BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PRIVACY PRESERVING SMART-CONTRACT PLATFORMS AND E-VOTING

SMART-CONTRACT PLATFORMY NA OCHRANU SÚKROMIA A ELEKTRONICKÉ HLASOVANIE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTOR **Bc. MAREK ŽIŠKA**
AUTOR PRÁCE

SUPERVISOR **Ing. MARTIN PEREŠÍNI**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

155970

Institut:	Department of Intelligent Systems (DITS)
Student:	**Žiška Marek, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Cybersecurity
Title:	**Privacy Preserving Smart-Contract Platforms and E-Voting**
Category:	Security
Academic year:	2023/24

Assignment:

1. Learn the principles of blockchains, smart contracts, electronic elections, and privacy-preserving platforms (PPPs).
2. Explore the performance and real privacy guarantees of privacy-preserving smart contract platforms for e-voting in the blockchain.
3. Study PPPs: Secret, Oasis, Integritee, Phala, and Hyperledger Fabric Private Chaincode, and choose 3-4 of them in which you will implement a simple e-voting application.
4. Implement simple e-voting (without any privacy) on chosen PPPs.
5. Evaluate theoretically and practically the performance in terms of transactions per second and the number of voters that can be processed.
6. Discuss the drawbacks, insights, and future improvements of e-voting within the given platforms.

Literature:

- Secret powering Web3 privacy; graypaper.
- Oasis smart privacy for dApps; docs.
- Integritee scalable and secure Web3.
- Phala network blockchain, pRuntime, and the bridge; paper.
- Hyperledger Fabric Private Chaincode, confidential chaincode execution using TEE (Intel SGX).
- Rust courses and tutorials, e.g., https://github.com/tweedegolf/101-rs in the case of some PPP that use Rust to develop smart contracts.
- Stančíková, Ivana, and Ivan Homoliak. "SBvote: Scalable Self-Tallying Blockchain-Based Voting." *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. 2023.

Requirements for the semestral defence:
1-3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Perešíni Martin, Ing.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	6.11.2023

Abstract

This work examines privacy-preserving platforms Secret, Phala, and Oasis Network, which provide frameworks for development of smart contracts with confidential storage and computation capabilities. We compare these platforms based on their features such as performance, usability, and additional factors within the context of an electronic voting use case. Firstly, we establish the theoretical foundations by introducing Voting Systems, then Blockchains, Smart Contracts, Trusted Computing, and Privacy-Preserving Platforms. We analyze the development capabilities, storage options, and other features of the selected platforms and propose the design of smart contracts for the e-voting application. Following this, we implement given smart contracts, detailing our experience, the tools used, testing procedures, contruct structure, and statistics collection methods. Proposed collected statistics allow us to estimate the vote-casting throughput of our implementations. Using this metric, along with other aspects, such as the development experience, storage options, community activity, documentation quality, we evaluate and compare these platforms. At the end we conclude the achieved results, key insights, reflections, and potentional areas for future improvements.

Abstrakt

Táto práca analyzuje platformy na ochranu súkromia ako Secret, Phala a Oasis Network, ktoré poskytujú nástroje pre vývoj smart kontraktov s možnosťou privátneho úložiska a dôverných výpočtových schopností. Platformy porovnávame na základe ich vlastností, ako je výkon, použiteľnosť a ďalšie faktory v kontexte prípadu použitia v elektronickom hlasovaní. Najprv predstavíme teoretické základy v oblasti volebných systémov, blockchainové technológie, smart kontrakty, technológie dôverného výpočtu a v neposlednom rade jednotlivé platformy na ochranu súkromia. Na základe zistení navrhneme volebný proces nášho smart kontraktu elektronického hlasovania, ktorý budeme implementovať na všetkých platformách. Následne analyzujeme možnosti a schopnosti vývoja každej z platforiem, najmä pokiaľ ide o definíciu štruktúry úložiska. Okrem toho navrhneme aj scenár hodnotenia, ktorý budeme vykonávať na každom z vyvinutých smart kontraktov. Po návrhu kľúčových častí smart kontraktov prechádzame k implementácií, kde diskutujeme o našich skúsenostiach, o použitých nástrojoch, o spôsobe zbere a vyhodnotenia štatistík a o metódach testovania. V rámci tejto fázy sme taktiež vyvinuli aj skripty, ktoré zbierajú štatistiky z navrhnutého scenára hodnotenia elektronického volebného systému, ktoré použijeme na vyhodnotenie a porovnanie týchto platforiem. Na záver zhrnieme dosiahnuté výsledky, kľúčové poznatky, úvahy a potenciálne oblasti pre budúce zlepšenia.

Keywords

blockchain, cryptography, smart-contracts, security, e-voting, Secret, Oasis, Phala, Web3, Solidity, Rust, performance analysis, gas usage

Klíčová slova

blockchain, kryptografia, smart-contract, bezpečnosť, elektronické hlasovanie, Secret, Oasis, Web3, Solidity, Rust, analýza výkonnosti, spotreba gasu

Reference

ŽIŠKA, Marek. *Privacy Preserving Smart-Contract Platforms and E-Voting.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Martin Perešíni

Rozšířený abstrakt

V tejto práci sa zaoberáme štúdiou volebných systémov, elektronického hlasovania, blockchainových technológií, schémam trusted computingu a blockchainovými platformami na ochranu súkromia. Blockchain je charakteristický svojou transparentnosťou, decentralizovanosťou, sledovateľnosťou a bezpečnosťou, čo sú esenciálne vlastnosti pre budovanie dôvery systémov. Jednou z oblastí kde sa môžu tieto technológie obzvlášť uplatniť sú systémy elektronického hlasovania. Dnes systémy elektronického hlasovania stále čelia problémom s legislatívnymi predpismi, hlavne pokiaľ ide o dôvernosť účastníkov hlasovania, integritu údajov, spoľahlivosť alebo tajnosť hlasovania. Hlasovacie systémy založené na blockchainových technológiách môžu vzhľadom na ich vlastnosti vyriešiť veľkú časť týchto výziev až na transparentnosť, a tajnosť hlasovania. Z tohoto dôvodu blockchainové realizácie elektronického hlasovanie často inkorporujú zložité kryptografické schémy ako *MPC*, *zk-SNARK*, dôkazy *NIZK* a iné s cielôm zaručiť privátnosť dát a vykonávaných operácií. Tieto schémy sa zvyčajne nasadzujú do smart kontraktov štandardných platforiem, ktoré ale majú limitované výpočetné schopnosti. Toto má za príčinu, že tieto metódy síce vyrieša problém s ochranou súkromia a privátnosti, ale zároveň zväčšia problém so škálovaním a nízkou rýchlosťou transakcií. Platformy na ochranu súkromia boli identifikované ako potenciálne riešenie tohto problému, pretože sa spoliehajú na dôverné a súkromné enklávy alebo *Trusted Execution Environments* namiesto komplikovaných a nákladných kryptografických schém.

Táto práca sa zameriava na ich výskume a na štúdiu konkrétnych platforiem ako *Secret*, *Oasis* a *Phala*, pre ktoré sme sa najmä zaoberali možnosťami vývoja smart kontraktov, pretože pre každú z nich sme implementovali jednoduchú aplikáciu elektronického hlasovania. Ako typ hlasovacieho systému sme zvolili jednoduchý variant jednočlenných pluralitných systémov, kde nešpecifikovaný počet voličov volí z viacero kandidátov, z ktorých na konci zvíťazí len jeden. Pri návrhu smart kontraktov na daných platformách sme sa zaoberali analýzou základných komponent používaných pri vývoju smart kontraktu, ako sú úložiskové štruktúry, zaužívané návrhové vzory, alebo mechanizmy na kontrolu prístupových práv k privátnym dátam. Tieto komponenty sa medzi platformami rapídne líšili, čo je spôsobené nie len tým, že sa používali rúzne programovacie jazyky (v našom prípade to boli *Solidity*, *Rust*, alebo *ink!*), ale hlavne na základe rôznych frameworkov, na ktorých sú tieto platformy postavené. Z týchto analýz sme navrhli podobu smart kontraktov, ktoré sme aj implementovali.

Implementované aplikácie boli použité vo fáze evaluácie, kde sme zbierali štatistiky z experimentov skladajúcich sa z určitých operácií počas volebného procesu (napr. odovzdávanie hlasov alebo výpočet výsledkov). Tieto štatistiky sa skladajú z údajov o spotrebe gasu pozorovanej operácie, ktorú sme spolu s parametrami platforiem, ako maximálny limit gasu v bloku a časový úsek potrebný na sfinalizovanie bloku, použili na výpočet transakčnej priepustnosti operácií za jednu sekundu. V našom vyhodnotení sme následne pracovali s voľbami, ktoré prijímajú hlasy počas dvojdňového časového okna. Na to aby sme zistili limity implementácie v tomto časovom okne, tak sme vypočítanú transakčnú priepustnosť vyextrapolovali do tohoto časového úseku. Tieto údaje o priepustnosti sme si vizualizovali do grafov, z ktorých sme vyhodnotili ktorá platforma má najlepšiu výkonnosť a najväčšie prevádzkové limity v aplikácií elektronického hlasovania.

Výsledkom tejto práce sú nové poznatky v platformách na ochranu súkromia, tri naimplementované aplikácie elektronického hlasovania, a výsledky spojené s ich výkonnosťou, z ktorých sme zistili že *Oasis* dokáže obslúžiť najväčší počet voličov, v tesnom závese je *Secret* a na poslednom mieste je *Phala*.

Privacy Preserving Smart-Contract Platforms and E-Voting

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Martin Perešíni. Supplementary information and constructive feedback was provided by Mr. Ivan Homoliak. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Marek Žiška
May 17, 2024

Acknowledgements

I would like to thank Ing. Martin Perešíni for leading me during this work, for all the helpful advice he provided me during our consultations. Additionally, I express my gratitude Ing. Ivan Homoliak Ph.D., for our consultations and for much-needed and constructive feedback. I want to mention my lovely laptop, the MacBook Pro 13" 2016 version, it was a love at first sight I still remember the day when my dad brought it home. I want to express my gratitude towards this faithful companion, as it unfortunately ceased to function during the writing of this thesis. It has been an exceptional partner, accompanying me not only through the completion of this thesis but also on countless projects throughout my bachelor's and master's journey. It (almost) never failed to deliver when the deadlines were the closest, may it rest well. You will be remembered forever. I also want to express my sincere gratitude to my brother Peter who supported me in the final hours of the thesis finalization, his help has not left an empty space in my heart ;) Last but not least I cannot forget to thank my family, friends and my lovely girlfriend for her unwavering support and companionship in all of our extensive study and work sessions, that at the end made most of this theses a reality.

Contents

Chapter 1

Introduction

Despite the fact that the popularity of blockchain and its associated technologies remains an up-and-down story, its meteoric rise over the past decade is an undisputable proof that millions of people recognized the vast potential this technology has. The main driving force behind this success is the benefits that blockchain is known for, such as transparency, increased efficiency, decentralization, traceability, and security, which are essential in building trust in the systems. These traits are desired in many fields that span a wide range of applications, ranging from financial transaction processing, healthcare records storage, identity management, or possibly even essential government systems such as electronic voting (e-voting) systems.

There is continuous research on the topic of e-voting systems from the perspective of improved security, increased voter turnout, and confidence. Although people have successfully incorporated Internet solutions into their lives, the Internet is still considered an unsafe place for a complex process such as elections [2]. Online e-voting systems still face problems with legislative regulations that are very strict with respect to the confidentiality of the voting participants and other challenges such as data integrity, reliability, transparency, the secrecy of the ballot, security, consequences of fraud, and others. Blockchain-enabled voting systems were proposed as a solution to most of these challenges, as their characteristics and immutable nature, can turn the blockchain into a role of a distributed ballot box [36]. In the real setting, blockchain-based voting systems have frequent problems with privacy protection, integrity, scaling, and slow transaction speeds. Several studies [16] [34] have been conducted on the topic of improving blockchain-based electronic voting solutions with the main focus of improving scalability issues related to cryptographic schemes used to guarantee privacy and verifiability. Privacy-preserving platforms have been identified as one of the potential solutions to these issues, as they rely on confidential and private *Trusted Execution Environments* instead of complicated and demanding cryptographic schemes on standard smart contract platforms [3]. This work focuses on research of these platforms and their application to the e-voting setting. We will go over *Secret Network*, *Oasis Network*, and *Phala Network* and their smart contract development capabilities, and for each, we will implement a simple e-voting application. These applications will be used in an experimentation and evaluation phase, where we will host elections under various settings and perform certain operations during the electoral process(e.g. casting votes or calculating results). We will track, collect, and evaluate the statistics of gas consumption and transactional throughput of these operations carried out during the electoral process. This comparison should determine which platform performs better in the e-voting setting and also what the operational limits are of the state-of-the-art privacy-preserving smart contract solutions.

Chapter 2

Theory

2.1 Voting Systems

Voting or electoral systems are an essential part of public life in a democratic society. They are traditionally tied to political elections, in which participants elect their government representatives by voting, but their application in various decision-making processes also takes place in business, nonprofit, or other organizations [30].

The selection of the best form of voting mechanism available requires a detailed evaluation of its security, accessibility, and transparency to avoid losing public trust in the democratic process. An election system with structural problems can lead to an election win of a fraudulent candidate and, potentially, to significant consequences to the democratic society. Its not exaggerated to state that there are individuals, organizations, or even whole nations that might have desires to manipulate the election results to their benefit. To prevent such desires, these systems incorporate a set of rules that prevent fraud and govern the entire execution of the electoral process. These rules must be carefully chosen so that participating voters have confidence and trust that electoral integrity cannot become questionable. At the same time, the rules must reflect a balance between the restrictions that voters must follow and the accessibility of the election. Researchers around the world are highly optimistic that the use of technology for election systems is the inevitable future [13]. However, finding viable technology turned out to be a harder task than anticipated.

2.1.1 Paper Ballot Voting

Over the years, a variety of new voting methods have been proposed, but traditional voting using paper ballots remains the most common form of voting. Of 227 countries and territories for which *Electoral Knowledge Network* collects data, 209 still cast votes manually by marking paper ballots [30]. It may seem surprising that, despite all the technological advances, organizations still consider paper ballots to be the safest and easiest option used in the electoral process.

The main advantage of these systems is the existence of physical evidence behind every vote, so the electoral process has the ability to start a physical recount of the ballots. The physical aspect also builds trust of the voters, as humans in general tend to have greater trust in something perceptible by touch, and paper ballots are no exception. Paper ballots are also considered to be very transparent, as every voter has the right to oversee the entire electoral process with his own eyes, seeing each ballot being casted and counted. The same applies to the representative of the election organizers, who can supervise and guard the whole process. Only the voters' choices remain to be hidden in order to guarantee their safety and prevent votes from being sold. However, there are still downsides:

- *Ballot box stuffing* - Is a type of electoral fraud, where a greater number of ballots was casted than the ammount of legitimate voters in given polling place. This term is usually used when the voter illegally submits more votes than is allowed, but sometimes it refers to any illegal votes.
- *Marketing the votes* - Individuals participating in the election influence each other to exchange votes for money or favors.
- *Ballot box tampering* - Unauthorized interference with the ballot box that breaks the integrity of the cointainer storing ballots. This usually occurs during transport or during counting.
- *Hygienic issues* - COVID pandemic showed us how gatherings like elections can result in crowded polling places and thus become a critical point of virus transmission. Handling physical paper ballots by election workers and voters is another possible point of contact that could lead to germ spread.
- *Human errors* - Many operations performed in paper ballot system are performed by humans, which result in a range of possible errors (e.g. misinterpreting ballots, sorting errors, or data entries into electronic system errors).

2.1.2 Voting by Mail

Pilot *Voting by Mail* (VBM) programs began in 1981 in the state of Oregon in the United States [2]. Since then, Oregon has remained a pioneer in the use of VBM and is an example of the success that this method has achieved. Many studies carried out in the state of Oregon showed that costs were considerably reduced and attendance increased compared to the previous election process without VBM [1]. Furthermore, the number of early votes submitted with VBM has shown that people prefer not to vote on election day and use *absentee ballots*[1]. Despite these positives, most states in the US use this method only for absentee voting, and the state of Oregon remains an exception with the use of VBM as a primary voting method [30].

The VBM process starts with ballots being mailed to voters, who, in turn, fill them out from the comfort of their home. Subsequently, voters have the option of mailing the ballot back or dropping them off at a predetermined place. The great benefit of this method is that people decide on their own when they want to hand in the ballots and that there is no need to travel to vote on election day. But it is still not a perfect system, as the entire process of delivering and returning the ballot can take several days, which is a significant time window. Compared to e-voting systems over the Internet where its a matter of seconds. In addition, there is a justified concern for potential frauds, as the electoral process is not as transparent as in paper ballot systems. These concerns may also be related to the uncertainty as to whether delivery services will not lose or deliver the ballot late. Some voters may also miss assistance from election workers and make errors when filling out the ballot, resulting in rejected ballots.

[1]Absentee ballots helps voters cast a vote even though they are unable to attend in person on the election day.(this might include military personel overseas and voters with disabilities)

2.1.3 Electronic Voting

Due to the ever-increasing digitalization and interest in modernization, e-voting systems have gained continuous attention and traction around the world. Its caused by many improvements e-voting systems can bring, including faster trial times, much easier accessibility, and experiments have shown increased voter turnout [2]. It might come as a surprise, but in certain countries voting machines have a long history behind them. For example, the Netherlands has a long tradition dating back to the 1950s and 1960s, when the government expressed their first interest in voting machines [17]. In 1965 authorities made it legally possible for Dutch municipalities to use the voting machines, and only a year later machines of this type were produced. The popularity of these methods over the years continued to rise to a point where, in 1999, the Netherlands started discussions about allowing voters from abroad to vote over the Internet (known as remote elections). Previously, voters abroad had limited options to vote by mail or require another person to vote on their behalf. In the 2004 European Parliament elections, voters abroad were used as an experimental group for remote elections. In addition to selecting their candidates, voters completed a poll to determine which method they prefer. The results turned out to be a huge success, as almost all voters preferred the elections over the Internet [17]. However, the success was only short-lived, as in 2006 the Netherlands returned to conventional voting methods using paper ballots due to suspicions of election fraud, and it remains the main voting method to this day [38].

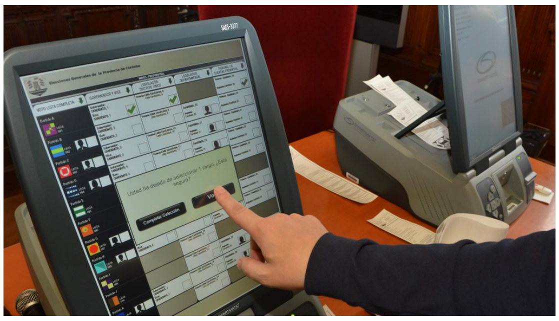


Figure 2.1: Voting machine in Argentina.

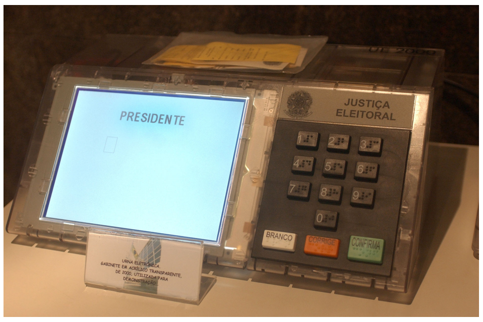


Figure 2.2: Voting machine in Brazil.

The general public has a common misconception that these systems exclusively identify Internet elections (referenced as i-voting), but most e-voting systems used in practice today still operate in person, as in conventional voting systems. Elections that occur remotely over the Internet through some form of secure platform are not common and are mostly used for absentee ballots only. The reason why the in-person form of e-voting is used more frequently is that it maintains the voters' physical presence of the traditional voting systems, which people got used to and tend to trust more. Its a reason why it is often considered a transitional step to i-voting. These voting systems are classified according to the manner in which the electronic machine is employed in the process, either being a direct recording electronic (DRE) or an electronic voting machine (EVM)[2]. EVM is a term that represents a broader range of electronics used to collect and calculate votes (e.g. optical scans or finger-print scanners) from paper ballots and eventually transform them to electronic format. Systems using DRE as their voting technique do not use physical ballots, but instead the voters directly select their preferred candidates from the digital interface like touch screen displays, and all the processing of the votes electronically.

As the example from the Netherlands described, the actual practice of deploying voting systems over the Internet is a technological feat. The main reason why it is difficult to implement such a system is that this kind of election requires a significant degree of security, while maintaining accuracy and secrecy of the electoral process [2]. Furthermore, voting regulations prohibit the release of any kind of information on how voters participated in the election. The voter has no other option besides simply trusting the system to correctly record and count his ballot as intended, without anyone being able to modify its contents. Researchers have been working on methods that would present voters with an indisputable proof of ballot integrity and elections as a whole.

One of the proposed solutions is that this problem could be solved by *end-to-end cryptographic voting systems*. These systems benefit from the use of encrypted proofs that provide assurance of the integrity of the ballot to voters [2]. Blockchain technologies with existing privacy solutions like *Zero knowledge proofs* (ZKPs) are identified as a potential solution, fulfilling many desired characteristics like transparency, secrecy, non-repudiation, and strong end-to-end verification capabilities that are important for voting systems. However, due to the scaling limitations of the blockchain, most of the research articles propose small-scale boardroom e-voting protocols such as *BBB-voting* or *Open Vote Network* (OVN). Only a handful of research papers proposed the design of a large-scale blockchain-based e-voting system (e.g. SBvote [34]).

2.1.4 Electoral Process Classification

Although previous sections presented the classification of voting methods based on the technological form used, the methodology of the electoral system itself is another form of classification. There are a significant number of various electoral systems around the world, while new breakthrough systems are constantly being proposed. Therefore, finding the appropriate typology might seem rather difficult. The study by Elisabeth Carter and David M. Farrell [7] has presented a viable classification approach based on the output of the system. For example, the result of government elections is classifiable as a *proportional* or *non-proportional* outcome. The difference is that in the *proportional* systems, the final distribution of elected representatives is directly related to the votes submitted, in contrast to the *non-proportional* system, where importance is given to the parties as a whole. The intent was to ensure that the one party has a clear majority over its competitors by allocating the seats based on the principle of winner-takes-all. Douglas Rae broke down the electoral process into three main components [25]:

- *District magnitude* (M) - Refers to the number of representatives to be elected in given district. In a single member plurality system, voters elect a single legislator ($M = 1$), while systems of proportional representation have each district elect multiple legislators ($M = n$), where n represents the number of legislators to be elected.
- *Ballot structure* - Determined on the type of ballot voters use to cast their votes. *Categorical ballots* consist out of various candidates or a list of or parties, allowing voters to make a simple choice of either / or. The *Ordinal ballots* are similar to the *categorical* with the exception that the voter can choose all candidates, and the determining factor is the ranking based on the order of preference.
- *Electoral formula* - Manages the deciding relation between votes and seats.

This was classification based on a components, but more frequent classification of electoral processes is based on the electoral formula. This classification divides the electoral process types into seven categories, but for the perspective of our study, we will only go over two foundational types:

1. *Plurality systems* - This class gathers systems where voters give their vote to their preferred candidate/s and the winner of the election becomes the candidate/s with the most votes, regardless of the total tally not achieving an absolute majority (over 50% of votes). To this day, these systems remain to be favored by more than 20% of world democracies worldwide [7]. Further distinctions of these systems are made on top of the number of elected representatives in a given electoral process. In an instance where the outcome results in a single winner, we talk about *single member plurality system* (SMP). A close relative of SMP is *block vote system* (BVS), which is related to district elections, where there are multiple seats to be filled by elected representatives and voters vote for as many candidates as there are seats available.

2. *Majority systems* - Second major category used in significant democracies such as France and Australia [7]. To win an election in the majority system, the candidate must gather a majority of the total votes submitted. There are two forms of these systems: *runoff system* and *alternative vote systems.* The *runoff system* consists of two rounds of elections, where the first round determines the top two candidates that proceed to the second round. The second round takes place just a couple of weeks after the first one, and its goal is to ensure the majority result. The *alternative vote system* (AVS), popular in Australia, provides a special ballot structure that allows voters to numerically sort all candidates based on their preference. The process of determining the winner starts by counting the number one selection. If there is no candidate who achieves a majority of the vote (again more than 50%), then the candidate with the least number of votes is eliminated from the election, and the votes of the eliminated candidate are distributed proportionally to other candidates. This process repeats itself until a winner with a majority of the votes appears.

2.2 Blockchain

The *Bitcoin*, and *Ethereum* cryptocurrencies are the first things people visualize whenever a blockchain is mentioned,even though a blockchain can be constructed without a currency or value tokens. This comes as no surprise since both *Bitcoin* and *Ethereum* have been filling the headlines of all noteworthy newspapers around the globe ever since the prices of these currencies skyrocketed in 2017 and 2021 respectively[2].

The history of blockchain extends beyond 2017, and it roots back to January 2009, when the famous *Satoshi Nakamoto* published the paper *Bitcoin: A Peer-to-Peer Electronic Cash System* [20], where he described his vision of electronic cash intended to be used in online payments without the need to certify transactions by a centralized financial institution. This article is considered a major milestone that has sparked intense research and development of blockchain technologies. Despite the resonant success of the paper, the world still does not know the real identity behind the author, which remains a mystery until today. Another breakthrough in blockchain technologies arrived in 2013 when *Vitalik Buterin* released *Ethereum* which brought a new paradigm that made possible the creation of Turing-complete decentralized applications with smart contracts [4].

[2]The price statistics of both *Bitcoin* and *Ethereum* is available over at https://coinmarketcap.com.

Generally, the blockchain acts as a trusted and reliable third party to maintain shared data, facilitate exchanges, and provide a secure computation [5]. The fundamental characteristic of the blockchain design is that the network of nodes is distributed over a peer-to-peer network without a central management unit [9]. Its a form of a distributed database that holds an ever-growing list of records, controlled by multiple entities. The participating entities must align with their counterparts by having a distributed trust process or consensus mechanism in place. These mechanisms act as guarantors of the integrity of the data stored on the blockchain, even in *Byzantine environment* such as peer-to-peer networks.

