University of Zurich^{UZH}

Privacy, Verifiability, and Auditability in Blockchain-based E-Voting

Raphael Matile, Christian Killer
Zurich, Switzerland
Student ID: 12-711-222, 12-738-118

Supervisor: Prof. Dr. Thomas Bocek, Bruno Rodrigues
Date of Submission: April 4th, 2018

MASTER PROJECT THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Project Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Elektronisches Abstimmen ist seit geraumer Zeit in aller Munde. In der kryptografischen Forschung bietet das elektronische Abstimmen ein Paradebeispiel um verschiedenste Sicherheitsansätze anschaulich zu evaluieren. In der Praxis führen erste Länder Abstimmungen mit dem elektronischen Stimmkanal auf nationaler Ebene durch. In technologischer Hinsicht ist der Rückgrat der Abstimmungen oft ein proprietäres System, welches das Nachvollziehen der Stimmergebnisse bei Unklarheiten für den einzelnen Wähler erschwert. Jedoch ist in den letzten Jahren mit der Blockchain ein neues technologisches Instrument soweit entwickelt worden, sodass erste Anwendungen auf seine vielversprechenden Eigenschaften vertrauen: eine garantierte Unveränderbarkeit einmal geschriebener Werte sowie die dezentrale Betriebsweise. Das Ziel dieser Arbeit besteht darin, diese Bereiche zusammenzuführen, um ein transparentes und sicheres Abstimmen auf dem elektronischem Stimmkanal zu ermöglichen. Das entwickelte System speichert dabei die abgegebenen Stimmen in verschlüsselter Form mit einem zugehörigen Beweis für ihre Korrektheit auf der Ethereum Blockchain. Eine Serverkomponente übernimmt dabei die Verschlüsselung der Stimmen, das Berechnen des dazugehörenden Beweises und schlussendlich auch das Berechnen des Abstimmungsresultats. Für die Abstimmungsverwaltung auf Seiten der Administratoren sowie die Stimmabgabe seitens der Wähler sind weiter zwei unterschiedliche Browser-Applikationen entwickelt worden. Durch das transparente Speichern der Stimmen auf der Blockchain, können einerseits Aufsichtsparteien die korrekte Auszählung überprüfen, andererseits ermöglicht dies jedem Stimmbürger seine verschlüsselte Stimme in der totalen Menge aller Stimmen wiederzufinden, sowie die Unveränderlichkeit dieser zu validieren.

Abstract

Since many years, electronic voting is a common topic in cryptographic research. Electronic voting systems offer an optimal way to evaluate security attributes. In practice, some countries have started to use an electronic channel for nationwide elections and votes. Often, the technological backbone of these electronic systems is proprietary software. Hence, such voting systems lack transparency from a voter's perspective. In recent years, the blockchain technology has been actively developed and matured to a level where first applications build on its promising features: guaranteed immutability of written values as well as its fully decentralized architecture. This work focuses on bringing these two areas together, working towards a transparent and secure voting system. Once submitted, encrypted votes are stored on the blockchain, along with evidence proving their validity. The architecture consists of multiple entities. Whereas a server component encrypts the votes and generates a corresponding proof, two distinct browser applications allow voting authorities to administer the vote and voters to submit their choices. The correctness of the voting result can eventually be retraced by any supervisory body. This is possible due to the transparent persistence of each vote on the blockchain. Further, this allows each voter to locate his own vote in the total set of submitted ones and further enables him to verify the immutability of his vote.

Contents

Chapter 1

Introduction

Electronic voting schemes are widely researched, covering many different areas from cryptography, security, usability and even consider legal and social aspects [4]. The main technical challenge in the design of electronic voting systems is to achieve privacy and verifiability, ultimately conflicting properties. In simple terms, privacy should assure that a ballot can not be traced back to the voter, but then again, verifiability requires that a voter can check that a ballot was indeed casted, recorded and counted as intended [5]. In recent years, research further revised the notions of privacy and verifiability. Thus, the resulting refinement of these properties led to advances and development of cryptographic building blocks. Still, practically implementing an electronic voting scheme is complex, especially while addressing nation-wide electoral processes in which the trust in governments and democracy is at stake. In recent years, blockchains have gained attention on a global scale. These distributed ledgers are very interesting in the context that they provide a fully decentralized, tamper-proof infrastructure to execute code and store data [6]. Electoral processes should not depend on trust but rather produce substantial evidence that can be individually and universally verified [4]. Instead of relying on central authorities, evidence can be generated and stored by such a fully decentralized infrastructure. Consequently, this work aims at offering a blockchain-enabled electronic voting system that satisfies a limited level of privacy and verifiability.

1.1 Motivation and Description of Work

There is only few research in combining electronic voting with distributed ledgers. Most implemented voting systems rely on trusted centralized client-server architectures with single points of failures. The goal of this work is to use a private proof-of-authority blockchain as a public bulletin board and use smart contracts to encrypt, collect and verify the ballots. The system should achieve an acceptable level of privacy, verifiability and auditability under certain assumptions. In order to do that, homomorphic encryption will be used as primary cryptographic primitive. The following work describes the steps that were taken to achieve this objective. It further outlines the final implementation and required theoretical building blocks to engineer the final architecture which was evaluated

Figure 1.1: Provotum logotype

in a small pilot voting. The name of the system in this work is *Provotum* and the logo is displayed in Figure 1.1. All the code referenced in this work is published with the Apache License 2.0 and freely available on GitHub [1].

1.2 Thesis Outline

In chapter 2, the background and related work is denoted. Then, in chapter 3, the architecture of the proposed system is described in detail. Subsequently in chapter 4, the voting process is described. Chapter 5 evaluates the proposed *Provotum* system. Finally, this work concludes in Chapter 6.

---

[1] `https://github.com/provotum`

Chapter 2

Background and Related Work

Voting schemes are an actively researched and very broad field. Prior research covers various areas, thus to get an understanding of the most important properties and building blocks, it is crucial to use surveys as a first starting point [4] [7] [25] [40].

2.1 Related Work

Most research in the field of secret-ballot voting schemes is very theoretical in nature. The most important notions of privacy have been further separated into ballot-secrecy, receipt-freeness, coercion-resistance and everlasting privacy [2]. One of the most prominent methods to achieve privacy is homomorphic encryption, since it enables the direct operation on encrypted data [1]. Additional methods include re-encryption, blind signatures, zero-knowledge and designated verifier proofs or mixnets [25]. Naturally, secure and private communication channels are a prerequisite of most electronic voting systems. In the context of electronic voting, verifiability states that a voter can trace the effect of his or her vote on the final tally [25]. In other words, verifiability assures that it verifiable that the final tally is correct, even if parts of the voting scheme are partially untrusted [29]. Additionally, the term end-to-end verifiability defines the three characteristics cast-as-intended, recorded-as-cast and tallied-as-recorded [5]. Further, verifiability can be divided into individual and universal verifiability. According to Sako and Kilian [37], Individual verifiability is achieved if a sender can verify whether or not his message reached the destination, but cannot determine if this is true for the other voters. Additionally, universal verifiability guarantees that it is possible to publicly verify that the tally of the ballots is correct. These refined notions of privacy and verifiability have become very popular to assess the functionality of electronic voting systems. A crucial part in achieving verifiability is a tamper-proof secure public bulletin board. Verifiable systems should be auditable in order to offer the possibility to verify proper functionality. Hence, there is active research in ballot auditing techniques, such as Markpledge-style auditing [24]. Due to the high variety of well-defined properties, the implementation of practical systems poses an extremely difficult and interdisciplinary task. Prior research combined electronic voting with decentralized ledgers, showcasing the feasibility of using Ethereum

Smart Contracts [30]. Finally, the challenge is to bring theoretical building blocks together to form a working piece of software that is tamper-proof and privacy-preserving while being verifiable.

In Switzerland, many efforts were made to bring theory closer together to practice. Additionally, the progress of electronic voting in Switzerland is driven by increased efforts of the Federal Council. The most relevant regulations by the Federal Council were first published in 2013 and are aligned with theoretical research, thus referring to privacy and verifiability [13]. The first main developer is The Swiss Post. Their system is developed as a closed-source system in partnership with Scytl. Their system is currently ceritifed for 50% of eligible voters [43]. On the other hand, there is the radically transparent open-source CHVote system which is developed by academic researchers and cryptographers from the University of Applied Sciences Bern [36] in collaboration with the canton of Geneva [9]. Meeting the requirements posed by the regulatory landscape is a real challenge for all competitors and researchers in the electronic voting market.

2.2 Regulatory landscape in Switzerland

Legal regulation sets the target boundaries for any adaption, development and future integration of electronic voting. Initiated by the Federal Council, the first serious efforts to test electronic voting were made in 2001 by the consortium *Vote Électronique*. Following that, three pilot projects were started in different cantons. The three cantons Geneva, Zurich and Neuchâtel pioneered in testing the viability of electronic voting in real-world elections. The canton of Geneva was the first project that was started in 2001 and is still under development under the name CHVote [9]. The Canton of Neuchâtel initiated a holistic approach called *Guichet Unique* [23]. The vision of *Guichet Unique* is to implement a virtual office counter, offering digitalized services to its citizens and electronic voting simply as one of many services. Due to the increased complexity, an external company called Scytl joined the project [38]. In 2015, the Canton of Neuchâtel partnered with Scytl and The Swiss Post to implement a new generation of the Post's electronic voting system [42]. The canton of Zurich developed its system in a partnership with the Swiss company Unisys. Unisys was responsible for the operation and maintenance of the system [41]. However, the canton of Zurich discontinued the project due to operational problems and new federal regulations [20].

These various projects helped to build up experience and knowledge in operating and running electronic voting systems. The repeated positive assessment by the government led to the adoption by other cantons.

In the beginning of 2014, the new *Ordinance of the Federal Chancellery on Electronic Voting* (VEleS) came into effect [13]. VEleS defines how each canton is allowed to test different prototypes of e-Voting systems. Thus, it specifies the conditions for the expansion of the electronic voting channel in nation-wide elections. The content of VEleS is heavily aligned with the notions of privacy and verifiability in research [19]:

- **Cast-as-intended** describes the ability of a voter to ensure that the casted and encrypted vote was in fact cast as intended.

- **Recorded-as-cast** verification allows the voter to review that the voting server correctly received his vote.
- **Counted-as-recorded** provides auditors, as well as voters with the ability to verify that all votes were received and correctly influenced the voting result.

In order to allow systems to work with up to 50% of all eligible voters, the system needs to assure at least *cast-as-intended* verification [13]. The attributes *recorded-as-cast* as well as *counted-as-recorded* have to be provided to license the system for participation by more than 50% of all eligible voters [13].

In 2015 the consortium Vote Électronique was discontinued and left only two main competitors in the Swiss electronic voting space: CHVote [9] and The Swiss Post [43]. Overall, the regulatory requirements act as boundaries for the development of electronic voting systems. A constant update of regulations with regards to technological advances is necessary to keep regulations in sync with realistic expectations towards real-world electronic voting systems. In conclusion, electronic voting is a highly regulated topic in Switzerland. These regulations cause long and costly pilot projects and an expensive licensing procedure. The future will show if all cantons will use the same system or if competition will be sparked by the spirit of cantonal federalism.

2.3 Cryptographic Fundamentals

In order to implement an electronic voting scheme, cryptographic building blocks are necessary. In the following sections, the utilized cryptographic fundamentals are introduced and explained in detail.

2.3.1 Homomorphic Encryption in the ElGamal Cryptosystem

In computing, encryption is essential to preserve confidentiality of sensitive data and information. The emergence of cloud services raises issues regarding the privacy of user and business data. Ordinary encryption schemes cannot operate on encrypted data without first decrypting it. However, the application of practical and efficient homomorphic encryption schemes can solve these problems.

Homomorphic encryption (HE) can be described as follows:

> *Homomorphic encryption is a form of encryption which allows specific types of computations to be carried out on ciphertexts and generate an encrypted result which, when decrypted, matches the result of operations performed on the plaintexts* [45].

In mathematical terms, homomorphic encryption can be defined as follows [18]:

> Let M) denote the set of the plaintexts (resp., ciphertexts). An encryption scheme is said to be homomorphic if for any given encryption key k the encryption function E satisfies.
>
> $$\forall m_1, m_2 \in M, \quad E(m_1 \odot_M m_2) \leftarrow E(m_1) \odot_C E(m_2)$$
>
> for some operators $\odot_M$ in M and $\odot_C$ in C, where $\leftarrow$ means "can be directly computed from", that is, without any intermediate decryption. If $(M, \odot_M)$ and $(C, \odot_C)$ are groups, we have a *group homomorphism.* We say a scheme is *additively homomorphic* if we consider addition operators, and *multiplicatively homomorphic* if we consider multiplication operators.

The ElGamal cryptosystem was developed by Taher ElGamal in 1985 [14], hence its name. The original version was only multiplicatively homomorphic. Later, the cryptosystem was extended to also support addition over encrypted data. Usually, computation is performed over a finite set of numbers, a cyclic group. Data is therefore first transformed to a member of such a group, before being encrypted or decrypted. These steps are defined below:

Cyclic Groups A cyclic group G of a particular order q is generated from a single element: the generator g. In particular, for a multiplicative group of integers modulo p, i.e. $(\mathbb{Z}_p)^*$, the finite amount of group elements are obtained by applying as operation the nth-multiplication to g and taking the modulus. However, such a group is only cyclic whenever q is one of the numbers described in the sequence `A033948` [31] and $p = 2q + 1$. Consider the following example: Select $g = 2$ and $q = 5$, $p = 2q + 1 = 11$, then

$$\begin{aligned} G &= g^0, g^1, g^2, g^3, g^4 \\ &= 2^0, 2^1, 2^2, 2^3, 2^4 \\ &= 1, 2, 4, 8, 16 \text{ //apply modulus } p \\ &= 1, 2, 4, 8, 5 \end{aligned}$$

Key Generation For key generation, a generator g, a cyclic subgroup G of order q of $(\mathbb{Z}_p)^*$ must be defined, with q being co-prime to p.

Private Key One chooses a random x from the set $\{1, ..., q - 1\}$ and further keeps x secret.

Public Key To obtain the public key, one generates $h = g^x$ and publishes the set of public parameters (G, q, g, h).

Multiplicative Variant

In the multiplicative variant of the ElGamal encryption the basic operations are defined as follows:

Encryption In order to encrypt a message $m \in \mathbb{Z}_q$ with a given public key, one generates $G = g^r$ with r being selected randomly from the set $\{1, ..., q-1\}$. Then, one calculates the shared secret $s = h^r = (g^x)^r = g^{xr}$. Eventually, the ciphertext is defined as $E(G, H) = \big(g^r, m \cdot s\big)$.

Decryption To decrypt a ciphertext $E(G, H)$, one calculates the shared secret $s = (g^r)^x = g^{rx}$ and following $m = H \cdot (s^{-1}) = m \cdot h^r \cdot (g^{xr})^{-1} = m \cdot g^{xr} \cdot g^{-xr}$ with s^{-1} being the modular multiplicative inverse of s.

Homomorphic Multiplication The multiplication of the plaintext value over two ciphertexts is then performed as follows [33]:

$$\begin{aligned} E(m_1) \cdot E(m_2) &= E(G_1, H_1) \cdot E(G_2, H_2) \\ &= (g^{r_1}, m_1 \cdot h^{r_1}) \cdot (g^{r_2}, m_2 \cdot h^{r_2}) \\ &= (g^{r_1+r_2}, (m_1 \cdot m_2) \cdot h^{r_1+r_2}) \\ &= E(m_1 \cdot m_2) \end{aligned} \tag{2.1}$$

Additive Variant

The additive alternative of ElGamal takes the message m to the power of the generator g. Therefore, the basic operations result in the following [11]:

Encryption In order to encrypt a message $m \in \mathbb{Z}_q$ with a given public key, one generates $G = g^r$ with r being selected randomly from the set $\{1, ..., q-1\}$. Then, one calculates the shared secret $s = h^r = (g^x)^r = g^{xr}$. Eventually, the ciphertext is defined as $E(G, H) = \big(g^r, g^m \cdot s\big)$.

Decryption To decrypt a ciphertext $E(G, H)$, one calculates the shared secret $s = (g^r)^x = g^{rx}$ and following $g^m = H \cdot (s^{-1}) = g^m \cdot h^r \cdot (g^{xr})^{-1} = g^m \cdot g^{xr} \cdot g^{-xr}$ with s^{-1} being the modular multiplicative inverse of s. Then, the discrete logarithm has to be solved in order to obtain m.

