, the program has to generate the circuit and the corresponding public parameters (proving key and verifying key) for each different election. In this case, the constrain system and the verifying key will not be able to be hard-coded in the program or Smart Contract.

### 5.2.2 Generator of result computation Proof $\pi_{Result}$

The ZK-Proof generator of $\pi_{Result}$ is also implemented in `Java` using `Jsnark`. As the original `Jsnark` library only provides RSA encryption gadget, we first have to implement a gadget for RSA decryption according to RSA PKCS# 1 v1.5 standard. The decryption gadget `RSADecryptionV1_5_Gadget_Free_Exponent` is very similar to the encryption gadget since the main computation of RSA decryption is also a computation of modular exponentiation. The implementation of decryption gadget uses lots of code from encryption gadget `RSAEncryptionV1_5_Gadget` only with different variable names. For the exponentiation parts, we use the same code as listed in Listing 5.8 since the exponent in the decryption key varies from election to election. Lastly, the output of the generated circuit is not `ciphertext` but the decrypted `plaintext`.

Due to the similarity, the code for RSA decryption gadget will not be listed again. Analogy to the class `RSAV1_5_Enc`, a similar class that extends `CircuitGenerator` should be implemented for $\pi_{Result}$ as well. In the following, I'll focus on explaining the class `Result_zkp_generator`, whose instance is responsible for generating the whole constrain system for $\pi_{Result}$.

An instance of `Result_zkp_generator` is expected to get `RSAmodulus`, `private exponent`, `cipherTexts`, `plainTexts` and `result` as input. The parameter `RSAmodulus` and `private exponent` are similar to `n` and `e` in Section 5.2.1. The parameter `cipherTexts` is a set of ciphertext that contains all encrypted vote from Blockchain. The parameter `plainTexts` is a set of plaintext decrypted from the ciphertexts above by Admin server. Notice that if the decryption is correct, each plaintext should have the same length as the length of the candidate list. The last parameter `result` is an integer array of length equals to the length of the candidate list, the value of each entry of the array represents the final votes the corresponding candidate got in the election.

**Listing 5.13** `buildCircuit` function for `Result_zkp_generator`

```
@Override
protected void buildCircuit() {
  correct_plaintexts_in_bytes = createProverWitnessWireArray(
                                                    plainTextLength * num_plainText);
  for(int i = 0; i < plainTextLength;i++){
    correct_plaintexts_in_bytes[i].restrictBitLength(8);
  }

  cipherTexts = createInputWireArray(this.rsaKeyLength / 8 * num_cipherText);
  for(int i = 0; i < this.rsaKeyLength / 8;i++){
    cipherTexts[i].restrictBitLength(8);
  }

  w_result = createInputWireArray(plainTextLength);
  addAssertion(w_num_plainText, oneWire, w_num_cipherText);
  for(int i = 0; i < plainTextLength; i++ ) {
    Wire sum = zeroWire;
    for(int j = 0; j < num_plainText; j++) {
      sum = sum.add(correct_plaintexts_in_bytes[i+j*plainTextLength]);
    }
    addAssertion(sum, oneWire, oneWire.mul(48*num_plainText).add(w_result[i]));
  }

  rsaModulus = createLongElementInput(rsaKeyLength);

  for(int i = 0; i < num_cipherText; i++) {
    rsaDecryptionV1_5_Gadget = new RSADecryptionV1_5_Gadget_Free_Exponent(
        rsaModulus, new BigInteger(this.exponent, 16).toString(2),
        getWireSegment(this.cipherTexts, i*this.rsaKeyLength/8, (i+1)*this.rsaKeyLength/8),
        rsaKeyLength);

    Wire[] plainTextInBytes = rsaDecryptionV1_5_Gadget.getOutputWires(); // in bytes

    //constrains to check the ciphertext is decrypted to expected plaintext
    for(int j = 0; j<this.plainTextLength; j++ ) {
      addAssertion(plainTextInBytes[j], oneWire,
              correct_plaintexts_in_bytes[this.plainTextLength * (i+1) - 1 - i]);
    }

    addZeroAssertion(plainTextInBytes[this.plainTextLength]);
    addZeroAssertion(plainTextInBytes[plainTextInBytes.length - 1]);
    addAssertion(plainTextInBytes[plainTextInBytes.length - 2], oneWire,
                                                      createConstantWire(2));
  }
}
```

Listing 5.13 shows the `buildCircuit` function of `Result_zkp_generator`. Similar to the `buildCircuit` in Listing 5.10, line 3 to line 12 restricts input `plainTexts` / `cipherTexts` as byte array. Notice that the `plainTexts` is implemented as a prover witness and `cipherTexts` is considered to be a public input. Then line 15 checks that the number of all given ciphertexts equals to the number of all given plaintexts. This constrain donates the first statement of $\pi_{Result}$, i. e., all recorded

votes are decrypted. After that line 16 to line 22 check the correctness of the result computation by recomputing the result from the given `plainTexts` and comparing it with the given `result`. Notice that the `plainTexts` are still given as `String`, thus, the sum of all `plainTexts` on a certain position doesn't equal to the integer value from the `result` on the same position. Hence, we need a conversion between these two values similar to line 13 in Listing 5.10. At the end, for each ciphertext a `RSADecryptionV1_5_Gadget_Free_Exponent` will be used to generate circuit for a correct decryption. Line 32 to line 38 shows that, for each ciphertext in `cipherTexts`, we get the plaintext from `RSADecryptionV1_5_Gadget_Free_Exponent` gadget and compare it with the corresponding plaintext from `plainTexts`. If it is the same, the decryption is then correct. The last three lines in the outer for-loop (line 40 to line 42 ) is used to check the padding of decrypted plaintext.

The `generateSampleInput` of `Result_zkp_generator` is also similar to the `generateSampleInput` of `RSAV1_5_Enc`. In this function, the values that are used to initialize the instance of `Result_zkp_generator` will be assigned to corresponding variables for circuit evaluation. Analogously, in the `main` function of $\pi_{Result}$ ZK-Proof generator, the expected input will be read from a file and be used to initialize the instance of `Result_zkp_generator`, after that, four functions `generateCircuit`, `evalCircuit`, `prepFiles` and `runLibsnark` will be called in sequence to generate the $\pi_{Result}$ ZK-Proof with given inputs.

In the testing phase, we noticed that to generate a $\pi_{Result}$ which involves only one decryption with RSA key length 1024 bits already takes nearly half an hour. This is because the private exponent is usually much bigger than the public exponent. As we use RSA with key length 2048 bits in Admin server and it takes hours to generate a $\pi_{Result}$ that involves only one decryption with 2048 bits key length, we decided not to integrate this part to the demo. That means, in this demo, after Admin server decrypts all votes and computes the result, it won't generate a ZK-Proof and the result will be simply saved in the shared database but not on the Blockchain.

A big drawback of these 3 ZK-Proof generators is that, the public input in the `xxxx_proof.json` file, which is produced by `libsnark`, represents the public input as field elements that `libsnark` supports. This leads to the problem that to check whether the public input is the desired input as described in Section 4.3.2, one needs to convert the original public input values to the corresponding field elements or vice versa to do the comparison. This conversion is however not implemented in this demo, as it requires too much effort to do this conversion outside `libsnark`. This is a huge security vulnerability for this demo, since a Voter could then use a single valid proof for multiple elections or an attack can even provide valid proofs for false statements. Thus, this demo is not suitable for practice use and to use PEES in practice, one must implement this conversion in Smart Contract so that the public input variables of a ZK-Proof can be compared with the values that have been saved on the Blockchain before the verification begins.

## 5.3 Smart Contract

Smart Contracts in PEES are written in solidity as they will be deployed on Ethereum by Admin server during an election. The following implementation servers as the template for Registration Smart Contract and Voting Smart Contract. The hard-coded values such as election Id `eid` will be set during the election. In the following, I will focus on the main functions in both Smart Contracts, functions like standard getter are omitted in this section.

### 5.3.1 Registration Smart Contract

Listing 5.14 shows the variables in Registration Smart Contract. `Client_server_addr` and `Admin_server_addr` are two variables that save the Ethereum account of the Client server and Admin server respectively. In solidity, it is saved as `address` type. As mentioned in Section 4.3.3, `eid` represents the election Id and it will be set to the corresponding value during the election by Admin server. The type `bytes32` in solidity is an alternative to type `string` as it uses less gas.

**Listing 5.14** Registration Smart Contract: Variables

```
1   address Client_server_addr;
2   address Admin_server_addr;
3   bytes32 eid;
4
5   bytes32[] leafs;
6   bytes32[] used_tokens;
7   bytes32 public root_hash;
8
9   bool reg_is_end = false;
10  uint reg_end_time;
```

The array `leafs` and `used_tokens` are two arrays of `bytes32` that save the valid coin commitment of VCs and checked token respectively. `root_hash` saves the root hash value of the MT over all saved coin commitments, i. e., over all element saved in `leafs`. It is set to be public, which means everyone has the access of the value of this variable.

The last two variables are used to control the time period of registration phase. The variable `reg_end_time` saves the end time of the registration phase in an integer as the number of seconds since 1970-01-01. The other variable `reg_is_end` is a boolean that enables EA to end the registration phase earlier than the previously set end time. This variable can be only set to `true` by the function as shown in Listing 5.15. Line 2 in Listing 5.15 guarantees that function `endReg` can only be called from EA's Ethereum account. As soon as this variable is set to `true`, the Registration Smart Contract will no longer save any incoming VCs.