2.2.1 Public and Private Blockchain

Both *Bitcoin* and *Ethereum* fall into the category of public blockchains. These networks are considered to be public, since anyone on the Internet is able to access the blockchain network, inspect the state, and modify the state by committing transactions(e.g., exchanging monetary value or smart contract execution) and anyone can become an authorized node that participates in consensus. These networks typically use *Proof of Stake* (PoS) and *Proof of Work* (PoW) as their consensus mechanisms, and their validator network consists of a large number of nodes. Public blockchains are suitable for use cases where transparency, accountability, and decentralization play an important role.

Private blockchains have a central authority that restricts access to a predefined set of participants with limited permissions. Usually, participants are required to be fully identified and trusted to guarantee the privacy of the networks. This branch of blockchain also finds use cases, especially in organizations that are not interested in sharing their confidential data, but still want to use blockchain. Examples of such networks are *Hyperledger Fabric*, *R3 Corda*, or *Quorum* [5]. The privacy features are not tied to private blockchains only, the popular trend of incorporating privacy features to public blockchains gained traction as of late. These are the so-called privacy-preserving platforms, which make use of *Trusted Execution Environments* or *Zero Knowledge Proofs* to power this strong feature (see Subsection 2.3.2 and Subsection 2.3.1).

2.2.2 Architecture of Blockchain

As the term blockchain suggests, the core component of the blockchain's architecture is the block itself. The structure consists of a continuously growing set of these blocks, where each block is being chained with its predecesor and succesor using cryptographic operations. The block is a form of abstraction that bundles two separable components, a list of transactions, and a range of metadata stored inside the block header (see Figure 2.3). These blocks are linked using a cryptographic hash function and timestamps to form a nearly unalterable data structure. Meta-data stored inside of the block header include:

- *Previous hash* - Computed from the previous block, used as bond between blocks to assure their tamper resistance.
- *Nonce* - Short equivalent of *number used once*. It is a critical random number that is used in the mining process (cryptographic puzzle) of blockchains based on *PoW*.
- *Version* - Specifies version present at the time of block creation
- *Mining difficulty* - Defines how hard was the mining process of the given block.

- *Merkle tree root* - Represented by a cryptographic hash that was computed from the pool of transactions stored in one block.

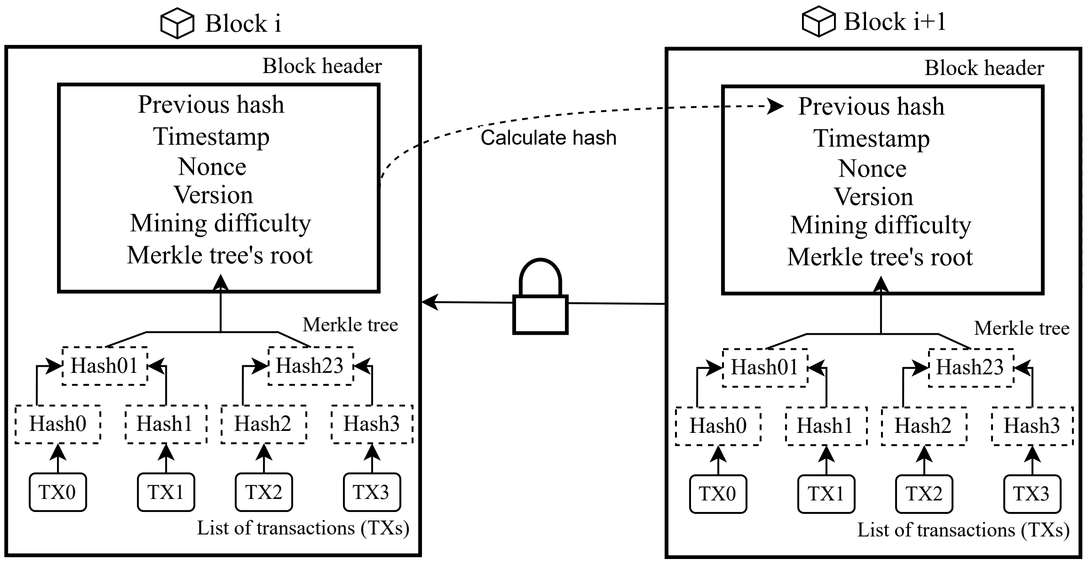


Figure 2.3: Block structure in PoW blockchain like *Bitcoin* [43].

The list of committed transactions describes the changes to the state of the blockchain in the current time window. In this way, it is possible to facilitate the tracking and secure transfer of digital assets or other arbitrary data. The hashes that bind to the previous blocks theoretically form a nearly unbreakable link (see 51% percent attack in Item 2.2.4), which ensures the integrity and security of the blockchain. The genesis block is an exception that may not point to the hash of a previous block and is usually created by the founders of the blockchain network.

2.2.3 Consensus Mechanisms

In theory of distributed computation, consensus mechanisms play a vital role as they enable a large-scale fault-tolerant network of machines or servers to work as a coherent group that agrees on the state of the system, even when some of the machines or servers fail or behave as an adversary [5]. These mechanisms are also an essential part of blockchain technologies, as they prevent double spending and guarantee the integrity of data and the trustworthiness of the shared ledger. This is only possible by having a strict policy and predefined sets of rules and techniques combined with cryptographic operations that all participants must follow. Each node stores an image of the latest state of the blockchain that is continually updated with newly appended blocks. But before the block is appended, the entire network must agree on its integrity using the rules defined by the consensus mechanism.

There is a wide range of consensus variants out there. The most prevalent is *PoW* mainly due to its association with Bitcoin. This mechanism was a breakthrough because it solved two problems that previous designs such as Adam Back's *HashCash* or Wei Dai's *b-money* failed to solve [4]. The first problem was to find a simple but effective algorithm that would allow the nodes in the network to agree on the order of the transactions. The second problem was making the network available easily and freely to anyone while preventing *sybil attacks*[3].

Other instances of consensus mechanisms are *Proof of Stake* (PoS), *Proof of Identity* (PoI), *Proof of Authority* (PoA) and many others. These mechanisms do not rely on miners to produce valid blocks, but rather on validators that are selected based on factors such as staked tokens (its an economic incentive where validators risk losing staked collateral[4] in case of malicious activity or failure to validate transactions), authority (reputation of their historical behavior), or their proven real-world identity.

2.2.4 Proof of Work

The nodes in networks utilizing *Proof of Work* (PoW) are continuously incentivized to compete in the process of solving cryptographic puzzles by promising rewards for providing the right solutions. The difficulty of these puzzles is set within the defined rules, so its resolution requires significant computational power that only a majority of nodes control. The process during which a block is created and stored on the blockchain follows these steps [20]:

1. *Transactions propagation* - All users using the blockchain initiates transactions by broadcasting them over the network.

2. *Transactions collection* - Broadcasted transactions are collected into pools, where they wait until their turn to be bundled into a new block arrives.

3. *Block creation and mining* - Nodes participating in the mining process or miners select transactions from the pool and create a block with all its accompanying metadata, but the block is not yet ready to be appended to the blockchain yet. Miners have to work and compete in the cryptographic puzzle based on the contents of the block (e.g., it involves finding a hash with certain characteristics calculated from the contents of the block including the nonce, which is iteratively incremented until the desired characteristic is met). In *Bitcoin*, the characteristic is to find a hash with a certain number of zeros in its prefix [5].

4. *Block validation* - The first miner to solve the puzzle publishes the solution to all others, who in turn carry out the validation steps. The first validation step is to check whether the received solution to the cryptographic puzzle is valid and actually meets the set requirements. The second step is to check if all the transactions contained in the block are confirmed and not spent on some of the previous transactions (a crucial step in eliminating double spending).

5. *Block acceptance* - If the validation in the previous block was successful, then all the nodes accept that block as the last one appended to the blockchain by working on the next block where previous hash .

6. *Rewards* - The miner who found the valid solution get rewarded by receiving tokens or cryptocurrency of the given blockchain platform.

[3]Its a type of attack where single adversary controls multiple nodes in the network, with intent to gain control over the consensus and manipulate the integrity of the blockchain [15]

[4]Collateral are valuable belongings or assets that are pledged as security that would be lost in favor of the blockchain network in the event of a failure to conform to set rules.

The nodes follow this process in order to continuously attempt to create blocks, where in the case of a PoW networks a block creation takes roughly ten minutes. The underlying cryptography of the mechanism is known and is considered secure and vigilant to attacks. Attackers have only one option to try to create a fork of the blockchain with a new block that contains broken order of transactions. But this attack is also not feasible, as PoW's employ the rule where miners consider the longest blockchain to be the truth. The attacker would have to control 51% of the network in order to be successful, and that is in most cases an impossible task.

2.2.5 Smart Contracts

Another crucial and enabling concept in blockchain technologies are smart contracts. Smart contracts can be defined as a digital version of a standard legal paper contract with a set of conditions agreed by the participating parties that is encoded, stored, and executed on a blockchain. This brought advantages over their paper counterparts, like automatic enforcement of the agreed terms, which deemed any central unit unnecessary. The smart contracts were a huge leap in blockchain designs, they are even considered the next generation of blockchains or blockchain 2.0, extending the basic token transfers of first-generation blockchains (for example, Bitcoin) to also have the ability to perform computations in a turing complete manner [4].

The original idea behind Smart Contracts was presented by a scientist and cryptographer, *Nick Szabo*, in 1997, as a form of digital vending machine [35]. The contract is basically a script written in a programming language that is stored and executed on a blockchain. These programs are typically used as a form of electronic agreement that automatically follows the workflow and triggers the next actions based on the conditions and terms established in the contract. The missing piece was the establishment of a decentralized system with a proven trust model, like blockchain, as the foundation for smart contracts. Correct smart contract execution guarantees have rendered a central supervisory party unnecessary, as both participants place their trust in the contract and blockchain instead. The analogy of a digital vending machine, originally envisioned by *Nick Szabo*, perfectly matches the previous definitions, as these machines also have a pre-programmed logic that is guaranteed to be followed, while there is no employee assistance needed (representation of the centralized party). Besides trust, the blockchain also introduces transparency and immutability, meaning that all of the code of contracts is publicly available, and once its published no one can change it.

There are many use cases for smart contracts, for example, banks could use them to trade finances, logistic companies could use them to track and trace the movement of their shipments, and energy companies could incorporate peer-to-peer energy trading between consumers that own a renewable source, but the list goes as there are several other fields where smart contracts could make impact.

2.2.6 Smart Contract Code Execution

The programming language used to write the logic of a smart contract depends on the hosting platform. Due to the popularity of *Ethereum*, the most widely used language is *Solidity* [4], but other languages such as *Rust* or *GoLang* (Chaincode) are also used in some cases (these are used in Polkadot, Hyperledger Fabric, and Cosmos [5]). In the following text, we will look at the *Ethereum* platform and its code execution capabilities. The execution of code starts with the compilation of high-level code written in *Solidity* into a stack-based bytecode, known as the *Ethereum Virtual Machine* code or *EVM* code, where each byte in the resulting bytecode represents a single operation. The execution of the code is performed in a loop, where the program counter determines which code is currently executed. The counter is iteratively incremented by one until the end of the code, the return instruction, or the error is reached [4]. In this fashion, the execution of *EVM* code is replicated by each validator on their local copy of the blockchain. Replication must be deterministic, so that the validators will not come up with different results for the same code. Computation has three storage options at its disposal:

- *Stack* - Standard last-in-first-out structure, where data values can be appended and popped.
- *Memory* - Expandable linear byte array that is used for temporary data storage. The size of the expansion is always multiples of the 32 bytes, until a *EVM* maximum memory size is reached. Each expansion increases the gas costs, so programmers must be mindful of the memory usage in their contracts.
- *Contracts long term storage* - Its a key-value store that persists even after the transaction is completed.

2.2.7 Fees System

Validators who perform the computation receive proportional compensation for their work. This serves a dual purpose, the first being that the validators are motivated to provide their computational resources, and the second being that there is some cost for performed computation in order to prevent unnecessary use. To represent and quantify the computation performed, smart contract platforms standardized a unit of computational work that is referred to by a term *gas*. Each instruction executed within the *EVM* code has a predetermined cost, which is quantified in a number of these gas units. In addition, each gas unit has its own price, represented and paid in the native currency of the underlying platform (e.g., Ethereum coins). Users interact with the smart contract by sending *contract-invoking* transactions, where they must specify the price of gas, and the maximum amount of *gas* that the user is willing to pay for the execution of the transaction [37]. If the execution exceeds the maximum gas limit, the execution is stopped, and all operations already performed are reverted to obtain the original state before the transaction. The sender of the transaction still gets charged the maximum gas limit, in order to prevent a resource-exhausting attack [12].

The gas limit and gas price metrics are used to calculate the resulting fee charged. The price of gas must be higher than the current base fee established by the protocol. The base fee estimation is determined by the gas size of the previous block (the amount of gas used for all the transactions) with a target size. The maximum increase in the base fee between blocks is 12.5% per block [39]. Anything more than the base fee acts as a tip directly to the validator of a given transaction, who prioritizes transactions with higher gas prices.

2.2.8 Homomorphic Encryption

Homomorphic encryption is a popular cryptographic technique in the context of blockchain technology, due to its security and privacy features. In mathematics, *homomorphism* is used to characterize two operations that have the ability to receive the same result, even after the order in which they have been originally performed has been changed [26]. *Homomorphic encryption* is a specific cryptographic encryption technique that uses *homomorphism* properties so that mathematical operations can be performed directly on top of encrypted data without knowing their encryption key. An interesting takeaway is that when the result of the operation performed on homomorphically encrypted data is decrypted, we will receive the same result as if we had performed the same operation on the raw data. This is an important privacy aspect that makes it possible to maintain data confidentiality while performing mathematical operations.

Definition 1 *An encryption is homomorphic if: from $Enc(a)$ and $Enc(b)$ it is possible to compute $Enc(f(a, b))$, where f can be: $+$, $\times$, $\oplus$, and without using the private key [40].*

As seen in Definition 1 homomorphic encryption operates with three basic functions: additive (known as the *Pailer cryptosystem* [22]), multiplicative (referenced as the *ElGamal cryptosystem* [11]) and *XOR*. The additive version of the encryption has an interesting application in e-voting, as these systems have to keep votes private, but at the same time they have to get counted to get the results. Homomorphic encryption is a perfect solution for this specific problem, as it has the capability to preserve privacy of the votes that remain encrypted while only the sum of the votes would be decrypted and shown to the public. The three functions represent the three main categories of homomorphic encryption, but the number of allowed operations and the amount of possible repeated calculations in the cryptosystem could differ. Thus, another categorization exists [40]:

- *Partially Homomorphic Encryption* - Only one type of operation is supported, but it is possible to repeat the calculation unspecific amount of times.
- *Somewhat Homomorphic Encryption* - Allows multiple types of operation, but a limited amount of calculation is allowed.
- *Fully Homomorphic Encryption* - There are no limitations in terms of operation type with unlimited repeated calculations.

2.2.9 Zero Knowledge Proofs

Zero knowledge proofs (ZKPs) is another cryptographic technique that aims to improve privacy for sensitive data shared in a public setting. ZKPs allow one party (provers) to prove to others (verifiers) that a statement is true without revealing any information about the statement in question, hence the name *Zero Knowledge* [40].

Today, many cryptographic protocols that require privacy use this technique, including the popular cryptocurrency *Zcash*. In relation to our topic of voting systems, ZKPs make it possible to unlink the voter and his vote from each other. Murtaza in his paper [18] presents a setting in which each entity participating in e-voting would have their own ID token used as an authentication for the election and a vote token that could be transferred as if we had given a vote to someone. To make sure that the vote token and ID token are unlinkable, one would create ZKPs from the process of burning their own ID token. This ZKP would then be presented as evidence that the voter has not voted yet and that he is eligible to vote. Upon successful verification, the voter receives the vote token used to vote for his preferred candidate. In this way, the ID token used to create the vote token remains private.

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zkSNARKs) is a lightweight form of a zero-knowledge proof, where proof verifications take only a couple of milliseconds [26]. The popular showcase of its utilization is the crypto-currency Zcash, where it is used to keep the transactions private. Each component of the zkSNARKs title identifies unique properties:

- *Succinct* - Size of messages, used to convince the verifier are significantly shorter, when compared to the input and its subsequent time complexity to compute. These small proofs are the reason behind the time-effective verification.

- *Non-interactive* - Interaction between prover and verifier is reduced to only one message, without counting the messages needed in the setup phase.

- *Arguments* - Provers are able to create arguments or proofs about wrong statements, but it takes significant computational power. Otherwise, the verifiers do not have to worry about such proofs. This is known as computational soundness [26].

- *of Knowledge* - Construction of the proof requires to have knowledge about the statement we wish to convince the verifier.

2.3 Trusted Computing

As the technologies and computer systems continue to become more complex and their adoption becomes more widespread, the security requirements are ever so high. Trusted computing was introduced to improve computation security, privacy, and data protection. At first, the segment relied on tamper-evident hardware modules, separated from the system, that provided an interface for platform security mechanisms. This approach had the problem that the functionality was limited by the pre-defined APIs. The most recent approach tried to tackle this problem by providing a confined environment used for code execution. The main benefit was that the applications had a tamper-resistant run-time that was protected by having custom security mechanisms in place. These confined environments are referenced in the literature in various ways, but most common one is *Trusted execution environment* [28].

2.3.1 Trusted Execution Environment

Trusted Execution Environment (TEE) is known as an isolated processing environment that protects its runtime state and assets from other untrusted parts of the system [28]. A simple explanation can be made in analogy to smart contracts, since smart contracts are a trusted neutral description for the processing of transactions on blockchain networks, TEE is a trusted neutral party for secure and private computations on hardware components [42]. The isolation of the execution process within the environment is handled by *Separation Kernel*, which separates the system into strongly isolated parts, which may have various levels of security mechanisms in place. Originally, *Separation Kernel* was developed to simulate a distributed system on a single machine, but later its potential to even other areas was discovered, including applications within TEEs. The origin of TEE is tied to the *Secure Execution Environment* (SEE) that does not consider the trust aspect as is the case in TEEs, but ultimately promises to guarantee properties such as:

1. *authenticity*: executed code should not be modifiable.

2. *integrity*: only executed code can change the stored state.
3. *confidentiality*: information about the state or the code executed should be kept secret from unauthorized applications (including the operating system).

TEE incorporates these attributes while also building trust in the computations performed within the environment. The concept of building trust depends on the existence of a trustworthy and reliable mechanism capable of presenting evidence of the computations being performed. In *TEE* the mechanism for presenting evidence is managed by a trustworthy entity called *Root of Trust* (RoT) [28].

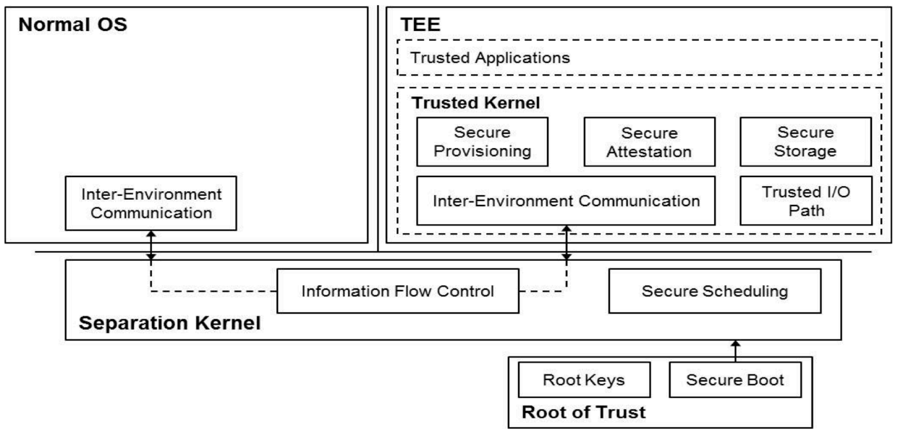


Figure 2.4: Architecture of the Trusted Execution environment [28].

The RoT is a tamper-resistant hardware module that executes a trusted measurement and computes a trust score. Tamper resistance is significant here, as the trustworthiness of the trust score itself is directly affected by the reliability of the trusted measurement. In the real world, we tend to measure the trust of a certain system by its reliability, which is basically a measurement of how much the system behaves as expected. The trust measurement in TEEs is determined by the integrity measurements that inspect if the loaded TEE is conforming to the one certified by the given platform provider before deployment. The outcome or the trust score is a Boolean value that indicates if the TEE is trusted (trust score being True) or not (trust score being False). Notable representatives that provide TEE technologies are *Intel SGX* (Software Guard Extensions), *AMD TrustZone*, or *Trusty TEE* used in mobile devices. All of those companies enable TEE's by incorporating sets of security-related instruction codes into their CPUs into a special isolated section referred to as *enclave*, so that its execution cannot be affected by other processes, nor even the underlying operating system. However, it is still necessary to provide a procedure that would be able to supervise and confirm the correct execution within the isolated enclave. In Intel SGX the procedure is called *Remote Attestation* [8].

The hardware supporting Intel SGX can generate the so-called *attestation quotes* that are based on the details of the hardware, firmware, the code executed inside the enclave, and other user-defined data produced during the code execution. The quotes are signed with private credentials embedded during the manufacturing process, to accomplish a unique bond to an Intel CPU unit. The quotes are then sent to *Intel Remote Attestation Service*, where the authenticity of the quote is verified. The positive outcome of the verification process provides a proof that a certain code has been executed within the given enclave and that the execution on its own was confidential and correct. Afterwards, the quote

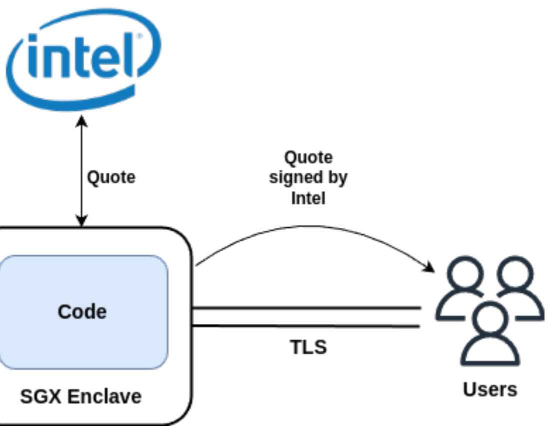


Figure 2.5: Remote attestation process in Intel SGX [41].

will be signed and published by Intel so that anyone with viable hardware can verify the execution process within the enclave.

2.3.2 Privacy-Preserving Platforms

Besides all the positives, blockchain and its decentralization and public nature face problems with legal regulations (e.g. General Data Protection Regulation or GDPR) and privacy issues. Privacy-preserving platforms are sets of technologies that focus their attention on the privacy protection of user data while allowing processing on top of it. These platforms may employ various cryptographic techniques, such as ZKPs, end-to-end encryption, or secure multiparty computation (MPC) techniques to anonymize users and give users the control which sensitive information will be disclosed publicly [3].

2.3.3 Secret Network

The *Secret Network* was the first protocol to provide confidential smart contracts on the mainnet [33] and is a leader in the space of confidential blockchain networks. Secret Network is a Layer 1 solution built with Tendermint's Byzantine fault-tolerant consensus algorithm [42], which is known for its high throughput and great scalability. The secret network utilizes cryptographic mechanisms and TEE enclaves to achieve data confidentiality while keeping the ledger public. Thus, users are able to perform transactions without revealing the outcome or content, providing a confidentiality factor that is missing in traditional public blockchains. The developers of the Secret contracts may use Secret's programmable privacy controls that allow us to specify which subsets of data will be encrypted and not.

Smart contracts are based on *CosmWasm* platform, which is a standalone module in the *Cosmos-SDK*[5] framework. The deployment of smart contracts follows a chain of events that are visualized in the Figure 2.6. The smart contract is first compiled into the WebAssembly (Wasm) bytecode, which is a binary instruction format designed to improve the performance of code execution directly in web browsers, for programming languages like `C`, `C++` and `Rust`. The compiled Wasm bytecode uses an intermediary Cosmos SDK module `x/wasm` to manage the execution of the smart contract in a *CosmWasm Virtual Machine* module (CosmWasm

[5]Cosmos SDK which is an open source framework used to build public Proof-of-Stake (PoS) or permissioned Proof-of-Authority (PoA) blockchain systems in the Cosmos ecosystem.

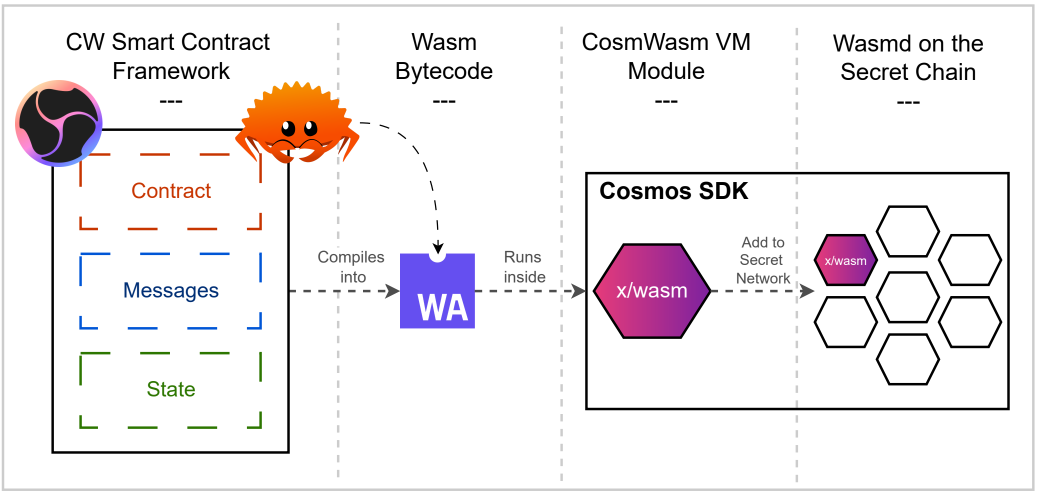


Figure 2.6: Overview of the CosmWasm involvement in the Secret Network [29].

VM). The role of the CosmWasm VM is to securely encapsulate the execution of the smart contract and to ensure that the code follows the same execution semantics. This grants that any CosmWasm contract will function the same on all Cosmos blockchain platforms (including Secret Network).

The validators in the Secret Network cooperate in the computation to achieve verifiability while securely preserving the privacy of the data in the TEE. Each node in the Secret Network inherited Cosmos's unique interoperability feature in the form of *Inter-Blockchain Communication Protocol* (IBC), which defines how two blockchains can handle authentication and data transfer between each other. Users of such networks make transactions between different chains almost seamlessly. The nodes in the network employ Intel's SGX, so its guaranteed that each must have gone through a remote attestation process (see Figure 2.3.1) to prove that the TEE is genuine.