Homomorphic Addition The addition of the plaintext values then follows the proof below [33]:

$$\begin{aligned} E(m_1) \cdot E(m_2) &= E(G_1, H_1) \cdot E(G_2, H_2) \\ &= E(g^{r_1}, g^{m_1} \cdot h^{r_1}) \cdot E(g^{r_2}, g^{m_2} \cdot h^{r_2}) \\ &= E(g^{r_1+r_2}, g^{m_1+m_2} \cdot h^{r_1+r_2}) \\ &= E(m_1 + m_2) \end{aligned} \tag{2.2}$$

In this work the additive variant is used. The additive variant allows a simple tallying process in order to obtain the final tally of a voting.

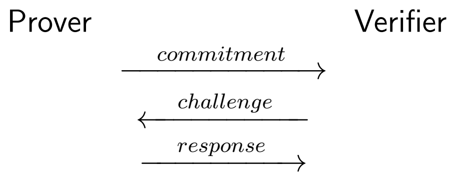


Figure 2.1: The three steps of a Σ-protocol

2.3.2 Proofs of Knowledge

In an electronic voting scheme, encrypted votes ensure the privacy of a voter's ballot. Orthogonally, the system has to ensure that only valid votes are eventually counted to the end-result of any vote. Proofs of knowledge may ensure such a property in an elegant way.

In an interactive proof system, two parties are involved: A prover P claiming a particular statement and a verifier V verifying the validity of this statement. In literature, two main types of such proofs are distinguished: A *proof of knowledge* is a proof which guarantees a particular statement whereas the statement itself may be required to be made public, whereas a *zero-knowledge proof of knowledge* (ZKP) allows the verifier to successfully verify the validity of a particular claim without needing the prover to disclose any of the private information. [21]

An interactive proof system is called a Σ-protocol if it follows a three-way protocol of interactions between the prover and the verifier. All required properties are outlined in the definition from [12]:

> A protocol $\mathcal{P}$ is said to be a Σ-protocol for relation R if:
>
> - $\mathcal{P}$ is of the above 3-move form, and we have completeness: if P, V follow the protocol on input x and private input w to P where $(x, w) \in R$, the verifier always accepts.
> - From any x and any pair of accepting conversations on input x, $(a, e, z), (a, e', z')$ where $e \neq e'$, one can efficiently compute w such that $(x, w) \in R$. This is sometimes called the special soundness property.
> - There exists a polynomial-time simulator M, which on input x and a random e outputs an accepting conversation of the form (a, e, z), with the same probability distribution as conversations between the honest P, V on input x. This is sometimes called special honest-verifier zero-knowledge.

The aforementioned 3-move form is shown in Figure 2.1 and involves a commitment of the prover to the verifier which the verifier can challenge in turn. Finally, a response is sent back to the verifier. Based on that response the verifier can decide whether the claimed statement is correct.

Since the verifier can challenge the prover multiple times in a sequence, the verifier gains trust in the prover every time the prover can return a valid response. A cheating prover has an average probability of 0.5 of answering correctly to an arbitrary challenge. Therefore, the verifier can repeat the challenge i times in order to obtain any desired target probability v of having an honest prover: $v = \Sigma_{n=1}^{i}(0.5)^{i}$. [25]

However, the interaction between $\mathcal{P}$ and $\mathcal{V}$ is not always desired: Once having a communication channel established, both, the prover and the verifier need to stay connected during the entire procedure in order to exchange messages between them. Further, in an interactive zero-knowledge proof, the transcript of the communication cannot convince any third party of the correctness of the proof itself, since a malicious verifier could also have computed the entire communication by itself. Therefore, interactive zero-knowledge proofs are non-transferable. [25].

This interaction between parties can be avoided by following a procedure outlined in [17] which was applied to the problem of proving the knowledge about the discrete logarithm for a particular number. By replacing the challenge chosen by the verifier with a value computed by a cryptographic hash function, the proof can still be correctly computed and verified.

Proof of Plaintext Validity

For a particular ElGamal ciphertext, a non-interactive proof allowing a verifier to ensure that an encrypted ballot actually contains either zero or one is outlined in Algorithm 1. It is based on the mathematical outline of proofs for ElGamal ciphertexts from [28]. For each domain variable which is not the plaintext, the algorithm will generate fake commitments (y, z), fake challenges (c) as well as fake responses (s). After having computed the challenge (the step usually performed by the verifier in an interactive Σ-protocol), the challenge as well as the response are adjusted to values appropriate to the plaintext message.

Algorithm 2 shows how a verifier can perform a check for the validity of the proof. First he reconstructs the commitments (y, z) obtained from the prover. Then, he recalculates the hash. If the sum of the challenges for each of the domain values is equal to the reconstructed challenge, then the proof is valid.

Algorithm 1: Creation of a non-interactive proof for ballot validity

```
Data: ElGamal public key parameters $p, g, h$, Plaintext m, Ciphertext $E = (G, H)$,
      Domain $D$ of valid messages $\{0, 1\}$
begin
    $sb \leftarrow \|g\|h\|G\|H$
    $t \in_{random} \mathbb{Z}_{q-1}$
    for $i \in |D|$ do
        if $D_i = m$ then
            $s_i \leftarrow 0$
            $c_i \leftarrow 0$
            $y_i = g^t$
            $z_i = h^t$
        else
            $s_i \in_{random} \mathbb{Z}_{q-1}$
            $c_i \in_{random} \mathbb{Z}_{q-1}$
            $y_i \leftarrow g^{s_i} \cdot G^{-c_i}$
            $z_i \leftarrow h^{s_i} \cdot \left(\frac{H}{g}\right)^{-c_i}$
    c $\leftarrow Hash(sb, y_1, z_1, y_2, z_2)$
    $c_0 \leftarrow c - c_1 - c_2$
    $s_{D_m} \leftarrow c_0 \cdot r + t$
    $c_{D_m} \leftarrow c_0$
return $s_1, s_2, c_1, c_2$
```

Algorithm 2: Validation of a non-interactive ballot validity proof

```
Data: ElGamal public key parameters $p, g, h$, Ciphertext $E = (G, H)$, Domain $D$
      of valid messages $\{0, 1\}$, proof parameters $s_1, s_2, c_1, c_2$
Result: True, on a valid proof for the specified ciphertext. False otherwise.
begin
    $sb \leftarrow \|g\|h\|G\|H$
    for $i \in |D|$ do
        $y_i \leftarrow g^{s_i} \cdot G^{-c_i}$
        $z_i \leftarrow h^{s_i} \cdot \left(\frac{H}{g}\right)^{-c_i}$
    c $\leftarrow Hash(sb, c_1, c_2, y_1, z_1, y_2, z_2)$
return $c_1 + c_2 \equiv c$
```

Chapter 3

Architecture

The following sections describe the architecture of the *Provotum*[1] blockchain-enabled electronic voting system. The backend components provide various core functionalities to two different browser single page applications (SPA) which were developed using ReactJS [2] and interact with the private Proof-of-Authority Ethereum blockchain. In conclusion, *Provotum* provides a verifiable and auditable voting system while retaining privacy.

The main architecture is illustrated in Figure 3.1. At the top, the worker nodes are forming a private Proof-of-Authority (PoA) Ethereum network (cf. Section 3.2). Each node is running an instance of the *Go Ethereum* (`geth`) client[3]. These nodes are interconnected using the Ethereum Wire Protocol[4] and publish their Remote Procedure Call (RPC) interface to clients.

The backend service component manages the interaction with the smart contracts (cf. Section 3.3). It also provides a websocket and a RESTful interface for the communication with frontend clients. In order to enable an encrypted communication channel using Transport Layer Security (TLS) to the voting system, a reverse proxy is integrated on all nodes. For this task, the Apache HTTP Server[5] is used. It terminates all secure connections on the hosts and forwards packets to the running applications.

A similar setup is also provided on the mock identity provider. Its primary purpose is determined by serving requesting clients with the private key of a unique pre-allocated wallet.

Eventually, the two frontend SPA's are the last two components of *Provotum*. The admin frontend[6] serves the voting authorities with the possibility to set up a new vote, while the voter frontend[7] is served to the voter and allows him or her to submit his ballot.

---

[1] `https://provotum.github.io`

[2] `https://reactjs.org/`

[3] `https://geth.ethereum.org/`

[4] `https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol`

[5] `https://httpd.apache.org/`

[6] `https://github.com/provotum/admin`

[7] `https://github.com/provotum/frontend`

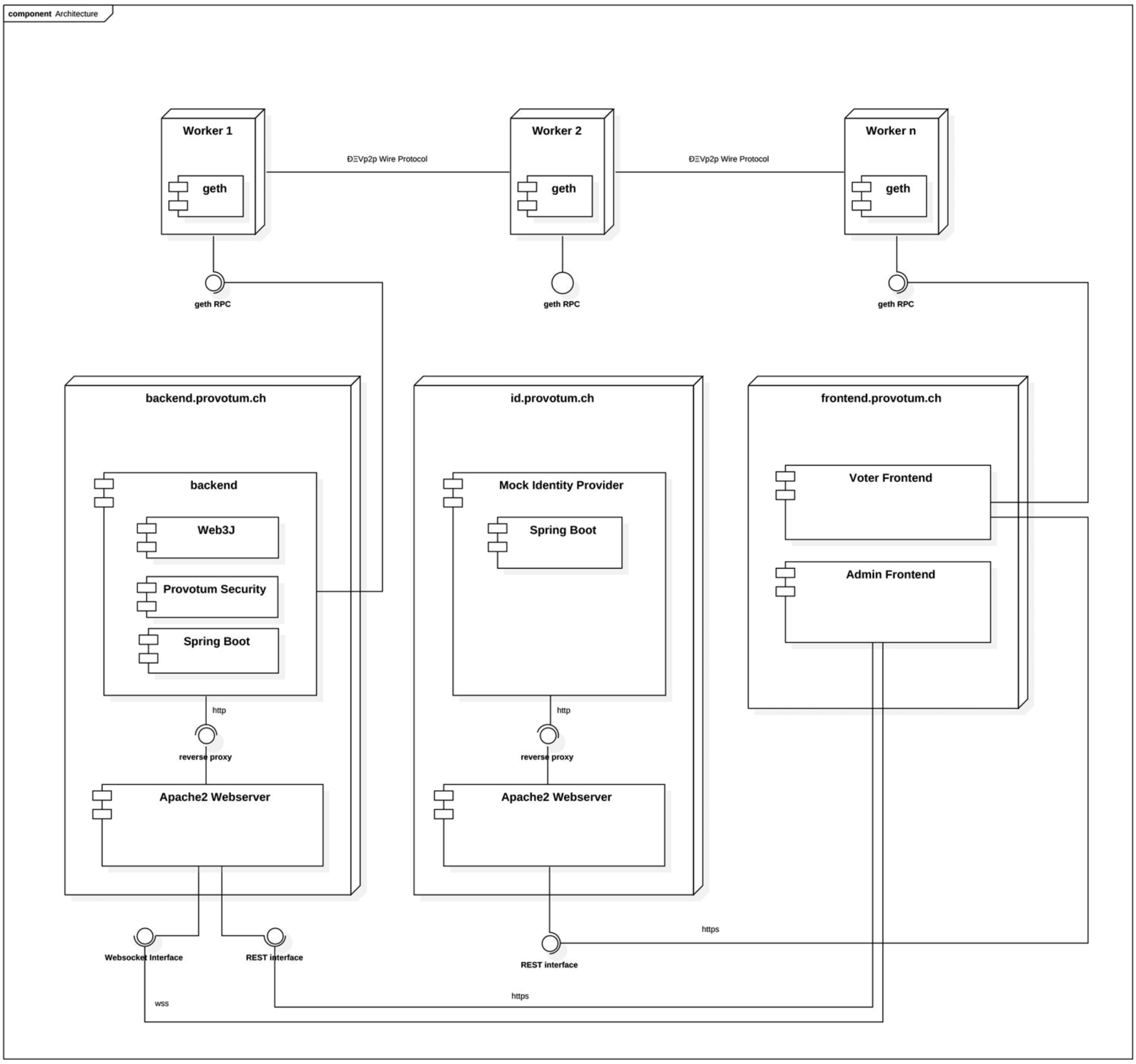


Figure 3.1: Systems Architecture

3.1 Stakeholders

In order to execute a successful voting, multiple stakeholders have to interact together. In the current *Provotum* system, there are three independent entities. First, the Voting Authority, which could be a canton or any other legal entity executing a voting. Second, the Mock Identity Provider, which is a placeholder for a real identity provider that can be replaced in future versions. The last stakeholder is an eligible voter that is voting on the subject matter.

3.1.1 Voting Authority

The voting authority (VA) is the main stakeholder of every vote. Before a new vote can be performed, a new private Proof-of-Authority (PoA) blockchain needs to be initialized and configured (cf. Section 3.2). The purpose of the VA is to initiate the voting by first setting up the vote through defining the question to vote on, then opening the ballot to accept incoming votes and finally closing the ballot. In the end, the VA is also responsible to initiate the evaluation process after the voting was closed. To finish the voting process, the VA publishes the results on the Public Bulletin Board (PBB): the private PoA blockchain.

3.1.2 Mock Identity Provider

In practise, an identity provider determines whether a particular person is eligible to vote, based on various criteria such as nationality or age. However, in this work, the sophistication of the identity provider is very limited. Thus, instead of effectively verifying that a voter is eligible to vote, the mock identity provider simply returns a valid authorization on any request. However, since this component is loosely coupled to all other parts of the voting system, it could be rather easily replaced by an identity provider verifying voters' eligibility.

3.1.3 Voter

The voter himself only has to submit his encrypted ballot to the PBB. At the end of the voting process, he is expected to verify that the voting result is correct.

3.2 Private Proof-of-Authority Blockchain

The main goal of a PBB is to provide a publicly verifiable log of communication [25]. Most voting schemes make use of a PBB to communicate the status of an election, its progress and the final tally.

Outlined in [26], the main properties of a secure PBB are:

- information cannot be removed or modified, only added to the board
- anyone may read the board
- the board provides a consistent view to everyone

Blockchains can offer a fully decentralized and tamper-proof infrastructure to execute, verify and store data [6]. Based on the previously mentioned properties of a secure PBB, blockchains form a viable foundation for electronic voting systems [35].

Ethereum is a general purpose public blockchain offering Turing-complete scripting languages [6]. Smart Contracts on Ethereum run in a sandboxed Ethereum Virtual Machine (EVM) and every operation in the EVM has to be paid for. This principle prohibits Denial-of-Service attacks on the public blockchain. In this work, the PBB is implemented as a decentralized private Proof-of-Authority blockchain, based on Ethereum [8]. Each member of the voting authority is in charge of running one or multiple Ethereum `geth` nodes [15]. For each election, a dedicated genesis block is generated, initialized and subsequently forms the starting point for a new private blockchain.

3.2.1 Proof-of-Authority

The Proof-of-Work (PoW) algorithm is the main consensus mechanism in Bitcoin and Ethereum [6]. A difficult crypto puzzle assures that double spending attacks are infeasible. The drawback of PoW is the huge amount of energy necessary to solve these crypto puzzles [6].

A public network without any value running the PoW protocol usually doesn't offer large enough computing barriers to fence off malicious attackers [34]. Because of the missing economic incentives, most miners mine in the live network. Therefore, PoA offers a viable alternative that doesn't require huge amounts of computing power.

The PoA protocol *Clique* is a very simplistic and effective protocol [34] for private blockchains. In PoW miners race against each other [8]. Contrary, in PoA backed networks authorized signers can create new blocks whenever it's their turn. The exact algorithm is outlined in section 3.2.2.

The main challenge is to set proper parameters to achieve a certain minting frequency, distribute minting load and how to dynamically adapt the list of the signers [34]. For our voting system, the clique PoA protocol makes it possible to pre-define all authorized signers, which in practise could be authorities and infrastructure that are controlled by government and election officials.

3.2.2 The genesis block

The genesis block is the first block of an Ethereum blockchain [44] and offers a wide array of configuration possibilities. The following list explains the most important parameters of the genesis block. Most parameters only apply if the `ALGORITHM` is set to `POA` and not

`POW`. These values can be adjusted in the setup component of this project. A shortened example of such a genesis block is outlined in Listing 3.1.