**Listing 5.15** Registration Smart Contract: function `endReg`

```
1   function endReg() public {
2       require(msg.sender == EA_server_addr);
3       reg_is_end = true;
4   }
```

The main functions in Registration Smart Contract are `Mint` and `GenerateMT` as shown in Listing 5.16 and Listing 5.17. Listing 5.16 takes the election Id (`eid`), variables that are related to the coin commitment (`cm, k, s`), the token issued from Client server (`tokenHash`) and the signature on the token (`sign_v, sign_r, sign_s`) as input and verifies the validity of the input as described in Section 4.3.3. It first recalculates the coin commitment using received `k` and `s` (line 4). Then, it checks the validity of the signature using function `checkSignature`. In order to check the signature in Smart Contract, we use the signature scheme provided by Ethereum to sign on the token and use the corresponding verify function to verify the signature. The token is signed by Client server

using its Ethereum account, later when verifying the signature, the verify function will return an Ethereum account. If the returning account is the same as the Ethereum account that is used to sign the token, the signature is valid. The calling of the verifying function and the comparison of the returning account and the previous hard-coded `Client_server_addr` are all done within function `checkSignature`, it only returns a boolean to indicate the validity of the signature. After that, function `Mint` checks whether the received `tokenHash` is already used or not using function `checkToken`. On receiving `tokenHash`, `checkToken` checks whether `tokenHash` is a member of `used_tokens` or not and if it is not, `checkToken` will save `tokenHash` to `used_tokens` and return `true`, otherwise, it will return `false`.

**Listing 5.16** Registration Smart Contract: function `Mint`

```
1    function Mint(bytes32 election_id, bytes32 cm, bytes32 k, bytes32 s,
2             bytes32 tokenHash, uint8 sign_v, bytes32 sign_r, bytes32 sign_s) public{
3
4        bytes32 cm_check = sha256(abi.encodePacked(k, s));
5        bool verifiedSign = checkSignature(tokenHash, sign_v, sign_r, sign_s);
6        bool tokenIsNotUsed = checkToken(tokenHash);
7
8        if( !reg_is_end && now < reg_end_time){
9            if(eid == election_id){
10               if(verifiedSign){
11                   if(tokenIsNotUsed){
12                       if(cm == cm_check){
13                           leafs.push(cm);
14                           emit leafCreated(true, "Save successful");
15                       }else{
16                           emit leafCreated(false, "Save failed : invalid Commitment");
17                       }
18                   }else{
19                       emit leafCreated(false, "Save failed : invalid Token");
20                   }
21               }else{
22                   emit leafCreated(false, "Save failed : invalid Signature");
23               }
24           }else{
25               emit leafCreated(false, "Save failed : Wrong election");
26           }
27       }else{
28           emit leafCreated(false, "Save failed : Registration time ended");
29       }
30
31   }
```

When all checks are done, `Mint` will emit the event `leafCreated` to the transaction sender based on the check result. If all checks pass, the received coin commitment `cm` will be saved in array `leafs` and the sender will be informed that the VC has been successfully minted. If one of the checks is invalid, the coin commitment will not be saved and the sender will be informed of the corresponding error message.

**Listing 5.17** Registration Smart Contract: function `GenerateMT`

```
function GanarateMT(bytes32 election_id) public {
    require(msg.sender == EA_server_addr);
    bytes32 parent;
    bytes32 padding = 0x0000000000000000000000000000000000000000000000000000000000000000;

    if(eid == election_id){
        if (leafs.length > 1){
            bytes32[] memory temp_nodes;
            bytes32[] memory parents;
            temp_nodes = leafs;
            uint end;
            uint parent_length;

            do{
                if(temp_nodes.length % 2 == 0){
                    end = temp_nodes.length;
                    parent_length = end / 2;
                }else{
                    end = temp_nodes.length - 1;
                    parent_length = (end / 2) + 1;
                }
                parents = new bytes32[](parent_length);
                uint j = 0;

                for(uint i = 0; i < end; i=i+2){
                    parent = sha256(abi.encodePacked(temp_nodes[i], temp_nodes[i+1]));
                    parents[j] = parent;
                    j++;
                }
                if(end < temp_nodes.length){ //there is one nodes left
                    parent = sha256(abi.encodePacked(temp_nodes[end], padding));
                    parents[j] = parent;
                }
                temp_nodes = parents;

            }while (parents.length > 1);

            root_hash = parents[0];
            emit treeGenerated(true, "Generation successful");
        }else if (leafs.length == 1){ //if there is only one leaf
            parent = sha256(abi.encodePacked(leafs[0], padding));
            root_hash = parent;
            emit treeGenerated(true, "Generation successful");
        }else{ //leafs.length == 0, i.e. no Leaf
            emit treeGenerated(false, "Generation failed : there is no registered voter");
        }
    }else{
        emit treeGenerated(false, "Generation failed : Wrong election");
    }

}
```

Similar to function `endReg`, line 2 in `GenerateMT` guarantees that this function can only be called from EA's Ethereum account. Upon receiving the transaction from EA with the same `eid` as hard-coded in the Smart Contract, `GenerateMT` begins to generate the MT over all commitments that saved in array `leafs` till the time it is called. To generate the MT, we differ from 3 situations:

`leafs` has more than 2 members
: In this case, `GenerateMT` generates the MT for all coin commitments in `leafs` from bottom to top with the same sequence as the coin commitment is saved. Each new node (`parent`) has two children nodes from the previous layer and it contains the hash value of its children. If the number of all nodes in previous layer is odd, there will be a node in current layer that only has one child. In this case, the hash value of this node is the hash value of its child concatenate with `padding`. The variables `temp_nodes` and `parents` are two temporary arrays that save the interim results. The key word `memory` explicitly indicates that these two variables exist only in function `GenerateMT`, once the function call is ended, the storage of these variables are freed.

  After the MT is generated, `GenerateMT` saves the hash value of root node in `root_hash` and emits the event `treeGenerated` to the transaction sender to indicate that MT has been successfully generated.

`leafs` has only 1 member
: This case handles the situation that only one Voter has registered to the election during the registration phase. In this case, `GenerateMT` directly calculates the `root_hash` as the hash value of the single member from `leafs` concatenated with the padding and then emit the sender that MT has been successfully generated.

`leafs` is empty
: This case handles another special situation where on one has registered to the election during the registration phase. In this case, `GenerateMT` does nothing and emits the sender that MT generation has failed due to 0 registered Voter.

Notice that the generated MT is not saved in Registration Smart Contract as the writing operation on Ethereum is too expensive. The only value that is saved in this Smart Contract is the root hash of the MT. In order to generate the ZK-Proof, one needs to download all coin commitments from `leafs` and rebuild the MT locally. To check whether the same MT is generated, one needs to compare the root hash he calculated with the root hash saved in `root_hash`.

`GenerateMT` should be called after the registration phase is ended. If the MT of all saved coin commitment are generated, but this Smart Contract kept receiving and saving new incoming VCs, no one can generate valid ZK-Proofs for these new VCs, as they are not contained in the MT.

### 5.3.2 Voting Smart Contract

All ZK-Proofs are verified in Voting Smart Contract. To verify a zk-SNARK proof, we use the pairing library from [Chr], which provides the necessary functions to verify the proof.

**Listing 5.18** Voting Smart Contract: Variables

```
1    address Client_server_addr;
2    address Admin_server_addr;
3    bytes32 eid;
4
5    bytes[] votes;
6    bytes32[] used_coins;
7    bool vote_is_end = false;
8    uint vote_end_time;
9
10   mapping (address => bool) vote_format;
11   VerifyingKey vk;
12   bool public verifyingKeySet = false;
```

Listing 5.18 shows all variables that are used in Voting Smart Contract. Similar to Registration Smart Contract, Voting Smart Contract has variables `Client_server_addr`, `Admin_server_addr` and `eid` that will be hard-coded by Admin server during an election. The two arrays `votes` and `used_coins` are used to save the encrypted vote and the serial number of the VCs respectively. The variables `vote_is_end` and `vote_end_time` are also similar to the variables `reg_is_end` and `reg_end_time` from Registration Smart Contract. For `vote_is_end`, there is a function `endVote` similar to `endReg` that can only be called from EA's Ethereum account and set `vote_is_end` to `true`.

As mentioned in Section 5.2, the ZK-Proof $\pi_{POUR}$ is split into two proofs. As solidity doesn't support giving too many parameters (limit is around 16 parameters) to a function, these two proofs must be verified separately. Hence, to connect the result of these two proof verifications, we need a variable `vote_format` to save the verification result of the vote-format proof for the transaction sender, so that when the same sender verifies his coin proof, the Smart Contract knows his verification result for vote-format proof. The type `mapping` is like the `HashMap` in Java, it takes an account address as the key and the verification result of the vote-format proof as the value.

The variable `vk` of type `VerifyingKey` saves the corresponding verifying key of the proof temporarily. The variable `verifyingKeySet` indicates whether a verifying key is correctly set or not. It is set to be `public` so that everyone can check before verifying the proof. In this demo, a Voter is asked to first set the verification key, then upload the proof to verify. As vote-format proof and coin proof have fixed constrain systems, the verifying keys can be hard-coded in the Smart Contract to reduce transactions needed to cast a vote. However, as the result computation proof $\pi_{Result}$ doesn't have a fixed constrain system, to verify such a proof, EA must first set the verifying key. Another constrain in this demo is that, we only consider the case that at the same time only one Voter tries to verify his ZK-Proofs. If it is not the case, the verifying could went wrong as it may happen that Voter 1 has set his verifying key and then Voter 2 uploaded his proof. To avoid this conflict, it is recommended to hard code the verifying keys for vote-format proof and coin proof in the Smart Contract.

**Listing 5.19** Voting Smart Contract: function `checkVote`

```
1    function checkVote(uint[2] a, uint[2] a_p, uint[2][2] b, uint[2] b_p, uint[2] c,
2                              uint[2] c_p, uint[2] h, uint[2] k, uint[] input) public{
3        bool vote_well_formed = verifyProof(a, a_p, b, b_p, c, c_p, h, k, input);
4        if (vote_well_formed){
```

```
5            vote_format[msg.sender] = true;
6        }else{
7            vote_format[msg.sender] = false;
8        }
9    }
```

We use the same `setVerifyingKey` and `verifyTx` (renamed to `verifyProof` in PEES) functions from [CS] to set a verifying key and verify the proof. Listing 5.19 shows the function `checkVote` that checks the vote format proof. This function assumes that the corresponding verifying key is already set. The parameters `a, a_p, b, b_p, c, c_p, h` and `k` are concrete component of a zk-SNARK proof, the parameter `input` is the public input for the proof. Upon receiving the proof, `checkVote` checks the validity of the proof and then saves the result in mapping `vote_format` for later use.