The ledger in its nature remains public, so anyone can lookup information like the transaction sender's address, their public transaction history, or the size of the encrypted inputs and outputs of executions. For governance and gas fees, the network uses the Secrets native *SCRT* token. There is no fee market in the Secret Network, which means that the size of the fee does not affect the prioritization of the performed transaction, but it might speed up the finality. The size of the fee is determined by the smart contract engine, which takes into account the computational resources required and the fee that validators ask for these resources. [6]

Privacy Preserving Smart Contracts

The binary code of the compiled smart contract in Secret Network remains to be public similarly as in any other popular smart contract network, but it differs in the way the data are managed. The state, input, and output of secret smart contracts are encrypted with a forge-proof encryption key that is unique for each smart contract [42]. The primary input component that gets encrypted are the user's messages. To maintain transparency and traceability within the network, addresses, block height, and sent funds remain unencrypted. The contract state is consistently encrypted, and its content is only known by the contract

[6]Gas fees are listed on Secret Network's github page https://github.com/SecretNetwork/gasFees.

itself. Additionally, users have the ability to view their own dedicated state in the contract. This design brought privacy risks during the migration of contracts[7], because anyone who would migrate the existing contract can become an administrator and thus gain read access to the data of the old contracts. This breaks the core features of Secret and is the reason why this feature is disabled. The output is encrypted similarly to the state, with the addition that the transaction senders can inspect the output of the transaction they invoked.

Storage

Storage is a form of abstraction that provides read and write access to persistent storage, in this case the blockchain. Smart contract platforms use this abstraction to store various data, such as configuration, metadata, or historical data, but mainly the overall state of the blockchain [23].

The storage in *Cosmos-SDK* is based on a key-value store, where each value is stored under a certain key [23]. It has a tree-like structure so that the Merkle root hash can be computed for the secure verification of the integrity of the data. It is based on the self-balancing binary search tree *IAVL+* an immutable version of the tree *AVL+* (named after its inventors Adelson-Velsky and Landis), where all operations performed take at worst $\mathcal{O}(\log n)$. *AVL* tree is defined as a balanced binary search tree where the height of the two subtrees (children) of a node differs by at most one and the inner nodes can also hold key-value pairs [10]. *AVL+* is an *AVL* modification in which only the leaf nodes store all values, while the branch nodes hold the keys [27]. The simplified example of the *AVL* tree can be seen in Figure 2.7, where each node is identified with a letter (key) and the green numbers denote the balance property (negative balance indicates the right imbalance, positive balance indicates the left imbalance).

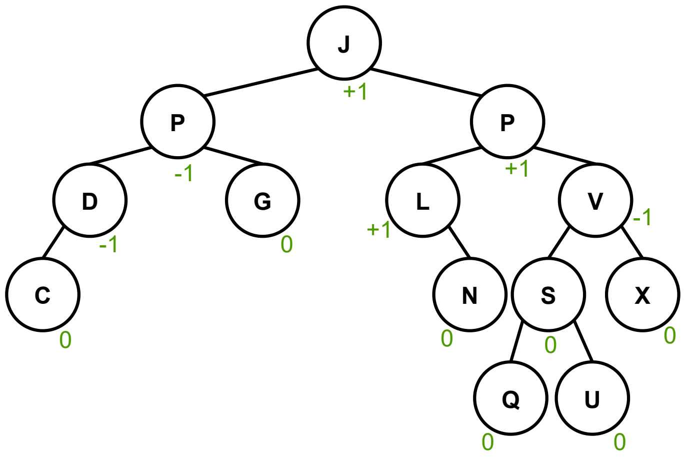


Figure 2.7: Simple example of how a key-value storage may look like [23].

The keys saved to storage have fixed-length prefixes containing metadata (e.g. information about the contract that owns the storage) based on which we retrieve values. When it comes to retrieving values, there are only limited options. Either we perform a single value retrieval, which is a very inexpensive operation (search with a known key has a time complexity of $\mathcal{O}(1)$), or we iterate over a set of keys to access multiple values.

[7]Contract migration is a process during which existing contracts are moved into newly created contracts, along with the stored data on the old contract.

The iteration process works on top of key prefixes, where we sequentially visit each node in the tree and compare their prefix with the one we searched for. In the instance shown in Figure 2.7 an iteration through `range('JPV')` would return keys `JPV`, `JPVS`, `JPVX`, `JPVSQ` and `JPVSU`. This process uses the fact that the keys have a fixed length (assuming 8 letters), so when the `range(JPV)` request is made, the algorithm looks for keys starting with `JPV00000` to `JPVFFFFF`.

2.3.4 Oasis Network

The Oasis platform is another platform that represents the new generation of blockchains focused on providing verifiable and confidential smart contract execution [32]. Similarly to the Secret platform, Oasis is a Layer 1 solution that focuses on the scalability, flexibility, and privacy of the network.

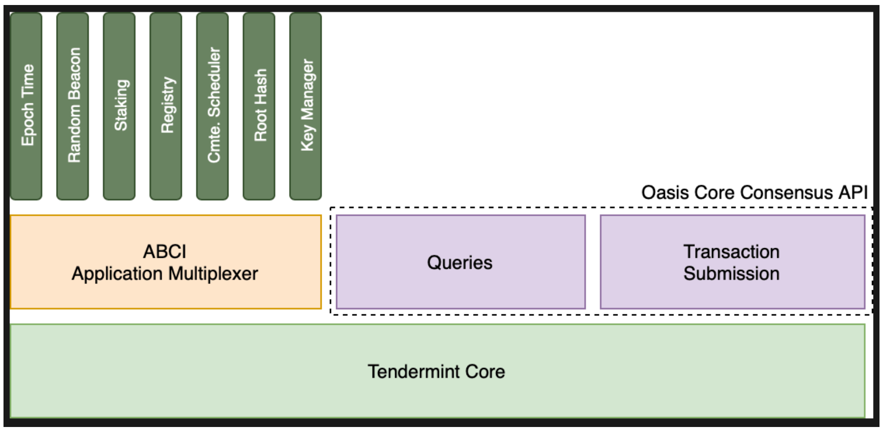


Figure 2.8: Architecture of the consensus layer [14].

The architecture of the platform consists of two major layers (see Figure 2.8). First, the consensus layer orchestrates stateful computations, and the runtime layer is where these computations are performed. Separation of the consensus layers from the smart contract execution layer is intended to bring convenience to being able to switch to different consensus mechanisms if necessary. Currently, this functionality is limited and Oasis only supports the *CometBFT*[8] consensus back-end. This brings a form of future proofing of the platform from possible breakthroughs in the space of consensus mechanisms in the future.
The consensus layer facilitates the runtime layer only with essential services that handle the functionality required for secure operation of the runtime layer:

- *Random beacon service* - Provides unbiased randomness for each epoch. This service has a scheme that guarantees that as long as there is a predefined number of participants and at least one participant is honest, the beacon will generate a secure entropy.

[8]Its a protocol based off of Byzantine fault tolerance, which is a property of distributed systems that ensures their security even in the presence of malicious members in the network. [32]

- *Staking service* - Guards correct operation of the PoS consensus mechanism that manages the staking ledger and handles operations such as stake transfer or stake escrowing [9].

- *Registry service* - Manages the distribution of public keys and metadata for the entity, node, and runtime.

- *Committee scheduler* - Periodically schedules a committee (i.e., validator or key manager) based on the epoch of the timekeeping service and the entropy of the random beacon. After that, the scheduler selects the validator committee and assigns a certain voting power.

- *Time-keeping service* - Responsible for the time measurement.

- *Governance service* - Provides chain governance or root hash service that manages runtime commitment processing, or minimal runtime state keeping.

The runtime layer consists of separate runtimes that have the ability to run simultaneously with other runtimes, each of which possible shares the same consensus layer. Runtime is an executable that runs in a sandbox environment, optionally combined with Intel SGX enclaves.

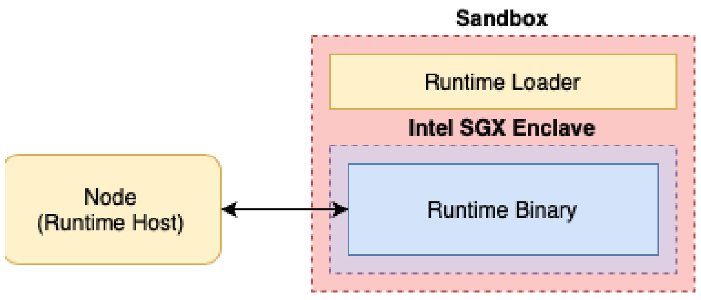


Figure 2.9: Architecture of the runtime layer [14].

The runtime state and logic are decoupled from the consensus layer, as the state and state transitions are stored inside of every runtime independently of the consensus layer. However, the consensus layer is leveraged for finality and the canonical state. It receives submitted values and short proofs of performed computations made on the runtime layer, and the consensus algorithm decides which values will be stored on the blockchain.

The integrity of the run-time is established with a replicated computation among multiple nodes with *discrepancy detection*. All nodes execute the same runtime and produce a result that is compared to determine if there are some discrepancies. In case of discrepancies, the execution is repeated with different compute committees. Each runtime can choose which verifiable and confidential computing technique will be used without having to make any changes to the interface that connects the layers. These techniques are used to verify the results of the executed transactions and to determine how strict the data privacy policy will be.

[9] Escrow refers to a neutral third party holding assets or funds before they are transferred from one party in a transaction to another. [6]

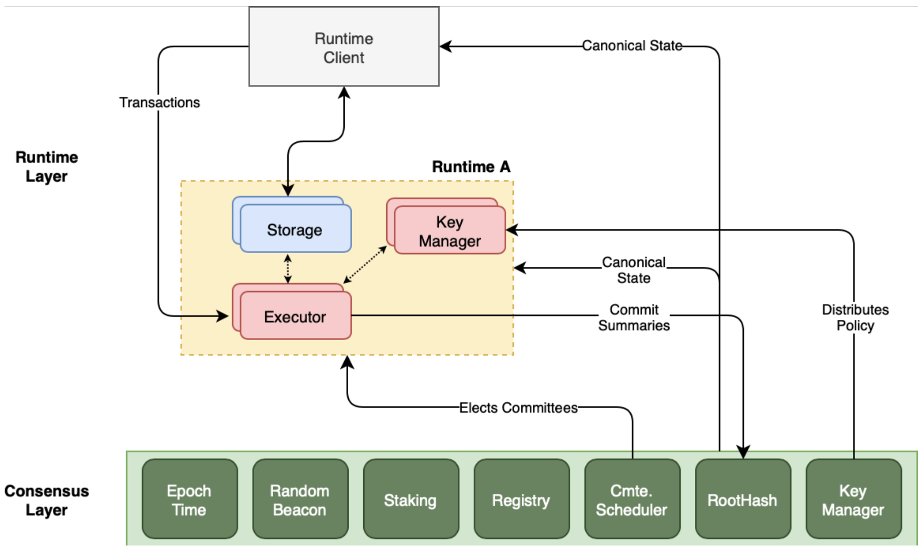


Figure 2.10: Architecture of the runtime layer [14].

2.3.5 Phala Network

Phala Network is a novel cross-chain confidential and interoperable smart contracts network introduced as *Polkadot Parachain*[10] with the *Event Sourcing* and *Command Query Responsibility Segregation* (CQRS) architecture in a hybrid TEE-blockchain system [41]. The combination of these characteristics makes the network scalable, resistant to conflicts, and facilitates confidential smart contracts with cross-contract and cross-chain interoperability [41]. The problem the Phala platform tries to solve is tied to building decentralized applications. Most use cases today require off-chain components to serve robust feature sets, such as storing large amounts of data or performing complex programs. Executing such features exclusively on-chain is not so feasible and efficient, which is why the Phala Network decoupled the smart contract computation between off-chain and on-chain. For example, a smart contract could read some data stored on-chain, then perform complex computation on the data on remote workers off-chain, then write the results back on-chain. The whole process would be much more efficient than using smart contracts, but at the same time remaining secure and confidential.

Enclave Architecture

The foundation for the confidential smart contracts is an infrastructure based on the Intel SGX's TEE technology that not only prevents leakage of information during contract execution, but also guarantees availability, authorization, and correctness of input data. The infrastructure is facilitated by a group of non-Byzantine worker nodes with TEE-compatible hardware. These nodes perform computations in the enclaves independently of the blockchain with a special *pRuntime* program that runs inside the enclave. The *pRuntime* as the name suggests is a run-time environment that exports a set of APIs that connect the program to the blockchain so that the contracts have access to the shared state, and

[10]A Parachains are data structures, usually blockchains, that are globally coherent and can be validated by the validators of the Relay Chain(the central chain of Polkadot). They take their name from the concept of parallelized chains that run parallel to the Relay Chain [24].

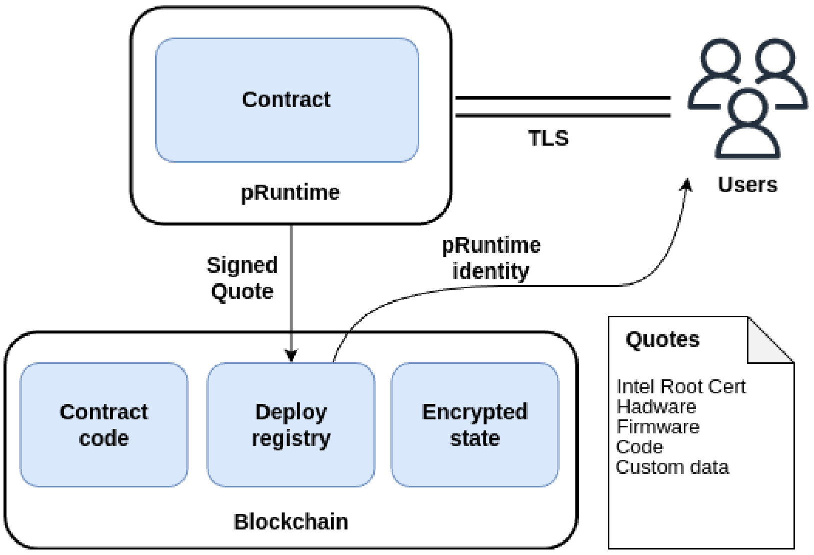


Figure 2.11: Architecture of the enclaves and blockchain within the Phala network [41].

users may securely query the data. The worker nodes are stateless, meaning they have to sequentially execute all of the input events stored on the blockchain (or cached contract state within the node) to obtain the latest state.

To maintain the confidentiality of the enclaves, the blockchain state must be encrypted, so *pRuntime* stores a set of secrets or keys used to decrypt the state, which are generated during the *pRuntime* identity registration on the blockchain. Since the keys are stored inside the enclave, and no one else has access, it is impossible to pretend to be an adversary *pRuntime*. There are two types of keys generated, the first using symmetric cryptography to secure the channel between *pRuntime* and the blockchain, and the second set is a pair of public and private keys of asymmetric cryptography that represent the *pRuntimes* identity. Thus, users can verify the authenticity of each *pRuntime* on the blockchain and use the public keys available *pRuntimes* to establish a TLS-like channel that serves their requests. All cryptographic secrets are managed by another type of nodes, called „gatekeepers", which are responsible for the security and availability of the network.

Confidential Smart Contracts

The process of contract deployment is started by the contract's bytecode being stored in the blockchain. Once the Gatekeepers notice the new contract on blockchain, they supply the contract bytecode to a worker node's *pRuntime*. In addition to bytecode, *pRuntime* receives a symmetric encryption key that will be used to decrypt and encrypt the state of the given contract on the blockchain. On the blockchain side, all of the encryption keys get stored in the chain state that in order to stay private has to be secured by another level of encryption, due to blockchain public nature [41].

For the contracts execution, the Event Sourcing / CQRS architecture is adopted. *Event Sourcing* is a design pattern in which the state is represented with subsequent transitions in the append-only log of timestamped events. The timestamps ensure that the network remains deterministic and that it is possible to reconstruct any state at any time. The CQRS is a pattern that specifies that the network handles read/write operations separately, where writes operations represent events (represent various write commands such as user invocations, blockchain events, and ingressive messages) that determine the state, and read operations a certain view of the state. The *pRuntime* observe all the events that target the contract deployed within its enclave, and in case passes them to the contract, where they get processed to form a *Chainview*. Users may query *Chainview* at any time, but they are required to provide proof of their identity during the query call, which contract uses to determine whether they will respond or not. The *pRuntime* additionally produces various side effects, such as updating the blockchain with a new encrypted state or sending messages to other blockchains or services.

Chapter 3

Design

The first section of this chapter starts with a specification of the electoral process implemented within the e-voting system. Afterwards, we will layout the foundational structure of the implemented smart contracts, and analyze entities that are taking part in the system and their mutual relationships in order to determine the architecture of the storage within the contracts. Then we will describe the structure of methods that facilitate the e-voting functionality in each of the smart contracts. The last three sections explain the significant design choices made in the given PPP. These include key programming, access control, or storage concepts offered in the respective platform, and analyze their differences in terms of performance, privacy, or relevancy. Since we evaluate the performance of the respective platforms, we will not incorporate a privacy-risk-free design that would require extra precautions like padding the messages to the same length and deliberate gas evaporation within the called methods (due to potentional side-channel attacks through publicly available information like gas and contract addresses).

3.1 E-voting System

E-voting systems and their underlying electoral process can attain several formats. These formats are adjusted according to the various contexts in which the elections take place. as each context may have different objectives from the electoral process. Some organizations may focus on transparency, security, and trust, while others might emphasize electoral tradition.

For our analysis evaluation and implementation, we have chosen an e-voting system with an electoral process inherited from standard single-member plurality systems (see Subsection 2.1.4). Its a type of an election system where eligible voters are casting votes for only one preferred candidate, and at the end of the election only one winner will be chosen from all the aspiring candidates. The system will support the organization of multiple elections that can occur at the same time. The user must be enrolled in the system before being able to be chosen to participate in the election. Administrators are responsible for the organization of newly held elections, which includes the following duties (see Figure 3.1):

- Registering new users to the system.
- Assignment of voters eligible to vote and running candidates to the given election, from the enrolled users.

- The creation of the elections and their conclusion of the election, after which no further votes will be accepted, and the results will be counted.

The list of administrators has been initialized by the contract owner (account which has deployed the contract) and can be afterward extended by all the administrators. Initially, we thought of an automatic procedure for ending elections that would end at a particular time in the future. The end date would be set during the election creation along with the start date, ultimately creating a time window during which the system would accept votes from voters. This design has proven to be an obstacle during testing and evaluation, so we decided to simplify the process by having a specific transaction to end the election.

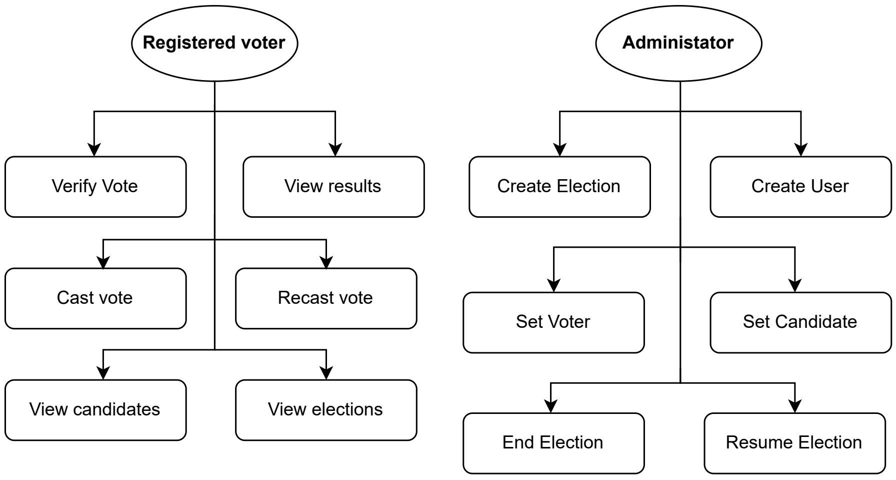


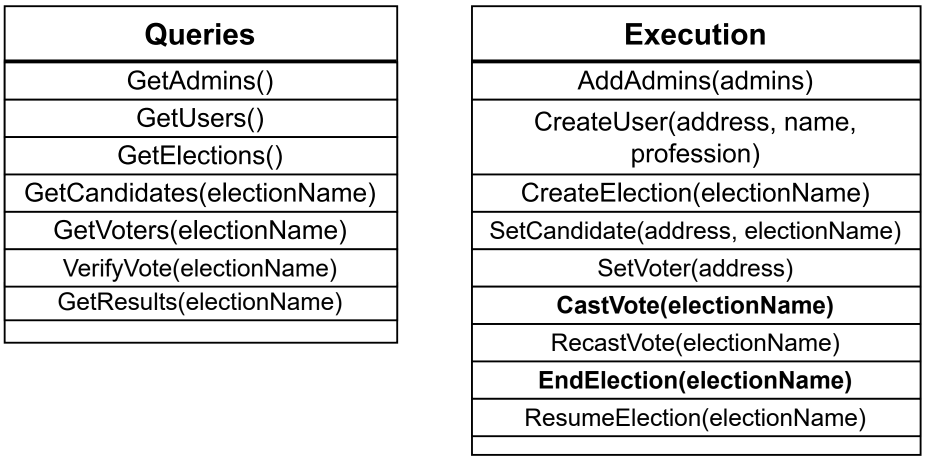
Figure 3.1: Access to methods based on the role in the e-voting system.

As the election occurs over a trusted and confidential blockchain platform, it is desirable that the voters of this system can recast their votes. This acts as a preventive measure against voter intimidation[1], which is one of the benefits that electronic voting brings to the standard process. Furthermore, the confidential blockchain platform facilitates a powerful feature of ballot verification without losing the required secrecy. Only the voter who has casted the given ballot will be able to inspect its contents. After the end of the election the result is queriable to anyone able to query the contract. The format of the result consists of the winner and a vote counts for all the running candidates.

To accommodate all of the functionality, we have proposed a template of methods that will serve the execution and query processing logic of the implemented smart contracts. In total, there are 16 methods (see Figure 3.2), without counting the internal or initialization methods. Most of the methods represent methods used to create or retrieve values of stored structures inside the contract (e.g., user creation and user retrieval). The methods expected to be more complex and integral to the e-voting system are the methods shown in bold, as they are expected to operate with the largest structures (list of votes, voters, or candidates) or they might be called frequently compared to the others. We have evaluated the performance of these methods in the last Chapter 5. These include methods that facilitate functions such as vote casting, vote verification, or the calculation of results, but

[1]A situation when someone puts a voter or group of voters under pressure or force to vote in a particular way [1].

we have also defined a `resume_election` that resumes the already ended election. We have used this function extensively in our evaluation, to repeatedly calculate election results without having to configure the election from the start again(e.g., we evaluated results calculation for different amounts of votes).



Queries
GetAdmins()
GetUsers()
GetElections()
GetCandidates(electionName)
GetVoters(electionName)
VerifyVote(electionName)
GetResults(electionName)

Execution
AddAdmins(admins)
CreateUser(address, name, profession)
CreateElection(electionName)
SetCandidate(address, electionName)
SetVoter(address)
CastVote(electionName)
RecastVote(electionName)
EndElection(electionName)
ResumeElection(electionName)

Figure 3.2: Signatures of methods facilitating the e-voting functionality.

Data storage within the contract is divided into five different entities, starting with a root state that holds all data together (see Figure 3.3 and configuration (*State* table) and remaining four structures will represent a particular entity within the system.

- *User* - Represents enrolled user, that is identified by its unique identifier or address. Additionally, the structure holds meta-information about the user's name and additional info like name or profession. All candidates and voters within the elections will be represented by this structure.
- *Election* - The integral entity of the e-voting system that groups all the participating voters, ballots, results, and candidates together.
- *Vote* - A separated structure that represents a ballot casted by one of the voters. It consists of the address of a voter that casted the ballot and his selected candidate.
- *Results* - Holds the final vote counts for each of the running candidates and the overall winner of the election.

During the initial stages, the representation of ballots/votes was designed in a different way. The ballots or votes were stored in the form of a counter in a structure representing the candidate. The counter was simply incremented for each vote the candidate had received. The main rationale behind this was performance optimization, as the vote-counting process has been distributed to all transactions that cast votes. In the end, we have opted for a separate structure, mainly for vote verification purposes, because for such functionality, a simple counter would not be sufficient, as the relationship between the voter and his selected candidate would not be retained. Also, this design had a privacy motivation as we separated the structure representing confidential information that is accessible only to the owner (voter).

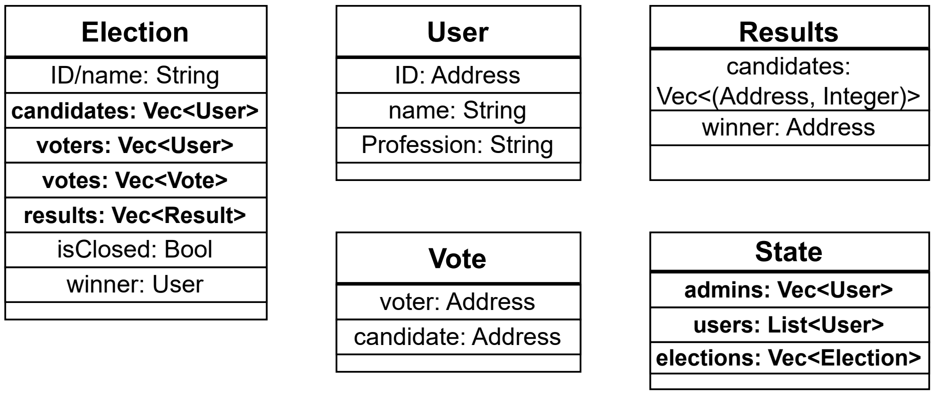


Figure 3.3: Structure of the storage that will be representing the state of the smart contract.

3.2 Secret

The main design choices that we have made on *Secret Network* were the storage options and the permissioned viewing tools. Compared to other platforms, the storage possibilities allowed iteration over map or set structures, which is very useful and not provided on the other evaluated platforms. On the other hand, the underlying system has one disadvantage related to permissioned viewing, which *Secret Network* compensated by providing tools that we had to choose and analyse (we discuss this in Subsection 3.2.2).