`GENESIS_CONFIG_CHAINID` defines the `chainId` field (e.g. the mainnet `chainId` is 1) and was introduced in EIP 155[8].For a private ethereum network it is suggested to use a rather unique `chainId` in order to mitigate networking issues with other chains and networks. Even if `geth` is started with the `no discover` flag which should prevent random peers from connecting, it is recommended to use something unique as `chainId`.

`GENESIS_CONFIG_CLIQUE_PERIOD` defines the `period` which is the minimum difference between two consecutive block's timestamps. It is suggested to use 15s in order to remain analogous to the mainnet ethash target [34].

`GENESIS_CONFIG_CLIQUE_EPOCH` defines `epoch`, the number of blocks after which to checkpoint and reset the pending votes. Please note that votes in this context refer to the clique algorithm (i.e. to propose new signers) and not the electronic voting system proposed in this work. It is suggested to use 30000 in order to remain analogous to the mainnet ethash epoch [34].

`GENESIS_NONCE` defines the `nonce` field. In PoW, the nonce is a 64-bit hash proving, in combination with the mix-hash, that it satisfies the Block Header Validity and allows to verify that a block has really been mined [44]. However, this field (and also the `miner` field) is repurposed in PoA. During regular blocks, both fields are set to zero. But if a signer wishes to propose a change to all other authorized signers, it can set the `miner` field to the signer it likes to vote about and set the `nonce` to `0x0` or `0xfff...f` to vote for adding or removing the proposed signer [34].

`GENESIS_TIMESTAMP` defines the `timestamp` field and is a scalar value to the reasonable output of Unix `time()` function at the block inception and is used in a mechanism to achieve homeostasis in terms of time between blocks [44].

`GENESIS_EXTRADATA_TWO_SEALERS` is written into the `extradata` field of the genesis block. The PoA protocol extends this field to on additional 65 bytes with the purpose of adding secp256k1 miner signatures[9]. This allows other nodes to verify the list of authorized signers. Additionally, it makes the miner section in block headers obsolete because the address can be derived from the signature [34]. The extradata used in this work was generated using the `puppeth`[10] helper script and contains the signatures of the first two nodes, thus authorizing them to sign blocks.

`GENESIS_DIFFICULTY` defines the initial `difficulty`. With PoW, `difficulty` represents a scalar value corresponding to the difficulty level applied during the nonce discovering of this block. The higher the difficulty, the more calculations have to be performed to mine a valid block. The value of `difficulty` is used to adapt the generation time of a block, thus keeping the frequency within a target range. In

---

[8] `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md`

[9] `https://github.com/ethereum/go-ethereum/blob/master/consensus/clique/clique.go`

[10] `https://github.com/ethereum/go-ethereum/tree/master/cmd/puppeth`

PoA, the `difficulty` contains one of two constants that signify the quality of a chain [34].

In PoA `difficulty` equals either $\text{DIFF\_NOTURN} = 1$ or $\text{DIFF\_INTURN} = 2$, which both are constants. These constants signify whether they contain in-turn or out-of-turn signatures. For example, if difficulty of a block equals 2, the signer was in-turn, hence `difficulty` must equal `DIFF_NOTURN if` $\text{BLOCK\_NUMBER} \bmod \text{SIGNER\_COUNT} \neq \text{SIGNER\_INDEX}$ [34].

`GENESIS_GASLIMIT` defines the `gaslimit` which is a scalar value equal to the current chain-wide limit of Gas expenditure per block [44]. However, the gas limit is dynamic and changes per block, the initial value is only the starting point. Using the `targetgaslimit` flag, `geth` can be instructed to diverge towards a constant `gaslimit` across all blocks.

`GENESIS_MIXHASH` is a Keccak 256-bit hash of the entire parent block header and a Pointer to the parent block, thus effectively building the chain of blocks. Only in the genesis block, this value is set to 0 [44].

`GENESIS_COINBASE` is a 160-bit address where the mining rewards are collected. This field can also be anything for the genesis block [44].

`GENESIS_ALLOC_BALANCE` defines the amount of ether that is pre-allocated to all the accounts in the `alloc` tag [44].

```json
{
  "alloc": {
    "0x0016b3226a4613e547bc958d915684742906b95d": {
      "balance": "888888888888888888888888"
    }
  },
  "coinbase": "0x00000000000000000000[...]",
  "config": {
    "chainId": 187,
    "clique": {
      "epoch": 30000,
      "period": 15
    }
  },
  "difficulty": "0x1",
  "extradata": "0x766972657320696e206[...]",
  "gaslimit": "0x47b760",
  "mixhash": "0x000000000000000000000[...]",
  "nonce": "0x0",
  "timestamp": "0x5aa98ca1"
}
```

Listing 3.1: Shortened Example genesis.json

3.2.3 Remote Deployment

The `remote` branch[11] of the setup repository is dedicated to the deployment on a remote infrastructure and was tested on five nodes running with 1GB RAM and 1vCPU each. The nodes were running `Ubuntu 16.04.4 x64` and `geth 1.7.3-stable-4bb3c89d`.

Configuration

The setup consists of a few `bash` shell scripts and a small Javascript file. In order to configure the deployment, the `.env` file contains various parameters that will be passed along to the execution of the JavaScript code in `steps/01-deploy-poa-net/src/generateKeys.js`. The parameters in the bash script have to be changed directly at the top of `steps/01-deploy-poa-net/run.sh`.

Step 1: Installation

The script installs the npm project and all necessary dependencies. The most important dependency is `keyethereum`, which is necessary to easily generate Ethereum account. After successfully running the `install.sh` script, `setup.sh` can be started.

Step 2: Initiate Setup

After starting `setup.sh`, `steps/01-deploy-poa-net/src/generateKeys.js` is executed and generates the amount of `NUMBER_OF_KEYS` and writes the output into the `privatekeys.json` file. This JSON is then sent to the `MOCK_IDENTITY_PROVIDER`. Next, the `genesis.json` file is generated and all public account addresses from the corresponding accounts in `privatekeys.json` will be pre-allocated in the genesis block which is written to the disk.

Step 3: Preparation on remote nodes

In order to prepare all nodes for the new election, all running `geth` processes are killed. Also, the actual blockchain data (`chaindata`), old log files and the old genesis block are wiped from the nodes. Now the nodes are in a clean slate state to start a new election.

Step 4: Distributing data to remote nodes

In this step, multiple files are copied to the nodes via SCP. The files include the new genesis file, a boot key for the bootnode, a script that will start `geth` and a javascript file `identities.js` holding the public addresses of the accounts that are initialized on the nodes. Preloading these identities eases testing and assigning wallets.

---

[11] `https://github.com/provotum/setup/tree/remote`

Step 5: Initialize genesis block

Before `geth` can be started, the genesis block needs to be initialized on all nodes. Since the genesis block is the first block of any Ethereum blockchain, this assures a common starting point.

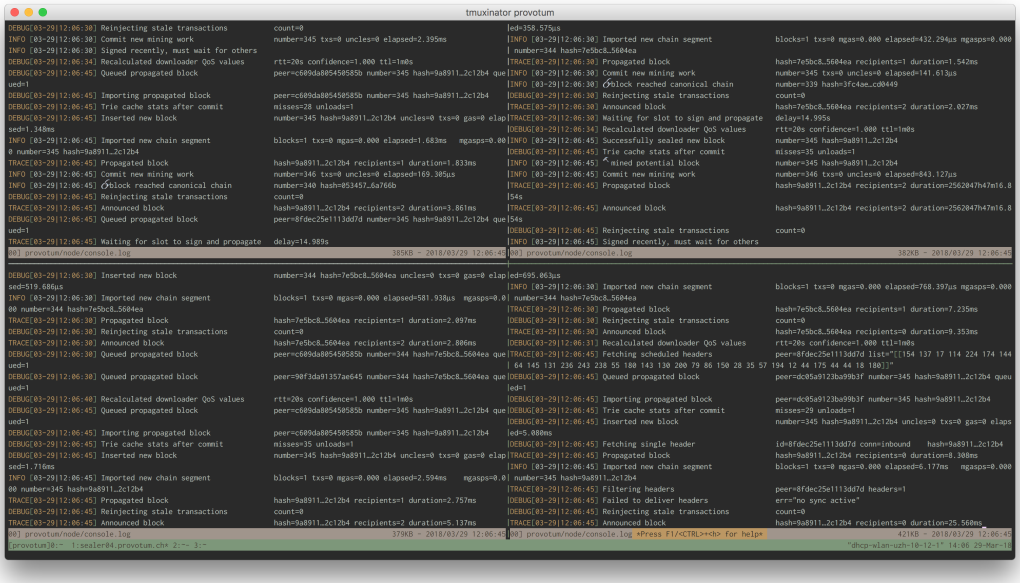
Step 6: Start geth

Next, `geth` is started on all five nodes. Initially, we start the first authorized node with the `mine` flag. After that, the script sleeps for a couple of seconds before starting the second node in order to let the first miner propagate his first mining work. Now the three other nodes are started exactly the same way, except they are not mining.

Step 7: Attach nodes

Right after step 6 when all `geth` nodes are running, we don't rely on the bootnode to connect all peers. Therefore, the nodes are attached using the `attach-nodes.sh` script which collects all `admin.nodeInfo` from the `geth` instances and adds them to each other executing `admin.addPeer("enode://1b2dad8...")` on every node. After that, the nodes will start to seal, accept and propagate blocks rather quickly and the private network is ready for the deployment of smart contracts.

Step 8: Monitoring



```
DEBUG[03-29|12:06:30] Reinjecting stale transactions        count=0
INFO [03-29|12:06:30] Commit new mining work                number=345 txs=0 uncles=0 elapsed=2.395ms
INFO [03-29|12:06:30] Signed recently, must wait for others
DEBUG[03-29|12:06:34] Recalculated downloader QoS values    rtt=20s confidence=1.000 ttl=1m0s
DEBUG[03-29|12:06:45] Queued propagated block               peer=c609da805450585b number=345 hash=9a8911…2c12b4 que
ued=1
DEBUG[03-29|12:06:45] Importing propagated block            peer=c609da805450585b number=345 hash=9a8911…2c12b4
DEBUG[03-29|12:06:45] Trie cache stats after commit         misses=28 unloads=1
DEBUG[03-29|12:06:45] Inserted new block                    number=345 hash=9a8911…2c12b4 uncles=0 txs=0 gas=0 elap
sed=1.348ms
INFO [03-29|12:06:45] Imported new chain segment            blocks=1 txs=0 mgas=0.000 elapsed=1.683ms   mgasps=0.00
0 number=345 hash=9a8911…2c12b4
TRACE[03-29|12:06:45] Propagated block                      hash=9a8911…2c12b4 recipients=1 duration=1.833ms
INFO [03-29|12:06:45] Commit new mining work                number=346 txs=0 uncles=0 elapsed=169.305µs
INFO [03-29|12:06:45] 🔗block reached canonical chain        number=340 hash=053457…6a766b
DEBUG[03-29|12:06:45] Reinjecting stale transactions        count=0
TRACE[03-29|12:06:45] Announced block                       hash=9a8911…2c12b4 recipients=2 duration=3.861ms
DEBUG[03-29|12:06:45] Queued propagated block               peer=8fdec25e1113dd7d number=345 hash=9a8911…2c12b4 que
ued=1
TRACE[03-29|12:06:45] Waiting for slot to sign and propagate   delay=14.989s
00] provotum/node/console.log                                                         385KB - 2018/03/29 12:06:45
```

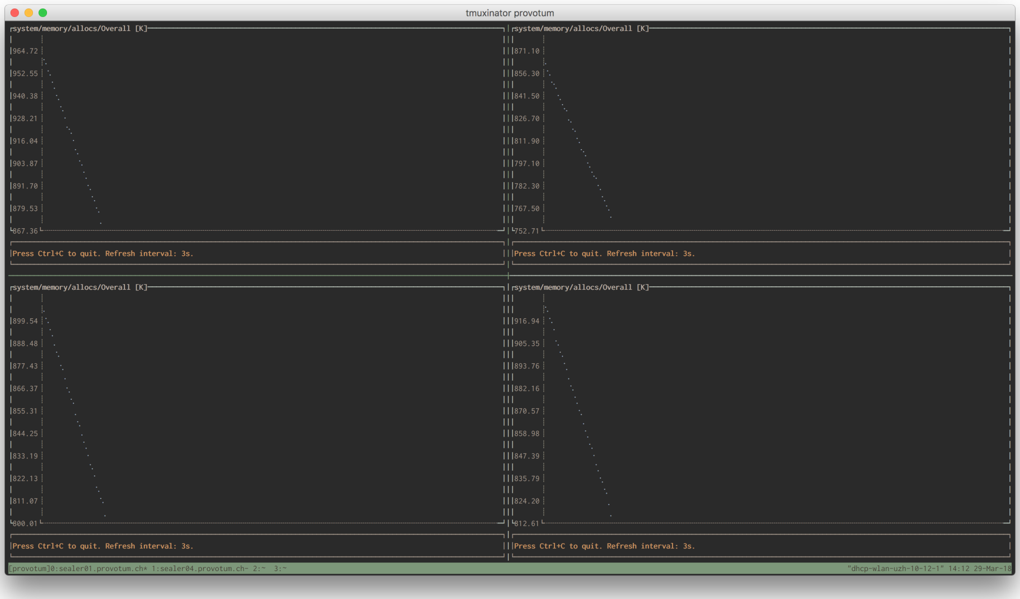
```
ed=358.575µs
INFO [03-29|12:06:30] Imported new chain segment            blocks=1 txs=0 mgas=0.000 elapsed=432.294µs mgasps=0.000
 number=344 hash=7e5bc8…5604ea
TRACE[03-29|12:06:30] Propagated block                      hash=7e5bc8…5604ea recipients=1 duration=1.542ms
INFO [03-29|12:06:30] Commit new mining work                number=345 txs=0 uncles=0 elapsed=141.613µs
INFO [03-29|12:06:30] 🔗block reached canonical chain        number=339 hash=3fc4ae…cd0449
DEBUG[03-29|12:06:30] Reinjecting stale transactions        count=0
TRACE[03-29|12:06:30] Announced block                       hash=7e5bc8…5604ea recipients=2 duration=2.027ms
TRACE[03-29|12:06:30] Waiting for slot to sign and propagate   delay=14.995s
DEBUG[03-29|12:06:34] Recalculated downloader QoS values    rtt=20s confidence=1.000 ttl=1m0s
INFO [03-29|12:06:45] Successfully sealed new block         number=345 hash=9a8911…2c12b4
DEBUG[03-29|12:06:45] Trie cache stats after commit         misses=35 unloads=1
INFO [03-29|12:06:45] 🔨 mined potential block               number=345 hash=9a8911…2c12b4
INFO [03-29|12:06:45] Commit new mining work                number=346 txs=0 uncles=0 elapsed=843.127µs
TRACE[03-29|12:06:45] Propagated block                      hash=9a8911…2c12b4 recipients=2 duration=2562047h47m16.8
54s
TRACE[03-29|12:06:45] Announced block                       hash=9a8911…2c12b4 recipients=2 duration=2562047h47m16.8
54s
DEBUG[03-29|12:06:45] Reinjecting stale transactions        count=0
INFO [03-29|12:06:45] Signed recently, must wait for others
00] provotum/node/console.log                                                          382KB - 2018/03/29 12:06:45
```

```
DEBUG[03-29|12:06:30] Inserted new block                    number=344 hash=7e5bc8…5604ea uncles=0 txs=0 gas=0 elap
sed=519.686µs
INFO [03-29|12:06:30] Imported new chain segment            blocks=1 txs=0 mgas=0.000 elapsed=581.938µs  mgasps=0.0
00 number=344 hash=7e5bc8…5604ea
TRACE[03-29|12:06:30] Propagated block                      hash=7e5bc8…5604ea recipients=1 duration=2.097ms
DEBUG[03-29|12:06:30] Reinjecting stale transactions        count=0
TRACE[03-29|12:06:30] Announced block                       hash=7e5bc8…5604ea recipients=2 duration=2.806ms
DEBUG[03-29|12:06:30] Queued propagated block               peer=c609da805450585b number=344 hash=7e5bc8…5604ea que
ued=1
DEBUG[03-29|12:06:30] Queued propagated block               peer=90f3da91357ae645 number=344 hash=7e5bc8…5604ea que
ued=1
DEBUG[03-29|12:06:40] Recalculated downloader QoS values    rtt=20s confidence=1.000 ttl=1m0s
DEBUG[03-29|12:06:45] Queued propagated block               peer=c609da805450585b number=345 hash=9a8911…2c12b4 que
ued=1
DEBUG[03-29|12:06:45] Importing propagated block            peer=c609da805450585b number=345 hash=9a8911…2c12b4
DEBUG[03-29|12:06:45] Trie cache stats after commit         misses=35 unloads=1
DEBUG[03-29|12:06:45] Inserted new block                    number=345 hash=9a8911…2c12b4 uncles=0 txs=0 gas=0 elap
sed=1.716ms
INFO [03-29|12:06:45] Imported new chain segment            blocks=1 txs=0 mgas=0.000 elapsed=2.594ms    mgasps=0.0
00 number=345 hash=9a8911…2c12b4
TRACE[03-29|12:06:45] Propagated block                      hash=9a8911…2c12b4 recipients=1 duration=2.757ms
DEBUG[03-29|12:06:45] Reinjecting stale transactions        count=0
TRACE[03-29|12:06:45] Announced block                       hash=9a8911…2c12b4 recipients=2 duration=5.137ms
00] provotum/node/console.log                                                         379KB - 2018/03/29 12:06:45
```

```
ed=695.063µs
INFO [03-29|12:06:30] Imported new chain segment            blocks=1 txs=0 mgas=0.000 elapsed=768.397µs mgasps=0.000
 number=344 hash=7e5bc8…5604ea
TRACE[03-29|12:06:30] Propagated block                      hash=7e5bc8…5604ea recipients=1 duration=7.235ms
DEBUG[03-29|12:06:30] Reinjecting stale transactions        count=0
TRACE[03-29|12:06:30] Announced block                       hash=7e5bc8…5604ea recipients=0 duration=9.353ms
DEBUG[03-29|12:06:31] Recalculated downloader QoS values    rtt=20s confidence=1.000 ttl=1m0s
TRACE[03-29|12:06:45] Fetching scheduled headers            peer=8fdec25e1113dd7d list="[[154 137 17 114 224 174 144
 64 145 131 236 243 238 55 180 143 130 200 79 86 150 28 35 57 194 12 44 175 44 44 18 180]]"
DEBUG[03-29|12:06:45] Queued propagated block               peer=dc05a9123ba99b3f number=345 hash=9a8911…2c12b4 queu
ed=1
DEBUG[03-29|12:06:45] Importing propagated block            peer=dc05a9123ba99b3f number=345 hash=9a8911…2c12b4
DEBUG[03-29|12:06:45] Trie cache stats after commit         misses=29 unloads=1
DEBUG[03-29|12:06:45] Inserted new block                    number=345 hash=9a8911…2c12b4 uncles=0 txs=0 gas=0 elaps
ed=5.080ms
DEBUG[03-29|12:06:45] Fetching single header                id=8fdec25e1113dd7d conn=inbound    hash=9a8911…2c12b4
TRACE[03-29|12:06:45] Propagated block                      hash=9a8911…2c12b4 recipients=0 duration=8.308ms
INFO [03-29|12:06:45] Imported new chain segment            blocks=1 txs=0 mgas=0.000 elapsed=6.177ms   mgasps=0.000
 number=345 hash=9a8911…2c12b4
TRACE[03-29|12:06:45] Filtering headers                     peer=8fdec25e1113dd7d headers=1
DEBUG[03-29|12:06:45] Failed to deliver headers             err="no sync active"
DEBUG[03-29|12:06:45] Reinjecting stale transactions        count=0
TRACE[03-29|12:06:45] Announced block                       hash=9a8911…2c12b4 recipients=0 duration=25.560ms
00] provotum/node/console.log *Press F1/<CTRL>+<h> for help*                           421KB - 2018/03/29 12:06:45
```

```
[provotum]0:~  1:sealer04.provotum.ch* 2:~- 3:~                                   "dhcp-wlan-uzh-10-12-1" 14:06 29-Mar-18
```