After the vote format proof is checked, a Voter then calls the `Pour` function (as shown in Listing 5.20) to verify his coin proof. `Pour` function also assumes that the corresponding verifying key is already set.

**Listing 5.20** Voting Smart Contract: function `Pour`

```
1    function Pour(bytes32 election_id, bytes32 sn, bytes enc_vote, uint[2] coin_a,
2              uint[2] coin_a_p, uint[2][2] coin_b, uint[2] coin_b_p, uint[2] coin_c,
3              uint[2] coin_c_p, uint[2] coin_h, uint[2] coin_k, uint[] coin_input) public{
4        bool coinIsNotUsed = checkCoin(sn, coin_a, coin_a_p, coin_b, coin_b_p,
5                                       coin_c, coin_c_p, coin_h, coin_k, coin_input);
6        bool voteIsWellFormed = vote_format[msg.sender];
7
8        if(!vote_is_end && now < vote_end_time){
9            if(eid == election_id){
10               if(coinIsNotUsed){
11                   if(voteIsWellFormed){
12                       votes.push(enc_vote);
13                       emit voted(true, "Vote successful");
14                   }else{
15                       emit voted(false, "Vote failed : invalid Vote (Coin is consumed)");
16                   }
17               }else{
18                   emit voted(false, "Vote failed : invalid Vote coin");
19               }
20           }else{
21               emit voted(false, "Vote failed : Wrong election");
22           }
23       }else{
24           emit voted(false, "Vote failed : Voting time ended");
25       }
26
27   }
```

The `Pour` function takes `eid`, VC's serial number `sn`, encrypted vote `enc_vote` and the coin proof, which is similar to the vote format proof, as input. It then checks the coin proof using `checkCoin`. This function is similar to `checkVote`, it checks the coin proof and then saves the serial number to

used_coins and return true if the proof is valid, otherwise, it sn won't be saved and false will be returned. After Pour gets the validity of coin proof, it retrieves the validity of the vote format proof of the same Voter from vote_format. Then based on the validity of both proofs, Pour emit the transaction sender (i. e., the Voter) the corresponding message for vote casting result.

Notice that the enc_vote will only be saved when both proofs are valid, and it is cast during the voting phase. If any proof fails, the enc_vote won't be saved and if a Voter provides a valid VC but casts an invalid vote (i. e., coin proof is valid but vote format proof is invalid), the enc_vote will not be saved but the serial number of the VC will be saved to used_coins as the punishment for the dishonest Voter.

Due to technical problems (solidity not allow putting both proofs and verifying keys as the input parameters for one function) and simplicity, the Pour function described in Section 4.3.3 are spilt into three functions in this implementation. The correct sequence of the function calls is: call setVerifyingKey to set verifying key for vote format proof, call checkVote to verify vote format proof, call setVerifyingKey to set verifying key for coin proof and finally call Pour to verify coin proof and cast vote.

In this demo, the saveResult function mentioned in Section 4.3.3 is not implemented, as the generation of ZK-Proof $\pi_{Result}$ takes too much time and the verifying key is also too big. The ZK-Proof $\pi_{Result}$ is only verified locally in this demo. The implementation of saveResult should be similar to function checkVote, where the ZK-Proof should be verified and not only the validity of $\pi_{Result}$ is saved but the result of the election is also saved in an array. The implementation of saveResult may come up in the later updates of this demo.

## 5.4 Web Servers

PEES provides two web servers: Admin server and Client server. These two servers are configured by PEES, which means that EA and Voters can only interact with the server but not be able to change how these servers behave. The code of these two servers should be public so that everyone can check the behavior of the servers. In the following, I'll first introduce the implementation of Admin server in Section 5.4.1 and then describe the implementation of Client server in Section 5.4.2.

### 5.4.1 Admin Server

Admin server in PEES is a web server that EA uses to configure and control the election. It is connected to the shared database. In this implementation, we use MangoDB [DED] as the shared database and we assume that the configuration of the election is already saved in this database.

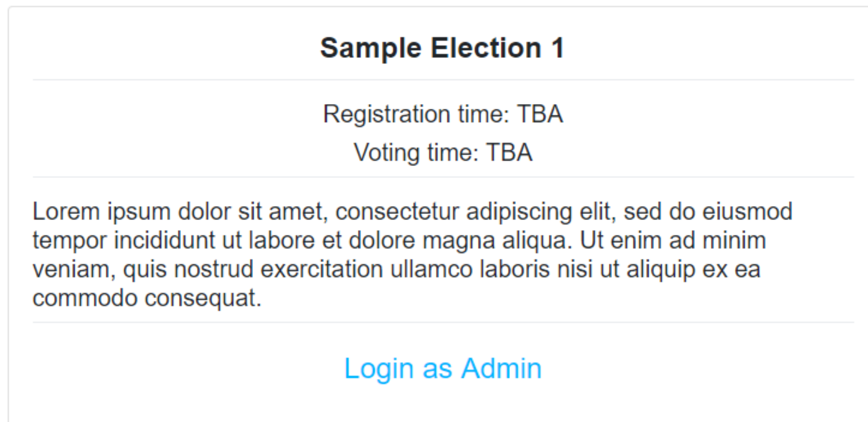



**Figure 5.1:** Login button

To make sure that only members of EA have access to `Admin Interface` of an election, there is a `Login as Admin` button for each election (Figure 5.1). After a user has logged in to an election as Admin, he can see all possible actions in this `Admin Interface` (Figure 5.2). Each option represents a phase from Section 4.4, where `Set up` stands for phase 1, `Result` shows the result of the selected election and `Log out` is used to let an Admin sign out. In the following, I'll explain the implementation of the Admin server according to different phases.

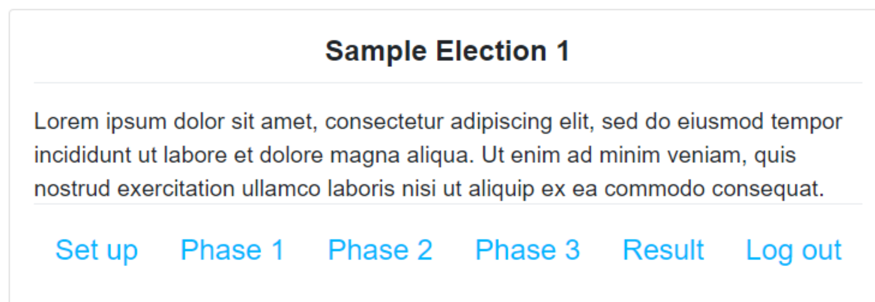



**Figure 5.2:** Admin Interface

#### Phase 0: Set up

After an Admin clicked `Set up` button, the server first checks whether the selected election has already been set up or not. If it is not, the Admin is then asked to give a number of shares that the decryption key will be split to and the threshold for recombination in a html-form. After Admin Server received share number and threshold, it executes the following codes.

**Listing 5.21** Admin Server: Set up

```
1  let result = await db.collection('Elections').find({id :id}).toArray();
2  if(result.length === 0){ //if no Election with given id is found
3    let errmessage = encodeURIComponent('Election not found');
```

```
4     return res.redirect('/admin/?err=' + errmessage);
5   }else{
6     let election = result[0];
7     let key = new rsa.Key();
8     key.generate(1024, "10001"); //set public key exponent to a fixed exponent of "0x10001"
9
10    let encryptionKey = {
11      n : key.n.toString(16),
12      e : key.e.toString(16)
13    };
14
15    let decryptionKey = {
16      n: key.n.toString(16),
17      e: key.e.toString(16),
18      d: key.d.toString(16),
19      p: key.p.toString(16),
20      q: key.q.toString(16),
21      dmp1: key.dmp1.toString(16),
22      dmq1: key.dmq1.toString(16),
23      coeff : key.coeff.toString(16)
24    };
25
26    let toSplit = secrets.str2hex(JSON.stringify(decryptionKey));
27
28    //Shamir's threshold secret sharing scheme -- split decryption key
29    let shares = secrets.share(toSplit, Number(num_shares), Number(threshold));
30
31    await db.collection('Elections').updateOne({id :id},
32                  {$set: {encryptionKey: encryptionKey, threshold : threshold, state : 0}});
33
34    //send share to admins through email
35    let admins = election.admins;
36    let contentstr = 'Admin <b><%= generator%></b> just authorized key generation for Election
37                      <b><%= electionname%></b>' +
38                      '<br>id: <%= id%>' +
39                      '<br><br>' +
40                      '    your share for the decryption key is:\n' +
41                      '<br><br>' +
42                      '    <%= shares%>' +
43                      '<br><br>' +
44                      'please save it <b>carefully</b>,
45                      it will be later needed to regenerate the key.'
46
47    let titlestr = 'Encryption Key for <%= electionname%> is generated';
48
49    let adminsHasShare =[];
50    for(let i=0; i<num_shares; i++){
51      let title = ejs.render(titlestr, {electionname: election.title});
52      let content = ejs.render(contentstr,{generator: sess.user,
53                        electionname:election.title, id:election.id, shares:shares[i]});
54      sendEmail(admins[i].email,title,content);
55      adminsHasShare.push(admins[i].email);
56    }
```

```
57
58      db.collection('Elections').update({id:id}, {$set: {admins_with_share : adminsHasShare}});
59
60      return res.render('setup/generated', {electionname:election.title, id:election.id,
61                  numshares : num_shares, admin: sess.user, already:false, loggedin : true});
```

First of all, it gets the selected election from shared database (Line 1 to Line 6), then it generates a RSA encryption / decryption key pair for the election (Line 7 to Line 24). As we have to save the RSA modulus and public exponents for ZK-Proof generation, we use the library from [Lau] to generate the RSA keypair as this library provides a simple interface to get all details of encryption / decryption keys. After that Admin server splits the decryption key into desired number of shares and sets the threshold as the given threshold according to Shamir's threshold secret sharing scheme [Sha79] using library [Gab]. Then the shares will be sent to corresponding Admins. In this implementation, the shares will be sent to first *n* (*n* represents the number of shares) Admins from the Admin list of the configuration for this election, this can be implemented using a given order as well. At the end, the Admin server saves the encryption key and a list of Admins, who have a share, to the shared database and render the success page for the Admin who issued this set up phase.