3.2.1 Contract Storage

As we have learned in Section 2.3.3 the platform uses persistent key-value storage, where each key is associated with a stored piece of data that can be easily and directly accessed using a storage key lookup. The storage keys are formatted by default as byte arrays, and the values can acquire arbitrary-type, but they have to be serializable to/from binary, as that is the format they are being actually stored in. The elementary value storage types include string, integer, boolean, and the classic array that we know from other programming languages.

Basic storage wrappers

Due to the repetitiveness and inconvenience of unavoidable data serialization/deserialization, a range of wrappers have been defined that make the process of accessing data in storage easier. These include the retrieval, insertion, and deletion methods, but also the corresponding methods that work on top of JSON data[2].

[2]Storage wrappers in Secret Network https://docs.scrt.network/storage-wrappers

```
pub fn save<T: Serialize, S: Storage>(storage: &mut S, key: &[u8], value: &
    T) -> StdResult<()> {
    storage.set(key, &Bincode2::serialize(value)?);
    Ok(())
}
pub fn load<T: Deserialize, S: Storage>(storage: &S, key: &[u8]) ->
    StdResult<T> {
    Bincode2::deserialize(
        &storage
            .get(key)
            .ok_or_else(|| StdError::not_found(type_name::<T>()))?,
    )
}

// Save data to storage
save(&mut deps.storage, KEY, &"Some Data")?;
// Load the data from storage
let config: Config = load(&deps.storage, KEY)?;
```

Listing 3.1: Basic wrappers used to provide easier access to the persistent storage.

These wrappers operating on top of the low-level binary keys and values technically provide efficient data storage and fulfill all of the developers needs. However, practice has shown that usage of these basic wrappers in complex use cases results in cumbersome experience. That is a reason why there is a range of additional storage structures built on top of the basic wrappers. The `cosmwasm_storage`[3] is an original library that defines a set of standard structures that provide an additional layer of functionality on top of the wrappers. Although we haven't used these structures in our final implementation, their comprehension is vital for understanding the underlying architecture. These four are the standard structures defined in this library:

1. *PrefixedStorage* - Foundational storage space that is semantically separated into different sub-stores by their unique prefixes/keys, similarly as its done in *MongoDB* collections or *SQL* tables. As discussed in the Section 2.3.3, the structure is based on *IAVL+* trees that keeps track of all the storage keys and their respective values of various data types. It is possible to define composite keys, which are basically keys composed out of two or more semantically different parts. For example we might create a composite key by creating a *PrefixedStorage* with the first part of the key designating a certain namespace(e.g. passwords), and the second part associates a specific value within that namespace(e.g. users password in the passwords namespace, see Listing 3.2).

```
pub const PREFIX: &[u8] = b"passwords_prefix";
let mut pwd_store = PrefixedStorage::new(PREFIX, &mut deps.storage);
let composite_key: &[u8] = b"sender_address";
save(&mut pwd_store, composite_key, &msg.password)?;
```

Listing 3.2: Demonstration of the PrefixedStorage.

[3]https://docs.rs/cosmwasm-std/storage

Since the prefixes must be unique we cannot construct different sub-store with the already existing prefix, nor we cannot have more than one mutating reference to the underlying store of certain prefix. The object *PrefixedStorage* supports basic types of methods that are used to retrieve, delete, or insert values.

2. *TypedStorage* - Creates a subspace wrapped in a type-aware structure that explicitly declares the datatype stored within the storage space and provides higher-level access methods. The creation of such a structure was based on the fact that each subspace works on a unique type that needs to be serialized/deserialized with each access. This resulted in repetitive processing of the data that could be easily solved with a special structure such as *TypedStorage*. The *TypedStorage* provides the same methods as *PrefixedStorage*, with the exception of the additional `update` method that loads and modifies the stored data at the same time (see Listing 3.3]).

```
pub const PREFIX: &[u8] = b"data_prefix";
let mut storage = PrefixedStorage::new(PREFIX, &mut store);
let mut bucket = typed::<_, Data>(&mut storage);
// save data
let data = Data { name: "Maria".to_string() };
let loaded = bucket.load(b"maria").unwrap();
assert_eq!(data, loaded);
```

Listing 3.3: Demonstration of the TypedStorage.

3. *Bucket* - Provides the same methods and works just as *TypedStorage* with the exception being that it is possible to return the instance of *Bucket* from a function. The previous structures were limited by having references to storage that cannot live longer than the local variable and thus didn't have such ability.

```
fn people<'a, S: Storage>(storage: &'a mut S) -> Bucket<'a, S, Data> {
    bucket(b"people", storage)
}
// save data
people(&mut store).save(b"john", &Data{
    name: "John",
})?;
```

Listing 3.4: Demonstration of the Bucket structure.

4. *Singleton* - Another wrapper around the *TypedStorage* structure that uses only a single storage key for key-value lookups, instead of providing a whole subspace of arbitrary key-values. This is especially usable for cases where we want to store sets of data that will not change over time (for example, the configuration of a contract).

```
let config = singleton(&mut store, b"config");
// save data
config.save(&Data{ name: "Admin" })?;
```

Listing 3.5: Demonstration of the Singleton structure.

The *cosmwasm_storage* library was showcased in various templates[4] in the official Secret Network development documentation. We took this recommendation and used this library in our initial design and development cycles, and we have managed to implement a greater portion of the e-voting functionality. Then we realized that the library is marked as deprecated since the newer versions of *CosmWasm* and that there exists a newer library called `coswamsm_storage_plus`[5]that is supported in the *Secret Network* as well.

Standardized Storage Wrappers

The `coswamsm_storage_plus` library is considered to be a standard storage layer for contract development, although the storage access patterns share many similarities with the ones we have inspected previously. At that time, we used arrays to store various data, like the users or elections. When we inspected the usage of these patterns, we saw that for most use cases there exists a better solution than arrays, which we have used extensively at that point. The problem was that when the size of the data stored in the array becomes large, a problem with efficiency arises, because to save and load a single item within an array, one needs to load the entire array, which logically can lead to an enormous amount of inefficiency (see Figure). The `coswamsm_storage_plus` library defines various structures, but we have identified the following tree as the best suited for our use case:

- *Item* - Similarly to the *Singleton* structure wraps around a single storage key and provides a simple interface for the manipulation of stored data. The difference here is that the structure no longer stores a *Storage* object during initialization (see Listing 3.6). This fact allowed *Item* to be defined as a static global constant that can be processed early and once during the compile time, instead of the repetitive construction in the body of functions, thereby conserving gas expenditure. This property is shared among all the structures defined in this library. In our design we have chosen this structure to hold smart contract configuration, which has small size and needs to be defined only once.

```
// global constant
pub const OWNER: Item<Data> = Item::new(b"owner");
fn foo(storage: &mut Storage) -> Result<> {
    // save data
    OWNER.save(storage, &{ name: "Jozef" })?;
}
```

Listing 3.6: Demonstration of the Item structure.

- *KeyMap* - Hashmap-like structure that stores objects under a typed key and allows iteration over keys and/or values without keeping their order. The typical use case for *KeyMap* is the storage of large amounts of data that could be iterated over in the future, while also performing key lookups. The *KeyMap* might be configured with the disabled iteration feature or with a custom paging size (size of fragments retrieved during iteration).

[4]For example a simple smart contract that works as counter with a frontend part of the application implemented in *Vue.js*. See repository github.com/secret-counter-vuejs-box

[5]https://github.com/cw-storage-plus

- *KeySet* - Stores typed data into a hashset-like structure with the same settings options as *KeyMap*, but the iterator over the structure is not in mutable form. The example usages for *KeySet* are sets of data, where it is necessary to keep track of a membership within the group (access control lists), while not relying on key-value lookups.

We followed up the characteristic comparison of the structures by doing an experiment in which we compared the gas performance in the critical operations of iteration and insertion. These operations will be heavily used in our contract design and may be the probable cause of reaching the block gas limit, which is a reason why these statistics are important to be determined. We have setup the experiment in the following way:

1. We created three structures - array(vector), map (KeyMap) and set (KeySet). Each of the structures holds the same string datatype.
2. We have also created functions of an element insertion, and element search&replace for the given structure.
3. We have iteratively inserted continuously increasing data in the underlying structure and after each insert performed the search&replace function. For each operation performed, we have collected the gas usage statistic and plotted a graph that can be seen in Figure 3.4.

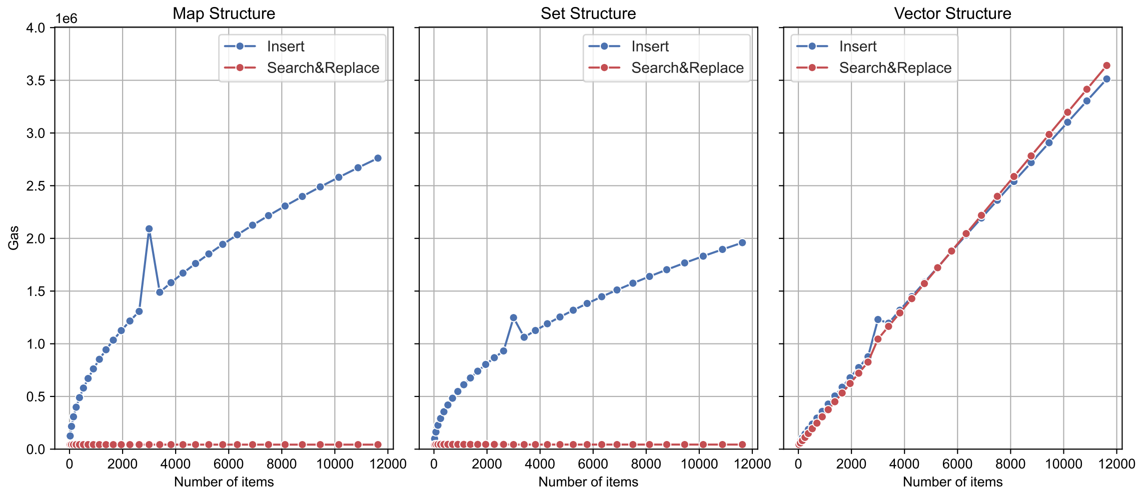


Figure 3.4: Gas usage comparison of certain operations.

This experiment has shown that the array is not an effective storage option, as was expected, but more importantly, we found that it is meaningful to take considerations when deciding which data will be stored within the *KeyMap* or *KeySet* structure. Since the structures provided in `coswamsm_storage_plus` are standardized and have better gas saving abilities, we have opted to use this library exclusively. In the following list, we analyze each entity stored in the contract and propose the best-suited storage structure variant.

- *Users* - A collection of users enrolled in the system are mostly used as an access control to the contract, but also as a source of potential voters and candidates participating in elections. We have chosen *KeyMap* because we use this storage mainly to verify existing users, which is effectively done using this structure. Because we will be doing a lot of lookups for existing users by their address and we want to store a potentially large set of `User` objects, the use of *KeySet* would not be ideal.

- *Elections* - To represent the elections, we have used the structure *KeyMap* with the same reasoning as above in the representation of users.
- *Votes* - They are expected to be of significant size during the electoral process. We have chosen a *KeyMap* for their storage as we need to maintain the relationship between the voter and the candidate effectively. Another reason being that we want to perform lookups for the voters selected candidate, when users verify or retrieve their votes.
- *Candidates* and *Voters* - These two entities are formally a subset of the users that will be mainly used to control access to the authorized operations in a given election. All that is needed is to check whether a given user belongs to the set of candidates or voters, which is a perfect use case for *KeySet*, which we have used to represent these two entities.
- *Results* - Similar design choice as the *Vote* structure as we need to represent a relationship between the candidate and the number of votes he received. To do so, we identified *KeyMap* as a perfect solution, as all that was needed was to map the addresses of the candidates to their count of received votes.

As can be seen in the list above, we haven't really mentioned how we represent the specific substores within the proposed structures(e.g., candidates or voters in a given election). This was due to the fact that all the structures may have another prefix appended that would identify such substore.

3.2.2 Permissioned Access Control

Because the computation is performed securely within the TEE of the network nodes, the blockchain state is encrypted, and the infrastructure of *Cosmos SDK* on its own does not make it possible to cryptographically authenticate the identity of a querier without the ability to control who is able to retrieve data from defined queries, our e-voting system would not be able to distinguish admins from other users, and voters would not be able to verify their votes. To compensate for this problem, *Secret Network* has provided access control management tools in the form of *ViewingKeys* and *Permits* [29].

- *ViewingKey* - It is a signature generated inside a transaction from a random input and the address of its owner, which is a very similar concept to password. The signature gets stored in the state of the contract, together with the public address of the owner. Afterwards a permissioned query requires the viewing key and the public address to retrieve a piece of data owned by the given individual. The passed *ViewingKey* is compared with the one stored in the contract to determine whether the querier is authorized to ask for that piece of data or not. The disadvantage of this tool is that the user must send a transaction before performing the authorized query.
- *Permit* - A successor method of the *ViewingKey*introduced in the SNIP24 specification[6] that increased efficiency and is considered as a viewing permission method

[6]SNIP24 - a new standard for permission viewing, available at https://github.com/SecretFoundation/SNIP24.md

for all access control situations. Its basically a formatted message that provides several arguments such as what exactly the permit applies to and what permissions the permit should allow (e.g., should the permit allow the querier to view the ballot). Permits are not saved in the smart contract state and do not require the initiation of a blockchain transaction. Therefore, permits are a less permanent way to gain viewing access with less network strain.

We have selected *Permit* as access control management in our smart contract, because they do not have to be stored inside the contract and the user is not required to perform a certain transaction to create *ViewingKey*. The queries that require access management are identified in Figure 3.2. In case of message calls that modify the state of the contract, we are able to check their sender address, so we do not have to incorporate the *Permit* logic into those.

3.3 Oasis

The platform has the ability to run independent parallel runtimes (ParaTimes), which enable multiple different computation models on the main *Oasis* blockchain. At the time of writing, the platform had officially supported two EVM-compatible ParaTimes called *Emerald* and *Sapphire*, but also a Wasm-compatible ParaTime named *Cipher*. The following subsection looks at the process of choosing the best *ParaTime* suitable for our development and evaluation. The great benefit that was not present in *Secret Network* was that this platform provided a very easy means to control access management. We have presented this capability and our utilization of it in Subsection 3.3.2. When it comes to other notable design choices, we had to analyze the different concepts brought with *Solidity* language for the storage structure of our contract. All other aspects of the implementation of smart contracts were straightforward and did not require intense design analysis.

3.3.1 ParaTime Selection

Our first steps took the direction of the Wasm-based ParaTime called *Cipher*[7], mainly because their smart contracts are written in the *Rust* language similarly to the other evaluated platforms. Since we have already finished the implementation in *Secret Network*, the use of the same language would allow us to reuse logic and patterns, which would make the implementation much easier. Unfortunately, we have struggled to find code examples that would show us how to implement and deploy a contract in this ParaTime. In addition, the available documentation is very limited, and finding resources on the Web or in *Oasis* community channels like Discord has also not helped. We have not found an SDK / API for developing automated scripts for smart contract deployment, interaction, and evaluation comparable to other platforms, so we ruled out *Cipher* and begin evaluating other options. The only real option left was confidential EVM-based *Sapphire ParaTime*[8] that offers much more extensive documentation and tooling support, not to mention the activity and size of the community. The *Sapphire* ParaTime also provided *Javascript* API for automated deployment that is missing in *Cipher*. The reason why *Sapphire* still remains at the forefront of the attention of *Oasis* staff is that there is a great market value in being the only confidential network with interoperable on-chain privacy compatible with all

[7]https://github.com/oasis/cipher
[8]https://github.com/oasis/sapphire

EVM networks [14]. These are the reasons why we have chosen *Sapphire* ParaTime for our e-voting smart contract implementation.

3.3.2 Confidentiality and Permission Access

The ParaTime provides means of confidential state, end-to-end encryption, and confidential randomness that are easy to integrate within EVM-dapps like *DeFi*, *NFT* or *blockchain gaming*. The smart contracts within *Sapphire* are programmable using EVM-based languages such as *Solidity* or *Vyper*. In addition, the developer can take advantage of development environments like *Hardhat* or *Foundry* that are also popular among developers of *Ethereum* smart contracts.

When it comes to confidentiality, *Sapphire* hides the state of the deployed contracts from everyone except the contract itself, and all transactions sent to ParaTime are encrypted from end-to-end, making it available only to the sender and the contract itself. In *Sapphire* it is possible to select which data will be retrievable and by whom determined by the implemented permission control within the query functions. This is possible since both transactions and queries have the ability to view the `sender` address attached to each call, and the developers are then able to implement the permissions and provide data and secrets to only authenticated callers. Thus, to have an access control management in our contract we did not have to incorporate any special structures, unlike in *Secret Network*, all that should be needed is to conditionally check the `sender` parameter. But since there are four different ways [14] of transaction invocation, we had to take each one into account to be sure about this assumption.

1. *Inter-contract call* - Whenever a contract creates a transaction or query to other contract, the `sender` field is always set accordingly. In this case, it is sufficient to have an access control method based on the `sender` field.

2. *Unauthenticated view call* - These are query calls performed using the `eth_call`[9] RPC method, which by default sets the `sender` parameter to null address(`adress(0)`). Since the null address is a valid address, we cannot allow its use during the registration of a new user. In case we allowed it, all the data viewable to this user would be visible to all unauthenticated queries.

3. *Authenticated view call* - These are queries that are signed by existing users, so the parameter `sender` is correctly set and no special handling is needed.

4. *Transactions* - They are also signed by existing users, so the parameter `sender` is set accordingly, so again no special handling is needed besides the above mentioned.

We also have to keep in mind that Solidity language creates queries or getter functions for all `public` state variables [21]. It is important that all our state variables are set to *private*, so that the access control implemented in the queries cannot be bypassed. Similar caution has to be taken in the function visibility settings. All defined functions that are meant to be used within the contract and come in contact with sensitive data have to be set as `internal` or `private`, so that they are not accessible outside the contract.

[9]https://www.quicknode.com/docs/eth_call

3.3.3 Contract Storage

State variables in the EVM are compactly stored in a single storage slot, which are chunks of data (a size of 32 bytes) that the EVM packs together to perform one large operation (e.g. write to the contracts storage) instead of multiple small ones. This becomes a problem when dealing with mappings and dynamically-sized arrays.

These two structures interact with the data in the memory using offsets calculated by an underlying *keccak256* operation. Each mapping and array access requires such computation under the hood. Since our design will use many instances of the mapping or dynamic array structures, it is important that our design reflects this fact. That is, in cases where we access the same data multiple times in a single scope, it is a good practice to cache the lookup result locally using the *memory* keyword. This is especially important when such lookups are performed in loops. Such a small change can result in significant gas savings.

Similarly to the structure *KeyMap* from *CosmWasm*, *Solidity* has its corresponding structure called *Mapping* that is also used for efficient key value search in larger data collections. *Mappings* are virtually initialized so that every possible key is assigned a default value of the stored data type, while not storing the keys directly (as mentioned previously, the position of the corresponding data is determined by *keccak256* calculation). These are the reasons why by default it is not possible to determine the length or size of *mapping*, nor is it possible to iterate over their keys or values. Since our e-voting smart contract design contains queries and transactions that require iteration, and there is no other effective option besides arrays or a custom version of iterative mapping, we had to opt for the use of arrays. Similarly as in *Secret Network* design, we will bundle key entities and the storage structure we used to store that entity:

- *Users* - A collection of users enrolled in the system are mostly used as to control the access of only enrolled users to the contract, but also as a source of potential voters and candidates participating in elections. We have chosen to use both `mapping` and `array` for storage of users. The `array` was chosen because administrators need to have the ability to retrieve enrolled users, which would not be possible with `mapping` only. The `mapping` is used for efficient lookups, where the key is the address of the existing user.

- *Elections* - Stores all the elections that are taking place within the contract. We will be using both `mapping` and `array` to store them, the reasoning remains the same administrator functionality. In this case, we use the name of the election as the mapping key.

- *Votes* - Compared to *Secret Network* we do not have the option to identify specific election substores by appending prefixes during runtime to underlying structures. We had to solve this during the contract storage structure definition. We have defined two `mapping`:

  - First one is holding array of all the votes under the elections name key. The array is needed for iteration over all the casted votes during results calculation.
  - The second `mapping` is utilized for quick lookups of casted votes(used in vote casting or recasting methods).

- *Candidates* and *Voters* - We have designed an equivalent storage structure for these two entities as in the case of *Votes*, meaning that we represent them by four `mapping` structures.

- *Results* - Similarly as in the previous entities, we define two `mappings`, one for retrieving the list of candidate results and the second one for quick lookups. In addition, we further enhance the results representation by a third `mapping` that holds the winners of the election under an election name key.

3.4 Phala Network

Developers in *Phala Network* have two different ways of building and deploying *Phat contracts*[10]:

1. *Phat Contract 2.0* - allows developers to write *Typescript* script that behave as an *Oracle*[11] while running on a decentralized infrastructure of *Phala Network* with possible confidential data secured by the platform. Its goal is to allow the deployed script to respond to requests from some on-chain smart contract side (e.g. contracts in EVM-compatible blockchains like Ethereum, Polygon, or Arbitrum). The script is free to call any APIs during its processing and respond to the contract side in any format the developer has defined.

2. *Phat Contract Rust SDK* - offers customizable development instruments based on the *Rust* programming language.

We have ruled out option *Phat Contract 2.0*, because *Rust SDK* is a more familiar development form that has many advantages and we are already familiar with it from *Secret Network*. Contracts are written in *Rust*-based smart contract language called *ink!* that is standardly used in all of the *Substrate*[12] based blockchains. Compared to other platforms, we did not have to choose between specific tools to manage access within the contract, as *ink!* automatically supplies the caller or sender address in all messages.

3.4.1 Contract Storage

The storage is organized as a key value database, where keys are arbitrarily long and the values are encoded [41]. In addition to all common *Rust* data types, the standard `ink_prelude` crate defines types like `Balance`, `Hash`, or `AccountId`. We will be using all basic types in our implementation, including the `AccountId` that will identify all registered users in our e-voting system.

The storage API operates with a standard unit called *storage cell* where data entries are stored and loaded using a dedicated key. All data entries that are stored within a single *storage cell* are considered to conform to the *Packed* storage layout [19]. By default, smart contract storage conforms to the *Packed* layout, which means that *ink!* tries to store all the variables defined in the contract storage structure as a single *storage cell*. As a consequence, each message that interacts with the contract storage will always interact with the entirety of the storage structure. This form of storage access is referenced as *eagerly loading*. The problematic nature of this is drawn in Listing 3.7, where each `get_admin` call would require the load of the entire `users` array. Not only does it cause a substantial increase in gas

[10]Atleast at the time of writing it was structured this way, because it seems like the *Phala* team is rebranding to *AI Agent Contracts*, see archive https://web.archive.org/docs.phala.network/phat-contract.

[11]https://chain.link/education/blockchain-oracles

[12]https://docs.substrate.io/

costs, but it can also break the contract as a whole. As in our example Listing 3.7, an ever-growing array will at some point reach the limitation of the *Packed* layout storage capacity of 16KB (maximum size of the buffer used to decode and encode storage items [41]). This is a problem that needs to be solved by converting the storage to *non-Packed* form.

```
#[ink(storage)]
pub struct PackedStorage {
    admin: AccountId,
    users: Vec<User>,
}
#[ink(message)]
pub fn get_admin(&self) -> AccountId {
    self.admin
}
```

Listing 3.7: Ineffective storage structure.

This is the reason why *ink!* data storage incorporated a concept of *lazy loading* that breaks the storage into smaller pieces, which can be loaded on demand. These concepts are provided in *ink_storage* in the form of the following *lazy loaded* storage types:

- `StorageVec` - Its a *lazy loaded* version of the Rusts `Vec`, that allows access to each element individually with theoretical limitation of storing 2^{32} elements. Unfortunately, during experimentation, we have not been able to utilize this structure due to errors tied to the import of the structure and we have not managed to find relevant information about `StorageVec` integration within *Phala Network*.

- `Mapping` - As well as in other frameworks, *ink!* defines a structure similar to hash tables that maps key value pairs directly to the storage. It offers high-level functionality for storing large sets of values, while still being efficient in terms of gas costs and code size. The drawbacks of this structure are that it is not iterable and is not possible to create nested mappings. In addition, the mapping values must be of *Packed* type, which was quite problematic to implement for the custom data structures that we have designed in our smart contract. This must have been reflected in a different design of the storage structure, so that it facilitates the needs we have carved in Subsection 3.2.1.

- `Lazy` - Its a structure that wraps any storage field, which is in turn transformed into a *non-Packed* layout, meaning that it will be stored in a separate *storage cell.* This results in slightly more effective storage management, but this solution is still not perfect for large sets of data (the `Mapping` structure is the right choice).

Application of the *lazy loaded* structure to our previously showcased example would either substitute the `users` with `Mapping` or `StorageVec`, or wrap into the `Lazy` wrapper structure (see Listing 3.8). Sending message `get_admin` would no longer require the load of a *storage cell* containing the ever-growing `users` array.

```
#[ink(storage)]
pub struct NonPackedStorage {
    admin: AccountId,
    users: Lazy<Vec<User>>,
}
#[ink(message)]
pub fn get_admin(&self) -> AccountId {
    self.admin
}
```

Listing 3.8: Storage structure optimized with *Lazy* primitive.

Similarly as in the previous platforms, we list the entities and propose a design of their storage structure:

- Users - For the collection of users we have chosen to use classic `Vec` structure that we wrap into a `Lazy` wrapper to store it in a *non-Packed* layout. The reasoning being that our design requires queries that involve retrieval of the users, or their subsets (candidates, voters). The `mapping` would have to support iteration, which would not be possible with `mapping` only. We have designed a separate `mapping` user storage that maps user addresses as a key to boolean that identifies whether the user was previously registered or not. This mapping is used for efficient lookups in messages not retrieving data (e.g. casting votes) as a form of user access control.

- Elections - We will be using both `array` to satisfy the administrators retrieval and `mapping` as a form of admin access control.

- Voters and Candidates - We have chosen a *mapping* representation that will store a subset of *users* that were enrolled by the administrators. Each entry within *mapping* will be identified by an election and the user address of the candidate/voter.

- Results and Winners - For the chosen format of the results consisting of an undetermined number of candidates and their received vote counts, we will not avoid the use of the *Vec* array structure. Each results array a will be paired with the overall winner address and mapped to name of the given election.