Figure 3.2: Tmuxinator tailing `geth` logs



```
tmuxinator provotum
system/memory/allocs/Overall [K]                system/memory/allocs/Overall [K]
Press Ctrl+C to quit. Refresh interval: 3s.     Press Ctrl+C to quit. Refresh interval: 3s.
system/memory/allocs/Overall [K]                system/memory/allocs/Overall [K]
Press Ctrl+C to quit. Refresh interval: 3s.     Press Ctrl+C to quit. Refresh interval: 3s.
[provotum]0:sealer01.provotum.ch* 1:sealer04.provotum.ch- 2:~  3:~      "dhcp-wlan-uzh-10-12-1" 14:12 29-Mar-18
```

Figure 3.3: Tmuxinator displaying `geth` metrics `system/memory/allocs/Overall`

Next, we can easily monitor the network using `tmuxinator`[12] and tailing `geth` logfiles as displayed in Figure 3.4. `Geth` also allows to collect and display metrics as displayed in Figure 3.3.

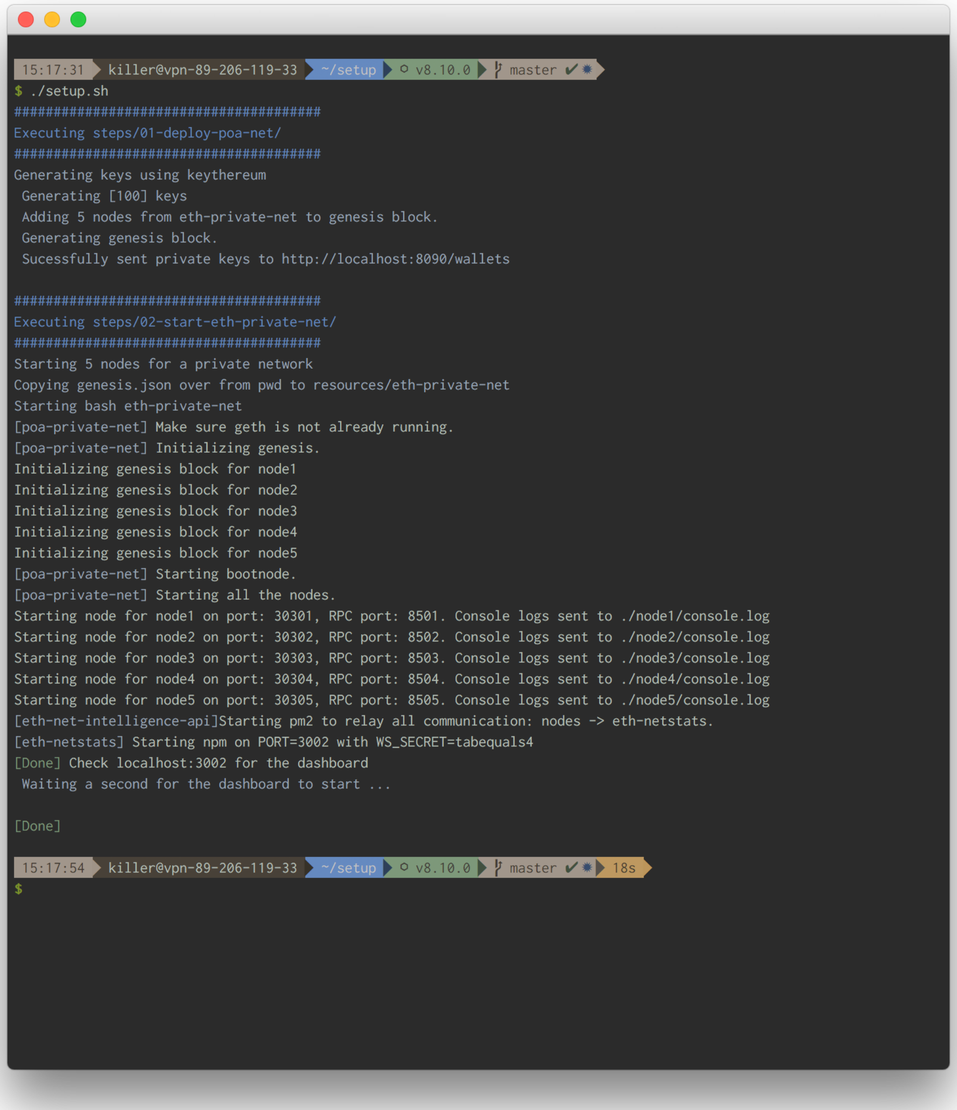
3.2.4 Local Deployment

Local deployment is rather straightforward, since all `geth` instances are running on `localhost` and can be easily observed with eth-netstats dashboard. The local deployment is only used for testing purposes and thus not further documented in this work. A detailed documentation is available on GitHub[13]. The steps are almost the same as with remote deployment. First run the `install.sh` script and start the `MOCK-IDENTITY-PROVIDER` on `localhost:8090`. After sucessfully running the script, you should be able to access the `eth-netstats` dashboard as displayed in Figure 3.5.

---

[12] `https://github.com/provotum/setup/blob/remote/provotum.yml`

[13] `https://github.com/provotum/setup/`



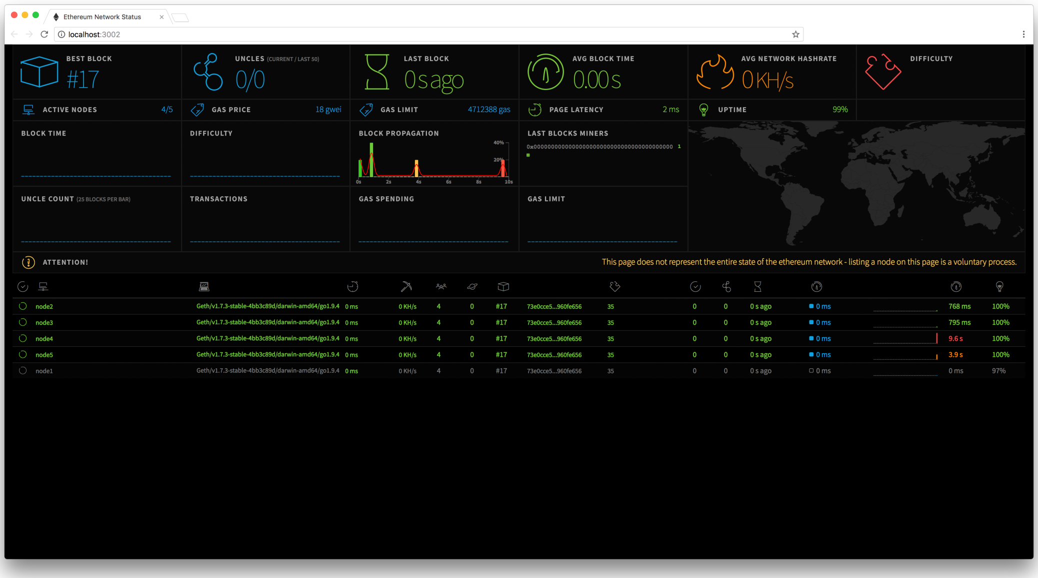
```
15:17:31  killer@vpn-89-206-119-33  ~/setup  ○ v8.10.0  master ✔ ✹
$ ./setup.sh
#######################################
Executing steps/01-deploy-poa-net/
#######################################
Generating keys using keythereum
 Generating [100] keys
 Adding 5 nodes from eth-private-net to genesis block.
 Generating genesis block.
 Sucessfully sent private keys to http://localhost:8090/wallets

#######################################
Executing steps/02-start-eth-private-net/
#######################################
Starting 5 nodes for a private network
Copying genesis.json over from pwd to resources/eth-private-net
Starting bash eth-private-net
[poa-private-net] Make sure geth is not already running.
[poa-private-net] Initializing genesis.
Initializing genesis block for node1
Initializing genesis block for node2
Initializing genesis block for node3
Initializing genesis block for node4
Initializing genesis block for node5
[poa-private-net] Starting bootnode.
[poa-private-net] Starting all the nodes.
Starting node for node1 on port: 30301, RPC port: 8501. Console logs sent to ./node1/console.log
Starting node for node2 on port: 30302, RPC port: 8502. Console logs sent to ./node2/console.log
Starting node for node3 on port: 30303, RPC port: 8503. Console logs sent to ./node3/console.log
Starting node for node4 on port: 30304, RPC port: 8504. Console logs sent to ./node4/console.log
Starting node for node5 on port: 30305, RPC port: 8505. Console logs sent to ./node5/console.log
[eth-net-intelligence-api]Starting pm2 to relay all communication: nodes -> eth-netstats.
[eth-netstats] Starting npm on PORT=3002 with WS_SECRET=tabequals4
[Done] Check localhost:3002 for the dashboard
 Waiting a second for the dashboard to start ...

[Done]

15:17:54  killer@vpn-89-206-119-33  ~/setup  ○ v8.10.0  master ✔ ✹  18s
$
```

Figure 3.4: Sucessfully finishing setup script



This page does not represent the entire state of the ethereum network - listing a node on this page is a voluntary process.

○	node2	Geth/v1.7.3-stable-4bb3c89d/darwin-amd64/go1.9.4	0 ms	0 KH/s	4	0	#17	73e0cce5...960fe656	35	0	0	0 s ago	■ 0 ms	768 ms	100%
○	node3	Geth/v1.7.3-stable-4bb3c89d/darwin-amd64/go1.9.4	0 ms	0 KH/s	4	0	#17	73e0cce5...960fe656	35	0	0	0 s ago	■ 0 ms	795 ms	100%
○	node4	Geth/v1.7.3-stable-4bb3c89d/darwin-amd64/go1.9.4	0 ms	0 KH/s	4	0	#17	73e0cce5...960fe656	35	0	0	0 s ago	■ 0 ms	9.6 s	100%
○	node5	Geth/v1.7.3-stable-4bb3c89d/darwin-amd64/go1.9.4	0 ms	0 KH/s	4	0	#17	73e0cce5...960fe656	35	0	0	0 s ago	■ 0 ms	3.9 s	100%
○	node1	Geth/v1.7.3-stable-4bb3c89d/darwin-amd64/go1.9.4	0 ms	0 KH/s	4	0	#17	73e0cce5...960fe656	35	0	0	0 s ago	□ 0 ms	0 ms	97%

Figure 3.5: Dashboard `eth-netstats`

3.3 Smart Contracts

In Ethereum, smart contracts represent a high-level abstraction of code statements which are compiled to Ethereum Virtual Machine (EVM) bytecode. This bytecode can be saved and then executed in order to transform the state of information stored on the blockchain. Most smart contracts are written in Solidity[14], which is a contract-oriented programming language having a syntax similar to JavaScript.

3.3.1 Zero-Knowledge Contract

The initial goal of this work was to execute as many computations as feasible on a decentralized infrastructure, i.e. within smart contracts. The Zero-Knowledge Contract should verify the encrypted votes. However, after having implemented the encryption with the corresponding proofs, technical limitations hindered the development of such a verification smart contract: On one hand, a library comparable to Java's `BigInteger` library is not available in Solidity. On the other hand, all computation during the encryption and proving are done within a cyclic field, requiring modular arithmetic for all common mathematical operations. Unfortunately the design and implementation of such a library would go beyond the scope of this project.

Therefore, the Zero-Knowledge contract is currently not accomplishing any verifications, but rather accepts all the votes it receives. Still, the contract is included in the actual voting process for later implementation. The actual validation of votes is performed in the Backend component instead (cf. Section 3.5).

---

[14] `https://github.com/ethereum/solidity`

3.3.2 Ballot Contract

The ballot contract keeps track of all submitted votes by using two structs shown in Listing 3.2. The `Voter` struct represents a single voter submitting his choice to the ballot contract. He is uniquely identified by his wallet address, i.e. the rightmost 20 bytes of the Keccak-256 hash (big endian) of the ECDSA public key [44]. Along with his address, the ciphertext encrypting the plaintext vote, the membership proof ensuring that the ciphertext represents a vote in the domain of the election as well as the encrypted random number used during encryption are stored.

The information of a running vote itself is represented by a struct called `Proposal`. It includes the current amount of voters which already have submitted a choice, a mapping of the address of a particular voter along whether he has already voted, the list of voters with their choices and concluding, a string storing the question on which all voters have to decide on.