#### Phase 1: Initializing Registration

Similar to `Set up`, after an Admin clicked `Phase 1` button, the server first checks whether phase 1 of the selected election has been already initialized or not. If it is not, the Admin is asked to give the begin data and the end data for the registration phase. The given data is then sent to Admin server and it will be hard-coded to the template of Registration Smart Contract, which is defined in Section 5.3.1, using the code from line 1 to line 4 in Listing 5.22. The election Id `eid` will also be hard-coded in Registration Smart Contract (line 5 to line 6), after that the Registration Smart Contract will be compiled using `solc`, a compiler for solidity, provided by Ethereum.

**Listing 5.22** Admin Server: phase 1

```
1   let rsc_source = fs.readFileSync("path/to/Registration.sol", 'UTF-8');
2
3   let sc_reg_end_time_str = 'uint reg_end_time = ' + Date.parse(end)/1000 + ';';
4   rsc_source = rsc_source.replace('uint reg_end_time;', sc_reg_end_time_str);
5   let eid_str = 'bytes32 eid = 0x' + id + ';';
6   rsc_source = rsc_source.replace('bytes32 eid;', eid_str);
7
8   console.log('++ Compiling Registration Smart Contract...');
9   let rsc_compiled = solc.compile(rsc_source, 1);
10  console.log('++ Done');
11
12  let contractName = ':Registration';
13  let rsc_bytecode = rsc_compiled.contracts[contractName].bytecode;
14  let rsc_abi = JSON.parse(rsc_compiled.contracts[contractName].interface);
15  let accounts = await web3.eth.getAccounts();
16  let addr = accounts[8];
17  let contract = new web3.eth.Contract(rsc_abi, null, {data: '0x' + rsc_bytecode});
18  let instance = await contract.deploy().send({from: addr, gas: 1000000});
```

```
19    let rsc_addr = instance.options.address;
20    console.log("++ Contract mined at " + rsc_addr);
21
22    let rsc = {
23                   abi : rsc_abi,
24                   addr : rsc_addr
25                 };
26
27    await db.collection('Elections').updateOne({id :id}, {$set: {reg_begin : beginstr,
28                                                            reg_end : endstr, rsc : rsc, state : 1}});
```

After the compilation, Admin server has to extract the ABI of the Registration Smart Contract and use EA's Ethereum account to deploy it to Ethereum Blockchain (line 12 to line 18). If Registration Smart Contract has been successfully deployed to the Blockchain, an instance of that Smart Contract, which contains the contract address, will be returned. The contract address and the contract ABI are important for those who want to call functions from this Smart Contract. Hence, these two values will be saved to the shared database so that Client server can retrieve them and pass them to Voters. At the end, a success page will be returned to the Admin.

#### Phase 2: Initializing Voting

Admin Server handles this phase very similar to phase 1. It first checks whether voting phase has been already initialized or not and if not, it will hard-code the `eid` and the given start / end date of voting phase in the Voting Smart Contract, then compiles and deploys the Smart Contract on the Blockchain, at last saves the needed data to the shared database as described above. The only difference between these two phases is that, before an Admin submits the start / end date of voting phase to the Admin server, he first needs to generate the MT over all received VCs and end the registration phase (if it is not ended) through Registration Smart Contract.

**Listing 5.23** Admin Server: phase 2 - generate MT

```
1    await contract.endReg();
2    contract.GanarateMT(eid);
```

To end the registration phase and generate the MT, one just needs to call the `endReg` and `GanarateMT` functions in sequence from the Registration Smart Contract as shown in Listing 5.23. Notice that `contract` in Listing 5.23 refers to the Registration Smart Contract of the selected election. The `endReg` is a synchronous function call, which means the code after this function call will first be executed when this function call is ended. However, the `GanarateMT` function call is asynchronous, to know the result of the MT generation, one has to wait until the event `treeGenerated` that is defined in Registration Smart Contract is triggered.

**Listing 5.24** Admin Server: phase 2 - EventListener for `treeGenerated`

```
1    let treeGenerated = contract.treeGenerated();
2    treeGenerated.watch(function(error, result){
```

```
3     if (!error){
4       if(result.args.successful){
5         console.log(result.args.info);
6         document.getElementById('vote-date').submit();
7       }else{
8         console.log(result.args.info);
9         errinfo.innerHTML = result.args.info;
10        errinfo.style.display = "block";
11        errinfodiv.style.display = "block";
12      }
13    } else {
14      console.log(error);
15    }
16  });
```

Listing 5.24 shows the event listener for event `treeGenerated`. The `result` contains the desired values that are sent back by `treeGenerated`. If the MT has been successfully generated, the form that contains the voting phase data will be submitted to the Admin server, otherwise an error information will be displayed to the Admin and the Admin has to resubmit the form.

#### Phase 3: Compute Result

Again, after an Admin clicked `Phase 3` button, the server first checks whether phase 3 of the selected election has been already done or not. If it is not, Admin server then checks whether the requested Admin is in the admin list, which records all Admins that have a share of the decryption key. If it is the case, the requested Admin is then asked to upload his share, otherwise the requested Admin has the ability to inform other Admins that he wants to start phase 3.

**Listing 5.25** Admin Server: phase 3 - Combine Decryption Key

```
1   if (shares.length >= threshold){ //if the server has enough shares
2                                    //to regenerate the decryption key
3
4     //Shamir's threshold secret sharing scheme -- combine decryption key
5     let combinedDecryptionKey = JSON.parse(secrets.hex2str(secrets.combine(shares)));
6
7     await db.collection('Elections').updateOne({id :id}, {$set: {shares:shares,
8           admin_uploaded: uploadedAdmins, decryptionKey : combinedDecryptionKey, state : 3}});
9     return res.render('phase3/combkey/combined', {electionname:election.title,
10                                          id:election.id, already:false, loggedin : true});
11  }
```

After Admin server collects enough shares from Admins, it uses line 5 in Listing 5.25 to combine the decryption and saves this key to the shared database. When the decryption key has been combined, there will be a new button `Authorize Result Calculation` (as shown in Figure 5.3) available for Admins. The Admin can choose to compute the result immediately or later.

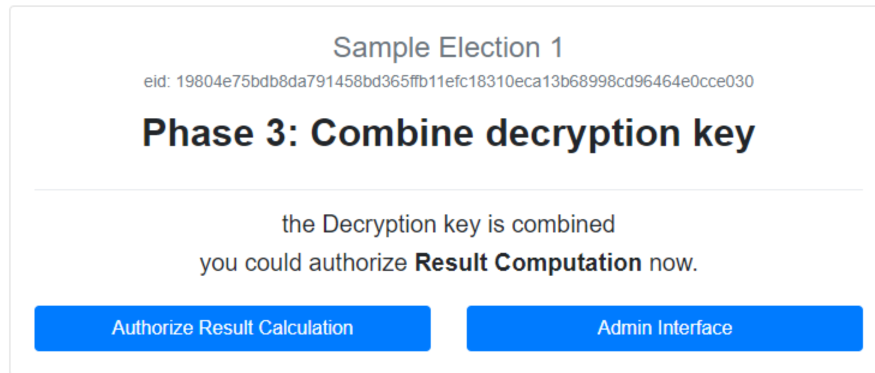



**Figure 5.3:** When decryption key has been combined

When an Admin authorizes Admin server to compute the result, it first ends voting phase (if it is not ended) in the same way as it ends registration phase in phase 2. After that, Admin server uses the getter for array `votes` from Voting Smart Contract to get all cast ballots from Blochchain (line 1 in Listing 5.26) and uses the combined decryption key to decrypt them (line 15 to line 17 in Listing 5.26). Then Admin server calculates the result of the election by adding every decrypted vote and saves the result in shared database.

**Listing 5.26** Admin Server: phase 3 - Compute Result

```
let votes = await getVotes(contract);
let votes_decrypted;
let combinedDecryptionKey = election.decryptionKey;

let privKey = new rsa.Key();
privKey.n = new rsa.BigInteger(combinedDecryptionKey.n,16);
privKey.e = new rsa.BigInteger(combinedDecryptionKey.e,16);
privKey.d = new rsa.BigInteger(combinedDecryptionKey.d,16);
privKey.p = new rsa.BigInteger(combinedDecryptionKey.p,16);
privKey.q = new rsa.BigInteger(combinedDecryptionKey.q,16);
privKey.dmp1 = new rsa.BigInteger(combinedDecryptionKey.dmp1,16);
privKey.dmq1 = new rsa.BigInteger(combinedDecryptionKey.dmq1,16);
privKey.coeff = new rsa.BigInteger(combinedDecryptionKey.coeff,16);

if(votes !== null){
  votes_decrypted = decrypt(votes, privKey);
}

if(votes_decrypted !== null && votes_decrypted.length >0){
  for(let i = 0; i<votes_decrypted.length; i++){
    if(votes_decrypted[i] < vote_result.length){
      vote_result[votes_decrypted[i]] ++;
    }
  }
}

await db.collection('Elections').updateOne({id :id},
```

```
28                          {$set: {result:vote_result, state : 4}});
```

As mentioned in Section 5.2.2, the ZK-Proof generator for $\pi_{Result}$ is not integrated in this demo since it takes too long time to generate a proof. Thus, the result is only saved in the shared database for everyone to check but not on the Blockchain as described in the protocol. This is the drawback of this implementation. Should PEES protocol be used in practice, the ZK-Proof generator provided in Section 5.2.2 must be integrated.