The design above is far from the ideal solution, since we still had to resort to storage of arrays in mappings, which is a problem from two perspectives. The first is the limitation related to decoding and encoding mapping values. In *ink!* there exists a special 16KB buffer that is used to store the encoded data, which is quite limiting, especially when we store a dynamically sized types like arrays. Reaching over this limit traps the contract. The second is tied to the situation where we want to append or insert some data to the existing mapping data. Each such insertion or append requires an insert of the whole value part again, which is very cosly (demonstrated in Listing 3.9).

```
pub fn transfer(&mut self) {
    let caller = self.env().caller();
    // ‘balance‘ is a local value and not a reference to storage!
    let balance = self.balances.get(caller).unwrap_or(0);
    let endowment = self.env().transferred_value();
    // The following line of code would have no effect to the balance of
        the
    // caller stored in contract storage:
    //
    // balance += endowment;
    //
    // Instead, we use the ‘insert‘ function to write it back like so:
    self.balances.insert(caller, &(balance + endowment));
}
```

Listing 3.9: Showcase of how SecretCLI connects and deploys a contract to the LocalSecret container.

Chapter 4

Implementation

This chapter presents details about the practical side behind the implementation, evaluation, and testing of e-voting smart contracts. We also share our development experience and provide details of how to setup the development environment with all the tools required for the process of uploading, deploying, and executing our smart contracts in the given PPP. Because each PPP platform uses a different framework for smart contract development, the development was always distinct, although the use case remained the same. The differences were not only in terms of the language used, but also in other aspects such as the level of documentation, the intuitiveness of the tools provided, or the activity of the community. These factors resulted in various time and effort requirements needed to complete the implementation, testing, deployment, and evaluation. We have used the version control management system *git* throughout the implementation process, resulting in four different repositories for three platforms that we have implemented the smart contract and the repository where we have contained the data processing, visualizations, and evaluation scripts. In the last section, we present the attempted development of a frontend application that aimed to integrate with the implemented smart contracts to demonstrate the potential of privacy-preserving platforms in an easy-to-use Web UI.

4.1 Secret Network

As mentioned previously in Subsection 2.3.3, the smart contracts in *Secret Network and* use the CosmWasm framework built on top of the *Rust* language that is compiled into a *WebAssembly* bytecode Rust is known for its strong memory safety, type safety, impressive tooling support, and ever increasing popularity among programmers. The combination of *WebAssembly* and *Rust* should guarantee optimized run-time performance, which in turn lowers gas costs. The available documentation is on good level, even though we had to dig deeper sometimes to find the information we were looking for, and there were also cases where we had to seek help from the community. Getting help was luckily not a problem as the community around the project is very active. The main remarks regarding the documentation are its rapid changes over time and chaotic organization. In our experience this resulted in that the templates referenced in the initial steps tutorial were not updated and not maintained anymore and also had unclear setup instructions.

4.1.1 Development Environment

The first step in setting up the development environment was to install the stable version of the *Rust* toolchain through its `rustup` toolchain manager[1]. Rust supports a large number of platforms[2], where some receive separate binary releases of the standard library, while others receive the full compiler. The `rustup` toolchain manager gives easy access to all of them. When we installed the Rust toolchain, `rustup` automatically installed the standard library for our host platform (the architecture and operating system of our system). Since we compiled the smart contracts into *WebAssembly*, it was necessary to install another target platform, for *Secret Network* and *CosmWasm* the expected target is `wasm32-unknown-unknown`. This identifier is known as *target triple* and consists of three strings separated by hyphens that represent the architecture, the vendor, and the operating system. The "wasm3" part means that the compilation will result in a *WebAssembly* binary that uses a 32-bit large address space. The remaining two „unknown" parts of the triplet mean that there is no limitation on the machine that is being compiled and run on. Additionally, we installed a developer tool `cargo-generate`, which helped us to quickly create a new folder structure of a new Rust project using a pre-existing git repository as a template.

4.1.2 Secret tools

Secret Network provides a command-line interface tool `SecretCLI`(Secret Network Light Client) and library `secretjs` that provide means of interaction with the Secret Network blockchain. We have used these tools to send transactions and queries to deployed smart contracts, to generate and manage user keys/wallets, or to create cryptographic signatures. We used the `SecretCLI` tool mainly with our first experimentation with contract deployment and execution, but it proved to be a good companion even in later stages. The `secretjs` on the other hand was helpful throughout the whole process, mainly due to its scripting capabilities which I will present in the following sections.

```
user@linux-MS-7B89:~$ secretcli config node http://localhost:26657
user@linux-MS-7B89:~$ secretcli config chain-id secretdev-1
user@linux-MS-7B89:~$ secretcli config keyring-backend test
user@linux-MS-7B89:~$ secretcli config output json
user@linux-MS-7B89:~$ secretcli tx compute store contract.wasm.gz --gas
    5000000 --from a --chain-id secretdev-1
user@linux-MS-7B89:~$ secretcli query compute list-code
[
    {
        "code_id": 1,
        "creator": "secret16u7w28vp68qmldffuc89am4f02045zlfsjht90",
        "code_hash": "2658699cea61120ac4411cdf1c05cdac3deceba8de0f6ce026d"
    }
]
```

Listing 4.1: Showcase of how SecretCLI connects and deploys a contract to the LocalSecret container.

[1] Step by step tutorial on the setup of the environment is published on official documentation of Secret Networkhttps://docs.scrt.network/set-up-env.

[2] Platform Support in the official documentation is available at https://doc.rust.org/platform-support.

During development, we extensively relied on a *Docker* container called *LocalSecret*, which is basically a containerized Secret Network testnet and pre-configured ecosystem with four accounts `a`, `b`, `c` and `d` and one validator. The requirements to run the container are approximately 2.5 GB of RAM, and that is the only real limitation besides the installation of Docker. This container provided us with the ability to deploy our contract on our local machine without the need to rely on availability of the testnet or the costs tied to the mainnet, and most importantly, we could test out the contracts in a sandbox environment.

4.1.3 Folder Structure

We have used the official template “Simple Counter Example” for Secret Network smart contracts with pre-configured packages, a build process and a predefined conventional file hierarchy. We have kept the hierarchy almost identical with couple of exceptions in the contract source files:

```
/
|
|__ node/
|   |__ queries_localsecret.js
|   |__ queries_testnet.js
|   |__ deploy_instantiate.js
|   |__ evaluation1.js
|   |__ utils.js
|__ schema
|   |__ execute_msg.json
|   |__ query_msg.json
|   |__ instantiate_msg.json
|__ src
|   |__ error.rs
|   |__ contract.rs
|   |__ msg.rs
|   |__ lib.rs
|   |__ state.rs
|   |__ utils.rs
|   |__ tests.rs
|__ Cargo.toml
|__ Makefile
```

The `schema` directory contains JSON schema files generated by the `cosmwasm_schema` and `schemars` packages. These files serve the purpose of validation and API documentation, as each JSON file represents a supported message that the contract expects. The `src` folder is the home of the contract code itself. The conventional structure of the contract's file tree in Secret Network consists of:

- `contract.rs` - Integral file that serves the incoming messages with core functional logic of the smart contract and tests.
- `state.rs` - Holds the representation and structures of the smart contract's long-term storage, with their constructors and methods.

- `msg.rs` - This module holds various structures that denote different types of entry point messages and the response formats of our query methods. We have organized the messages into three foundational blocks:
  - `InstantiateMsg` - Is a type of message that is required to be passed within the contract instantiation. It has a single `admins` property, which holds the array of administrators in the contract.
  - `QueryMsg` - Formulates the set of expected message formats for data retrieval methods that corresponds to Figure 3.2. Each entry also specifies the type of the returned value, by setting the `#[cw_serde]` and `#[returns(ResponseType)]` macros. There is a special case of the message format, defined as `WithPermit` that is unique to *Secret Network* smart contracts. This type distinguishes authenticated queries, as they require additional parameter of the `Permit` type that is used for the management of the access control (explained in Subsection 3.2.2).
  - `ExecuteMsg` - Similarly as `QueryMsg` defines set of message formats required in the transactional methods.
- `lib.rs` - Exports all of the modules defined in the „src" directory, to make the modules visible and accessible between each other.

We have extended this structure with `error.rs`, `utils.rs` and `tests.rs`. The reasoning was better readability and modularity of the custom errors, tests, and helper functions. The last "node" directory is composed of *Javascript* scripts, for automated deployment of the contract to either testnet or local docker instance and other scripts that interact with the deployed contract via message calls.

4.1.4 Smart contract structure

CosmWasm follows the *actor model* design pattern, where *actors* are individual entities that communicate by exchanging messages. *CosmWasm* denotes these messages as a special type of message called *entry-point*, which serves the same purpose as the `main()` functions do in most native programming languages. The exception being that smart contracts could have more than one entry point function. Its up to the contract implementation how many and which type of entry points will be defined to accommodate the contract interaction. Developers can choose between three types of entry point messages:

1. `contract::instantiate` - Holds *initialization* logic that is executed once, after the Secret contract gets deployed to the network. In our contract, it is used to set up the initial state configuration and assigns the sender of the *Instantiate* message as the initial user and administrator of the contract. The account that deploys the contract has the option to pass a list of admin accounts, the contract should store in addition to his address.

2. `contract::execute` - Bundles all transactional logic that modifies the contract state. In our implementation, we have several methods of this type, all of them correspond to the „Execute" table identified in the Figure 3.2.

3. `contract::query` - Collection of read-only functions that provide predefined answers of some collection in the stored state, without any state modification. Similarly, as in the `contract::execute` entry point, all the implemented methods are corresponding to the "Queries" table in Figure 3.2.

To make the entry points distinguishable for the *Wasm* run-time from other functions, they are decorated with a special attribute `#[entry_point]`. The purpose of the decorator is to make the function distinguishable for the *Wasm* runtime. The raw *Wasm* entry points support only the basic data types natively supported in the *Wasm* specification. Since we are working in *Rust*, we want to have the entry-points support advanced structures and enums defined in Rust. The macro solved this problem by calling an internal function that does all the magic of creating high-level structures. These entry-point functions have by default four arguments:

- `DepsMut/Deps` - Contains functions responsible for querying and updating the current contract state, querying other contracts, and provides an Api object with helper functions for dealing with CosmWasm addresses (e.g. address validation, signature verification). The Deps object represents a non-mutating version of DepsMut. This Deps object is usually used in queries, as these can never alter the internal state of the contract.

- `Env` - Represents the state of the blockchain at the time of the execution of the message. It contains metadata like the height of the chain, chain's id, current timestamp, and address of the called contract.

- `MessageInfo` - contains metadata about the sent message that triggered the entry point serving the transactional logic. The metadata consists of essential information for authorization: the address that has sent the message and payment in the form of chain native tokens sent with the message. This metadata has implication in the access control, even though this object is not available inside of query methods, where other techniques have to be used.

- `msg` - By standard configuration these messages hold an `Empty` type (equivalent to empty JSON object), but it can be modified to any structure or enum that implements serialize trait. Based on this argument, the following order of operations is followed. We have created three types of message types, each corresponding to different types of entry point, that are defined in the `msg.rs` module (see message types in Subsection 4.1.3).

4.1.5 Storage Structure

During the iterative development approach, storage was the most frequently altered part of the implementation. During our initial development, we relied on the `secret_storage`[3] package and its *Singleton* structures together with the arrays. For example, our state representation (visible in Listing 4.2) used the `Singleton` structure for state storage, which contained arrays for elections, users, and administrators. The state was accessible by a common pattern, consisting of two functions `config` and `config_read` that retrieved a mutable or immutable version of the state with the predefined key/prefix.

[3]https://crates.io/crates/cosmwasm-storage

```
pub struct State {
    pub owner: Addr,
    pub users: Vec<User>,
    pub elections: Vec<Election>,
    pub admins: Vec<Addr>,
}
pub const CONFIG_KEY: = Item::new(b"state");

pub fn config(storage: &mut dyn Storage) -> Singleton<State> {
    singleton(storage, CONFIG_KEY)
}

pub fn config_read(storage: &dyn Storage) -> ReadonlySingleton<State> {
    singleton_read(storage, CONFIG_KEY)
}
```

Listing 4.2: Initial state representation.

After our analysis described in Subsection 3.2.1, we have changed the structure to use the structures defined in the `cosmwasm_storage_plus` library. This library is not directly compatible with *Secret Network*, but luckily there is a `secret_toolkit` package that provides range of common tools used in development, including storage structures mentioned previously. After incorporating the structures defined in this library, our final implementation has completely removed the use of arrays, and instead relied on the `Keymap` and `Keyset` structures (see Listing 4.3). In similar manner we have implemented all of the structures we have chosen to use in our design (see Figure 3.2.1) to represent the contracts state.

```
pub struct State {
    pub owner: Addr,
}
pub const STATE: Item<State> = Item::new(b"state");
pub const ELECTIONS: Keymap<String, Election> = Keymap::new(b"elections");
pub const USERS: Keymap<Addr, User> = Keymap::new(b"users");
pub const ADMINS: Keyset<Addr> = Keyset::new(b"admins");
```

Listing 4.3: Restructed state representation in the final implementation.

4.1.6 Access Control Management

To control who has access, it is necessary to have means of authentication of the users who send messages to the contract. The methods used to control who is interacting with our contract depend on whether its a query or a transaction. In terms of transactions, the solution is simple, as one of the arguments to these methods contain an `MessageInfo` object that stores information about the identity of the sender. In our `utils.rs` module, we have implemented a few helper functions that are being used in conjunction with this information(`user_exists`, `validated_address` or `admin_exists`) to authenticate the sender with information stored in the contract.

```
// check repetitive users
if user_exists(&deps, &info.sender){
    return Err(ContractError::RepetitiveUser {
        user: validated_addr,
    });
}
```

Listing 4.4: Example how we utilize the sender identity to control access.

On the other hand, within the query methods we had to incorporate `Permits` structure to have the ability to check the queries identity (see Subsection 3.2.2). We already mentioned once that we use the `WithPermit` query message type to identify permissioned queries that require sending a special object called `Permit` in its arguments. The *Permit* object within these queries can be verified with a `validate` function, that verifies the signatures and the public key contained inside of it to determine who is the sender of the query message call. In this way, we can manage who have access within the queries to data stored in contract state. We have implemented this functionality for `get_admins` and `get_vote` functions only, as further incorporation made the evaluation process more difficult (we would have to create signatures and permits for each query call).

4.1.7 Testing

We have implemented unit tests for each of our key e-voting functionalities using the library `cosmwasm_std::testing`. This library provides methods for mockup objects creation required to pass in the arguments of the contract calls (objects described in Subsection 4.1.4). By mocking these objects, we are able to define the admin of the contract and senders of each transaction, thus enabling us to perform all the administrator functionality, without running into the access restriction problems. We have structured the tests into seven different test cases:

1. `tests::proper_initialization` - Tests the instantiation call used to initialize the contract. It does so by sending the `instantiate` transaction and querying the administrators with the `GetAdmins` message to see if the contract was successfully instantiated and if the correct administrators were set up.

2. `tests::user_creation` - Tests the user creation, by sending a `CreateUser` transaction and verifying the correct execution by querying the users with the `GetUsers` message and comparing the response to expected values. For the mockup user creation we have implemented a `tests::create_mockup_user` function, that is used throughout the test cases.

3. `tests::election_creation` - Another simple test case that verifies the functionality of election creation.

4. `tests::candidate_assign` - We perform the same as in the previous test case to create the user that will become the candidate. But before we nominate a candidate, we have to create an election by sending a `CreateElection` message. After checking the success of the election creation, we are able to set the created user as a candidate running in this election and also verify this action.

5. `tests::assign_voter_to_election` - Almost equivalent to the previous test case, with the exception that the user is assigned to be the voter in the create election, instead of candidate.

6. `tests::casting_votes` - Consists of four different users that are enrolled in the contract, where half of them will become candidates and the other half voters in a single election. Both voters will give their vote to a different candidate with the `CastVote` message. Afterwards, one of the voters will change its mind and recast their vote with `RecastVote` message. Lastly, the election is ended, and the results of the election are retrieved and checked if the numbers add up and the winner is determined correctly.

7. `tests::permit_ballot_query` - Sets up an election with one voter and one candidate, where the voter casts vote to the only candidate, and then he tries to retrieve and verify his vote. Verification of votes is one of the authenticated query calls that are required to pass a `Permit` object besides the standard arguments. We have created `Permit` with the help of `secretcli` and its `tx sign-doc` command. This command signs the provided file, in our case a JSON representation of the Permit with the provided address(we have given the address we are calling the contract with in the tests).

Each of the test cases required the mockup objects, set an administrator with valid address and instantiate the contract. We have implemented a function `init_things` that does all this initialization, and every test case then calls this function to set everything up.

4.1.8 Statistics Collection

We have implemented scripts that deploy and interact with the contract on *Testnet* or *LocalSecret* (locally running dockerized instance) and collect statistics that will be compared to the other platforms. These scripts were written in the *Javascript* language while heavily relying on the `secretjs` library for connection to *Secret Network*. We are going to explain the role of each script in the following list:

- `deploy_instantiate.js` - Deploys and instantiates the contract using the wallet instance generated by the mnemonic stored in the `.env` file via the `SecretNetworkClient` (see Listing 4.5) and the `contract.wasm.gz` file containing the contract compilation result. The client provides methods for deploying the contracts and sending queries or transactions to the deployed contract. In this script, we deploy and instantiate the contract using the `storeCode` and `instantiateContract` methods from the client. These methods return `codeId`, `codeHash` and `codeAddress` of the deployed contract. We store all of this information in the `.env` file using the `dotenv` package, so that the other scripts can then load this information and use it to interact with the contract.

- `utils.js` - Defines two helper methods:
  - `howLong` - This function wraps some of our contract API method with its expected arguments. The method is executed and the results are processed to retrieve the information about the gas and time expenditure required to perform that message call. These statistics are stored in the predefined location.
  - `updateEnvContent` - Used to update or create new *.env* variables.

- `queries_localsecret.js` - Represents an API to our contract, consisting of all possible message calls(both queries and transactions) to our contract deployed to the locally running *LocalSecret* instance.
- `queries_testnet.js` - Equivalent of `queries_localsecret.js`, but the configuration is configured to connect to the *pulsar-3* testnet network.
- `evaluation.js` - Each of the defined methods checks for possible errors and retrieves information about the message call performed (such as gas and execution time) and stores it in a file using the `howLong` wrapper.

```
// create wallet from mnemonic loaded from .env file
const wallet = new Wallet(process.env.A_ACCOUNT);
// connect to the Secret Network
const secretjs = new SecretNetworkClient({
    chainId: "secretdev-1", // "pulsar-3",
    url: "http://localhost:1317/", // "https://api.pulsar.scrttestnet.com",
    wallet: wallet,
    walletAddress: wallet.address,
});
```

Listing 4.5: Connection to the *Secret Network* using the *secretjs* methods.

4.2 Oasis Network

The smart contract in *Oasis Network* was implemented using *Sapphire ParaTime* and its *Solidity* programming language. We have also taken advantage of the supported development environment, in the form of *Hardhat*. The popularity of *Ethereum* has resulted in exceptional documentation and coverage of both *Hardhat* and *Solidity*, but the level of documentation of *Sapphire ParaTime* is also not lacking. This resulted in a comfortable development experience that was on another level compared to the other platforms. Finding the information we were looking for during development was never the problem. The only limitations that we have observed are the lack of configuration options when it comes to data storage structure and the lower flexibility of the contract size.

4.2.1 Development Environment

To setup the environment all is needed is to install tools like `sapphire-hardhat` npm package, the *Hardhat* environment, together with its dependencies (e.g. `@nomicfoundation / hardhat toolbox`). These can be afterwards uses to create and initialize a new project using a single command `npx hardhat@ 2.19.2 init`. We had to create our own wallet and charge it with tokens that were needed for the deployment of our contract(either buying ROSA token or using the faucet for testnet tokens). Afterwards, the only thing that separated us from deploying our first smart contract to *Sapphire ParaTime* was to change the *Hardhat* configuration settings. These changes included an import of the installed `sapphire-hardhat` npm package and the definition of the network parameter in the `HardhatUserConfig`, so it points to the *Sapphire* mainnet or testnet. In addition to this, we have initialized an empty *git* repository and created a *.env* file that we will be using to store our wallet credentials and the addresses of deployed contracts.

4.2.2 Folder structure

Based on the automatized project setup done by the *Hardhat* tool, we configured the file hierarchy into the following form:

```
/
├── artifacts/
│   ├── build-info
│   └── contracts/
├── contracts/
│   └── EvotingOasis.sol
├── test/
│   └── EvotingOasis.ts
├── scripts/
│   ├── deploy.ts
│   ├── evaluation.ts
│   └── utils.ts
├── evaluation-evoting/
│   └── oasisdata/
├── hardhat.config.ts
├── Makefile
├── package.json
├── .env
└── tsconfig.json
```

The `artifacts` directory is used to store the compiled contracts with all related build information (such as the compiler version or performed optimizations). The `contracts` and `tests` folders hold two files that contain the implemented contract code and its unit tests. The evaluation and data collection scripts are structured as three different files inside the :„scripts" directory, we will pay more attention to these in Subsection 4.2.7. These scripts collect data that are stored in the `evaluation-evoting` directory, which is a *git submodule* of our evaluation repository. In addition to this, the project contains automation, configuration, and package management files like `Makefile` or `hardhat.config.ts`.

4.2.3 Smart Contract Structure

Contracts in Solidity follow the same pattern as classes in object-oriented languages. They contain persistent data in state variables and functions that can modify or retrieve these variables [21]. Each contract has a `constructor` method that is called when the contract is created. After its execution, the contract will be added to the network. Analogously to `instantiate` in *Secret Network*, this function is used to configure the contract state and initialize the administrators list.

```
// Administrators list query call
function getAdmins() public view onlyAdmin returns (address[] memory) {
    return admins;
}
```

Listing 4.6: Showcase of the keywords used on one of the query functions.

The query and write message calls are differentiated by the keyword used in the function definition that sets the visibility. We define all functions that can be called via message calls (methods identified in Figure 3.2) to be `public`. The query subset has an additional `view` keyword that affirms that the function is not modifying the state simply reading from it. We have also defined `pure` function `areStringsEqual` that compares two strings with their hashes generated by a *keccak256* operation on top of them. We use this function extensively, mainly tied to functions involving elections, as the string comparison of the elections name uniquely distinguishes it from other elections.

4.2.4 Storage Structure

During the implementation, we have not changed the storage design and it remains the same as we have presented in Subsection 3.3.3. In our design, we reference nested mappings, but to show how it looks in code, we provide a following Listing 4.7. Here we can see that the `Vote` structure is contained in two mappings, as we have also described in the design.

```
struct Vote {
    address voterAddr;
    address selectedCandidate;
}
mapping(string => Vote[]) private votesArray;
// election name maps to the votes casted mapping that
// maps with voter address to selected candidate address
mapping(string => mapping(address => address)) private votesMap;
```

Listing 4.7: Showcase of the storage structure representing casted votes.

4.2.5 Access Control Management

The foundational prerequisite to having any access control within the contract was that we needed to set all state variables to be `private`. If we did not do so, the compiler would automatically create an externally accessible *getter* function for each `public` state variable.

Since we can verify the identity of the sender in all possible message calls (see Subsection 3.3.2) through a global variable `msg.sender`. Using this variable, we can decide who has access to confidential data stored in the contract. The *Solidity* language provides a concept of a function *modifiers* that is a declarative way of specifying a condition that must be met prior to the execution of the function. We have defined the `onlyAdmin` modifier that checks if the identity of the sender stored in the `msg.sender` variable matches with some of the stored administrators. Its a form of access control that we have used throughout the functions performing the administrator functionality (e.g. user or election management).

In addition to the `onlyAdmin` modifier, we also control access by checking the `msg.sender` variable using the `require` functions. These functions take two arguments, the first one being the condition where we compare the sender's identity, and the second one a description of the reason why the `require` condition has not been satisfied, which will be used to revert the function. We have used such `require` statements in the `castVote`, `recastVote` and `getVote` functions.

4.2.6 Testing

The use of the *hardhat* development environment also simplified the implementation of unit tests for our contract. It provides a helpful *Ethers.js* library that has many methods that allow interaction with EVM-based blockchains, including the *Hardhat Network* that simulates real-world blockchain networks for our test cases. We have also used a *Mocha* or `chai`[4] library to run and control our tests with series of `expect` function calls.

```
const { expect } = require("chai");
describe("Sample contract", function () {
  it("Deployment should assign tokens to the owner", async function () {
    const [owner] = await ethers.getSigners();
    const contractInstance = await ethers.deployContract("Sample");
    const retrievedOwner = await contractInstance.getOwner();
    expect(retrievedOwner).to.equal(owner);
  });
});
```

Listing 4.8: Showcase of contract deployment and interaction using the *Ethers.js* library.

On the example above we demonstrate a simple test case in the `Sample Contract` module where we retrieve the default account `owner` and deploy the contract. Since the `owner` is used by default to send message calls inside the simulated network, its address will be configured during contract deployment and instantiation (we assume that during the instantiation this state field is initialized). Then we retrieve the owners of that contract and compare them using the *Mocha* function `expect` to determine whether everything went well. This was a snippet of how we have implemented all of our test cases, that are almost equivalent to the test cases presented in the Subsection 4.1.7 and we will not go over each of them again.

4.2.7 Statistics Collection

Similarly as in the Subsection 4.2.6 above we take advantage of the *hardhat* development environment and its *Ethers.js* library to collect statistics of gas usage within the contract. The deployment of the contract within the script was managed by the `Contract Factory` object that was retrieved with the `getContractFactory` method. In case we have already deployed the contract, we retrieve the contract instance with the `getContractAt` method. The instance of the contract can later be used to send transaction and query calls. The collection of statistics was performed within the `evaluation.ts` *Javascript* script, which not only manages the deployment of the contract, but also sends transactions that simulate an ongoing election in the deployed contract. During execution, the scripts collect and store statistics from performed transactions, using helper methods defined in the `utils.ts` script file.

```
// Create Contract Factory object
const EvotingOasis = await ethers.getContractFactory("EvotingOasis");
let evotingOasis = await EvotingOasis.deploy();
evotingOasis.createElection("Volby Prezidenta 2024");
```

Listing 4.9: Showcase of contract deployment and interaction using the *Ethers.js* library.