Submitting a vote to the contract is as simple as sending a transaction to the method `vote`. The arguments are the ciphertext, the proof of the vote and the encrypted random number of the ciphertext. As shown in Listing 3.3, it is then verified that the submission of votes has been previously opened by the election authorities. Also, the voter is required to not have previously submitted a vote. Eventually, the proof is sent to the zero-knowledge contract where a validation is requested. On success, the voter and his choice is stored in the proposal. Since return values of methods are not retrievable by current client implementations, events[15] are fired at the same positions as the return statements, allowing a client to be notified about the state of a processed vote.

```
struct Voter {
    address voter;
    string ciphertext;
    string proof;
    bytes random;
}

struct Proposal {
    uint nrVoters;
    mapping(address => bool) voted;
    Voter[] voters;
    string question;
}
```

Listing 3.2: Structs in the Ballot contract

---

[15] http://solidity.readthedocs.io/en/v0.4.21/contracts.html#events

```
contract Ballot {
    address private _owner;
    bool private _votingIsOpen;
    Proposal private _proposal;
    ZeroKnowledgeVerificator _zkVerificator;

    /**
     * @param ciphertext   The ciphertext, i.e. a string
     *                     representing (G, H)
     * @param proof        The corresponding membership proof.
     * @param random       The random number used in the ciphertext.
     *                     Note, this value is encrypted.
     *
     * @return bool, string True if vote is accepted, false otherwise,
     *                      along with the reason why.
     */
    function vote(string ciphertext, string proof, bytes random)
                                external returns (bool, string) {
        // check whether voting is still allowed
        if (!_votingIsOpen) {
            return (false, "Voting is closed");
        }

        bool hasVoted = _proposal.voted[msg.sender];
        // disallow multiple votes
        if (hasVoted) {
            return (false, "Voter already voted");
        }

        bool validZkProof = _zkVerificator.verifyProof(proof);
        if (!validZkProof) {
            return (false, "Invalid zero knowledge proof");
        }

        _proposal.voted[msg.sender] = true;
        _proposal.voters.push(Voter({
            voter : msg.sender,
            ciphertext : ciphertext,
            proof : proof,
            random : random
        }));

        _proposal.nrVoters += 1;

        return (true, "Accepted vote");
    }
}
```

Listing 3.3: Voting in the Ballot contract

3.4 Security

The security component is a Java Maven artifact, packaging the required cryptographic methods for encrypting votes and creating corresponding membership proofs. Its implementation is based on a class providing similar functions as `BigInteger` but with support for modular arithmetic. This package provides three main interfaces:

IHomomorphicEncryption This interface specifies the signature for all implementations of a particular kind of homomorphic encryption. As a generic parameter, it requires the kind of ciphertext it operates on.

IHomomorphicCipherText Homomorphic cipher texts allow to operate on each other, abstracting the concrete mathematical details from the caller. It requires a concrete ciphertext as generic parameter.

IMembershipProof The interface for a membership proof requires a class implementing `IHomomorphicCiphertext` as generic parameter, restricting the classes it is able to generate proofs for.

As concrete implementations, this package bundles the additive variant of the ElGamal encryption among with the corresponding ciphertext (cf. Chapter 2.3). Additionally, a non-interactive membership proof following the Chaum-Perdersen protocol and made non-interactive using the Fiat-Shamir heuristic is implemented. Both implementations are partially based on the work from [27] and can be found on GitHub[16]. To avoid replicating security relevant functions to generate keypairs, this component utilizes BouncyCastle as security provider[17]. To ensure a consistent naming for all relevant encryption parameters in the source code, wrapper objects for a private resp. public key have been implemented.

Further, to allow for serialization resp. deserialization of ciphertexts, appropriate utilization classes are implemented. In order to reduce the amount of arguments which have to be passed to a client as well as to reduce their size, all relevant numeric parameters are transformed to base 36, the maximum Java supports with its built-in mechanisms[18]. Then, their representation are concatenated with uppercase letters as delimiters to form a string of defined format (Note, that uppercase letters describe their alphabetic counterpart instead of variables):

Ciphertext is a concatenation of g, the first component of an additive ElGamal ciphertext; h its second component and m the modulus used during encryption:

$$Gg \| Mm \| Hh \| Mm$$

Membership Proof is a concatenation of p, the prime modulus used during the encryption and y_i, z_i, s_i, c_i the commitments, challenges and responses used in a non-interactive $\sum$-proof, respectively:

$$Pp \| Yy_1, Yy_2, ..., Yy_n \| Zz_1, Zz_2, ..., Zz_n \| Ss_1, Ss_2, ..., Ss_n \| Cc_1, Cc_2, ..., Cc_n$$

---

[16] `https://github.com/FreeAndFair/evoting-systems/tree/master/EVTs/adder`

[17] `https://www.bouncycastle.org/java.html`

[18] `java.lang.Character.MAX_RADIX`

Private Key is a concatenation of p, the prime modulus used during encryption; q, the prime relative to p in the form of $p = 2q + 1$; g, the base generator of the cyclic group and x, the private key value:

$$Pp \| Qq \| Gg \| Xx$$

Public Key is a concatenation of p, the prime modulus used during encryption; q, the prime relative to p in the form of $p = 2q + 1$; g, the base generator of the cyclic group and h, the public key value $h = (g^x) \bmod p$:

$$Pp \| Qq \| Gg \| Hh$$

In the additive ElGamal variant (cf. Section 2.3.1), the publicly known parameters are included in the public key serialization specified above. In the presented system, a further value has to be published to the bulletin board: The random parameter r used during the encryption of the ciphertext $E(m) = (g^r, g^m \cdot h^r)$. This need initially roots in the way two ciphertexts are multiplied with each other: Although only the values of G_1, H_1 resp. G_2, H_2 are required to perform this calculation, one has to keep track of the sum of all random values $r_1 + r_2$ in order to generate a valid membership proof for the correct summation of the voting result. An approach not requiring to make the random value public involves publishing the private key x. However, this would allow all parties with access to the blockchain to decrypt single votes, removing a layer of privacy for the voter (Note, the vote is still not assignable to a particular person since the voter's wallet is assigned by the identity provider and not linked to a particular identity). Since r is generated during encryption of a plaintext vote and only reused after the final counting procedure, it can be encrypted and stored on the public bulletin board. In particular, these values are encrypted with a dedicated RSA private-public keypair owned by the voting authorities. Therefore, when trading off publishing the private key for making an encrypted random value public, the second approach is preferred.

3.5 Backend

The backend application is built on top of the Spring Boot[19] Maven artifact, and provides two main interfaces for interaction: Whereas a RESTful interface allows to initiate different actions on the Ethereum blockchain, websocket clients are notified about events published to particular topics[20].

The package **communication** holds all classes related to either the websocket or RESTful interaction. This includes the plain-old-java-objects (POJOs) which are serialized resp. deserialized using the Jackson JSON library[21]. Further, it holds the RESTful controllers based on a Spring `RestController` as well as a `TopicPublisher` service for publishing events to arbitrary websocket topics.

---

[19] `https://projects.spring.io/spring-boot/`

[20] For all available endpoints, see Appendix C

[21] `https://github.com/FasterXML/jackson`

For the interaction with the Ethereum blockchain, the backend component uses the Web3J[22] library. Besides of providing abstractions for RPC interfaces offered by an Ethereum node, Web3J allows to generate Java wrappers based on smart contract definitions written in Solidity. With these definitions, all publicly invocable methods of a smart contract are reflected by a corresponding method in a Java class, wrapping the smart contract on Ethereum.

As a last element, the security component is integrated: On application startup, a new ElGamal and RSA public-private keypair is generated if they do not already exist in a specified target directory. This assures that the voting authorities are protected from accidentally overwriting the voting parameters used for encryption, hence hindering themselves from ever decrypting the ballot. After defning these parameters, a RESTful interface allows to encrypt plaintext votes and generate corresponding proofs. In addition, clients to this interface may further request a validation for a ciphertext obtained by invoking the appropriate endpoint with a corresponding membership proof.

3.6 Frontend

Two distinct browser single page applications allow for managing votes and submitting votes. Whereas the former is run by the voting authorities, the latter is used by the voter. Both applications offer a similar graphical user interface (GUI), always displaying the latest occurred actions in an `Event Log`.

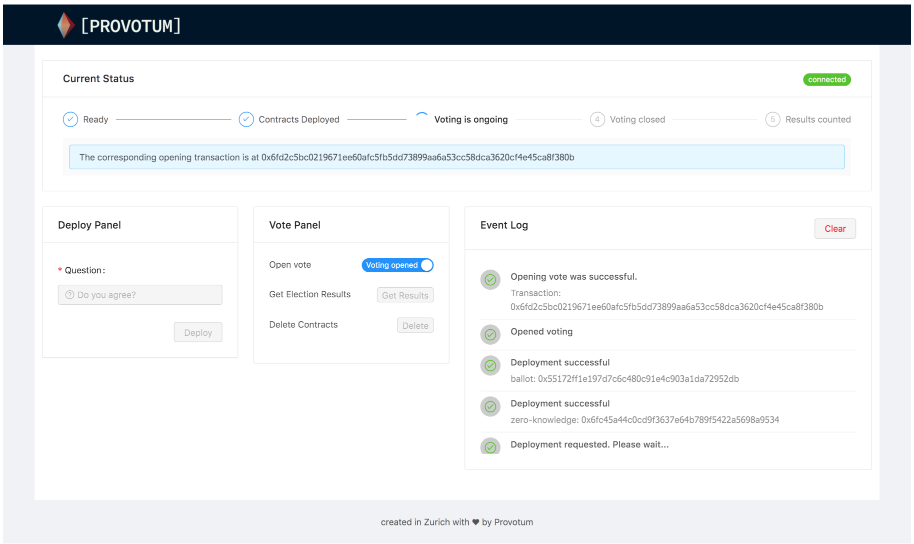
3.6.1 Administration Frontend

Once the administration frontend is fully retrieved from a web server, it automatically attempts to establish a websocket connection to the Java backend in order to subscribe the client application to updates published on particular topics. As illustrated in Figure 3.6, once the VA is connected to the backend, it may insert the voting question in the corresponding input field. Clicking on `Deploy` initiates the deployment of a new set of smart contracts on the private Ethereum blockchain. As soon as the two smart contracts are deployed, the new state is reflected on GUI. Then, the VA may open the vote for all participants with a transaction to the ballot contract, proxied by the backend. As soon as the corresponding event is shown in the Event Log, the VA can communicate the opening of the vote to all participants.

Once the voting period has been ended, the VA can initiate the formal closing on the smart contract. Clicking the slider button `Close Voting`, a transaction is sent to the ballot contract from the backend. When the transaction was mined, the votes can then be retrieved and the final result can be computed. Eventually, this is published to the ballot contract, allowing any outside observer to verify the voting outcome.

---

[22] `https://github.com/web3j/web3j`



The corresponding opening transaction is at 0x6fd2c5bc0219671ee60afc5fb5dd73899aa6a53cc58dca3620cf4e45ca8f380b

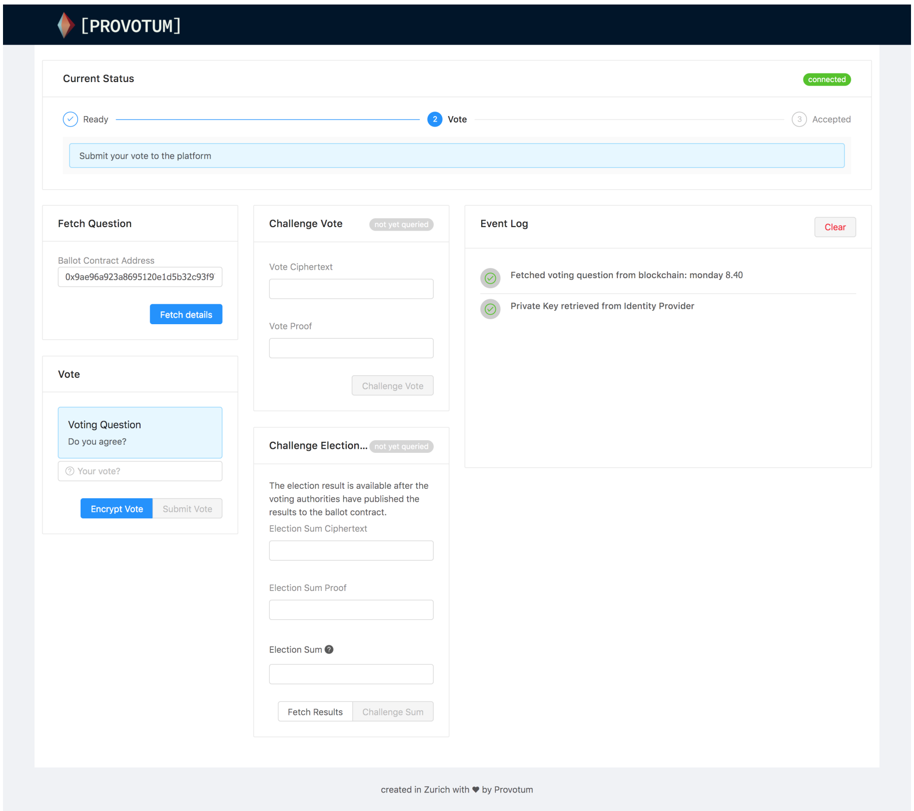
```
Opening vote was successful.
Transaction:
0x6fd2c5bc0219671ee60afc5fb5dd73899aa6a53cc58dca3620cf4e45ca8f380b
Opened voting
Deployment successful
ballot: 0x55172ff1e197d7c6c480c91e4c903a1da72952db
Deployment successful
zero-knowledge: 0x6fc45a44c0cd9f3637e64b789f5422a5698a9534
Deployment requested. Please wait...
```

Figure 3.6: Administration Frontend Application

3.6.2 Voter Frontend

Right away, the frontend retrieves the ballot contract address from the backend application. Further, it obtains a new private key of a pre-allocated wallet from the identity provider. As shown in Figure 3.7, the voter can request the voting question at hand directly from the ballot contract, being ensured that he will vote on the appropriate question. Having decided on the proposed question, he will type 0 for a vote indicating opposition to the question, 1 for support. By clicking on `Encrypt vote`, the plaintext representation is sent to the backend application where it is encrypted and a proof is generated. Subsequently, both values are then shown in the corresponding input fields. By clicking on `Challenge Vote`, each voter can verify that the encrypted vote is within the accepted boundaries of a valid vote, i.e. $\left[0, 1\right]$. By manipulating either the ciphertext or the vote and then re-submitting the verification request, the voter can increase his trust into the encrypted vote with each validation indicator matching the expected outcome. Once he is satisfied with the encrypted vote, he will submit the pair containing the ciphertext and the corresponding proof directly to the blockchain, by sending a raw transaction to the ballot contract address, initially obtained.

The parameters required by a raw transaction are shown in Listing 3.4. The nonce of a raw transaction is the amount of transactions which have been sent from the associated wallet and should not be confused with the nonce during the mining process which represents the running variable during generation of a valid proof of work. To allow a voter to send only a single transaction from the obtained wallet even if he would manipulate the disabled state of the `Submit Vote` button, the nonce is set to `0x00`. Any attempt to send a transaction again without adjusting the nonce will result in an error.
Whereas the gas price indicates the amount the voter is willing to pay for each operation caused by his transaction, the gas limit represents the amount the voter is willing to pay at maximum in order to get his transaction processed. [16]. To obtain a value which gets the transaction mined, the common approach is to use the amount of the last processed block.



Submit your vote to the platform

Ballot Contract Address 0x9ae96a923a8695120e1d5b32c93f9

Voting Question
Do you agree?

Your vote?

The election result is available after the voting authorities have published the results to the ballot contract.

Fetched voting question from blockchain: monday 8.40

Private Key retrieved from Identity Provider

created in Zurich with ♥ by Provotum

Figure 3.7: Voting Frontend Application

The receiver of the transaction is set to the address of the ballot contract initially retrieved from the backend component. The from address is then recreated based on the buffer holding the private key obtained from the Identity Provider.
In addition, the transaction data is retrieved from the Application Binary Interface (ABI) of the ballot contract. The estimated gas required to perform the actions on the ballot contract is then calculated by a utility function of web3js[23]. The chain id eventually specifies the identifier of the blockchain and must match the corresponding field of the genesis block.

```
const txParams = {
  nonce: "0x00",
  gasPrice: this.web3.toHex("22000000000"),
  gasLimit: this.web3.eth.getBlock("latest").gasLimit,
  to: contractAddress,
  from: ’0x’ + ethUtil.privateToAddress(privKeyBuffer).toString(’hex’),
  value: "0x0",
  data: rawTxData,
  gas: this.web3.eth.estimateGas({to: rawTxTo, data: rawTxData}),
  chainId: 187
};
```

Listing 3.4: Raw Transaction Parameters

After the vote has been closed by the voting authorities, the GUI also provides a mechanism in order to retrieve the final voting result. On the right-hand side in Figure 3.7, the

---

[23] https://github.com/ethereum/web3.js/

voter can obtain the sum of all votes. This sum is the plaintext value of the decrypted result of the homomorphic addition over all cipher texts and does therefore not represent whether the result is representing support or opposition to the initially asked question. This information can be recomputed by any party having access to the blockchain: First, by verifying all proofs of any cipher text, the verifying party will obtain v_I votes which were invalid. Secondly, challenging the proof published by the authorities will allow the voter to trust the total of yes votes v_Y. Then, the prover may obtain the v_T transactions to the publicly callable `vote` method. Eventually, the total number of opposing votes is then calculated by $v_T - v_Y - v_I$.