#### Result

As the computed result is not in a human readable manner, the Admin server provides a template to display the result in an understandable manner. A sample result of an election is shown in Figure 5.4.

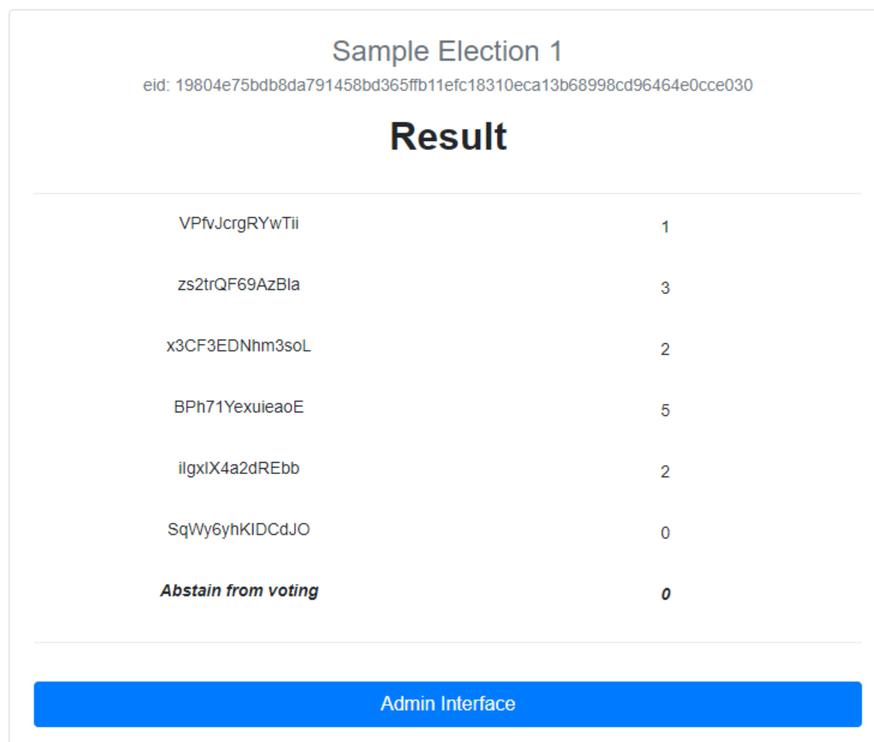



**Figure 5.4:** Result of an election

### 5.4.2 Client Server

Client server is used to interact with Voters. As mint VCs and generate ZK-Proofs are done on Voters browser and computers, the main functionality of Client server is to provide needed interface and JavaScript code for Voters so that every Voter is using the same code in an election. Similar to Admin server, I will introduce the implementation of the Client server according to Voters view of different phases in an election.

#### Phase 0: Set up

When an election is still waiting for EA to set up or to initialize the registration phase, Client server will return a page as shown in Figure 5.5 to interested Voters. Voters can see the description or introduction of the selected election, but as the election is not yet opened, they can't sign up to the election or vote.

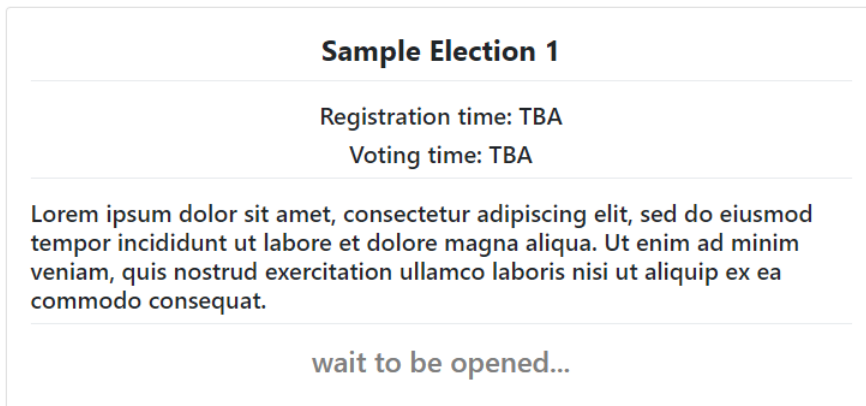



**Figure 5.5:** An election waiting to be opened

#### Phase 1: Registration Phase

After registration phase of an election is initialized, the view of Voters to that election will change to Figure 5.6. The registration time period is given under the election title and there will be a `Sign up` button at the bottom for Voters to sign up for this election. After a Voter clicked this button, he is asked to authenticate to Client server. In this implementation, the Voter is asked to give his email to Client server and the Client server is set to accept everyone who wants to vote, as long as he hasn't voted before.

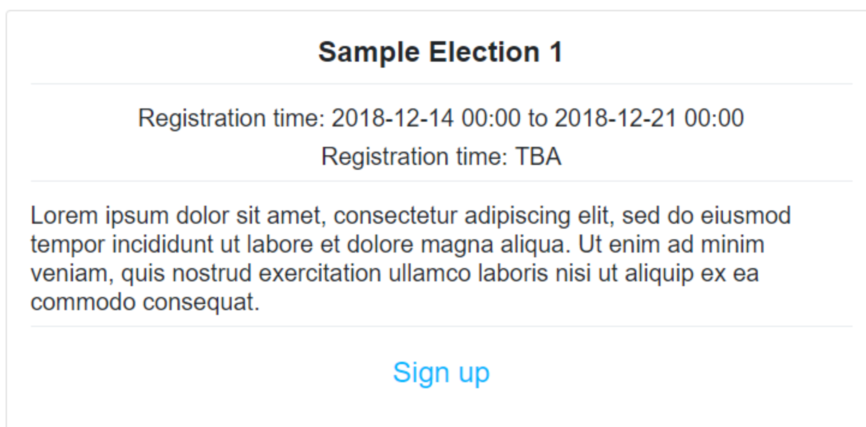



**Figure 5.6:** After registration phase of an election is initialized

Hence, after the Client server received the email from a Voter, it first checks whether this email has already been used to sign up for the selected election or not (line 1 to line 2 in Listing 5.27). If it is not, it generates a signature on the hash of the email address using `sign` function provided by `web3`. Then the Client server group the hash of the email address and the signature together as the `token`, and send it back to the Voter. In the meantime, the Client server saves the email and the corresponding token to the shared database to prevent a double sign up.

**Listing 5.27** Client Server: phase 1 - Sign on Token

```
1  let savedemail = await db.collection(collection).find({email : email}).toArray();
2  if( savedemail.length === 0 ){ // email not registered before
3    let hash = cryptofunc.hashUTF8(email);
4    let signature = await web3.eth.sign(hash, addr);
5
6    //generate token that signed by Registration Server
7    let token = {
8                  nonce : hash,
9                  signature : signature
10                };
11
12   let data = {
13                email : email,
14                token : token
15               };
16
17   try{
18     await db.collection(collection).insertOne(data);
19     return res.render('registration/reg2', {electionname: election.title, id : id,
20                                             token : token, email:email, contract: rsc});
21   }catch(error){
22     console.log(err);
23     return res.render('registration/reg1', {electionname: election.title, id : id,
24                                             error : "Sign-up failed, please try again"});
25   }
```

After the Voter's browser received the `token` from Client server, it uses `generateCoin` provided by Client server (Listing 5.28) to create a VC as described in Figure 4.2. Then in `mint` function, which is also provided by Client server, it extracts the needed variables from signature and calls the `Mint` function of Registration Smart Contract with the signature and `cm, k, s` from the newly generated VC for verification.

**Listing 5.28** Voter's Browser: mint VC

```
1  function generateCoin(token){
2    let rho = nonce(32);
3    let r = nonce(32);
4    let k = hash("SHA-256", "HEX", rho + r, "HEX");
5    let s = nonce(32);
6    let cm = hash("SHA-256", "HEX", k + s, "HEX");
```

```
7
8     let coin = {cm:"0x" + cm, k:"0x" + k, s:"0x" + s};
9     witness = {cm:"0x" + cm, k:"0x" + k, s:"0x" + s, rho:"0x" + rho, r:"0x" + r};
10
11    mint(coin, token);
12  }
13
14  function mint(coin, token, account){
15    let eid = '0x' + document.getElementById("eid").innerHTML.substr(5);
16    let msg = token.nonce;
17    let msg_to_hash = "\x19Ethereum Signed Message:\n" +msg.length.toString() + msg;
18    let hashed_msg = web3.sha3(msg_to_hash);
19
20    let signature = token.signature.substr(2);
21    let r = '0x' + signature.slice(0, 64);
22    let s = '0x' + signature.slice(64, 128);
23    let v = '0x' + signature.slice(128, 130);
24    let v_decimal = web3.toDecimal(v) + 27;
25
26    contract.Mint(eid, coin.cm, coin.k, coin.s, hashed_msg, v_decimal, r, s,
27                       {from: web3.eth.accounts[0], gas: 150000}, function (error, result) {
28      if (error)
29      console.error(error);
30    });
31  }
```

After that, the browser waits for the `leafCreated` event that is defined in Registration Smart Contract to get the result of mint of the newly generated VC. If it is successful, i. e., the newly generated VC is saved in the Registration Smart Contract, a json file `vote_coin.json` that contains the very detail of the generated VC will be generated for Voters to download and the same information will be saved in the browser's `localStorage` as a backup. If the VC is not saved in the Registration Smart Contract, the Voter will be informed of the error and he can try to re-mint it later.

Notice that function listed in Listing 5.28 are provided by Client server but executed locally on Voters browser. That means, the Client server has no access to the details, especially the exact values of `rho` and `r`, of the generated VC.