The helper methods are responsible for statistical collection, reading, and writing to and from *.env* file, and generating dummy data. The results are stored inside *csv* files that are

[4]Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, look at https://mochajs.org/

structured into two columns: execution time and gas spent. Each *csv* file is named after the message call of which the file is used to store statistics. The statistics are configured to be stored within the `evaluation-evoting` directory, which is an imported *git submodule* from our statistics evaluation repository.

4.3 Phala Network

The standard computation program within *Phala Network* is instead of smart contracts referenced as *Phat Contract* [41]. These contracts are written in the Rust-based *ink!* smart contract programming language, which is used by default in all *Substrate*[5] based blockchains and a default programming language in the *Polkadot* ecosystem. The *ink!* is an embedded domain-specific language (eDSL) that empowers all the benefits provided in *Rust* and extends it with various macros to customize and transform the *Rust* structures into smart contract elements. The language is popular among developers for its ease of use, extensive preliminary testing, and frequent updates. Its capabilities, correctness, and efficiency have been proven in a large number of NFT marketplace projects, but there are some flaws the developers have to endure, such as problematic error handling, high commitment to macro-specific logic, or poor extensibility[6].

4.3.1 Development Environment

Apparent prerequisite for the development of *Phala* smart contracts is the installation of *Rust* and its `cargo` package manager and `rustup` toolchain manager. These can be installed from available online sources and also via the provided `install-rust` *Makefile* rule that does it automatically. To develop and manage *ink!* smart contracts, we needed an additional installation of a `cargo-contract` CLI tool that provides a set of commands for compiling, deploying or initializing new *ink!* smart contracts projects.

The *Phala Network* had the option of running a local testnet network, but it required running a full stack of the core blockchain and its connection to the Web UI, and there is no *docker* alternative. We have chosen to test out the deployment of our contracts on the `Phala PoC6`[7] testnet network directly, as all we needed is an existing wallet and testnet tokens. To create a wallet, we have used `polkadot.js` extension for *Google Chrome* browsers and for the acquisition of testnet tokens the *Phala Testnet faucet*[8].

4.3.2 Folder Structure

We have based our project on the `phat-hello`[9] template that establishes a very basic folder structure where the smart contract code is contained within a single `lib.rs`. We have extended the smart contract code base with one more file, where we have separated our custom data structures definitions like `Election` and `User`. The contracts compiled together with other artifacts generated during the compilation process are stored inside the `target` folder. Similarly as in previous two platforms we have imported the *git* submodule

[5]Substrate is an open source framework that provides tools to build future proof blockchains optimized for various use cases. https://substrate.io/

[6]Good write up about these was put up by Oleksandr Mykhailenko in the article available at https://4irelabs.com/top-5-challenges-with-ink/

[7]https://phat.phala.network/

[8]https://phala.network/faucet

[9]https://github.com/phala/phat-hello

of our evaluation repository `evaluation-evoting` where we will store all the collected data. The scripts performing the election simulation and data collection within a deployed smart contract are located inside the `node` folder.

```
/
├── target/
│   └── ink
├── node
│   ├── .env
│   ├── package.json
│   ├── deploy.js
│   ├── logger.js
│   ├── evaluation.js
│   └── utils.js
├── evaluation-evoting/
├── lib.rs
├── data.rs
├── Makefile
└── Cargo.toml
```

4.3.3 Smart Contract Structure

Similarly to *CosmWasm* any blockchain built on *Substrate* has the option to add smart contract support to its blockchain by adding *Contract pallet*. There are two possible types of selected smart contract type, either *WebAssembly* or *EVM-compatible*. In *Phala Network* the type of smart contracts is WebAssembly (Wasm) similar to in *Secret Network*, but compared to *Cosmwasm* the architecture has notable differences (we discuss these in Subsection 4.3.3). The messages in the *ink!* contract do not require predefined *JSON* schemas for the message structure. Instead, they heavily rely on the use of *Rust* macros. The standard macros are as follows:

- `#[ink::contract]` - Annotates modules that implement *ink!* smart contracts to perform analysis on the written smart contract code and generates optimalized code.
- `#[ink(constructor)]` - Equivalently to *CosmWasm* `instantiate` message is a macro that denotes a function called when the contract is deployed, which sets the contract state by initializing all the structures that represent the storage, including the list of administrator addresses in our case.
- `#[ink(storage)]` - It is used to mark the structure representing the storage and the internal state of contracts.
- `#[ink(message)]` - Annotates functions exposed in the contract interface that behave equivalently to the `query` and `execute` messages in the *CosmWasm* framework. All the designed methods that facilitate the e-voting system (see Figure 3.2) will be marked with this macro.
- `#[ink::test]` - Annotates a separate test case functions within the `tests` module.

The *ink!* language exposes a several handy environment functions that for example return the address of the contract, the sender of the message, or the current block number. Depending on the type of message, these functions are accessed by either the `Self::env()`

or `self.env()`. We have used the `self.env().caller()` function to retrieve the address of a message sender, which is used to manage and control user access within the contract. This can be acquired in all of the messages defined in the contract interface. As a result, we were easily able to restrict who can perform certain operations or retrieve data.

```
#[ink(storage)]
pub struct SampleContract {
    admins: Mapping<AccountId, bool>,
}
impl SampleContract {
    #[ink(constructor)]
    pub fn new() -> Self {
        Self { admins: Mapping::new(Self::env().caller()) }
    }
    #[ink(message)]
    pub fn admin_operation(&self, some_data: Data) -> Result<Bool> {
        if(self.admins.contains(Self::env().caller())){
            // authorized implementation
        }
        Ok(True)
    }
}
```

Listing 4.10: Basic contract structure consisting of storage, its initialization and a query.

In the Listing 4.10 we demonstrate the use of most of the macros we have mentioned previously in a very simple contract consisting of a single `admins` state field, its initialization, and some function that is part of the contract message interface. In addition, we highlight the use of `Self::env().caller()` helper function to restrict access within the contract message only to authorized users.

4.3.4 Contract Storage

To reflect and satisfy our designed structure in Section 3.4.1 we had to make some adjustments during implementation. Mostly, when it comes to the representation of a relationship between certain elections and its entities. In previous platforms, there was the possibility of creating nested `mapping` or appending suffixes that we have used to identify a substore for specific elections (e.g., substore within candidate addresses `mapping` belonging to given elections). Although the *ink!* programming language provides `mapping` structure, these are unable to have another nested `mapping` within them.

```
let votes_mapping: Mapping<(String, AccountId), AccountId>;
// get the vote from the votes mapping -(voter, candidate) pair
let vote_key = (election_name.clone(), validated_voter_addr);
votes_mapping.get(vote_key)
```

Listing 4.11: Showcase of nested mapping.

We had overcome this problem by explicitly defining the keys of *mapping* as tuples, where the first part of the tuple identifies the substore and the second a specific value within that mapping (see Listing 4.11). In this way, we have implemented all the mappings identified in Section 3.4.1 and the other structures remain unchanged.

4.3.5 Testing

The *ink!* makes it possible to create test case functions that can be performed before deploying the contracts to one of the networks. Similarly as in smart contract structures, even tests have their own specific macros in *ink!*. The module that contains the test case functions is marked with a `#[cfg(test)]` macro and each test case with `#[ink::test]`. These macros let the special *ink!* compiler know that these test cases have to be executed in a simulated blockchain environment [41]. There are several functions that provide customization or retrieval of information from the simulated blockchain environment(e.g., topping up balance in certain account or creating new accounts).

In our unit tests, we have chosen a default initialization of the mock-up blockchain and only work with default test accounts, as they were guaranteed to be valid in the simulated blockchain. All unit tests were contained in a single function `main_functionality` and they are mostly equivalent to the test cases described in Subsection 4.1.7.

```
let mut evoting_contract = EvotingContract::new();
// retrieve default test accounts
let accounts =
ink::env::test::default_accounts::<ink::env::DefaultEnvironment>();
// Alice is the default account in tests - so she is the owner who
// instantiated the contract, thus an admin as well
let admins = evoting_contract.get_admins().unwrap();
assert!(admins.contains(&accounts.alice));
```

Listing 4.12: Contract initialization in tests and interaction with its instance.

4.3.6 Statistics Collection

To write the scripts that collect statistics, we have used the `@phala/sdk` library that builds on the `@polkadot` package. We have performed data collection on the testnet network. The implementation process was quite complex and required several steps:

1. Configure *WebSocket Provider* - This object allows sending requests using WebSocket to a TCP port of the Phala WebSocket RPC server `wss://poc6.phala.network/ws`.

2. Creation of the *Phala Registry* object - It is initialized with `ApiPromise` created from the *WebSocket Provider*.

3. *Keyring Pair* generation - The keyring is generated from a mnemonic of an existing account registered in the testnet network.

4. Upload and instantiate the contract - Previously created *Phala Registry* object is together with the ABI, metadata of the compiled contract, and the *Keyring* of our testnet account is passed into the `PinkCodePromise` object that will be used to deploy and instantiate the contract. The instantiated result `PinkContractPromise` will be our interface to the contract message calls.

5. Perform evaluation scenario - It consists of message calls via the `PinkContractPromise` object just as we defined in the evaluation design in Section 5.1.

We were unable to collect the statistics from the response of our message calls through the `PinkContractPromise` object. We have found a workaround in *Phala SDK Cookbook*[10], where a script `tail.js` uses a `PinkLogger` to collect all the metadata about the transactions performed on the configured network. We have configured this script file to listen to messages performed in our contract and the testnet.

There was one last problem related to the metadata collected from the logger, as this object did not have direct information about the name of the message performed. The only information available that we had was the *nonce* of the transaction performed. However, we were able to collect the nonce from the evaluation script through `PinkContractPromise` interface and thus we could connect the nonce and gas expenditure from the logger with a specific message. Afterwards we parsed the metadata about the gas usage and nonce of the performed transaction exclusively for the ones that modify the state(on-chain transactions with `MessageOutput` type).

4.3.7 Proof of Concept Frontend Application

Integration of smart contracts into a fully fledged frontend application was the goal we have been aiming for from the beginning. It would further demonstrate the capabilities of privacy-preserving platforms in the electronic voting use case, even to nontech people.

We have worked on this along with the implementation of smart contracts, and the development reached a point where we implemented several components, including the input forms, dynamic routing, navigation, integration to *Metamask*, and dynamic visualization of mockup API data. We have included screnshots of the implemented views in Appendix B. The technological stack of the app consisted of the *React* framework, *PatternFly*, *Webpack*, and *react-router*. These technologies facilitated the development of dynamically routed reusable components that could be seamlessly integrated and bundled for enhanced browser integration. Due to a lack of time, the front-end application has not been finished, there are missing views and missing back-end connection to the smart contract platforms.

[10]Example scripts for interaction with the *Phala Network*, available athttps://github.com/phala-sdk-cookbook

Chapter 5

Evaluation

Transactional throughput serves as a performance indicator that can be used to evaluate the scalability and efficiency of blockchain platforms. By calculating and comparing these metrics across the evaluated platforms, we are able to objectively evaluate the capabilities and limitations of the smart contract implementations (in our case, the e-voting system) on these platforms. Our evaluation is no different, and we based it on a methodology presented in the first Section 5.2, in which we outline a systematic approach for the calculation of transactional throughput.

In the next three sections, we theoretically evaluate the performance of all the platforms using the formula in the methodology, and we discuss each of their compelling features, but also limitations, the overall experience, not only when it comes to smart contract development, but also related to the quality of community, quality of documentation, and other aspects.

In the last two sections, we first visualize the collected data to help us understand the behavior and trends in the gas usage of the operations performed within the e-voting system and discuss whether the results have matched the theoretical calculations, which platform has performed the best, which one the worst, and what could be the reason why the platform has performed in such manner. And to conclude the evaluation, we summarize all the results and findings obtained throughout our assessment and reflect what could have been done better or differently, what are the key takeaways, and what other related platforms could be at the core of future research.

5.1 Evaluation Scenario

In our performance analysis, we focus on determining the vote-casting throughput and the number of voters the implemented e-voting contracts are capable of processing. We have devised an evaluation scenario in which we simulate an ongoing election that will enable the collection of statistics necessary for estimation of these metrics. The scenario consists of several steps involving the execution of smart contract operations (e.g., creating users), which are measured in terms of gas consumption and stored for later processing. The scenario consists of the following steps:

1. Create an election with one new user that will be configured to be a valid voter in election.

2. Perform the following sequence in loop for n times.

(a) Create a user and set him to be a valid voter within the election created in the previous step.

(b) Create a second user that will become a candidate within the same election.

(c) The newly created voter casts the vote to the newly created candidate.

(d) Every tenth iteration(including the first one), the election is ended and the results calculated. Afterwards, the election is resumed to allow the voters to cast their votes again.

(e) Return to the first step of the loop.

This way we can determine the effect of increasing number of voters and candidates to gas costs of performing vote casting and the effect of increasing votes at the gas cost of results calculation. The voter created initially is intended to vote and determine the winner of the election. This scenario is customizable, and we can perform variations of this scenario with other functionalities as well.

5.2 Methodology of Transactional Throughput Calculation

Throughput is a term frequently used in the discussion of distributed or embedded systems, and it defines the rate at which the system can perform a given number of tasks during a specified time period [31]. There are several variants such as data throughput, system throughput, network throughput, and throughput related to blockchains networks as well. The throughput on blockchains typically means the number of *transactions per second* (TPS) processed by a blockchain network. It is a critical metric for assessing the performance and scalability of blockchain platforms, so as part of our performance analysis, we had to develop a methodology for its calculation.

The principal component of the methodology is a formula underpinned by two pivotal equations, which utilize the configuration settings of the platforms, including *Block Gas Limit* (BGL) and *Block Execution Time* (BET). These parameters will allow us to theoretically calculate the number of *Transactions Per Block* (using Equation 5.1) and subsequently the number of *transactions per second* (using the second Equation 5.2).

$$Transactions\ per\ Block = \frac{Block\ Gas\ Limit}{Transaction\ Gas\ Cost} \tag{5.1}$$

Maximum transactions in one block.

$$Transactions\ per\ second = \frac{Transactions\ per\ Block}{BlockExecutionTime} \tag{5.2}$$

Transactional throughput.

Based on the TPS and the average transaction gas cost on a given platform, we can determine the theoretical limits of the number of operations performed within the blockchain platform. We will assume a 48 hour time window, which represents the duration election is opened. The estimation will be based on the TPS extrapolation to the duration of the time window, exhibiting a method to estimate the potential number of operations executed during this time window. The extrapolation enables a projections of behavior or performance over a longer period, from a small sample of data (e.g., avoiding millions of vote-cast transactions). For instance, this estimate could be the projected volume of vote casts or other operations carried out within the election period.

5.3 Secret Network

The smart contracts in the Secret Network are built on top of the *CosmWasm* framework, which is based on the *Rust* language that is compiled to *Wasm*. The framework delineates an easy-to-follow and understand programming patterns for transaction processing, message format specifications, and contracts state definitions, that integrate well within the *Rust* language. The offered storage structure options are extensive, which provides great contracts state design customization. The only downfall of the platform is the need to incorporate special mechanisms for confidential queries, which is not required in the other platforms.

The developer documentation provided is on a very solid level. It contains understandable guides and clearly explains the foundational concepts with relevant references to example projects. Although there were also instances where the documentation failed to cover certain topics (e.g., clear functioning example of confidential queries), we had to resort to the community for help. The main remark regarding the documentation is the chaotic organization. For example, the initial guides sometimes referenced templates that were not maintained anymore, which resulted in wasted time. The community around the project is very active, there are always ongoing discussions, and the responses to our inquiries were prompt.

5.3.1 Theoretical Performance

The *Secret* whitepaper declares that the transaction throughput can theoretically reach hundreds of transactions per second depending on the difficulty of the transactions performed [42] . The rate may be significantly improved with the vanilla engine `Wasmer`[1], which is approximately 10 to 15 times more performant than the `Wasm3`[2] engine used in Secret Network today. This newer engine is not yet supported, but the developers have its integration listed on their roadmap. The parameters of the network configuration can easily be found in the official documentation[3]:

- **Block Gas Limit** - 6 million.
- **Block Time** - Approximately 6 seconds.
- **Average gas cost of transaction** - 100,000 gas.

$$TPB = \frac{6000000}{100000} = 60 \tag{5.3}$$

$$TPS = \frac{60}{6} = 10 \tag{5.4}$$

$$TimeWindow = ((60 \times 60) \times 24) \times 2 = 172800 \tag{5.5}$$

$$VotesCasted = 172800 \times 10 = 1728000 \tag{5.6}$$

Theoretical calculation of the possible number of casted votes.

Based on the parameters of the platform and the Equations 5.2 and 5.1 we can state that the *Secret Network* can process 60 transactions per block, which at block production rate of

[1]Wasmer enables containers to run on multiple platforms, available at https://github.com/wasmer.
[2]https://github.com/wasm3
[3]https://docs.scrt.network/documentation/tps-and-scalability

6 seconds results in transactional throughput of ten transactions per second. If the average gas cost of 100,000 was the actual cost in the e-voting system to perform votes casting, then we would theoretically be able to process 1,728,000 votes in a two-day election period.

5.4 Oasis Network

The smart contracts in *Oasis* and *Sapphire ParaTime* are using the popular *Solidity* language and support powerful *EVM* tools *Hardhat* or *Foundry.* Due to the dominance of *Ethereum* and *EVM*-based chains these technologies have been highly utilized in a large number of real-world projects, which resulted in exceptional coverage and a tuned development experience. Our perspective has only proven this fact, since the shear time we spent on the development of the contract on this platform was a fragment of the time we spent on the other platforms. When it comes down to limitations of the platform, we can only mention the lack of configuration options in data storage structure and secondly the strict limitations of the contract size.

5.4.1 Theoretical Performance

The oasis network does not present information about the performance and configuration of their *Sapphire ParaTime* in their official documentation. In this case, we had to inspect the *Oasis Explorer*[4] that shows all the new blocks and transactions published together with their metadata and the source code of *runtime-sdk*, used to build *Sapphire ParaTime.* The amount of transactions performed by *ParaTime* can include in their blocks is capped to 1000[5], but the actual number will be different. From the *Oasis Explorer* metadata and the source code we can determine the parameters of the network configuration we were looking for:

- **Block Gas Limit** - 15 million.
- **Block Time** - Approximately 6 seconds.
- **Average gas cost of transaction** - 150,000 gas.

Based on the parameters above and the Equations 5.2 and 5.1 it is possible to derive that the *Oasis Network* can process 60 transactions per block, which at block production rate of 6 seconds results in transactional throughput of ten transactions per second.

$$TPB = \frac{15000000}{150000} = 100 \tag{5.7}$$

$$TPS = \frac{100}{6} = 16.6 \tag{5.8}$$

$$TimeWindow = ((60 \times 60) \times 24) \times 2 = 172800 \tag{5.9}$$

$$VotesCasted = 172800 \times 16.6 = 2880000 \tag{5.10}$$

Theoretical calculation of the possible number of casted votes.

If the average gas cost of 150,000 was the actual cost in the e-voting system to perform votes casting, then we would be theoretically able to process 2,880,000 votes in a two day election period.

[4]https://explorer.oasis.io/sapphire
[5]https://github.com/oasisprotocol/sdk/src/config.rs

5.5 Phala Network

The benefits of the *ink!* language used in the *Phala* contracts provide a variety of macros that enhance the *Rust* language to provide powerful smart contract customizations with the established features of *Rust* such as type control or increased security. But there are flaws as well, these include a problematic error handling, high commitment to macro-specific logic, poor modularization, and limitations of storage structure options. Our development experience with the platform underlines these issues, especially the difficulty of handling the macros across more modules and the storage options for designing the contracts state.

The community around *Phala* is active, during development we have reached out on multiple occasions, and in most cases we received assistance. The available documentation and guides should definitely be improved. It was clearly visible that the team behind *Phala* was rapidly changing its marketing according to current trends, and the documentation only reflected these shifts (e.g., rebrands of *Phat Contracts* into *AI agents*), resulting in situations where the information on which we based our development suddenly perished and we had to resort to the use of *Wayback Machine*.

5.5.1 Theoretical Performance

Compared to previous platforms, the parameters of the *Phala Network* configuration were difficult to determine, due to the deficient documentation mentioned above. Only related information in the documentation is on the calculation of transaction fees[6], but there are no references to the gas parameters. The *Phala* whitepaper [41] also does not mention anything related to block gas limits or block execution times. We've chosen to delve into the source codes used to build the network and the networks dashboard[7] similarly as in *Oasis*. We concluded that these sources provided conflicting information about *Block Gas Limit* (BGL) and *Block Execution Time*, so we tried to reach out to the community in *Discord* to resolve which information is correct, but even at the time of writing this thesis we still did not receive an answer. Thus for our evaluation, we have decided to use the information collected from the source codes[8], where the parameters are set in the following way:

- **Block Gas/Weigt Limit** - 1 500 000 000 000
- **Block Time** - Approximately 12 seconds.
- **Average gas cost of transaction** - 500,000 gas.

With the parameters specified above we calculated that the *Oasis Network* is configured to be processing 250,000 transactions per second. This number is probably inflated and does not reflect the real throughput of the network, as validators may not have enough time to process all the computational load in such a massive case *Block Gas Limit*. The block metadata from the *Phala* dashboard also suggest that this may be the case as the blocks never reach 3% usage of *Block Gas Limit*. We will use these parameters in our evaluation anyway to see how the *Phala* network performed in our implementation.

[6]https://docs.phala.network/references/support/transaction-costs
[7]https://polkadot.js.org/apps/explorer/phala
[8]These were determined with analysis of several files in https://github.com/Phala/blockchain repository

5.6 Implementation Performance

The data collected from our implemented smart contracts, consisting of the gas consumption of vote casting and counting results within the e-voting system (just as we defined in Section 5.1), form the basis of our evaluation. We have chosen to assess performance by visualizing all the data collected and conducting an analysis of these visualizations to obtain insights into the efficiency of implemented smart contract.

During data collection, we mainly resorted to the use of *Testnets*, instead of *Mainnets*, as a medium for our deployed contracts, since computational effort (represented in gas) is calculated deterministically on both occasions, but we evade the associated real-world value costs in *Mainnets*. The collected data was loaded into *pandas dataframes*, where they were processed in accordance with our methodology formula specified in Section 5.2, but now with the actual gas costs of our implementations. To visualize the data, we have used the `matplotlib` and `seabon` libraries for *Python*.

5.6.1 Analysis of Collected Data

The core of the evaluation scenario are vote casts (see Section 5.1), as its the integral operation within any voting system. We wanted to visualize how the growing number of votes, voters, and candidates in the system relates to the gas consumption of casting votes.

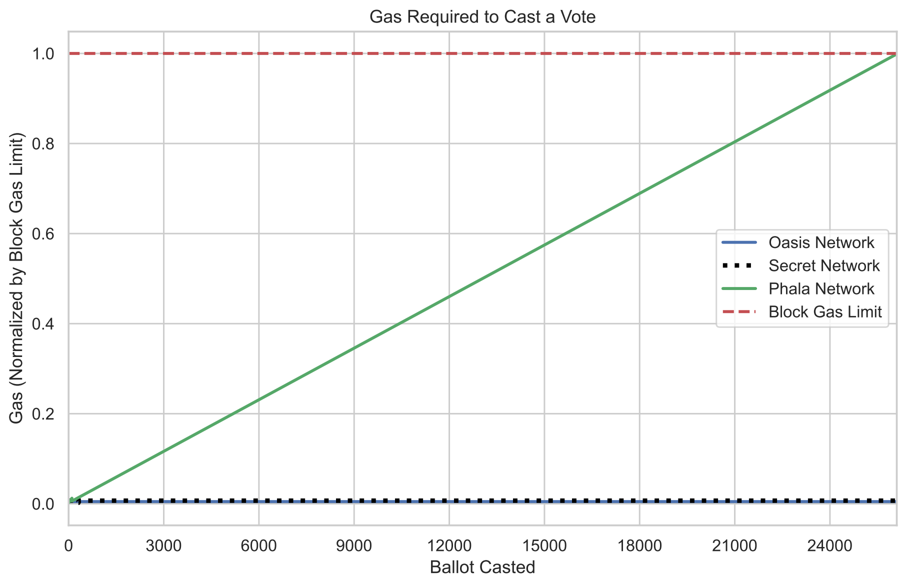


Figure 5.1: Gas usages trends of casting votes.

In the Figure 5.1 above is a visual representation of the gas requirements to cast vote on a particular platform. We have normalized the collected gas information with the maximum block gas limit on a given platform, so the data are in a comparable perspective between platforms. The data collected consisted of a small sequence of gas costs for a vote-casting operation, where the remainder was extrapolated. Compared to the proposed methodology, in the above Figure 5.1, instead of extrapolating to a time window, we extrapolated to a certain amount of votes casts, instead of a certain time frame. The interval we extrapolated was selected by hand, based on the trend of the extrapolation.

It is visible that the increase in the number of votes does not affect the gas required to perform vote-casting transactions in the implementation within *Secret Network* and *Oasis Network*. Since our evaluation scenario (inspect Section 5.1) also creates a new candidate and voter for each ballot cast, we can also state that an increase in the number of candidates and voters stored in the contract state also does not affect the gas required to cast votes.

On the other hand *Phala Network* experienced a linear increase in gas consumption and after storing around 26,000 votes reached *Block Gas Limit*. This will have significant effects on the overall performance of the implementation and is a reason why we have focused on improving the design of the contract (see Section 3.4.1). We have also tried a variation of the evaluation scenario, that iteratively created only a new voter that casted votes for the same candidate. The results have changed negligibly, which was a proof that the performance problem is definitely tied to the storage of casted ballots.

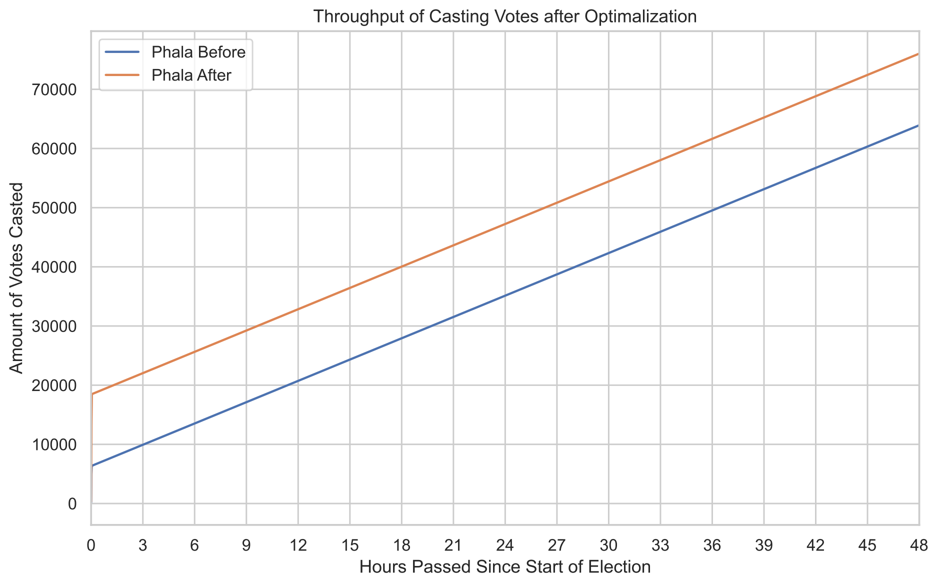


Figure 5.2: Vote casting throughput before and after optimizations.