3.7 Mock Identity Provider

The final component required by the proposed voting system is represented by an Identity Provider issuing private keys to voters. Similarly to the Backend application, this provider is also set up as a Spring Boot application. It provides two RESTful interfaces allowing a party to store a set of unique private keys of an Ethereum wallet and a second interface, returning one of these keys to the requester. The exact requests are documented in Appendix D. Note, that no authentication and authorization whatsoever is required for invoking these endpoints as this is out of scope for this project.

3.8 Assumptions and Limitations

In the current revision, *Provotum* only allows simple votes showing support either *for* respective *against* a particular question. However, many votings can not be done with just a binary answer, requiring a set of possible valid choices. In [28], two further cases are described:

Multi-Way Elections represent votes where a voter has to select one out of a possible set of candidates. Since the homomorphic operation of ElGamal is computed over $(\mathbb{Z}_p)^*$, an appropriate encoding must be found. An approach described by [3] encodes each vote in a base-representation in the form of $B_{|V|}$ whereas $|V|$ is a number greater or equal to the amount of voters participating. Then, for C candidates, there are $|V|^{C-1}$ of different valid ballots.

Consider the example of four candidates with six voters outlined in Table 3.1. Performing the addition of all drawn ballots with respect to the base $|V|$ yields:

$$0001_{|V|} + 1000_{|V|} + 0100_{|V|} + 1000_{|V|} + 0010_{|V|} + 1000_{|V|} = 3111_{|V|}$$

The result then announces that three voters have voted for candidate C_4 and one for candidates C_1, C_2, C_3.

Limited Votes are elections where the voter can choose K out of L candidates. This approach was developed in [22] and is outlined in Table 3.2. Instead of summing up a

Voter	Ballot	Candidate
V_1	0001	C_1
V_2	1000	C_4
V_3	0100	C_3
V_4	1000	C_4
V_5	0010	C_2
V_6	1000	C_4

Table 3.1: Multi-Way Election

Voter	Ballot	Candidates
V_1	$\big(1,0,1,0\big)$	C_4, C_2
V_2	$\big(0,1,1,0\big)$	C_3, C_2
V_3	$\big(0,0,1,1\big)$	C_2, C_1
V_4	$\big(1,1,1,0\big)$	C_4, C_3, C_2
V_5	$\big(0,0,1,0\big)$	C_2
V_6	$\big(0,0,1,0\big)$	C_2

Table 3.2: Limited Vote Election

number to a particular modulus, the ballots are encoded as vectors of size $|C|$, i.e. having a coordinate for each candidate. Counting the election result is then performed component wise:

$$\big(1,0,1,0\big)+\big(0,1,1,0\big)+\big(0,0,1,1\big)+\big(1,1,1,0\big)+\big(0,0,1,0\big)+\big(0,0,1,0\big)=\big(2,2,6,1\big)$$

Candidate C_4 and C_3 will therefore receive 2 votes, C_1 one vote and candidate C_2 a total of six votes, winning the election.

Chapter 4

Voting

In the following chapter, the different voting processes are outlined in detail. Most of the steps are initiated and executed by the voting authorities (cf. Sections 4.1, 4.2, 4.4, 4.5). On the voter's side, only a single procedure must be executed (cf. Section 4.3).

4.1 Initiate a Vote

The first process is described in Figure 4.1: The voting authorities request the administration frontend SPA from a webserver. Then, the voting question has to be specified, which is sent in turn to the backend (cf. step 5). Next, the backend service deploys the zero-knowledge contract to the private Ethereum blockchain. If this succeeds, the ballot contract deployment is requested on the blockchain as well. Failed deployments are detected and returned to the frontend in steps 8, resp. 13, allowing the authorities to request a further deployment.

4.2 Open the Vote

To provide the voting authorities with the possibility to set up the vote prior to the start date, the ballot contract does not accept incoming votes by default. On the contrary, the voting authorities have to explicitly open the vote. This procedure is outlined in Figure 4.2a. It requires a single call to the ballot contract by the voting authorities, proxied by the backend service to the Ethereum blockchain.

4.3 Vote

The voting process itself does not involve any active involvement by the voting authorities once the voting has been opened. However, it requires that a third party is involved,

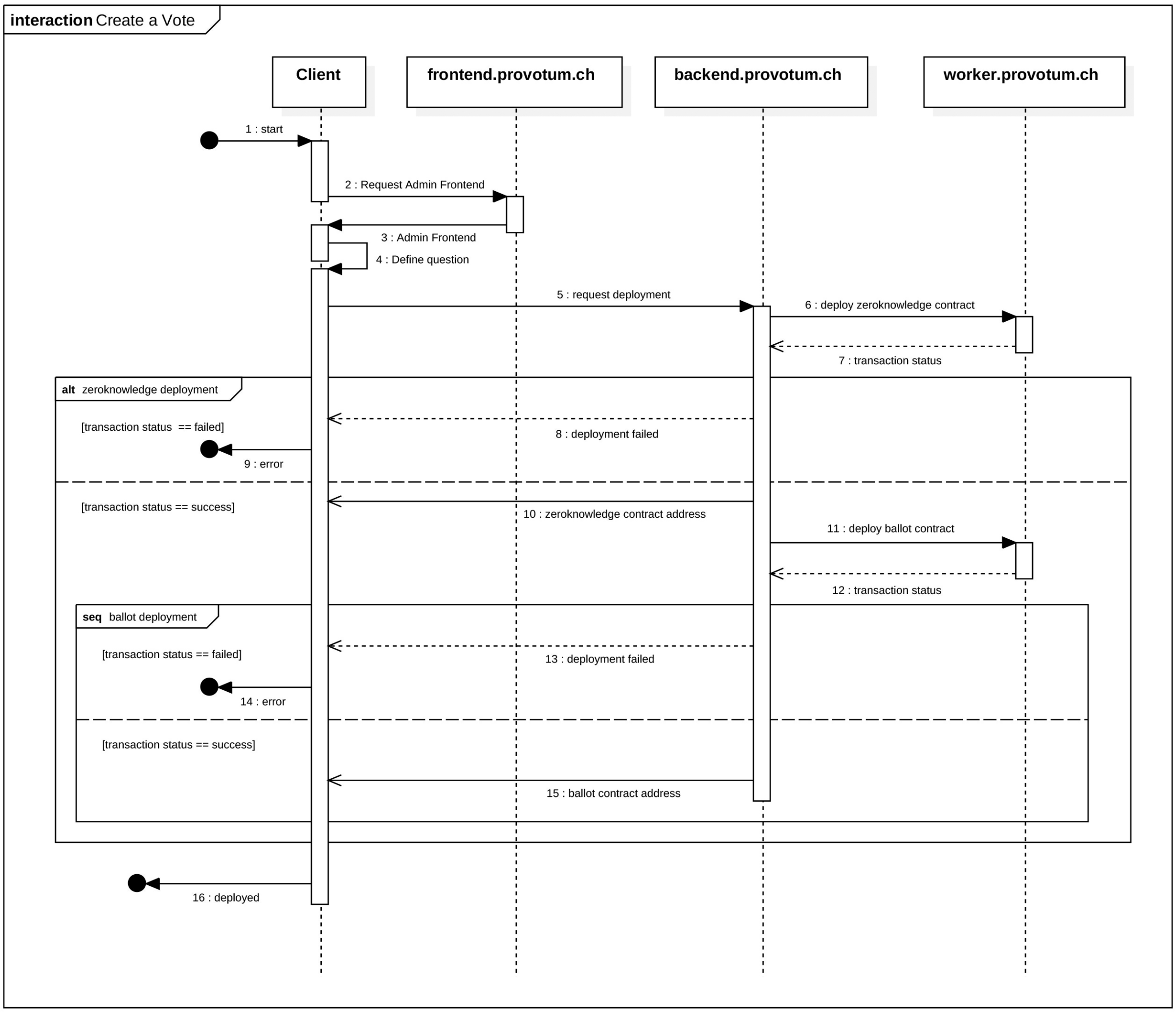
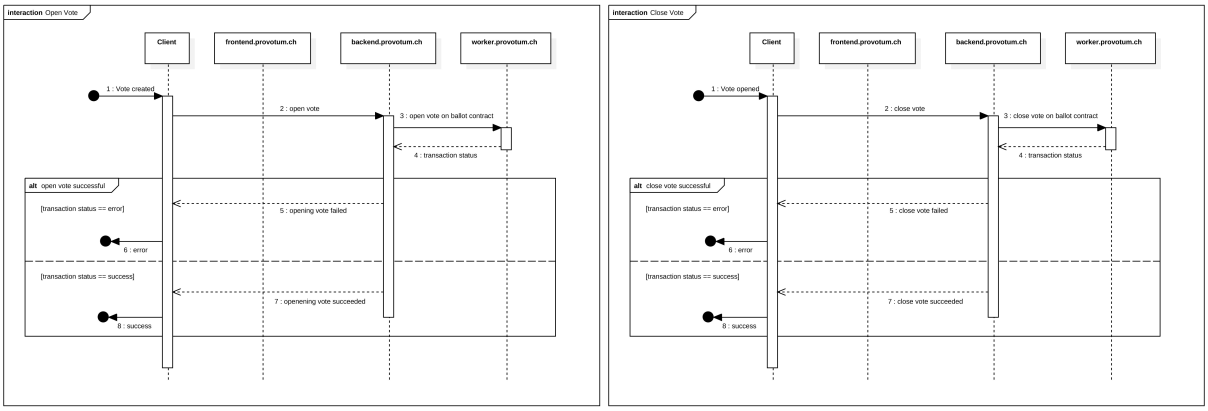

interaction Create a Vote
Client
frontend.provotum.ch
backend.provotum.ch
worker.provotum.ch
1 : start
2 : Request Admin Frontend
3 : Admin Frontend
4 : Define question
5 : request deployment
6 : deploy zeroknowledge contract
7 : transaction status
alt zeroknowledge deployment
[transaction status == failed]
8 : deployment failed
9 : error
[transaction status == success]
10 : zeroknowledge contract address
11 : deploy ballot contract
12 : transaction status
seq ballot deployment
[transaction status == failed]
13 : deployment failed
14 : error
[transaction status == success]
15 : ballot contract address
16 : deployed

Figure 4.1: Create a Vote



(a) Open the Vote

(b) Close the Vote

Figure 4.2: Opening resp. closing a Vote

authenticating the voters. In this work such an identity provider is stubbed. This means the mock identity provider is simply returning a unique private key of a pre-allocated wallet to a voter on each request. Having obtained the private key, the voter may now specify its choice for the question to vote on. Subsequently, this plaintext vote is sent to the backend (cf. step 9 in Figure 4.3) and returned in encrypted form, along with a membership proof for the range $[0, 1]$. The voter then has the possibility to manipulate the ciphertext, as well as the proof. Further, the voter can send these modified values to the backend again, obtaining information for the following cases depending on his undertaken manipulations:

Modified ciphertext, unmodified proof The backend will return a negative indicator stating that the modified ciphertext does not hold a vote in the range of $[0, 1]$ anymore, considering the modified ciphertext is still valid in terms of its deserialization format. Alternatively, the indicator may represent that the ciphertext does not follow its deserialization format anymore and the proof can not be verified.

Unmodified ciphertext, modified proof In case of an altered proof, the backend will return a negative indicator in all cases where the ciphertext does not encrypt a valid value (by means of what the modified proof expects). Alternatively, the indicator may return unsuccessfully if the proof was altered in a way such that it can not be deserialized anymore.

Modified ciphertext, modified proof If both values are modified, the backend will return a negative indicator whenever the ciphertext resp. the proof cannot be deserialized or the ciphertext does not match the proofs expected value. Further, the backend may return a positive indicator, if the proof and the ciphertext were altered in such a way that the expected value is indeed encrypted. Such a vote will eventually be invalidated during the counting procedure, as the proof must ensure a vote in the range $[0, 1]$.

Once the voter is satisfied with the ciphertext and proof returned by the backend (respectively his own alterations), he may submit his vote, independent of the indicator returned by the backend. Instead of relying on the backend for the connection to any Ethereum peer, the voter frontend itself sends a raw transaction to such a node.

4.4 Close the Vote

Similarly as described in Section 4.2, its counterpart process is represented by closing the vote once the voting period is over. The corresponding procedure is outlined in Figure 4.2b. The voting authority sends a close vote request to the backend which in turns submits a corresponding transaction to the Ethereum blockchain.