#### Phase 2: Voting Phase

Similar to phase 1, after voting phase of an election is initialized, the voting time period is given under the election title. Figure 5.7 shows the Voter's view during the voting phase. Different to the view for phase 1, there are two buttons here. The first button `Generate vote witness` is used to generated `vote_witness.json` file that is needed for $\pi_{POUR}$ ZK-Proof generator.

**Sample Election 1**

Registration time: Ended
Voting time: 2018-12-18 00:00 to 2018-12-23 00:00

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Generate vote witness   Submit vote

**Figure 5.7:** After voting phase of an election is initialized

After a Voter clicked `Generate vote witness` button, he is asked to upload his `vote_coin.json` file to the browser. If the Voter is using the same browser that he generated his VC, he has the chance to get his VC from browser's `localStorage`. In order to make sure that Voter uploaded a valid `vote_coin.json` file, the format of the VC in this file will first be validated. If the file has a correct format, Client server will provide the candidate list to the Voter as shown in Figure 5.8.

Sample Election 1
eid: 19804e75bdb8da791458bd365ffb11efc18310eca13b68998cd96464e0cce030

**Vote 1: Generate vote witness**

Step 2

Anim pariatur cliche reprehenderit, enim eiusmod high life accusamus terry richardson ad squid.

- VPfvJcrgRYwTii
- zs2trQF69AzBla
- x3CF3EDNhm3soL
- BPh71YexuieaoE
- ilgxlX4a2dREbb
- SqWy6yhKIDCdJO
- ***I don't want to vote***

Back   Vote

**Figure 5.8:** Candidate list

In this page, Voter should choose his preferred candidate and if he wants to abstain from the election, he can chose the option at the bottom which indicates his abstention. When the Voter has made up his mind, he should then click `vote` button. However, this button will not submit his vote directly, but generates the needed witnesses and public inputs for the Voter to download. As the browser

now has the uploaded VC, the encryption key of the election from Client server and the Voter's choice, it first encrypts the Voter's vote to obtain `vote_ct`, then it calculates the `sn` of the VC using the value `rho` from the upload file, and lastly computes the Merkle Authentication Path as listed in Listing 5.29.

**Listing 5.29** Voter's Browser: calculate Merkle Authentication Path

```
1  async function generate_merkle_authentication_path(leaf_hash){
2    let path = new Array();
3    let address_bits = new Array();
4
5    try{
6      let merkle_tree = await rebuild_merkle_tree();
7      let check = await check_root(merkle_tree.root);
8
9      if(check){
10       let node = find_leaf_with_hash(leaf_hash, merkle_tree.leafs);
11       let tree_depth=0;
12
13       while(!node.isRoot){
14         let sibling_hash = node.sibling.value;
15         path.unshift(sibling_hash);
16         address_bits.push(node.addr);
17         tree_depth++;
18         node = node.parent;
19       }
20     }
21
22     let address = compute_address(address_bits);
23     return {root: merkle_tree.root, tree_depth: tree_depth, path: path,
24                                         address_bits : address_bits, address : address};
25   }catch(err){
26     vote_failed = true;
27     failed('generate');
28     let text = document.getElementById("download-fail-text");
29     text.style.display = "block";
30     let download = document.getElementById('download');
31     download.style.color = 'grey';
32     document.getElementById("hint").innerHTML = 'looks like something has went wrong,
33                                                  please try again later';
34     console.log(err);
35   }
36   return null;
37 }
```

Recall that in Registration Smart Contract, the actual MT is not saved but only its root hash `root_hash`. Hence, to compute the Merkle Authentication Path for coin commitment `cm` and `root_hash`, the corresponding MT should be regenerated using the original data (i. e., coin commitments of all valid VCs) that are saved in Registration Smart Contract. After the MT is rebuilt (line 6), the root hash of the rebuilt MT should be compared with the root hash saved in Registration Smart Contract to see whether the same MT is generated. If it is the case, this function then finds the position of

given cm (i. e., leaf_hash in Listing 5.29) in the rebuilt MT and gets the Merkle Authentication Path from bottom up. The value of address and address_bits will also be computed during Merkle Authentication Path computation.

When all witnesses and public inputs are prepared, the browser will group them into different witnesses sets as listed in Listing 5.1, Listing 5.2, Listing 5.9 and prepare vote_witness.json file for Voters to download. After that, Voters are asked to run two ZK-Proofs generators that are mentioned in Section 5.2.1. With two generated ZK-Proofs, Voter can cast his vote through Submit vote as shown in Figure 5.7.

**Listing 5.30** Voter's Browser: submit vote

```
1   async function submitVote(){
2     let coin_vk = proof.coin_proof.vk;
3     let coin_proof = proof.coin_proof.proof;
4     let coin_pub_input = proof.coin_proof.pub_input;
5
6     let vote_vk = proof.vote_proof.vk;
7     let vote_proof = proof.vote_proof.proof;
8     let vote_pub_input = proof.vote_proof.pub_input;
9
10    await contract.setVerifyingKey(vote_vk.A, vote_vk.B, vote_vk.C, vote_vk.gamma,
11                          vote_vk.gamma_beta_1, vote_vk.gamma_beta_2, vote_vk.Z, vote_vk.IC,
12                          {from : web3.eth.accounts[1], gas : 9007199254740991});
13
14    await contract.checkVote(vote_proof.A_g, vote_proof.A_h, vote_proof.B_g, vote_proof.B_h,
15                    vote_proof.C_g, vote_proof.C_h, vote_proof.H, vote_proof.K, vote_pub_input,
16                    {from : web3.eth.accounts[1], gas : 9007199254740991})
17
18    console.log("vote_format_checked");
19
20    await contract.setVerifyingKey(coin_vk.A, coin_vk.B, coin_vk.C, coin_vk.gamma,
21                          coin_vk.gamma_beta_1, coin_vk.gamma_beta_2, coin_vk.Z, coin_vk.IC,
22                          {from : web3.eth.accounts[1], gas : 9007199254740991});
23
24    contract.vote("0x"+proof.eid, "0x"+proof.sn, "0x" + proof.vote_ct,
25                    coin_proof.A_g, coin_proof.A_h, coin_proof.B_g, coin_proof.B_h,
26                    coin_proof.C_g, coin_proof.C_h, coin_proof.H, coin_proof.K, coin_pub_input,
27                    {from : web3.eth.accounts[1], gas : 9007199254740991 }, function(err, res){
28        if(err){
29          console.log(err);
30        }
31    });
32  }
```

When Voters upload his ZK-Proofs $\pi_{Coin}$ and $\pi_{Voin}$, the browser again checks the format of these two proofs. If the format is valid, the browser then extracts verifying keys, public inputs and proofs from both proof files (line 2 to line 8 in Listing 5.30). Recall that the Voting Smart Contract is implemented in this demo in such a way that only after the verifying key of a proof has been set, the corresponding proof can be verified. Hence, there are two sets of function calls in Listing 5.30 that

first set the verifying key and then verifies the proof using the function provided by Voting Smart Contract. As mentioned in Section 5.3.2, the $\pi_{Voin}$ ZK-Proof should be first verified, no matter $\pi_{Voin}$ is valid or not, $\pi_{Coin}$ will be sent to Voting Smart Contract after the verification of $\pi_{Voin}$ is done, since Voting Smart Contract guarantees that the encrypted vote will only be saved to the Smart Contract when both ZK-Proofs are valid.

Again, the functions listed in Listing 5.29, Listing 5.30 and other actions described above are executed locally on Voter's browser but are provided by Client server.

#### Phase 3: Result Ready

When the election is over or closed by EA and the result is calculated, Voters will have the chance to see the result using `Result` button as shown in Figure 5.9. The result page for Voters is the same as the result page for EA (Figure 5.4). In this implementation, the ZK-Proof of Result is not integrated, hence there are no options for Voters to check the validity of the result.

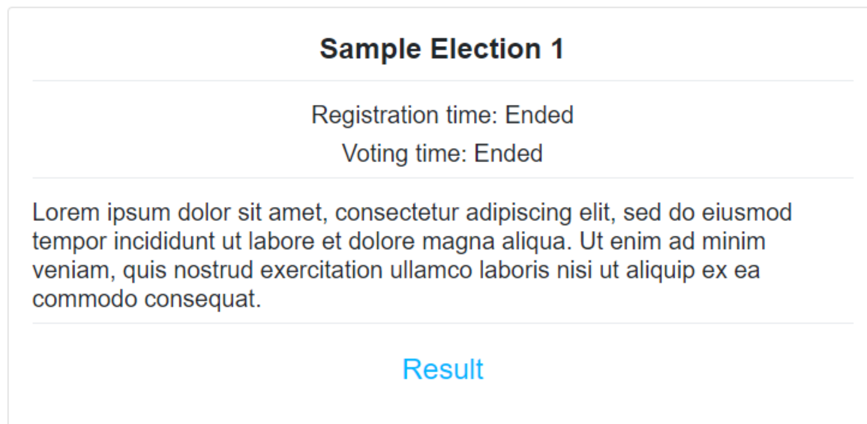



**Figure 5.9:** Result is ready

# 6 Security Analysis

Recall that the goal of PEES is to provide accuracy, verifiability and privacy against both public and EA. In this chapter, I'll explain how PEES achieves these security goals. A formal security analysis is out of the scope of the thesis, and the security analysis in this chapter is based on the protocol itself but not the implementation in Chapter 5.

### Accuracy

According to the protocol, Registration Smart Contract guarantees that each Client Server issued `token` can only be converted to one valid VC. The Voter is asked to consume this VC when casting his vote. Thus, it is guaranteed that only eligible Voters can cast votes to an election. Since the spending of VCs are saved on Blockchain, the prevention of double spending mechanism provided by Blockchain also guarantees that each eligible Voter can only cast one vote to an election.