Even after optimizing the implementation, we have managed only an insignificant increase in vote-casting throughput (see Figure 5.2). The reason why the *Phala* implementation failed to achieve a similar performance as the other platforms is that we have not managed to fully replace the use of array structures in our storage and that the map structures cannot insert individual items under the same key (we discussed these problems in 3.4.1).

Based on the outcomes observed in the Figure 5.1 we could perform the methodology presented in Section 5.2 with the actual gas costs of casting votes. The extrapolation mentioned in the methodology expects a single number to represent the cost of the given operation, which was casting votes in our case. Choosing such a number on *Secret* or *Oasis* was trivial since the gas consumption remained constant. In the case of *Phala* we had to make a compromise, since gas consumption was increasing linearly and reached a *block gas limit* around the 26,000 ballot casted mark (notice Figure 5.1). Thus we split the extrapolation into two parts, where the first part (up to the gas limit and 24,000 ballots cast) is approximated by an average gas cost to cast votes until that mark, and the remaining part is approximated using the block gas limit itself. Meaning that after 26,000 ballot mark, every new casted ballot will occupy the whole block (effectively the throughput is limited to block execution time).

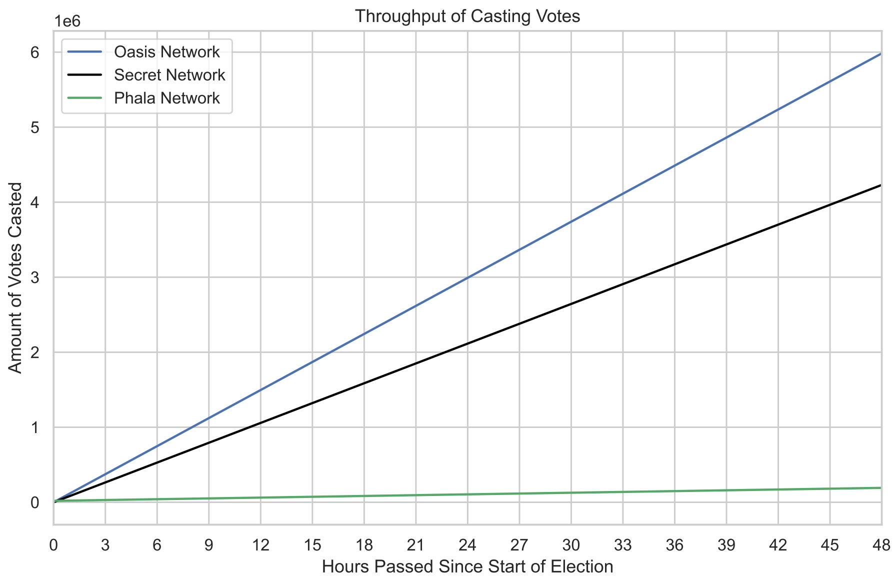


Figure 5.3: Vote casting performance in the implementations.

The results of the vote-casting throughput extrapolation to the time window of 48 hours, during which the election is open to receive votes, are visible in Figure B.9. We can clearly see that *Oasis* outperformed other platforms, but *Secret* does not lack significantly, ultimately these two platforms are capable of processing and storing four to six million votes in a two-day voting period. When examining the statistics for *Phala*, it is evident that the linear increase in the gas costs to cast votes had an effect on the total throughput, since the total number of ballots stored reaches only a fraction of the other platforms.

5.7 Summary of Results

Having evaluated the visualizations of transactional vote-casting throughput and the gas costs associated with the vote casting operation within our implemented smart contracts, we will summarize the results we can derive from our analysis and these visualizations.

5.7.1 Maximum number of candidates or voters

Even though we have not presented a standalone visualization of the relationship between the number of candidates or voters participating in the election and the gas costs related to the operations performed in the same election (adding another candidate, or casting votes), we are able to estimate it using the visualization in Figure 5.1. In the perspective of storage structure, the behavior of the gas costs during the enrollment of new candidates or voters in an election is similar to that of casting votes, which in the case of *Secret* and *Oasis* is a constant trend in gas costs. So, the actual maximum of candidates and voters participating in an election held in two days will not be significantly less than the calculated throughput in Figure B.9. This is not the case in *Phala*, as it has not performed well in vote-casting. In addition, the increasing number of candidates or voters stored in the state does not have an effect on the gas costs of casting votes. This claim holds because of the way our data collection methodology and implementation are set up. If such an effect existed, we would already see some form of increase in Figure 5.1.

5.7.2 Vote Casting

Compared to our theoretical throughput estimates in the previous sections and the actual statistics approximated from the data collected in our implementations, we can conclude that both *Oasis* and *Secret* almost doubled the throughput of transactions with the average gas cost. That means that the gas cost of casting votes in these two platforms is well below this and perform above standard. In the case of *Phala*, the results of the vote casting performance were significantly inferior, even after our optimizations.

5.8 Discussion

To conclude our research of privacy-preserving platforms, the overall outperformer in transactional vote-casting throughput in our implementation is clearly *Oasis*, the close second being *Secret*, and the last position is occupied by *Phala*. The other operations like vote recasting, and vote verification are not expected to yield different results, as the concepts of how they access the storage do not differ significantly.

The reason behind the deficient results for *Phala* is rooted in the provided storage structures that had several limitations, which we have tried to tackle with various optimizations (inspect 3.4.1), but to no prevail. Other platforms did not have problems to that extent. The reason why this is the case might be related to the actual target and characteristics of *Phala Network*. It is marketed as an off-chain computational platform or coprocessor that is empowered by the ability to perform traditional *HTTP* requests from smart contracts. This exclusive feature allows developers to form a *HTTP* connection to other storage services (compatible with AWS S3[9]), where the storage of the smart contract state can be offloaded, ultimately forming a stateless back-end[10]. The only information stored inside the private storage of the contract could be the access credentials to the connected storage service, and the only thing the contract focuses on is the computation. Clearly, the goal of the platform is not the storage of data, but the computation, and since we implemented the contracts equivalently across the platforms (not reflecting this powerful feature), the results were not comparable with *Oasis* and *Secret*.

The implementation of electronic voting has shown that privacy-preserving platforms have the ability to accommodate a large number of voters and candidates. Studies like [34] has shown that utilizing standard blockchain smart contracts with incorporated cryptographic schemes to provide secrecy within e-voting systems instead of trusted enclaves limits the overall throughput of the system based on the number of candidates the voters are selecting from. This has proven the potential and strength that the use of *Trusted Execution Environments* brings to the use case of electronic voting.

[9]https://aws.amazon.com/s3/
[10]https://docs.phala.network/build-stateless-backend

The drawback of *Trusted Execution Environments* is that we have to trust third-party companies (mainly Intel and its SGX TEE technology) to remain honest. These companies facilitate the isolation and verification of computation performed in isolated enclaves, which is the fundamental feature on which privacy-preserving platforms operate. If these companies developing or companies adopting these mechanisms failed to maintain the technology, a potential security breach would have severe consequences, and private data would most likely be leaked. The *Secret Network* is a live example of this fact, in 2022 they failed to mitigate publicly known vulnerabilites of the Intel SGX enclave at that time[11]. Researchers who discovered this vulnerability have managed to decrypt all the encrypted data on this network, including the contracts states or transaction contents. This serves as a memento that these technologies are far from foolproof.

[11]https://sgx.fail/

Chapter 6

Conclusion

The goal of this thesis was to study voting systems, blockchains, trusted computing, and the concepts of privacy-preserving platforms. We have compared existing privacy-preserving platforms such as Secret, Phala, and Oasis Network based on their features such as performance, usability, and other aspects in an electronic voting use case. We have analyzed the development capabilities, storage options, and other features of the chosen platforms and proposed a smart contract design for the e-voting application on which we evaluated these platforms. In the design, we have focused on the optimal structure of these contracts, to make sure there are no implementation bottlenecks that would decimate our evaluation objectivity. In addition, we have ensured that the privacy and confidentiality of the voting process would be guaranteed within the designed smart contracts. Thereafter, we have discussed the implementation of the designed smart contracts, what was the experience like, what problems have we encountered, how we tested our solution, and how we collected gas usage statistics of the operations perfomed within the evaluation scenario.

The result of this thesis are three fully functional smart contracts with automated testing, deployment, and statistics gathering infrastructures built around them. These smart contracts formed a basis in our evaluation, where we compared their transactional throughput and the number of candidates or voters that the contracts were able to process and store. Statistics have shown that *Oasis* can serve the largest number of voters, a close second being *Secret* and, the least performant was *Phala*. The analysis and comparison of these statistics have highlighted the potential these technologies have in the sphere of electronic voting, as they improve the limits of stored and processed candidates, voters, or casted votes compared to e-voting systems on standard blockchain platforms.

Future work can focus on improvements and optimizations of smart contracts in *Phala*, where a suitable integration of external storage services for the storage of non-private data should result in improved performance. Or another direction the future research could take is the analysis and evaluation of other existing platforms, like Integritee, Hyperledger Fabric, Private Chaincode, which offer similar privacy features within their respective architectures.

Bibliography

[1] Alvarez, R. M. and Hall, T. E. *Point, Click, and Vote: The Future of Internet Voting.* 1st ed. Brookings Institution Press, 2004. ISBN 978-0-8157-0369-3.

[2] Awad, M. and Leiss, E. L. The evolution of voting: analysis of conventional and electronic voting systems. *International Journal of Applied Engineering Research.* 1st ed. 2016, vol. 11, no. 12, p. 7888–7896.

[3] Bernal Bernabe, J., Canovas, J. L., Hernandez Ramos, J. L., Torres Moreno, R. and Skarmeta, A. Privacy-Preserving Solutions for Blockchain: Review and Challenges. *IEEE Access.* 1st ed. 2019, vol. 7, no. 1, p. 164908–164940. DOI: 10.1109/ACCESS.2019.2950872.

[4] Buterin, V. *Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform.* 1st ed. 2014. Available at: `https://github.com/ethereum/wiki/wiki/White-Paper`.

[5] Cachin, C. and Vukolić, M. *Blockchain Consensus Protocols in the Wild.* arXiv:1707.01873. ArXiv, july 2017. ArXiv:1707.01873 [cs] type: article. Available at: `http://arxiv.org/abs/1707.01873`.

[6] Carolina Banton, A. C. *How Escrow Protects Parties in Financial Transactions.* August 2023. Available at: `https://www.investopedia.com/terms/e/escrow.asp`.

[7] Carter, E. and Farrell, D. M. Electoral Systems and Election Management. In: LeDuc, L., Niemi, D. and Norris, P., ed. *Comparing Democracies 3.* London: Sage, 2009, chap. 2. ISBN 9781847875044.

[8] Costan, V. and Devadas, S. *Intel SGX Explained.* 2016. Publication info: Preprint. Available at: `https://eprint.iacr.org/2016/086`.

[9] Di Pierro, M. What Is the Blockchain? *Computing in Science & Engineering.* 1st ed. 2017, vol. 19, no. 5, p. 92–95. DOI: 10.1109/MCSE.2017.3421554.

[10] E. Black, P. *AVL tree.* November 2019. Available at: `https://www.nist.gov/dads/HTML/avltree.html`.

[11] ElGamal, T. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In: Blakley, G. R. and Chaum, D., ed. *Advances in Cryptology.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, p. 10–18. ISBN 978-3-540-39568-3.

[12] Grishchenko, I., Maffei, M. and Schneidewind, C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In: Bauer, L. and Küsters, R., ed. *Principles of Security and Trust*. Cham: Springer International Publishing, 2018, p. 243–269. ISBN 978-3-319-89722-6.

[13] Jafar, U., Aziz, M. J. A. and Shukur, Z. Blockchain for Electronic Voting System—Review and Open Research Challenges. *Sensors*. 1st ed. 2021, vol. 21, no. 17. DOI: 10.3390/s21175874. ISSN 1424-8220. Available at: `https://www.mdpi.com/1424-8220/21/17/5874`.

[14] Jekovec, M., Jeran, L. and Janez, T. *Oasis Core Developer Documentation / Runtime Layer / Consensus Layer*. May 2024. Available at: `https://docs.oasis.io/core`.

[15] Kasi, N. R., S, R. and Karuppiah, M. Chapter 1 - Blockchain architecture, taxonomy, challenges, and applications. In: Islam, S. H., Pal, A. K., Samanta, D. and Bhattacharyya, S., ed. *Blockchain Technology for Emerging Applications*. Academic Press, 2022, p. 1–31. Hybrid Computational Intelligence for Pattern Analysis. DOI: https://doi.org/10.1016/B978-0-323-90193-2.00001-6. ISBN 978-0-323-90193-2. Available at: `https://www.sciencedirect.com/science/article/pii/B9780323901932000016`.

[16] Kumar, R., Badwal, L., Avasthi, S. and Prakash, A. A Secure Decentralized E-Voting with Blockchain & Smart Contracts. In: Balvinder, S., ed. *13th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. 2023, p. 419–424. DOI: 10.1109/Confluence56041.2023.10048871. ISBN 9781479942350.

[17] Loeber, L. E-Voting in the Netherlands; from General Acceptance to General Doubt in Two Years. In: Gesellschaft für Informatik e. V. *Electronic Voting 2008 (EVOTE08). 3rd International Conference on Electronic Voting 2008, Co-organized by Council of Europe*. Bonn: [b.n.], 2008, p. 21–30. ISBN 978-3-88579-225-3.

[18] Murtaza, M. H., Alizai, Z. A. and Iqbal, Z. Blockchain Based Anonymous Voting System Using zkSNARKs. In: IEEE. *2019 International Conference on Applied and Engineering Mathematics (ICAEM)*. 2019, p. 209–214. DOI: 10.1109/ICAEM.2019.8853836. ISBN 9781728123547.

[19] Müller, M., Ruberti, S., Castano, H. and Weeg, P. *Overview of Storage and Data Structures*. 2024. Available at: `https://use.ink/datastructures/overview`.

[20] Nakamoto, S. *Bitcoin: A peer-to-peer electronic cash system*. 2009. Available at: `http://www.bitcoin.org/bitcoin.pdf`.

[21] Pace, K. and Wackerow, P. *Solidity Language Documentation*. May 2024. Available at: `https://docs.soliditylang.org/en/v0.8.25/introduction-to-smart-contracts.html`.

[22] Paillier, P. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In: Stern, J., ed. *Advances in Cryptology — EUROCRYPT '99*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, p. 223–238. ISBN 978-3-540-48910-8.

[23] Petroaie, M., Thompson, A. and Elesbao, A. *What is CosmWasm?* 2024. Available at: https://docs.archway.io/developers/cosmwasm-documentation/introduction.

[24] Pettinari, P. and Cook, J. *GAS AND FEES.* Apr 2024. Available at: https://wiki.polkadot.network/docs/learn-architecture.

[25] Rae, D., Hanby, V. and Loosemore, J. Thresholds of Representation and Thresholds of Exclusion: An Analytic Note on Electoral Systems. *Comparative Political Studies.* 1st ed. 1971, vol. 3, no. 4, p. 479–488. DOI: 10.1177/001041407100300406. Available at: https://doi.org/10.1177/001041407100300406.

[26] Reitwiessner, C. *ZkSNARKs in a nutshell.* 2016. Available at: https://blog.ethereum.org/2016/12/05/zksnarks-in-a-nutshell.

[27] Reyzin, L., Meshkov, D., Chepurnoy, A. and Ivanov, S. *Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies.* 2016. Publication info: Published elsewhere. Minor revision. Financial Cryptography 2017. Available at: https://eprint.iacr.org/2016/994.

[28] Sabt, M., Achemlal, M. and Bouabdallah, A. Trusted Execution Environment: What It is, and What It is Not. In: IEEE. *2015 IEEE Trustcom/BigDataSE/ISPA.* Helsinki, Finland: [b.n.], August 2015. DOI: 10.1109/Trustcom.2015.357. ISBN 978-1-4673-7952-6. Available at: http://ieeexplore.ieee.org/document/7345265/.

[29] Schultzie, D., Morami, A. and seanrad.scrt. *Secret Network Developer Documentation | Introduction | Overview.* August 2024. Available at: https://docs.scrt.network/secret-network-documentation/.

[30] Schumacher, S. and Connaughton, A. *How countries register votes and cast votes during elections | World Economic Forum.* 2020. Available at: https://www.weforum.org/agenda/2020/11/voter-registration-mail-ballots-countries-world-elections/.

[31] Shetty, N. *Understanding Latency and Throughput in Embedded, Computer, and Blockchain Networks.* 2023. Available at: https://shardeum.org/blog/latency-throughput-blockchain/#Latency_in_ComputerEmbedded_Systems.

[32] Song, D. *The Oasis Blockchain Platform.* Oasis Network, june 2020. Available at: https://assets.website-files.com/5f59478e350b91447863f593/628ba74a9aee37587419cf65_20200623%20The%20Oasis%20Blockchain%20Platform.pdf.

[33] Staff, C. *How Secret Network's Privacy as a Service Unlocks Web3 for the Next Billion Users.* June 2023. Section: Sponsored Content. Available at: https://www.coindesk.com/sponsored-content/how-secret-networks-privacy-as-a-service-unlocks-web3-for-the-next-billion-users/.

[34] Stančíková, I. and Homoliak, I. *SBvote: Scalable Self-Tallying Blockchain-Based Voting.* 2022.

[35] Szabo, N. Formalizing and Securing Relationships on Public Networks. *First Monday*. 1st ed. Sep. 1997, vol. 2, no. 9. DOI: 10.5210/fm.v2i9.548. Available at: https://firstmonday.org/ojs/index.php/fm/article/view/548.

[36] Tas, R. and Tanriover, O. O. A Systematic Review of Challenges and Opportunities of Blockchain for E-Voting. *Symmetry*. 1st ed. 2020, vol. 12, no. 8. DOI: 10.3390/sym12081328. ISSN 2073-8994. Available at: https://www.mdpi.com/2073-8994/12/8/1328.

[37] Tonelli, R., Pierro, G. A., Ortu, M. and Destefanis, G. Smart contracts software metrics: A first study. *PLOS ONE*. 1st ed. Public Library of Science. april 2023, vol. 18, no. 4, p. 1–31. DOI: 10.1371/journal.pone.0281043. Available at: https://doi.org/10.1371/journal.pone.0281043.

[38] Wikipedia contributors. *Elections in the Netherlands — Wikipedia, The Free Encyclopedia*. 2023. Available at: https://en.wikipedia.org/w/index.php?title=Elections_in_the_Netherlands&oldid=1190501158.

[39] Wood, G., Habermeier, R. and Czaban, P. *Polkadot Wiki*. Apr 2024. Available at: https://ethereum.org/en/developers/docs/gas/.

[40] Wu, Y. and Kasahara, S. Smart Contract-Based E-Voting System Using Homomorphic Encryption and Zero-Knowledge Proof. In: Zhou, J. and team, ed. *Applied Cryptography and Network Security Workshops*. Cham: Springer Nature Switzerland, 2023, p. 67–83. ISBN 978-3-031-41181-6.

[41] Yin, H., Zhou, S. and Jiang, J. *Phala Network: A Secure Decentralized Cloud Computing Network Based on Polkadot*. 2022. Available at: https://api.semanticscholar.org/CorpusID:233305130.

[42] Zyskind, G. *Secret Network: A Privacy-Preserving Secret Contract & dApp Platform*. 2024. Available at: https://scrt.network/graypaper.

[43] Žiška, M. *Biometric System Security Using Blockchain Technology*. Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/studenti/zav-prace/detail/146344.

Appendix A

Contents of the included storage media

A.1 Secret Smart Contract

```
/
├── Cargo.lock
├── Cargo.toml
├── LICENSE
├── Makefile
├── README.md
├── evaluation-evoting/
├── node/
├── schema/
├── src/
│   ├── contract.rs
│   ├── error.rs
│   ├── lib.rs
│   ├── msg.rs
│   ├── state.rs
│   ├── tests.rs
│   └── utils.rs
└── permits/
```

- **Cargo.toml** - contains dependencies of the smart contract.
- **Makefile** - automatized launch of local instances of secret network, build process with optimization, sending transactions and collecting data statistics.
- **evaluation_evoting** - git submodule of the data processing repository
- **node** - directory containing all of the scripts for deploying and collecting data.
- **schema** - json files representating the supported messages of the contract.
- **src** - directory containing all the source codes of implemented e-voting smart contract.
- **permits** - directory containing all the generated Permit objects that are used in authenticated queries.

A.2 Oasis Smart Contract

```
/
├── Makefile
├── README.md
├── contracts/
│   └── EvotingOasis.sol
├── evaluation-evoting/
├── hardhat.config.ts
├── package.json
├── evaluation-evoting/
├── scripts/
│   ├── deploy.ts
│   ├── evaluation.ts
│   └── utils.ts
├── test/
│   └── EvotingOasis.ts
├── tsconfig.json
└── yarn.lock
```

- **Makefile** - automatized launch of local instances of the network, build process, deployment of the contracts.
- **README.md** - guide how to run, test, or deploy the contract.
- **contracts** - directory containing the source code of implemented e-voting smart contract.
- **evaluation-evoting** - git submodule of the data processing repository.
- **hardhat.config.ts** - configuration of the hardhat development tool, local and testnet networks setup.
- **package.json** - list of javascript packages.
- **scripts** - directory containing all the scripts for deploying, sending transactions and collecting data statistics.
  - **deploy.ts** - script that deploys the smart contracts binaries into the testnet or localnet network.
  - **evaluation.ts** - script that connects to the network where our smart contract was deployed to perform the evaluation scenario.
  - **utils.ts** - defines helper functions for environment files interaction, collection of the statistics and file handling.
- **test** - directory containing the unit test file of our smart contract functionality.

A.3 Phala Smart Contract

```
/
├── Cargo.lock
├── Cargo.toml
├── Makefile
├── README.md
├── build.txt
├── data.rs
├── lib.rs
├── evaluation-evoting/
└── node
    ├── deploy.js
    ├── evaluation.js
    ├── logger.js
    ├── package-lock.json
    ├── package.json
    └── utils.js
```

- **Cargo.toml** - contains dependencies of the Rust smart contract.
- **Makefile** - automatized launch of the networks, build process, deployment of the contracts.
- **README.md** - guide how to run, test, or deploy the contract.
- **lib.rs** - main source file containing all of the smart contract logic, with the unit tests.
- **data.rs** - source file with a helper module containing definitions of custom structures.
- **evaluation-evoting** - git submodule of the data processing repository.
- **node** - directory containing all the scripts for deploying, sending transactions and collecting data statistics.
  - **deploy.js** - script that deploys the smart contracts binaries into the Phala testnet network.
  - **evaluation.js** - script that connects to the Phala testnet network and our deployed smart contract to perform the evaluation scenario.
  - **logger.js** - script that launches a logger instance that scans the Phala testnet for the transactions performed in our smart contract and collects their statistics.

A.4 Data Processing and Evaluation

```
/
├── oasisdata
│   ├── EndingElectionApproximation.txt
│   └── EvaluationFinal
├── phaladata
├── secretdata
│   ├── ComparisonItemConfig
│   ├── ComparisonSetVecMap
│   └── EvaluationFinal
├── plots
│   ├── evaluation
│   └── secret
├── README.md
├── requirements.txt
├── visualize.py
└── visualize_secret.py
```

- **oasisdata** - folder containing collected statistics from Oasis smart contract.
  - **EndingElectionApproximation.txt** - description of the format of the statistics data files.
  - **EvaluationFinal** - All the data used in the final evaluation.
- **phaladata** - folder containing collected statistics from Phala smart contract.
- **secretdata** - folder containing collected statistics from Secret smart contract.
  - **ComparisonItemConfig** - data used to compare the storage structures of Item and Config.
  - **ComparisonSetVecMap** - data used to compare the storage structures of KeySet, KeyMap, and array.
  - **EvaluationFinal** - data used in the final evaluation.
- **evaluation-evoting** - git submodule of the data processing repository.
- **plots** - directory containing all the visualized graphs.
- **README.md** - guide how to compile, install dependencies and run the visualization and data processing scripts.
- **requirements.txt** - description of all the packages needed to execute the program,
- **visualize.py** - script that processes data from all the platforms, performs extrapolation, and visualizes the graphs used in our final evaluation.
- **visualize_secret.py** - script that processes data from Secret Network, and visualizes the graphs used in the comparisons of the storage structures.

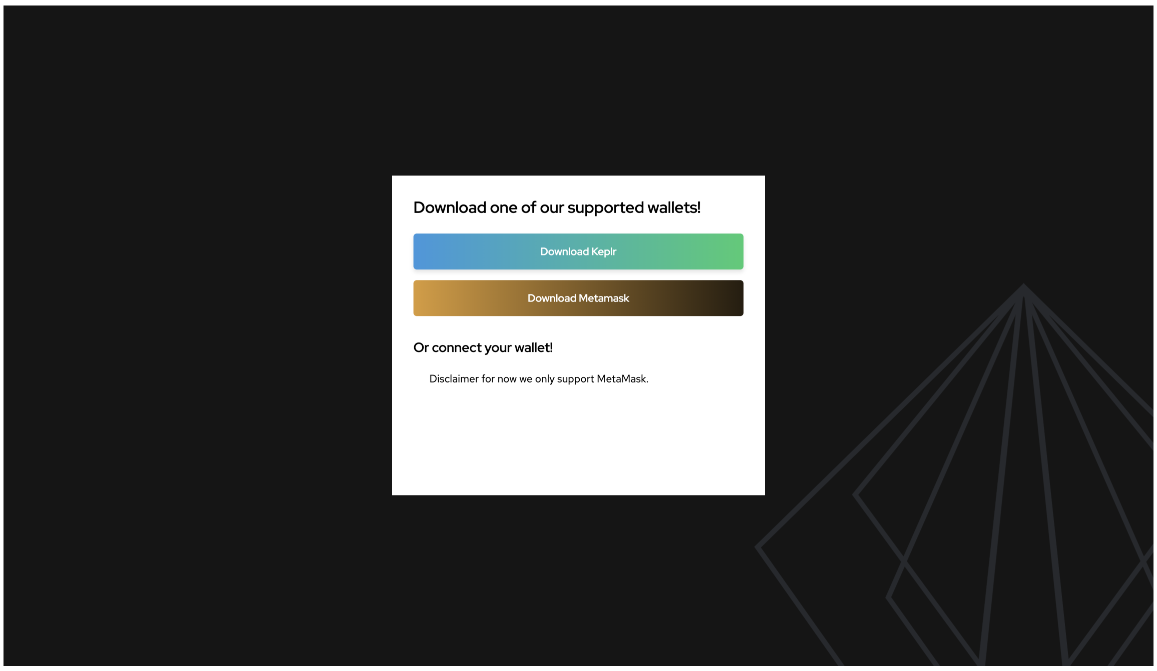
A.5 Frontend Proof Of Concept App

```
/
├── LICENSE
├── README.md
├── src/
│   ├── api
│   │   ├── deploy_instantiate.js
│   │   ├── instantiateKeplr.js
│   │   ├── package.json
│   │   ├── queries.js
│   │   └── utils.js
│   ├── app
│   │   ├── AppLayout
│   │   ├── Dashboard
│   │   ├── ElectionsPage
│   │   ├── LoginPage
│   │   ├── NotFound
│   │   ├── Settings
│   │   ├── Support
│   │   ├── UsersPage
│   │   ├── app.css
│   │   ├── assets
│   │   ├── index.tsx
│   │   └── utils
│   ├── favicon.png
│   ├── index.html
│   ├── index.tsx
│   └── mockdata
│       └── elections.json
├── stylePaths.js
├── tsconfig.json
├── package.json
├── webpack.common.js
├── webpack.dev.js
└── webpack.prod.js
```

- **src** - folder containing the codebase.
  - **api** - Folder containing scripts for connection to the networks of the privacy-preserving platforms.
  - **app** - List of directories, where each represents a certain component within the frontend application.
  - **index.html** - Root html file of the frontend application.
  - **index.tsx** - Root typescript file of the frontend application.
- **webpack*.js** - Configurations of the Webpack module bundler, used to build the project.

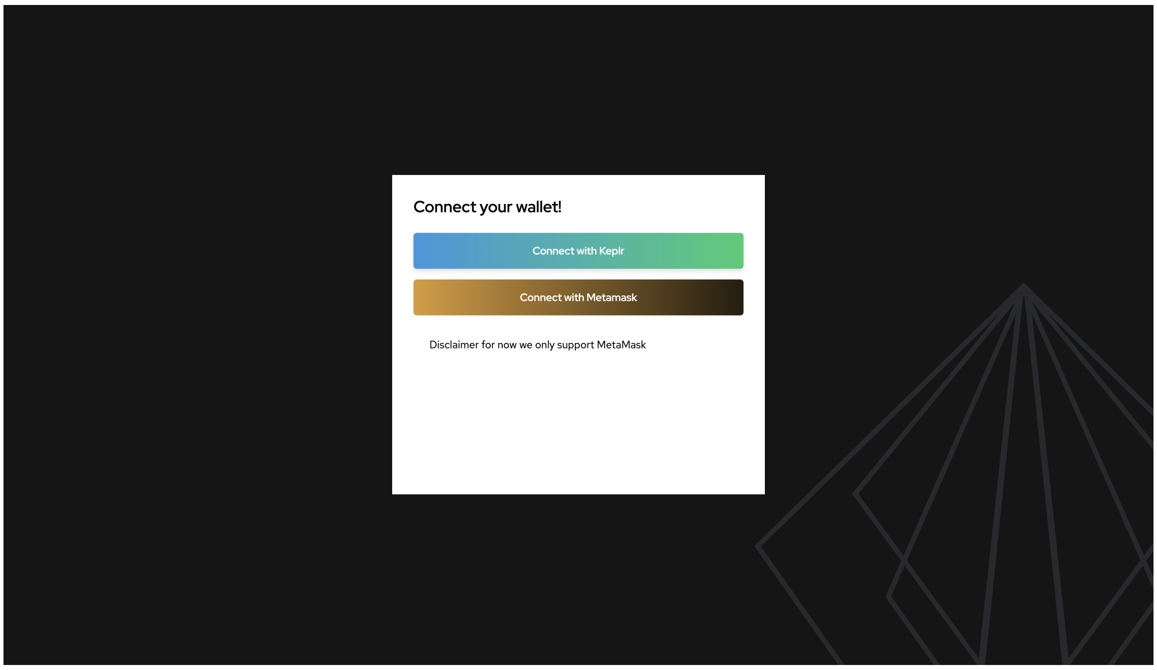
Appendix B

Views of proof of concept frontend application



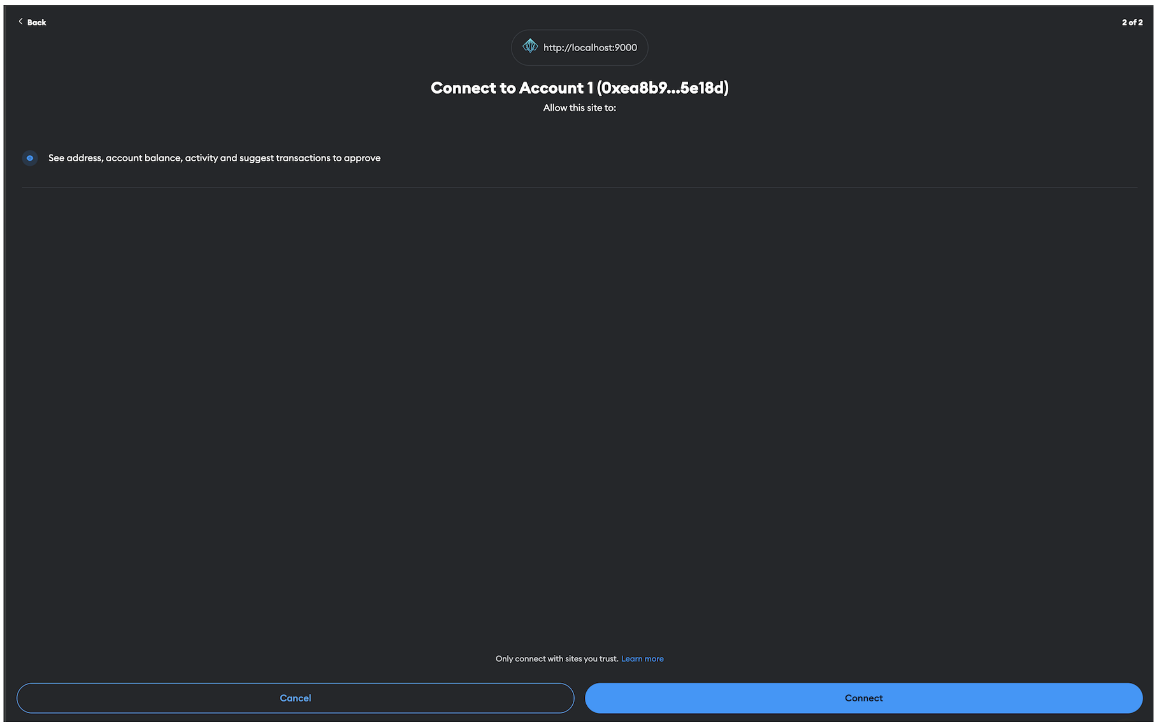
Disclaimer for now we only support MetaMask.

Figure B.1: View for users that do not have Metamask or Keplr extensions installed.



Disclaimer for now we only support MetaMask

Figure B.2: View for users that have one of extensions installed but are connected to our application.



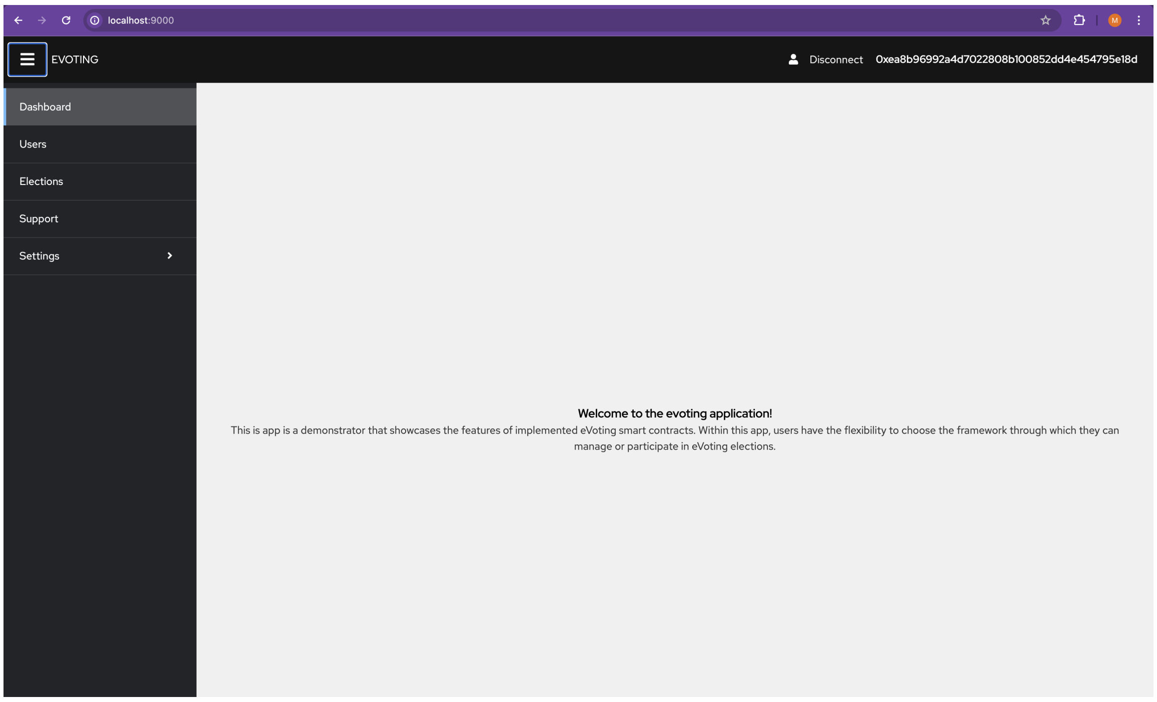
Connect to Account 1 (0xea8b9...5e18d)

Allow this site to:

See address, account balance, activity and suggest transactions to approve

Only connect with sites you trust. Learn more

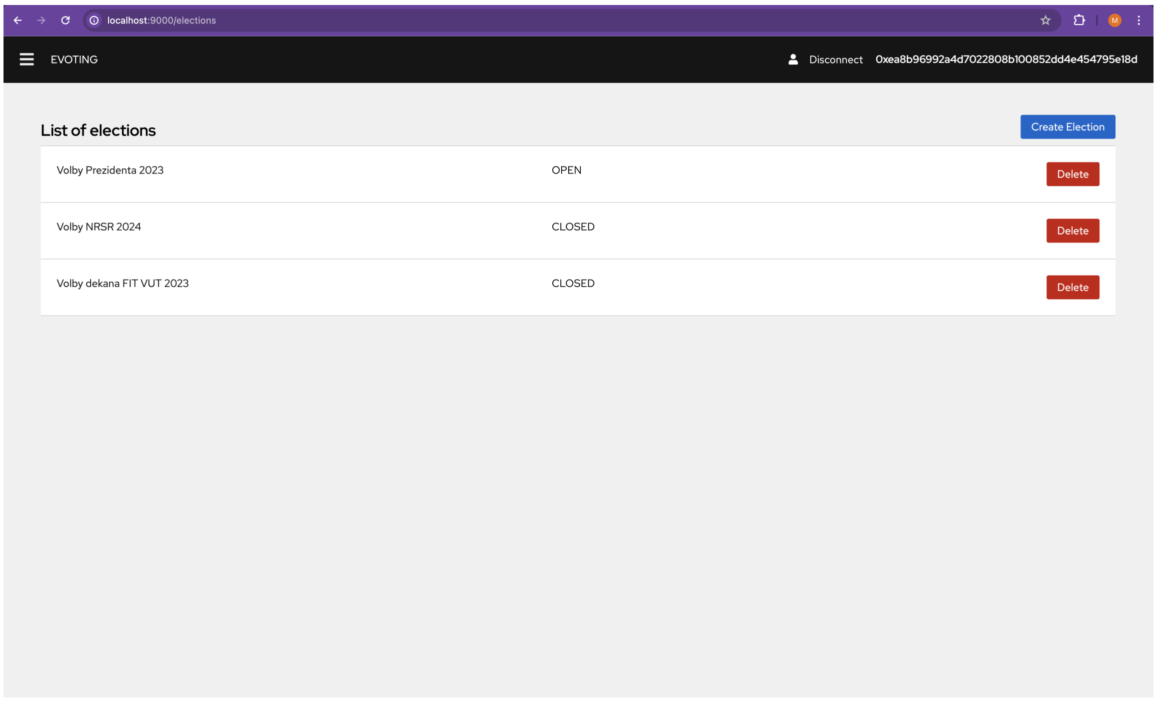
Figure B.3: Connection modal for connecting to the Metamask browser extension.



Welcome to the evoting application!

This is app is a demonstrator that showcases the features of implemented eVoting smart contracts. Within this app, users have the flexibility to choose the framework through which they can manage or participate in eVoting elections.

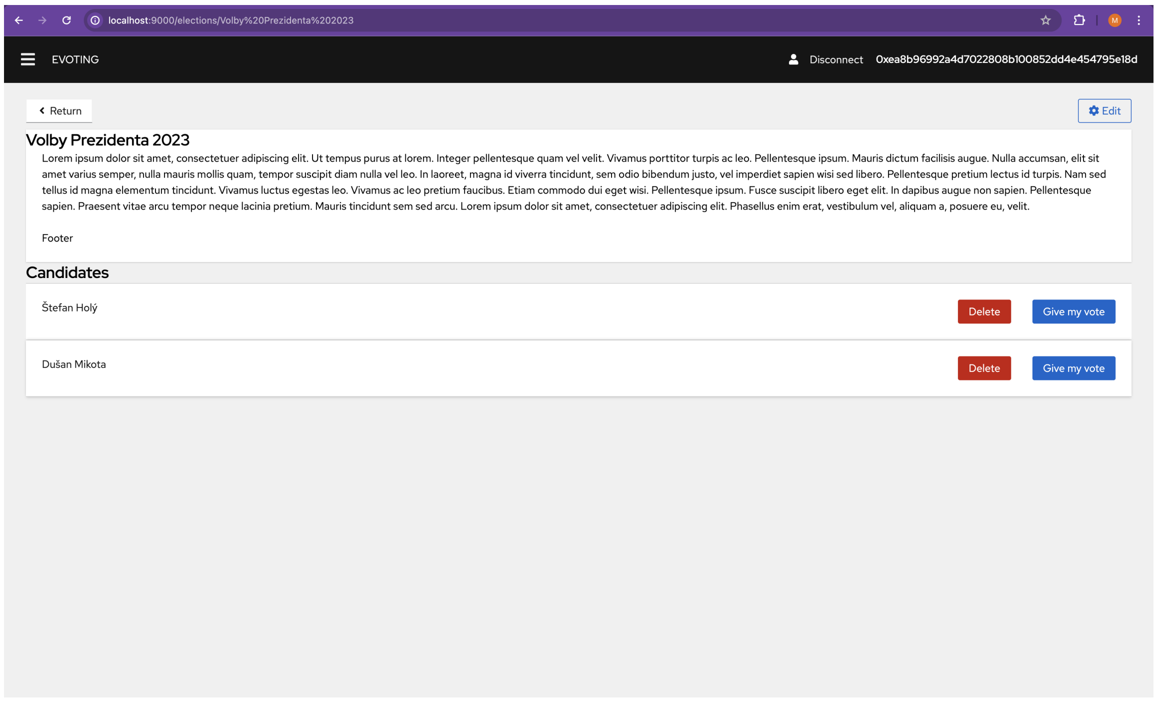
Figure B.4: The default view page together with the navigation panel.



List of elections

Volby Prezidenta 2023	OPEN	Delete
Volby NRSR 2024	CLOSED	Delete
Volby dekana FIT VUT 2023	CLOSED	Delete

Figure B.5: List of all the elections in the system.


localhost:9000/elections/Volby%20Prezidenta%202023
EVOTING
Disconnect
0xea8b96992a4d7022808b100852dd4e454795e18d
Return
Edit
Delete
Give my vote
Delete
Give my vote

Volby Prezidenta 2023

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut tempus purus at lorem. Integer pellentesque quam vel velit. Vivamus porttitor turpis ac leo. Pellentesque ipsum. Mauris dictum facilisis augue. Nulla accumsan, elit sit amet varius semper, nulla mauris mollis quam, tempor suscipit diam nulla vel leo. In laoreet, magna id viverra tincidunt, sem odio bibendum justo, vel imperdiet sapien wisi sed libero. Pellentesque pretium lectus id turpis. Nam sed tellus id magna elementum tincidunt. Vivamus luctus egestas leo. Vivamus ac leo pretium faucibus. Etiam commodo dui eget wisi. Pellentesque ipsum. Fusce suscipit libero eget elit. In dapibus augue non sapien. Pellentesque sapien. Praesent vitae arcu tempor neque lacinia pretium. Mauris tincidunt sem sed arcu. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus enim erat, vestibulum vel, aliquam a, posuere eu, velit.

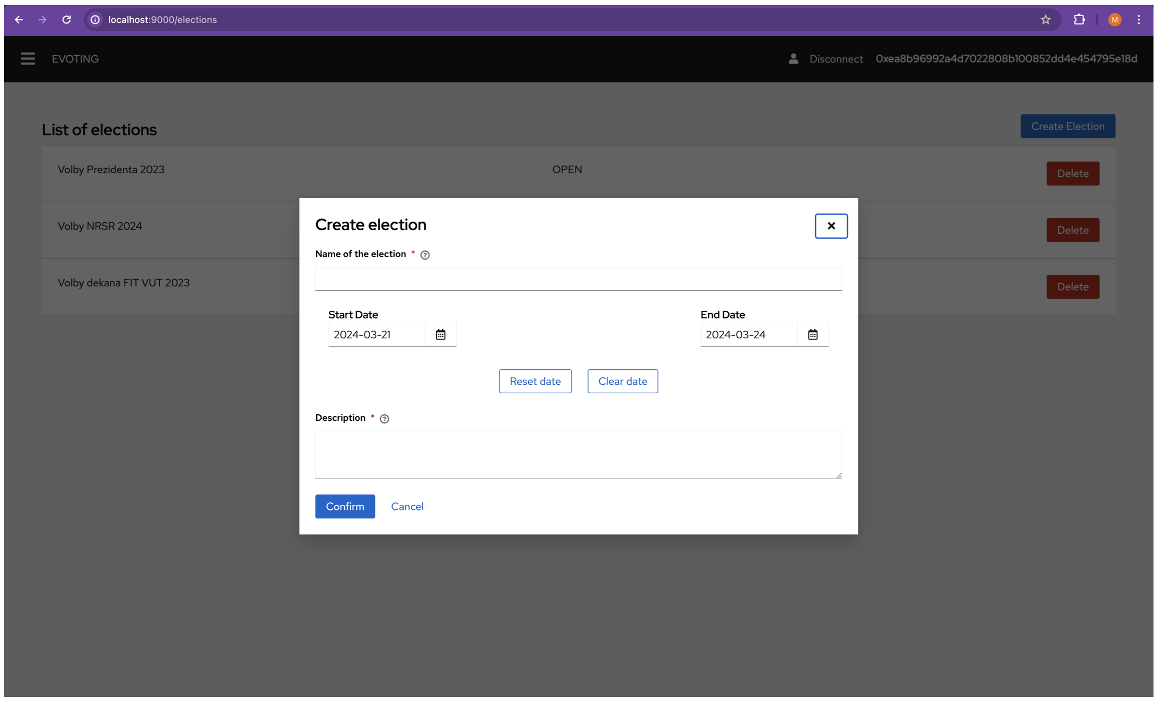
Footer

Candidates

Štefan Holý

Dušan Mikota

Figure B.6: Detail view of an election.



List of elections

Volby Prezidenta 2023	OPEN
Volby NRSR 2024	
Volby dekana FIT VUT 2023	

Create election

Name of the election *

Start Date 2024-03-21

End Date 2024-03-24

Description *

Figure B.7: Form for creating new elections.

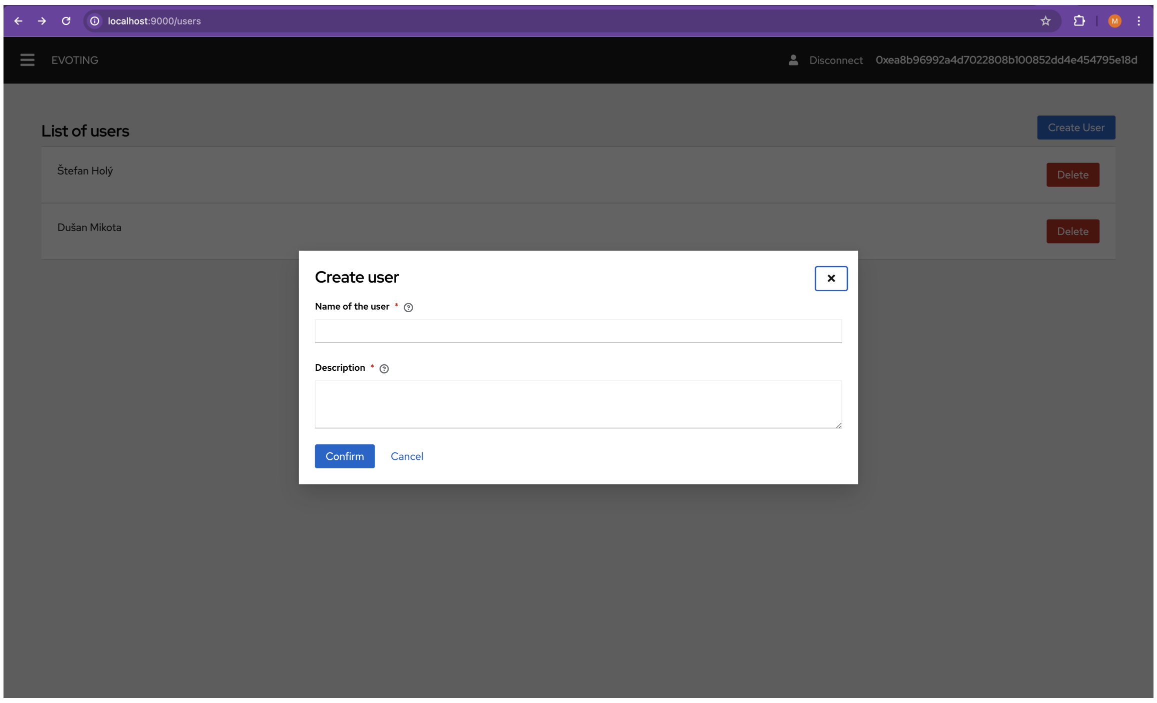


Figure B.8: Form for creating new users.

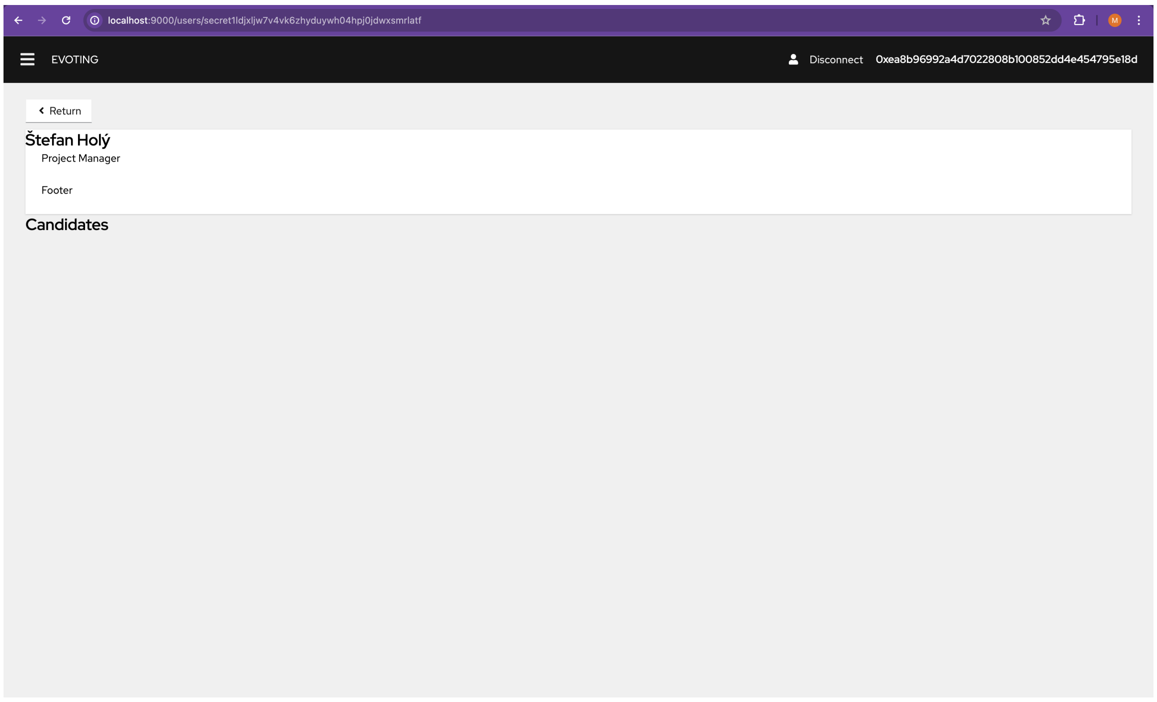

localhost:9000/users/secret1ldjxljw7v4vk6zhyduywh04hpj0jdwxsmrlatf
EVOTING
Disconnect
0xea8b96992a4d7022808b100852dd4e454795e18d
Return
Štefan Holý
Project Manager
Footer
Candidates

Figure B.9: Detail view for the enrolled users.