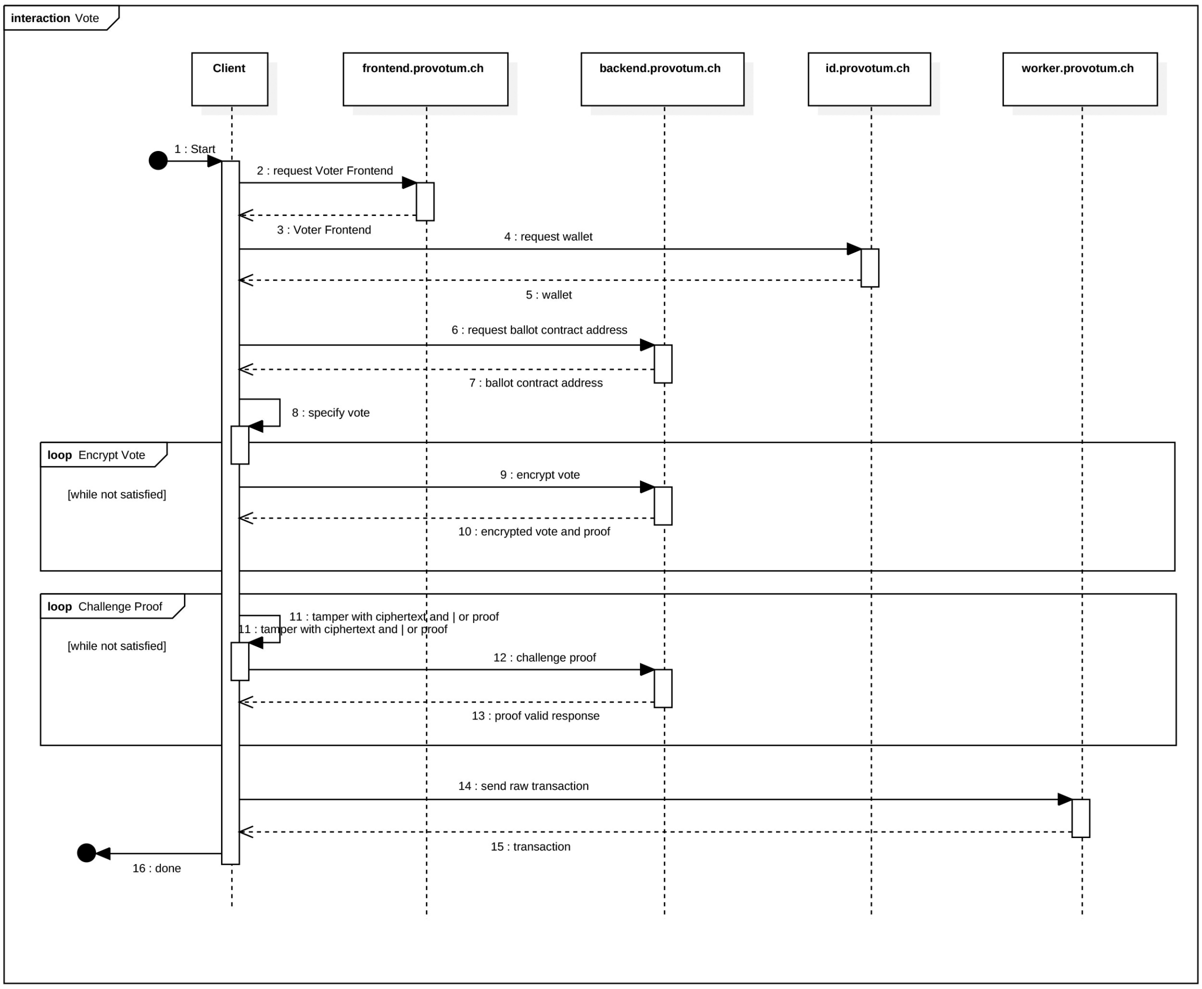


Figure 4.3: Vote

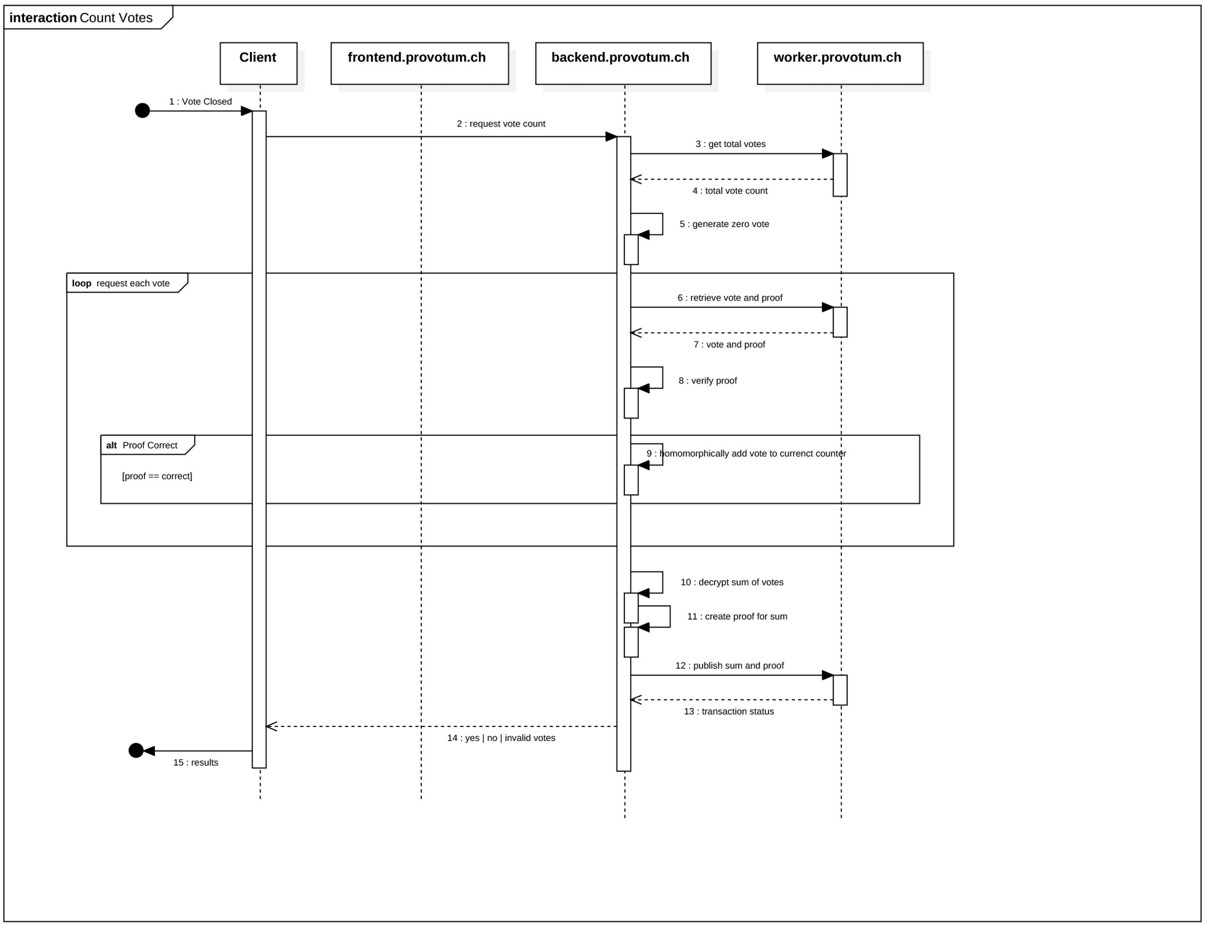


Figure 4.4: Count Votes

4.5 Count Votes

Once the voters have submitted their votes and the voting authorities have successfully closed the ballot contract from accepting any further votes, the counting process can be initiated. As shown in Figure 4.4, the voting authorities request the counting on the backend service. Since the ballot contract stores the received votes within a custom `struct` data type, the backend first retrieves the total number of votes in step three, before retrieving each vote (cf. step 6). In order to perform homomorphic addition (by performing a multiplication over the ciphertext), a zero vote has to be generated to which the first retrieved vote can be added to (cf. step 5). Before counting a vote to the total result, it is verified with the corresponding proof. Only on success, the vote is counted to either the supporting or the opposing vote count. If a vote is invalid, it is still counted to the total of retrieved votes. Once the sum of supporting votes has been determined, the sum, its corresponding ciphertext and a newly generated proof are published to the Ethereum blockchain. Eventually, the voting result is returned to the frontend.

Chapter 5

Evaluation

In [10], the perfect coexistence of verifiability and privacy is shown to be non-existent. Nevertheless, the notions of verifiability, auditability and privacy are properties that allow for a focused evaluation of any electronic voting system. Subsequent sections discuss these characteristics in more detail and show to what extent they are fulfilled. An overview is shown in Table 5.1.

5.1 Verifiability

With regards to verifiability, a set of different properties allow for a comprehensive comparison of systems:

Individual Verifiability (IV) Once a voter has submitted the encrypted ballot to the PBB (ref. Section 3.2), the voter will obtain a reference to his transaction, the transaction hash. As soon as one of the sealer nodes has also broadcasted the block that incorporates the transaction to all other nodes, the voter could explore the set of transactions sent to the ballot contract and reveal his particular transaction using the initially obtained reference. Therefore, this attribute is fulfilled by the proposed system.

Universal Verifiability (UV) Since all votes are transparently stored on the PBB, they are available to any third-party having an interest to validate the voting result. In order to reproduce the final outcome, the third-party would first fetch the sum of all votes v_Y along with its proof from the ballot contract. Once the proof is verified, the third-party would then retrieve all votes v_T sent to the ballot contract and their corresponding proofs. After having validated all ciphertexts, the amount of invalid votes v_I emerges. Finally, the number of votes rejecting the question can then be obtained by computing $v_T - v_Y - v_I$. Due to the ability of any third-party to verify these results, UV is also reached.

End-To-End Verifiability (E2E) In order to be classified as a fully E2E verifiable voting system, three attributes must be achieved:

		fulfilled
Individual Verifiability (IV)		●
Universal Verifiability (UV)		●
End-to-End Verifiability (E2E-V)		◕
	Cast-as-intended (CAI)	○
	Recorded-as-cast (RAC)	●
	Counted-as-recorded (CAR)	●
Auditability		●
Ballot-Privacy (BP)		●
Receipt-Freeness (RF)		●
Coercion-Resistance (CR)		○

● = fulfilled ◕ = partially fulfilled ○ = not fulfilled

Table 5.1: Overview of evaluation properties and their fulfillment.

cast-as-intended (CAI) Although the voting system allows to encrypt a particular plaintext vote multiple times, the backend component will always only return a proof that the ciphertext is within the range of $\left[0,1\right]$. Therefore, a voter cannot be ensured that a malicious backend component implementation would not just switch his vote to the inverse variant. Due to this, CAI verifiability is not fulfilled.

recorded-as-cast (RAC) Using the transaction hash (obtained once the vote has been submitted to the blockchain), the voter can retrieve the sent ciphertext c' and the corresponding proof p' at any given time. By also storing that data (c, p) before submitting the vote, he then can ensure that $c \overset{!}{=} c'$ and $p \overset{!}{=} p'$.

counted-as-recorded (CAR) or tallied-as-recorded (TAR) Similar to the auditing third-party, the voter can follow the same procedure to obtain the voting result as published by the voting authorities. In addition, by verifying that the voting transaction has been received by the ballot contract and the vote was accepted, the incorporation of his and all other votes is guaranteed.

5.2 Auditability

Proving that a vote is correct may not only be important during the time in which the voting process is running. Instead, any party should be able to verify that the announced outcome actually is true, even after the voting has been closed. In the research literature, no clear distinction between verifiability and auditability is made. However, [5] mentions Risk Limiting Audits (RLA), which purpose is to maintain a particular risk limit while not having to perform the same amount of work as posed by recounting the entire set of votes. Another approach is to examine an audit trail of an append-only datastructure. Since the outlined system heavily depends on the blockchain as its public bulletin board, such a data structure is given. Therefore, any third party can audit the voting result as long as the voting authorities maintain the set of nodes running the blockchain and access to them is provided.

5.3 Privacy

The privacy of a voter is an important key feature of any electronic voting system. Three main attributes can be distinguished according to [25]:

Ballot-Privacy (BP) The proposed system can guarantee BP under the assumption that the VA will not attempt to identify clients based on their IP address when connecting to the backend component: Since a voter will only contact this service to encrypt his vote and challenge the received proof but will not send any further data related to his identity (e.g. the public key of his wallet), no linkage between the the voter and the ballot can be made and the voting authorities will not know for whom a particular voter has voted.
In addition, an observer of the transaction sent from the voter directly to the ballot contract will not learn anything about his choice either since the vote is already encrypted with the public key of the voting authorities.
However, a more advanced attacker could also observe the communication between the voter and the backend component which would allow him to assign the ciphertext sent to the ballot contract to the plaintext vote. This issue can easily be mitigated by enforcing a secured connection.

Receipt-Freeness (RF) Once the voter has encrypted his vote, no information on his choice is available anymore: Only the ciphertext itself along with a corresponding proof that the vote is valid is available. Since each attempt of encrypting a vote is further requiring a new instance of a random parameter, a client would not even be able to prove to any third-party that the ciphertext returned from the backend component actually was a response to his encryption request, resulting in no receipt of his choice.

Coercion-Resistance (CR) Although BP and RF are guaranteed to a certain extent, CR is not achieved: If a coercer is at the same physical location as the voter, the entire voting process is observable and therefore, the voter can illustrate which choice he had supported. Since the wallet address of the voter is stored on the ballot contract to avoid duplicate submissions, he will not be able to submit a new decision during a later stage in the vote.

Further, the notion of *everlasting privacy* introduced by [32] requires no assumptions of cryptographic hardness for achieving privacy. However, the security of the ElGamal encryption depends on the discrete logarithm problem, i.e. finding x so that $h = g^x$. As stated in [39], this problem can be solved in polynomial time using Shor's algorithm with an appropriate quantum computer. Therefore, the presented system does not fulfill everlasting privacy.

Chapter 6

Summary and Conclusion

This work examined the use of distributed ledgers in the field of electronic voting. While prior research focused on the detailed design of cryptographic schemes and well-defined properties, this work aimed at a practical implementation of a blockchain-enabled electronic voting system. We showed that distributed ledgers offer great use for a transparent, tamper-proof and fully decentralized public bulletin board. However, our evaluation further expressed that the usage of blockchains can also introduce new security and privacy issues, expressing the inherent conflict between privacy and verifiability. More precisely, technical limitations in the cryptographical capabilities of Solidity have let to unfavourable architectural decisions. In the current implementation, the properties cast-as-intended and coercion-resistance could unfortunately not be achieved. With respect to the Swiss regulatory framework, our system would therefore not be allowed to any nation-wide votes. With these downsides in mind, the current architecture of the *Provotum* system still offers room for improvements.

6.1 Future Work

Although the current *Provotum* system enables electronic voting with a private blockchain as public bulletin board, not all initial objectives were achieved. Working towards a system providing E2E-V, many areas offer room for improvement. This section enumerates possible focus points for future work on the *Provotum* system.

Cryptography

Initially, most of the verification process was thought to be executed in a smart contract. Limitations in the available libraries providing the necessary mathematical operations were not available: One the one hand a library providing mathematical operations on numbers greater than 2^{256}, on the other hand an implementation of all common modular operations on these. Designing such libraries in Solidity is a key feature to bring the verification process to Ethereum.

In addition, the current ElGamal implementation could be extended to make use of operations on elliptic curves instead of a finite cyclic field of integers. Similarly, the encryption of the random parameter of each ciphertext could be implemented with ECDSA instead of RSA. This would also require research for appropriate libraries in Solidity.

Cast-as-intended Verifiability

To allow the *Provotum* system to be considered for Swiss electoral processes, cast-as-intended verifiability must be provided. Instead of only generating a range proof for a particular vote, the Backend component could also provide the voter with two dedicated proofs ensuring a supporting or opposing vote. These proofs may however not be stored on the blockchain, as they would allow any party to verify which sender has voted for which option. Therefore, a solution ensuring the current privacy level while maintaining cast-as-intended verifiability could be designed and integrated.

Identity Provider

Thirdly, the current implementation only uses a mock identity provider to assign wallets to arbitrary voters. In a real-world electronic voting system, only eligible voters should be able to submit votes. With the latest discussions on ERC 275[1], it may be feasible to implement this functionality on Ethereum and integrate it to the *Provotum* system.

Elections instead of Votes

The current implementation also only supports for simple yes/no questions. As outlined in Section 3.8, further work could focus on implementing Multi-Way Elections and Limited Votes. Having elections already in mind, strategies for majority and proportional elections could be developed.

User-Interface and User-Experience

Eventually, the end-user should not be left out of the equation. A good user experience and a well-designed user interface are important factors for adaption and acceptance of the electronic voting channel in society. The current implementation of the *Provotum* system offers room for improvement: To make testing easier, simply reloading the browser clears any state information in all the frontend SPAs. Persisting state information could be a starting point for improvements. Additionally, a voter must be instructed on how he can verify and challenge proofs sent by the voting authority for his encrypted vote. A refined guidance of the voter through all these steps may be a further key point.

---

[1] `https://github.com/ethereum/EIPs/issues/725`

Bibliography

[1] Acar, A., Aksu, H., Uluagac, A. S., and Conti, M. (2017). A survey on homomorphic encryption schemes: Theory and implementation. *CoRR*, abs/1704.03578.

[2] Ali, S. T. and Murray, J. (2016). An overview of end-to-end verifiable voting systems. *CoRR*, abs/1605.08554.

[3] Baudron, O., Fouque, P.-A., Pointcheval, D., Stern, J., and Poupard, G. (2001). Practical multi-candidate election system. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 274–283. ACM.

[4] Benaloh, J., Bernhard, M., Halderman, J. A., Rivest, R. L., Ryan, P. Y. A., Stark, P. B., Teague, V., Vora, P. L., and Wallach, D. S. (2017). Public evidence from secret ballots. *CoRR*, abs/1707.08619.

[5] Benaloh, J., Rivest, R. L., Ryan, P. Y. A., Stark, P. B., Teague, V., and Vora, P. L. (2015). End-to-end verifiability. *CoRR*, abs/1504.03778.

[6] Bocek, T. and Stiller, B. (2018). *Smart Contracts – Blockchains in the Wings*, pages 169–184. Springer Berlin Heidelberg, Berlin, Heidelberg.

[7] Brandon Carter, Ken Leidal, D. N. Z. N. (2016). Survey of fully verifiable voting cryptoschemes.

[8] Buterin, V. (2014). Ethereum: A next-generation smart contract and decentralized application platform. `https://github.com/ethereum/wiki/wiki/White-Paper`. Accessed: 2018-03-23.

[9] Canton of Geneva. E-voting system - CHVote. `https://republique-et-canton-de-geneve.github.io`. Accessed: 2018-03-23.

[10] Chevallier-Mames, B., Fouque, P. A., Pointcheval, D., Stern, J., and Traoré, J. (2010). On some incompatible properties of voting schemes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6000 LNCS:191–199.

[11] Cramer, R., Gennaro, R., and Schoenmakers, B. (1997). A secure and optimally efficient multi- authority election scheme. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1233:103–118.

[12] Damgard, I. (2002). On Σ-protocols. *Lecture notes for CPT*.

[13] Die Schweizerische Bundeskanzlei (vom 13. Dezember 2013 (Stand am 15. Januar 2014)). Verordnung der BK über die elektronische Stimmabgabe (VEleS). `https://www.admin.ch/opc/de/classified-compilation/20132343/201401150000/161.116.pdf`. Accessed: 2018-03-23.

[14] ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472.

[15] Ethereum (2017). Geth. `https://github.com/ethereum/go-ethereum/wiki/geth`. Accessed: 2018-03-23.

[16] Ethereum (2018). Ethereum - design rationale. `https://github.com/ethereum/wiki/wiki/Design-Rationale#gas-and-fees`. Accessed: 2018-03-21.

[17] Fiat, A. and Shamir, A. (1987). How to prove yourself: Practical solutions to identification and signature problems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 263 LNCS:186–194.

[18] Fontaine, C. and Galand, F. (2007). A survey of homomorphic encryption for non-specialists. *EURASIP J. Inf. Secur.*, 2007:15:1–15:15.

[19] Galindo, D., Guasch, S., and Puiggalí, J. (2015). 2015 neuchâtel's cast-as-intended verification mechanism. In *Proceedings of the 5th International Conference on E-Voting and Identity - Volume 9269*, VoteID 2015, pages 3–18, New York, NY, USA. Springer-Verlag New York, Inc.

[20] Giampiero Beroggi et al (2011). Evaluation der E-Voting Testphase im Kanton Zürich 2008-2011. `https://www.zh.ch/bin/ktzh/rrb/beschluss.pdf?rrbNr=1391&name=Evaluation_E-Voting_Z%C3%BCrich&year=2011&_charset_=UTF-8`. Accessed: 2018-03-30.

[21] Goldreich, O., Micali, S., and Wigderson, A. (1986). Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 174–187.

[22] Hirt, M. (2010). Receipt-free K-out-of-L voting based on ElGamal encryption. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6000 LNCS, pages 64–82.

[23] Im Namen des Schweizerischen Bundesrates, Der Bundespräsident: Moritz Leuenberger, Die Bundeskanzlerin: Annemarie Huber-Hotz (2006). Bericht über die Pilotprojekte zum Vote Électronique. `https://www.admin.ch/opc/de/federal-gazette/2006/5459.pdf`.

[24] Joaquim, R. and Ribeiro, C. (2012). An efficient and highly sound voter verification technique and its implementation. In Kiayias, A. and Lipmaa, H., editors, *E-Voting and Identity*, pages 104–121, Berlin, Heidelberg. Springer Berlin Heidelberg.

[25] Jonker, H., Mauw, S., and Pang, J. (2013). Survey. *Computer Science Review*, 10(Complete):1–30.

[26] Jonker, H. and Pang, J. (2011). Bulletin boards in voting systems: Modelling and measuring privacy. In *2011 Sixth International Conference on Availability, Reliability and Security*, pages 294–300.

[27] Kiayias, A., Korman, M., and Walluck, D. (2006). An internet voting system supporting user privacy. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 165–174.

[28] Korman, M. J. (2007). *Secret-Ballot Electronic Voting Procedures Over the Internet*. PhD thesis, University of Connecticut.

[29] Küsters, R. and Müller, J. (2017). Cryptographic security analysis of e-voting systems: Achievements, misconceptions, and limitations.

[30] McCorry, P., Shahandashti, S. F., and Hao, F. (2017). A smart contract for boardroom voting with maximum voter privacy. In *International Conference on Financial Cryptography and Data Security*, pages 357–375. Springer.

[31] McCranie, J. and Sloane, N. J. A. Numbers that have a primitive root (the multiplicative group modulo n is cyclic). `https://oeis.org/A033948`. Accessed: 2018-03-27.

[32] Moran, T. and Naor, M. (2006). Receipt-Free Universally-Verifiable Voting with Everlasting Privacy. *Advances in Cryptology - CRYPTO 2006: 26th Annual International Cryptology Conference. Proceedings*, 4117:373–392.

[33] NVotes (2016). Multiplicative vs additive homomorphic elgamal. `https://nvotes.com/multiplicative-vs-additive-homomorphic-elgamal/`. Accessed: 2018-04-02.

[34] Péter Szilágyi (2017). Clique poa protocol & rinkeby poa testnet. `https://github.com/ethereum/EIPs/issues/225`. Accessed: 2018-03-20.

[35] Riemann, R. and Grumbach, S. (2017). Distributed protocols at the rescue for trustworthy online voting. *CoRR*, abs/1705.04480.

[36] Rolf Haenni, Reto E. Koenig, P. L. E. D. (2017). Chvote system specification version 1.3.

[37] Sako, K. and Kilian, J. (1995). Receipt-free mix-type voting scheme. In Guillou, L. C. and Quisquater, J.-J., editors, *Advances in Cryptology — EUROCRYPT '95*, pages 393–403, Berlin, Heidelberg. Springer Berlin Heidelberg.

[38] Scytl Secure Electronic Voting, S.A. Customers - Canton of Neuchâtel. `https://www.scytl.com/en/customer/canton-of-neuchatel/`. Accessed: 2018-03-23.

[39] Shor, P. W. (1999). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM review*, 41(2):303–332.

[40] Smith, W. D. (2005). Cryptography meets voting.

[41] Statistisches Amt des Kantons Zürich (2018). E-Voting - Chronik. `https://wahlen-abstimmungen.zh.ch/internet/justiz_inneres/wahlen-abstimmungen/de/evoting/chronik.html`. Accessed: 2018-03-23.

[42] Swiss Post Ltd. (2015). Swiss Post promotes eVoting. `https://www.post.ch/en/about-us/company/media/press-releases/2015/swiss-post-promotes-evoting`. Accessed: 2018-03-23.

[43] Swiss Post Ltd. (2016). Swiss Post's e-voting solution. `https://www.post.ch/en/business/a-z-of-subjects/industry-solutions/swiss-post-e-voting`. Accessed: 2018-03-23.

[44] Wood, G. (2017). Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07). Accessed: 2018-03-23.

[45] Yi, X., Paulet, R., and Bertino, E. (2014). *Homomorphic Encryption*, pages 27–46. Springer International Publishing, Cham.

Abbreviations

BP	Ballot Privacy
CAI	Cast-as-intended Verifiability
CAR	Counted-as-recorded Verifiability
CR	Coercion-Resistance
E2E-V	End-to-End Verifiability
EVM	Ethereum Virtual Machine
IV	Individual Verifiability
PBB	Public Bulletin Board
PoA	Proof-of-Authority
POJO	Plain-old-Java-object
PoW	Proof-of-Work
RF	Receipt-Freeness
RAC	Recorded-as-cast Verifiability
SPA	Single Page Application
UV	Universal Verifiability
VA	Voting Authority
VEleS	Verordnung der BK über die elektronische Stimmabgabe
ZKP	Zero-Knowledge Proof of Knowledge

List of Figures

List of Tables

Appendix A

Installation Guidelines

In order to run the *Provotum* voting system, the following requirements must be met:

Frontend Applications require

- Node.js 9.4.0 or newer
- Node Package Manager (NPM) 5.6.0 or newer

Backend, Security and Mock Identity Provider require

- Java 8 or newer
- Apache Maven 3.5.0 or newer

Ethereum requires

- Go Ethereum Client (`geth`) in version 1.7.3
- Solidity Compiler (solc) in version 0.4.19
- Web3js in version 0.20.5
- Web3j in version 3.2.0

Then, you may use the contents of the attached CD to setup a local instance of the voting application. Note, that the steps below will only refer to the order on how you should set up all components. For detailed installation instructions, refer to the `README` in each of the projects.

- Install the Maven artifact of the Mock Identity provider on your system. Then run `mvn:spring-boot run` to start.
- Install the Maven artifact of the Security component on your local system using `mvn clean install`.
- Install the Maven artifact of the Backend component on your local system using `mvn clean install`.

- Run the local variant of the Setup script to start a set of sealer nodes with a predefined Genesis block. This will also send pre-allocated wallets to the Mock Identity Provider.
- Run the Backend component by executing `mvn:spring-boot run`.
- Start the voting authority frontend by invoking `npm start -s` from its root directory.
- Start the voter frontend by invoking `npm start -s` from its root directory.

Appendix B

Contents of the CD

The attached CD contains the following:

- A description of the CD contents
- All the source code for *Provotum*
- All *Related Work* papers as either PDF or HTML
- This report in source files, PDF and PS as well as all the contained figures in JPG, PNG and/or their sources files.

Appendix C

Backend Interface Specifications

C.1 RESTful interface specification

For requesting operations on the Blockchain, a RESTful interface is provided. The following subsections describe the available endpoints.

Note: Generally, there is no direct response to any request one makes to the RESTful API since operations on a blockchain may take an undefined amount of time. Therefore, the backend will process the incoming requests asynchronously and notify about results on a corresponding websocket topic.

On a successful request, all endpoints will return a HTTP status code of `202 Accepted` and an empty body, if not stated differently. The endpoints may return a HTTP status of `400 Bad Request` and an empty body if fields are missing or the request is malformed.

C.1.1 Zero-Knowledge Contract

Deploy Deploy a new zero-knowledge contract. The request should follow the form below with the appropriate host not set. The corresponding response will then be published to the Deployment Topic.

Request

```
POST /zero-knowledge/deploy HTTP/1.1
Content-Type: application/json; charset=utf-8
```

Remove Remove a zero-knowledge contract at a specific address. The corresponding response will be published to the Removal Topic.

Request

```
DELETE: /zero-knowledge/{contractAddress}/remove HTTP/1.1
Content-Type: application/json; charset=utf-8
```

C.1.2 Ballot Contract

Deploy Deploy a ballot contract. The zero-knowledge contract must be deployed previously as its address is required in the request shown below. Voters are not allowed to vote yet. The corresponding response will be published to the Deployment Topic.

Request

```
POST /ballot/deploy HTTP/1.1
Content-Type: application/json; charset=utf-8

{
  "election": {
    "question": "The question to vote on"
  },
  "addresses": {
    "zero-knowledge": "<zero-knowledge contract address>"
  }
}
```

Retrieve Ballot Contract Address Once deployed, third parties can fetch the latest contract address from this endpoint. Note, that each time a new ballot is deployed, the returned address is different and no history is provided. Note further, that the returned address may be null and a status code of `412 Precondition Failed`, is returned if the contract has not yet been deployed.

Request

```
GET /ballot/address HTTP/1.1
Content-Type: application/json; charset=utf-8
```

Response

```
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Connection: close

{
  "address":"a contract address"
}
```

Open Vote Allow voters to vote on a specific contract specified in the URL. The corresponding response will be published to the State Topic.

Request

```
POST /ballot/{contractAddress}/open-vote HTTP/1.1
Content-Type: application/json; charset=utf-8
```

Close Vote Close the ability of voters to vote on a specific ballot contract. The corresponding response will be published to the State Topic.

Request

```
POST /ballot/{contractAddress}/close-vote HTTP/1.1
Content-Type: application/json; charset=utf-8
```

Get Question Request the question from the specified ballot contract. The corresponding response will be published to the Meta Topic.

Request

```
POST /ballot/{contractAddress}/question HTTP/1.1
Content-Type: application/json; charset=utf-8
```

Get Results Request the results of the vote from the ballot contract specified. Note, that this will also publish the resulting sum to the ballot contract. The corresponding response will be published to the Meta Topic.

Request

```
POST /ballot/{contractAddress}/results HTTP/1.1
Content-Type: application/json; charset=utf-8
```

Remove Remove the contract at the given address. The corresponding response will be published to the Removal Topic.

Request

```
DELETE /ballot/{contractAddress}/remove HTTP/1.1
Content-Type: application/json; charset=utf-8
```

Encryption Encrypt the given vote and generate a corresponding proof, that the vote is within the boundary of a valid vote, i.e. $\left[0,1\right]$.

Request

```
POST /encryption/generate HTTP/1.1
Content-Type: application/json; charset=utf-8

{
    "vote": 1
}
```

The response will have a status code of 201 and contain the ciphertext as well as the corresponding proof in the body:

Response

```
HTTP/1.1 201
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Connection: close

{
  "ciphertext":"ciphertext",
  "proof":"proof",
  "random":"encrypted random value"
}
```

Verify a ciphertext The backend allows to verify a proof of a particular encryption, i.e. that the vote is within the boundary of a valid vote, i.e. $\left[0,1\right]$. Note, that this endpoint will only return valid responses for cipher texts and proofs which were generated with the same ElGamal public-private keypair as the backend is using at the time of the received request.

Request

```
POST /encryption/verify HTTP/1.1
Content-Type: application/json; charset=utf-8

{
  "ciphertext": "ciphertext",
  "proof": "proof"
}
```

If the proof is valid for the given ciphertext, then a status code of 200 is returned, 409 otherwise. The response body will be empty in all cases.

Response

```
HTTP/1.1 200
Content-Length: 0
Connection: close
```

C.2 Websocket interface specification

For listening for actions performed on the blockchain, the backend provides a websocket interface allowing clients to subscribe to different topics. The base path for all websocket connections is `websocket`. The topics described in the following are all assumed to be relative to this base path.

C.2.1 Topic Endpoints for RESTful intialized requests

The following subsections list events which are published as a response to a RESTful request.

Deployments The path for receving notifications about a successful resp. erroneous deployment is `/topic/deployments`. A message published on this topic follows the structure below:

```
{
  "id": "<UUID>",
  "responseType": "<ballot-deployed|zero-knowledge-deployed>",
  "status": "<success|error>",
  "contract": {
    "type": "<ballot|zero-knowledge>",
    "address": "<contract address>"
  },
  "message": "optional message, may be an empty string"
}
```

Note that the element address may be `null`, if the status equals to `error`.

Removals The path for receiving notifications about a removal of a contract is `/topic/removals`. A message published on this topic follows the structure:

```
{
  "id": "<UUID>",
  "responseType": "<ballot-removed|zero-knowledge-removed>",
  "status": "<success|error>",
  "transaction": "0x1234567890",
  "message": "optional message, may be an empty string"
}
```

Votes The topic to listen for events corresponding to votes is `/topic/votes`

```
{
  "id": "<UUID>",
  "responseType": "<vote",
  "status": "<success|error>",
  "transaction": "<sender addresss>",
  "message": "optional message, may be an empty string",
}
```

Opening resp. Closing Voting The topic to listen for events opening resp. closing the vote: `/topic/state`. A message will follow the format below:

```
{
  "id": "<UUID>",
  "responseType": "<open-vote|close-vote>",
  "status": "<success|error>",
  "transaction": "<sender addresss>",
  "message": "optional message, may be an empty string",
}
```

Meta A meta message usually includes information about the current state of a contract. Currently, the following events are emitted:

GetQuestionEvent: Contains the question of a ballot.

```
{
  "id": "<UUID>",
  "responseType": "get-question-event",
  "status": "<success|error>",
  "message": "optional message, may be an empty string",
  "question": "<question>"
}
```

GetResultsEvent: Contains the current votes of a ballot.

```
{
  "id": "<UUID>",
  "responseType": "get-results-event",
  "status": "<success|error>",
  "message": "optional message, may be an empty string",
  "votes": {
    "yes": 0,
    "no": 1,
    "total": 1,
    "invalid": 0
  }
}
```

C.2.2 Topic Endpoints for Blockchain Events

The following section describes events which happened on the blockchain and are forwarded to a topic.

Events The path for receving notifications about events is `/topic/events`. The following events may occur:

ChangeEvent: A event changing the current status of the Ballot contract. This event is usually emitted when the voting has been opened resp. closed on the contract or one of the contracts has been removed.

ProofEvent: An event indicating that a proof validation has taken place on the Zero-Knowledge Contract.

VoteEvent: The event on the blockchain, indicating that a vote has been processed.

In all cases, the event has the form:

```
{
   "id": "<UUID>",
   "responseType": "<vote-event|change-event|proof-event>",
   "status": "<success|error>",
   "senderAddress": "0x1234567890",
   "message": "optional message, may be an empty string"
 }
```

Appendix D

Identity Provider Interface Specification

The Identity Provider provides a RESTful interface for interacting. Its specification is outlined in the following section.

D.1 RESTful interface specification

Add Wallets In order to add new wallets (and remove the old ones), submit their private keys as shown in the request below.

Request

```
POST /wallets HTTP/1.1
Content-Type: application/json

{
    "wallets": [
        {
            "private-key": "<string>"
        },
        {
            "private-key": "<string>"
        },
        ...
    ]
}
```

Retrieve a Wallet In order to get a non-assigned wallet's private-key, use the following request:

Request

```
GET /wallets/next HTTP/1.1
Content-Type: application/json
```

Successful Response

```
{
  "private-key": "<string>"
}
```

Erroneous Response

```
{
  "timestamp": 1519138261748,
  "status": 404,
  "error": "Not Found",
  "exception": "org.provotum.mockidentityprovider.exception.
   NoWalletLeftException",
  "message": "No wallet left",
  "path": "/wallets/next"
}
```