Then for each saved votes on Blockchain, the demands of accuracy are what ZK-Proof $\pi_{Result}$ proofs. As long as the ZK-Proof $\pi_{Result}$ of an election is valid and every interested party follows the rule, that result of an election can only be trusted when the corresponding ZK-Proof is valid, then the accuracy is achieved. It is not demanded that PEES should only produce accurate result, as if the result is incorrect, it cannot provide a valid ZK-Proof that convinces everyone to believe it.

### Verifiability

Using Blockchain as the underlying technique makes the verifiability of PEES easy to achieve. As a Voter knows his encrypted vote, he can simply download all saved votes from Voting Smart Contract after an election is over and checks whether his vote is in the recorded votes or not. If it is not in there, the Voter knows that his vote has been altered during the election. However, as it is hard to find which node in the Blockchain network actually mined the Block that contains this Voters voting transaction, it is not impossible to identify which party is responsible for this data manipulation.

### Privacy

In this section, the privacy against public and against EA will be analyzed together. Recall that the privacy that PEES guarantees isn't equal to the anonymity. This means, it is not demanded that the identity of a Voter should be hide from the public. The privacy guarantees only, that the exact vote of a Voter is kept in secret to the Voter himself during and after an election. To do that we consider the views of the public and of EA during an election as shown in Figure 6.1.

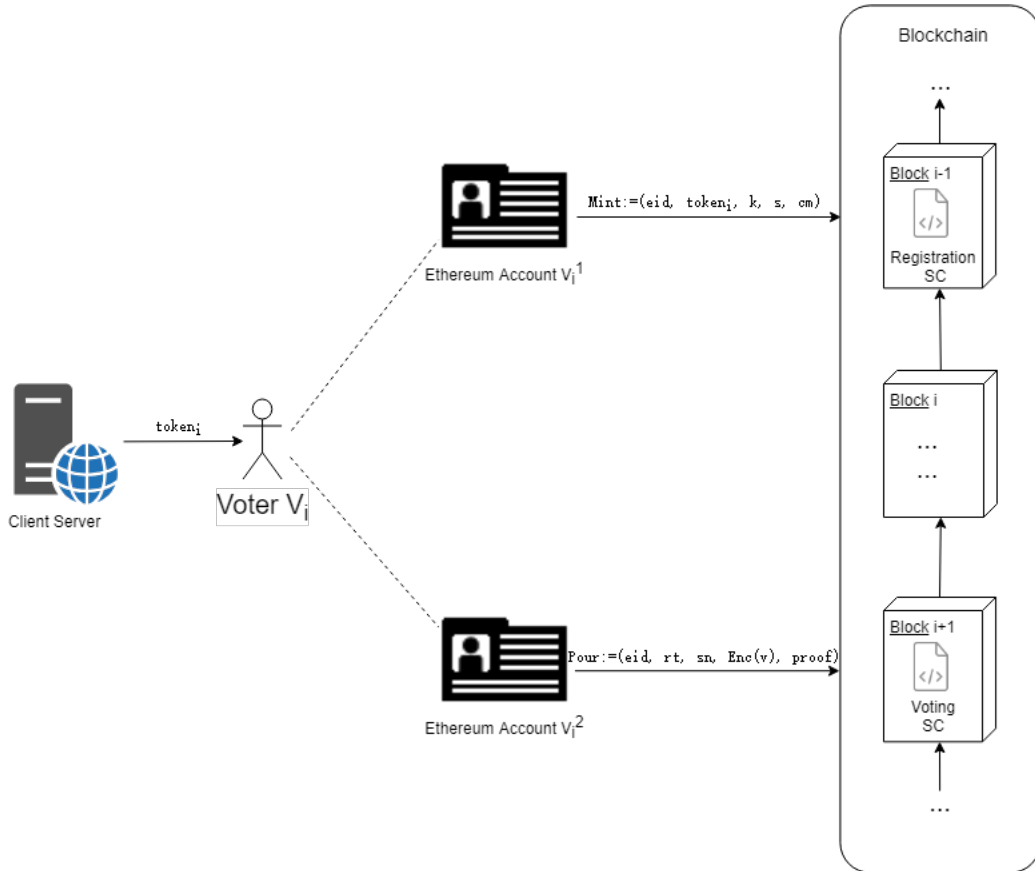



**Figure 6.1:** Views of the public and of EA during an election

According to the assumption in Section 4.2, the $\mathtt{token}_i$ from the registration phase is public, which means everyone in the network has access to that value. As this value is sent from Client server back to a Voter$V_i$, we conclude that everyone in the network knows that a Voter has received $\mathtt{token}_i$ during the registration phase and for the view of EA, it knows that the Voter $V_i$ has received $\mathtt{token}_i$ during the registration phase as EA knows the identity of each Voter.

Then after $V_i$ generated his VC locally, he sends a mint transaction that contains `eid`, $\mathtt{token}_i$, `k`, `s` and `cm` to Registration Smart Contract. This is again considered to be public. Since the same $\mathtt{token}_i$ appeared in this transaction, everyone in the network now has the knowledge that a Voter with his Ethereum account $V_i^1$ has minted a VC with coin commitment `cm`. Again, as EA knows each Voter's authentication data, it can additionally connect the Voter's identity $V_i$ with his Ethereum account $V_i^1$.

After that, Voter $V_i$ will cast his vote `v` encrypted on Blockchain using pour transaction through his another Ethereum account $V_i^2$. Notice that pour transaction doesn't contain any values that appeared before (except `eid` which contains no information about the Voter). In addition, it is not possible to retrieve the `cm` from `sn` and the public doesn't have the decryption key for the encrypted vote `Enc(v)`, we conclude that everyone in the network only knows that, a Voter with his Ethereum account $V_i^2$ has cast his vote, but the exact vote `v` and whether $V_i^2$ and $V_i^1$ belongs to the same Voter is not known.

To only protect privacy against public, a Voter can use the same Ethereum account for mint transaction and pour transaction, since the public can't decrypt the votes on Blockchain and they don't know the identity of a Voter anyway. However, to protect the privacy against EA, a Voter must use two different Ethereum accounts. Recall that PEES split the decryption key of an election into multiple shares so that one single Admin from EA can't decrypt any vote from Blockchain. However, we can't rule out the case that more than $n$ Admins from EA are malicious, where $n$ represents the threshold of the secret sharing scheme. In this case, these Admins could combine the decryption key outside of Admin server and hence they could decrypt any encrypted votes from the Blockchain. Notice that these Admins already knows that Voter $V_i$ has Ethereum account $V_i^1$ from registration phase, thus, although the ZK-Proof breaks the connection of `cm` and `sn`, these malicious Admins can still connect the actual vote `v` to the Voter $V_i$ through his Ethereum account, if Voter $V_i$ uses the same Ethereum account in both registration and voting phases. Therefore, we conclude that the privacy against EA can only be achieved, when Voters use different Ethereum accounts for registration and voting phases respectively.

# 7 Conclusion and Further works

E-Voting and Blockchain have been popular topics these years. There are some researches that build up E-Voting systems using Blockchain, however, as Blockchain keeps every recorded data public, it is hard to bring privacy, especially the privacy against election organizers, into E-Voting systems using Blockchain. In this thesis, we proposed a new E-Voting system PEES that is based on Blockchain and still provides privacy for Voters. With the help of ZK-Proof $\pi_{POUR}$, PEES not only provides privacy on-chain, and also provides the privacy against election organizers. With another ZK-Proof $\pi_{Result}$, PEES proofs to every interested party that the accuracy of an election holds. Using Blockchain as the underlying technology, PEES also provides verifiability, which allows Voters to check, whether their votes have been manipulated during the election or not.

Apart from the protocol of PEES, this thesis also provides a demo of PEES. In this demo, the fundamental part of PEES protocol is implemented, some code segments such like the code segment for Voter authentication can be changed to other specific rules for certain elections. However, this demo is not complete, as the generation of a ZK-Proof $\pi_{Result}$ takes too much time, this part is not integrated in the demo. In addition, $\pi_{Coin}$ and $\pi_{Vote}$ are developed in different program languages using different libraries, it requires a Voter to call two different programs during an election to generate needed ZK-Proofs, which makes this demo very inconvenient to use in practice. Another drawback of PEES is that, to verify a ZK-Proof on Ethereum Blockchain costs lots of Ether. If Voters don't get refund from EA, Voters might get less motivated to attend to an election.

### Further works

The current demo of PEES still has many places that can be improved. For ZK-Proof generators, the constrain system of $\pi_{Coin}$ can be hard-coded in the program and the corresponding verification key can be hard-coded in the Voting Smart Contract. As for ZK-Proofs $\pi_{Vote}$, $\pi_{Result}$ that involve constrains of RSA encryption and decryption, the demo has built the exponent `e` directly into constrain system but if it can be given as a public input (for encryption) or witness (for decryption), then the constrain system of these two ZK-Proofs can also be fixed and therefore be hard-coded into the program.

By the time this thesis is written, there are no other libraries apart from `libsnark` and `Jsnark` that can be used to generate ZK-Proofs from zk-SNARK. If the generation of ZK-Proofs can be provided by Ethereum, js/node.js or other back-end / front-end program languages, the proof generation can be embedded into the current implementation so that Voters don't have to run an external proof generator during an election.

ZoKrates, A toolbox for zk-SNARK on Ethereum [ETa] provides a high-level language for generating ZK-Proofs of computations and to verify those proofs in Solidity. Currently, it only has a proof-of-concept implementation [ETb], when all the testing of ZoKrates are done, one can consider using ZoKrates as the ZK-Proof generator for PEES.

In the current PEES protocol, the deployed two Smart Contracts are election related. This is aimed to separate the storage of minted VCs and recorded ballots from different elections. However, It is not compared with the approach that uses only two Smart Contracts for all elections and store the minted VCs and recorded ballots under different variables. One main aspect that needs to be compared is the overall Ether spent during an election. For the later approach, the maximal storage of a Smart Contract might also be a constrain, since if the maximum storage of a Smart Contract is reached, a new Smart Contract should be deployed as the replacement of the former one.

### Acknowledgement

First and foremost, I would like to express my sincere gratitude to Prof. Ralf Küsters for giving me such interesting topic. I would also like to expand my gratitude to my supervisor Dipl.-Math. Mike Simon for the continuous support of my bachelor thesis, and for his patience, motivation, and immense knowledge. His guidance helped me in all the time of developing PEES and writing of this thesis.

My sincere thanks also go to Jingxi Zhang for checking my thesis and offering me valuable advices.

Last but not the least, I would like to thank my family: my parents Zhiyong Li and Zhengrong Yang, for giving birth to me at the first place and supporting me spiritually throughout my life.

# Bibliography


[BACa] S. E. B, C. A, G. C, et al. *Zerocash*. URL: http://zerocash-project.org/ (visited on 11/01/2018) (cit. on p. 30).

[BC] L. Baptiste, D. Christopher. *Jsoncpp*. URL: http://open-source-parsers.github.io/jsoncpp-docs/doxygen/index.html (visited on 12/08/2018) (cit. on p. 52).

[BCCT11] N. Bitansky, R. Canetti, A. Chiesa, E. Tromer. *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Cryptology ePrint Archive, Report 2011/443. https://eprint.iacr.org/2011/443. 2011 (cit. on p. 21).

[BCI+12] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, O. Paneth. *Succinct Non-Interactive Arguments via Linear Interactive Proofs*. Cryptology ePrint Archive, Report 2012/718. https://eprint.iacr.org/2012/718. 2012 (cit. on p. 21).

[BS18] H. J. Bob Summerwill. *Ethereum Homestead Documentation*. 2018. URL: http://ethdocs.org/en/latest/index.html (visited on 10/31/2018) (cit. on pp. 29, 30).

[But15] V. Buterin. *Merkling in Ethereum*. 2015. URL: https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/ (visited on 10/20/2018) (cit. on pp. 17, 18).

[CAa] Chriseth, B. Alex, et al. *Solidity*. URL: https://solidity.readthedocs.io/en/v0.4.25/ (visited on 12/08/2018) (cit. on p. 51).

[CS] L. Christian, M. Sam. *libsnark tutorial*. URL: https://github.com/christianlundkvist/libsnark-tutorial/blob/master/src/ethereum/contracts/Verifier.sol (visited on 12/13/2018) (cit. on p. 73).

[Chr] R. Christian. *snarktest.solidity*. URL: https://gist.github.com/chriseth/f9be9d9391efc5beb9704255a8e2989d (visited on 12/13/2018) (cit. on p. 71).

[DED] I. Dev, H. Eliot, M. Dwight. *mongoDB*. URL: https://www.mongodb.com (visited on 12/26/2018) (cit. on p. 74).

[Doo] *Doodle*. https://doodle.com/. Accessed: 2018-10-26 (cit. on p. 13).

[ETa] J. Eberhardt, S. Tai. "ZoKrates-Scalable Privacy-Preserving Off-Chain Computations". In: () (cit. on p. 93).

[ETb] J. Eberhardt, S. Tai. *ZoKrates*. URL: https://github.com/Zokrates/ZoKrates (visited on 12/20/2018) (cit. on p. 93).

[ElG85] T. ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472 (cit. on p. 16).

[Eli] B. Eli. *Efficient modular exponentiation algorithms*. URL: https://eli.thegreenplace.net/2009/03/28/efficient-modular-exponentiation-algorithms (visited on 12/26/2018) (cit. on p. 61).

[Est] *e-Governance*. https://e-estonia.com/solutions/e-governance/i-voting/. Accessed: 2018-10-26 (cit. on p. 13).

[Eth] Ethereum. *Ethereum JavaScript API*. URL: https://github.com/ethereum/web3.js/ (visited on 12/06/2018) (cit. on p. 51).

[FIP95] P. FIPS. "180-1. secure hash standard". In: *National Institute of Standards and Technology* 17 (1995), p. 45 (cit. on p. 56).

[Fou] N. Foundation. *NodeJs*. URL: https://nodejs.org/en/ (visited on 12/06/2018) (cit. on p. 51).

[GGM84] O. Goldreich, S. Goldwasser, S. Micali. "How to construct Randolli functions". In: *Foundations of Computer Science, 1984. 25th Annual Symposium on*. IEEE. 1984, pp. 464–479 (cit. on p. 31).

[Gab] B. Gabriel. *Shamir's Shared Secred Scheme - JS version of B. Poettering's Linux CLI utility*. URL: https://github.com/gburca/ssss-js (visited on 12/17/2018) (cit. on pp. 53, 77).

[HL10] C. Hazay, Y. Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010. ISBN: 978-3-642-14302-1. DOI: 10.1007/978-3-642-14303-8. URL: https://doi.org/10.1007/978-3-642-14303-8 (cit. on p. 19).

[IOSGI06] O. (weizmann Institute Of Science Goldreich (Israel)). *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, 2006 (cit. on p. 15).

[JW17] w. Jeffrey Wilcke. *Contracts and Transactions*. 2017. URL: https://github.com/ethereumproject/go-ethereum/wiki/Contracts-and-Transactions (visited on 10/31/2018) (cit. on p. 29).

[KL07] J. Katz, Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. ISBN: 1584885513 (cit. on p. 15).

[KS98] B. Kaliski, J. Staddon. *PKCS# 1: RSA cryptography specifications version 2.0*. Tech. rep. 1998 (cit. on p. 16).

[KTV12] R. Kusters, T. Truderung, A. Vogt. "Clash attacks on the verifiability of e-voting systems". In: *Security and privacy (SP), 2012 IEEE symposium on*. IEEE. 2012, pp. 395–409 (cit. on p. 13).

[Kam18] Kamiov. *JavaScript*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 12/06/2018) (cit. on p. 51).

[Kos] A. Kosba. *Jsnark*. URL: https://github.com/akosba/jsnark (visited on 12/08/2018) (cit. on pp. 52, 61).

[Laba] S. Lab. *libsnark: a C++ library for zkSNARK proofs*. URL: https://github.com/scipr-lab/libsnark (visited on 12/08/2018) (cit. on pp. 20, 52, 57).

[Labb] R. Laboratories. *PKCS 1 v2.2: RSA Cryptography Standard*. URL: https://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf (visited on 12/16/2018) (cit. on p. 61).

[Lau] E. Laurent. *node-bignumber*. URL: https://github.com/eschnou/node-bignumber (visited on 12/17/2018) (cit. on pp. 53, 77).

[MAAS13] M. F. Mursi, G. M. Assassa, A. Abdelhafez, K. M. A. Samra. “On the development of electronic voting: a survey”. In: *International Journal of Computer Applications* 61.16 (2013) (cit. on p. 13).

[MVO96] A. J. Menezes, S. A. Vanstone, P. C. V. Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0849385237 (cit. on p. 56).

[Mod18] R. Modi. *Introduction to Blockchain, Ethereum and Smart Contracts — Chapter 1*. 2018. URL: https://medium.com/coinmonks/https-medium-com-ritesh-modi-solidity-chapter1-63dfaff08a11 (visited on 10/31/2018) (cit. on p. 28).

[NBF+16] A. Narayanan, J. Bonneau, E. Felten, A. Miller, S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton University Press, 2016. ISBN: 0691171696, 9780691171692 (cit. on p. 23).

[Nak08] S. Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: (2008) (cit. on p. 23).

[PHGR16] B. Parno, J. Howell, C. Gentry, M. Raykova. “Pinocchio: Nearly practical verifiable computation”. In: *Communications of the ACM* 59.2 (2016), pp. 103–112 (cit. on p. 52).

[RSA78] R. L. Rivest, A. Shamir, L. Adleman. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems”. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: http://doi.acm.org/10.1145/359340.359342 (cit. on p. 16).

[Rei16] C. Reitwiessner. *zkSNARKs in a nutshell*. 2016. URL: https://chriseth.github.io/notes/articles/zksnarks/zksnarks.pdf (visited on 10/26/2018) (cit. on p. 20).

[SCG+14] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, M. Virza. “Zerocash: Decentralized anonymous payments from bitcoin”. In: *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2014, pp. 459–474 (cit. on pp. 13, 30–33).

[Sha79] A. Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613 (cit. on pp. 19, 53, 77).

[Str] StrongLoop/IBM. *Express4.16.4 - Fast, unopinionated, minimalist web framework for Node.js*. URL: https://expressjs.com/ (visited on 12/06/2018) (cit. on p. 51).

[Tom] W. Tom. *RSA and ECC in JavaScript*. URL: http://www-cs-students.stanford.edu/~tjw/jsbn/ (visited on 12/17/2018) (cit. on p. 53).

[Vit] B. Vitalik. *Quadratic Arithmetic Programs: from Zero to Hero*. URL: https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649 (visited on 12/23/2018) (cit. on pp. 20, 21).

[Vox] *VoxVote*. http://www.voxvote.com/. Accessed: 2018-10-26 (cit. on p. 13).

[Woo14] G. Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. 2014. URL: https://gavwood.com/paper.pdf (visited on 10/31/2018) (cit. on p. 30).

[ZAAA+14] M. T. I. Ziad, A. Al-Anwar, Y. Alkabani, M. W. El-Kharashi, H. Bedour. “E-voting Attacks and Countermeasures”. In: *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. IEEE. 2014, pp. 269–274 (cit. on p. 13).

[ZXD+17] Z. Zheng, S. Xie, H. Dai, X. Chen, H. Wang. “An overview of blockchain technology: Architecture, consensus, and future trends”. In: *Big Data (BigData Congress), 2017 IEEE International Congress on*. IEEE. 2017, pp. 557–564 (cit. on pp. 25–27).

All links were last followed on December 27, 2018.

## Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature