

BLOCKCHAIN ELECTIONS: SMART CONTRACT ELECTORAL SYSTEM
DESIGN AND IMPLEMENTATION

A Thesis
by
NATHANIEL PATRICK HERNANDEZ

Submitted to the Graduate School
Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

May 2021
Department of Computer Science

BLOCKCHAIN ELECTIONS: SMART CONTRACT ELECTORAL SYSTEM
DESIGN AND IMPLEMENTATION

A Thesis
by
NATHANIEL PATRICK HERNANDEZ
May 2021

APPROVED BY:

Cindy Norris, Ph.D.
Chairperson, Thesis Committee

E. Frank Barry
Member, Thesis Committee

R. Mitchell Parry, Ph.D.
Member, Thesis Committee

Rahman Tashakkori, Ph.D.
Member, Thesis Committee

Rahman Tashakkori, Ph.D.
Chairperson, Department of Computer Science

Michael J. McKenzie, Ph.D.
Dean, Cratis D. Williams School of Graduate Studies

Copyright© Nathaniel Patrick Hernandez 2021
All Rights Reserved

Abstract

BLOCKCHAIN ELECTIONS: SMART CONTRACT ELECTORAL SYSTEM DESIGN AND IMPLEMENTATION

Nathaniel Patrick Hernandez
B.S., Appalachian State University
M.S., Appalachian State University
Chairperson: Cindy Norris, Ph.D.

Proponents of Internet-based voting systems suggest that it might offer solutions to alleviate some of the shortcomings previous democratic institutions have fallen victim to: gerrymandering, election fraud, malicious ballot-box zoning, etc. This has become especially relevant as governments and societies wrestle with the COVID-19 pandemic, state-sponsored election interference, and claims of election fraud. However, Internet voting systems are fraught with risks and there is a wealth of research and real-world incidents which demonstrate the plethora of issues one would face in implementing such a system. Voting systems are well-understood to have notoriously difficult to fulfill requirements, which are often at odds with one another, including security, privacy, and verifiability requirements. The rise in popularity of blockchain-based technologies has renewed interest in such systems, and although it is unlikely that publicly available blockchain-based solutions can meet the requirements demanded of large-scale elections, there is potential utility in having

on-chain electoral systems available for lower-stakes decision-making. This research investigates blockchain-based decision-making through electoral system design and implementation using the Ethereum blockchain and Solidity programming language. This research demonstrates that secure and verifiable small-scale voting systems can be built using “off-the-shelf” blockchain technologies when privacy constraints are loosened. This research further demonstrates that the costs associated with operating such voting systems are steep, likely rendering such systems impractical in most circumstances and therefore an inappropriate foundation for large-scale elections. Finally, this research identifies which electoral systems and features are viable within these blockchain environments and the associated costs of supporting and operating these electoral systems and features.

Acknowledgements

I would like to thank first and foremost, my thesis advisor, Dr. Cindy Norris, for the all of the help, encouragement, advice, personal support, and patience she has afforded me throughout this experience. I am sure we both have a few new gray hairs to show for it. I cannot thank you enough. I would also like to thank Dr. James Fenwick for all of the help provided which has enabled me to finish this work. Thank you both for all of the generosity and hospitality you have shown me, I would not have made it if not for both of you and I am forever grateful.

To my committee:

- Dr. Mitch Parry, thank you for all of your instruction over the years. Your presence has always motivated me to work harder and challenged me to be mindful of the things I am working on.
- Frank Barry, you are a fountain of knowledge with more experience than seems possible to contain. I always find your words and our conversations memorable. Thank you as well for everything you have taught me over the years.
- Dr. Rahman Tashakkori, words really cannot express enough how much of an influence you have had on my life and how grateful I am to have met you. You have been like a father to me and helped me to grow in more ways than I ever

imagined I could. I aspire to be as good a person as you one day. Thank you.

I would like to thank Google, for providing me with the opportunity to attend the USENIX conference in San Francisco and meet with significant members of this field. And finally, I would like to offer a huge thank you to the NSF S-STEM scholarship program which made my undergraduate and graduate studies possible at Appalachian State University.

Contents

Abstract	iv
Acknowledgements	vi
List of Figures	xi
List of Tables	xii
List of Listings	xiii
List of Abbreviations and Symbols	xiv
1 Introduction	1
1.1 Overview of Materials	6
2 Background.	9
2.1 Elections	9
2.1.1 A Taxonomy of Electoral Systems	13
2.1.2 Evaluating Electoral Systems	20
2.2 Blockchain Technologies	27
2.2.1 Bitcoin	29
2.2.2 Ethereum	38
3 Literature Review.	50
3.1 Internet Voting	50
3.1.1 Survey of Internet Voting Systems	51
3.2 End-to-End Verifiability	67
3.2.1 Requirements	67

3.2.2	Architecture	76
4	Methods	82
4.1	Requirements	83
4.1.1	Technical Requirements	85
4.1.2	Non-Functional Requirements	87
4.2	Assumptions and Objectives	87
4.3	Tooling	89
4.3.1	Frameworks	90
4.4	Architecture and Design	94
4.4.1	Authorization Components	95
4.4.2	Election Components	103
4.4.3	Delegation Components	110
5	Results	114
5.1	Test Results	114
5.1.1	Election Components	116
5.1.2	Authorization Components	126
5.2	Analysis	127
5.2.1	Requirements	127
5.2.2	Objectives	131
6	Conclusion	133
	Bibliography	138
A	Appendix Software Documentation	147
A.1	Authorization Components	148
A.1.1	Primitive Contract Ownership	149
A.1.2	Generalizing Contract Access Control	154
A.1.3	Access Control Lists	160
A.1.4	Registries	174
A.2	Election Components	179
A.2.1	Election Contracts	179
A.3	Delegation Components	182
A.3.1	Delegation Contracts	183

Vita	184
----------------	-----

List of Figures

2.1	A chain of linked Bitcoin blocks. [35]	31
2.2	A single Bitcoin block and its internal representation. [35]	32
2.3	A chain of signatures representing an electronic coin. [35]	34
3.1	VOI System Architecture [22]	54
3.2	SERVE System Architecture [49]	58
3.3	SERVE Voting Protocol [49]	59
3.4	DVBM System Architecture [50]	64
3.5	A specification of the possible variants for an E2E-VIV system. [16]	77
4.1	The Embark dashboard and application state.	92
4.2	Testing an election contract using the Truffle framework.	94
4.3	Authorization dependency graph modeling.	98
A.1	Authorization dependency graph modeling.	149
A.2	Contract ownership dependency graph modeling.	150
A.3	Generalized contract access control.	155
A.4	Generalized contract access control by way of access control lists.	161
A.5	Simplified election registry.	175

List of Tables

2.1	Electoral systems organized by families and characteristics. [39]	14
2.2	Electoral Criteria Compliance [44,39,20,31,36]	21
2.2	Electoral Criteria Compliance [44,39,20,31,36]	22
2.3	The contents of a block [2].	32
2.4	The contents of a block header [2].	33
2.5	Ether Denominations [3]	39
2.6	Fee Schedule [51]	48
4.1	Requirements targeted for fulfillment.	83
4.1	Requirements targeted for fulfillment.	84
5.1	Simulated gas consumption by contract function.	117
5.1	Simulated gas consumption by contract function.	118
5.2	Requirements targeted for fulfillment.	128

List of Listings

4.1	Guard Usage Example	99
4.2	Delegation Structure	111
4.3	Vote Delegation	112
5.1	RangeVoteConfiguration	122
A.1	contract RestrictTo	151
A.2	interface Ownership	152
A.3	contract Owned	153
A.4	interface Authority	156
A.5	interface Authorization	157
A.6	contract Guarded	158
A.7	interface AccessControlList	162
A.8	contract BasicACL	165
A.9	contract ACLAuthority	170
A.10	interface Registry	176
A.11	contract VoterRegistrationAuthority	177
A.12	contract FirstPastThePost	179
A.13	Vote Delegation	183

List of Abbreviations and Symbols

Blockchain Technologies

ASIC Application-Specific Integrated Circuit

Ethereum

Ξ/Ð/◆ **ether**, also referred to as `eth` or ETH

EVG Ethereum Virtual Machine

OOG Out-of-Gas Exception

Legislation

VRA Voting Rights Act (1965)

UOCAVA Uniformed and Overseas Citizens Absentee Voting Act (1986)

NVRA National Voter Registration Act, also known as the Motor Voter Act (1993)

HAVA Help America Vote Act (2002)

MOVE Military and Overseas Voter Empowerment Act, amended UOCAVA (2009)

VEA Voter Empowerment Act (2015)

Legislative Organizations

DoD Department of Defense

EAC Election Assistance Commission, established under HAVA (2002)

FVAP Federal Voting Assistance Program, established under UOCAVA (1986)

NIST	National Institute of Standards and Technology
SPRG	Security Peer Review Group, technical group assembled by FVAP to review SERVE
(DC)BOEE	Washington D.C. Board of Elections and Ethics
OSDV	Open Source Digital Voting Foundation

Voting

LEO	Local Election Official
VVPAT	Verifiable Voter Paper Audit Trail
VVSG	Voluntary Voting System Guidelines
DRE	<i>Direct Recording Electronic</i> voting machines

Voting Systems

IVAS	Integrated Voting Alternative Site
VOI	Voting Over the Internet project (2000)
SERVE	Secure Electronic Registration and Voting Experiment (2004)
DVBM	D.C. Digital Vote-by-Mail system

Elections

MMD	<i>Multi-Member District</i> , a district with multiple seats available for candidates to hold or multiple choices available for selection.
SMD	<i>Single-Member District</i> , also known as single-winner district
MMP	Mixed-Member Proportional
PR	Proportional Representation

Electoral Criteria

MC	<i>Majority Criterion</i>
MMC	<i>Mutual Majority Criterion</i>
ISDA	<i>Independence of Smith-dominated Alternatives</i>

IIA	Independence of Irrelevant Alternatives (IIA)
LIIA	Local Independence of Irrelevant Alternatives (LIIA)
CC	Consistency Criterion
PC	Participation Criterion (PC)
PC	No Favorite Betrayal (NFB)

Electoral Systems

LPR	List Proportional Representation
STV	Single Transferable Vote
FPTP	First Past the Post
BV	Block Vote
PBV	Party Block Vote
AV/IRV/RCV	<i>Alternative Vote</i> , also known as <i>Instant-Runoff Voting</i> , <i>Ranked Choice Voting</i> , transferable vote, or preferential vote
RV/SV	<i>Range Vote</i> , also known as <i>Score Voting</i>
TRS	Two-Round System

Chapter 1 – Introduction

This research investigates blockchain-based decision-making through electoral system design and implementation. Decision-making, in its idealized form, is a process which entails the evaluation of some problem's context and environment in order to identify relevant criteria and variables worth considering. An analysis of these criteria and variables can be used to identify a problem's decision-space: the individual impact of each variable, their trade-offs, and ultimately a set of viable choices to pick from. Weights might be applied to the criteria — determined by the choice-maker's personal values, beliefs, and preferences — which, when considered with each of the viable choices' risks and rewards, could be used to produce a final choice with an optimal outcome. This decision-making process is a non-trivial procedure which becomes increasingly difficult to consider as the consequences of the decisions become more significant; and the analysis process harder to evaluate as the variables, factors, goals, and the general environment of the problem becomes more complex. For example, it is much more difficult to make decisions that have long-lasting impacts than when the impacts are strictly immediate; likewise, decision-making which involves complex

ethical concerns, e.g., life or death, are much more difficult to consider than those which do not.

Decision-making in reality rarely lives up to the standards required to meet that of the idealized process; full-knowledge of an environment is rare and constraint satisfaction is often sufficient, i.e., most decisions require only a “good enough” solution. Heuristics are often used to trade optimality, completeness, accuracy, or precision for speed. Tacit knowledge or intuition is used to fill the gaps in one’s explicit knowledge or substituted for reliable information. And even thorough decision-making processes can still produce poor results: over-analysis can cause “analysis paralysis,” preventing one from arriving at any decision at all; too much information can cause information overload, preventing one from properly assimilating knowledge, or worse, producing an illusion of knowledge; and still, unconscious biases, instincts, and emotions can subvert well-intended decision-making processes.

All of the complexities regarding decision-making become more complex at scale as we attempt to manage group-based decision-making processes: the processes themselves become fractured as they are distributed across individuals, personal biases manifest themselves as group biases, and the various procedures and algorithms used to collect, accumulate, and evaluate individuals’ preferences are themselves flawed and biased in their own ways. [39] Democratic communities aim to make de-

cisions by granting and counting individuals' votes; and although voting may seem conceptually simple, there are practical and theoretical complexities which are difficult to overcome. The complexities with respect to actually implementing such systems often result in systems which are mired with imbalances, fraud, and corruption. The disenfranchisement of citizens is one example of this and it occurs in many ways: lobbying, vote buying, gerrymandering, malicious ballot-box zoning, and overly aggressive voter identification requirements to name a few. Many other forms of voter suppression and election fraud exist which provide real-world examples of the shortcomings a democratic-process might incur and demonstrate the complexity of designing such decision-making systems. [26,39,12]

Issues such as these in conjunction with the growing usage and near-ubiquitous nature of the Internet and personal computers has led many to consider whether the Internet could offer a new medium through which votes could be cast. Proponents of Internet-based voting systems suggest that it might offer solutions to alleviate some of the shortcomings previous democratic institutions have fallen victim to. This has become especially relevant as governments and societies wrestle with the COVID-19 pandemic and state-sponsored election interference. However, online voting has been shown to be fraught with risks, and there has been a wealth of research and events which demonstrate the plethora of issues one would face in implementing such a

system. [30,50]

The research with respect to large-scale Internet-based voting systems is clear: the potential rewards of such systems do not currently outweigh the risks. [29] This research does not intend to retread this ground, nor does it set out to advocate for such a system. However, digital organizations and communities which exist and operate entirely “on-chain” are everyday springing into existence; this has become especially true since the advent of blockchain technologies such as Ethereum, which provides decentralized foundations for securely executing arbitrary code. [17,15] Like all organizations these decentralized organizations and communities need decision-making processes and systems to reach group consensus. To that end, this research explores blockchain-based electoral systems: their feasibility, scalability, security, and potential implementations. The unique constraints and costs stemming from the underlying blockchain systems and the nascent nature of decentralized organizations demand focus; thus, the viability and implementation of several electoral systems is explored. Smart contract implementations, written in Solidity, are provided which offer insights into the potential designs, pitfalls, and costs required to support various electoral systems and features. This research demonstrates that secure and verifiable small-scale voting systems can be built using “off-the-shelf” blockchain technologies which support code-execution; however, this is likely only

possible where privacy, anonymity, and receipt freedom constraints are loosened. This research further demonstrates that the costs associated with operating such voting systems are steep, likely rendering such systems impractical in most circumstances and therefore an inappropriate foundation for large-scale elections. Finally, this research identifies which electoral systems and features are viable within these blockchain environments and the associated cost of supporting and operating these electoral systems and features.

Some of the questions this research initially sought to answer includes:

1. What modalities of governance and electoral systems exist?
2. What are the advantages and disadvantages of these electoral systems?
3. What kinds of election processes and procedures, governance modalities, and electoral systems are appropriate within blockchain ecosystems and decentralized organizations?
4. Of the election processes available, which are blockchain ecosystems capable of supporting?
5. What is the current state of research with respect to digital and online voting?
6. Can blockchain technologies be leveraged to improve the reliability, verifiability, or security of elections?

1.1 Overview of Materials

This material is divided into six chapters:

1. *Introduction*, which introduces this material, poses initial questions, and an offers overview of the subject matter to be reviewed.
2. *Background*, which introduces background materials regarding:
 - *Elections and Electoral Systems*, their modalities, component parts, risks, an overview of some of the most common implementations of electoral systems, and an introduction to the tools available for analyzing and selecting electoral systems.
 - *Blockchain Technologies*, Bitcoin and Ethereum, their basic concepts and abstractions, internal data structures, algorithms, network architectures and topologies, the properties thereof, and the Ethereum Virtual Machine (EVM).
3. *Literature Review*, which introduces and explores:
 - *Internet Voting*, its general procedures and requirements, reviews several novel Internet voting experiments: their architectures, criticisms, and outcomes.
 - *End-to-End Verifiable Voting*, its purpose, the requirements to support end-to-end verifiability (technical and non-functional), architectural op-

tions, existing systems, and common cryptographic techniques available for use in electoral systems.

4. *Methods*, which covers the methodologies used within this work, borrowing from the tools and methodologies, introduced in Chapter 2's *Background*, for selecting electoral systems, and from the technical and non-functional requirements introduced in Chapter 3's *Literature Review* on end-to-end verifiability:

- *Requirements and Design Principles*, the guiding principles and requirements which this research aims to support.
- *Architecture and Implementation*, the overall design to support various electoral systems: registration and authorization of voters, voter representation and vote delegation, vote casting and tallying processes, and contracts for holding elections backed by various electoral systems: First-Past-the-Post (FPTP), Range Vote (RV), etc.

5. *Results*, which reviews the results produced through methodologies covered in Chapter 4's *Methods* in two parts:

- *Test Results*, the testing methodologies and frameworks leveraged, an analysis of the algorithmic complexities, election simulations and costs, and test results of the various electoral systems implemented.
- *Analysis*, which reviews the shortcomings, limitations, and areas of im-

provement required with respect to the methodologies chosen in Chapter 4's *Methods*, offers an interpretation of the results produced in Chapter 5's *Results*, takes a look back at the initial questions introduced in Chapter 1's *Introduction*, offers some context and perspective with respect to how this material fits into the greater body of previous work, and offers some suggestions for future work.

6. *Conclusion*, summarizes the material covered in the previous chapters, reviews the major results produced in this work, and offers closing thoughts.

Chapter 2 – Background

This chapter is broken into two major sections:

- *Elections*, which examines the modalities, component parts, and risks involved when selecting electoral systems; provides an overview of some common and relevant electoral systems; and introduces tools available for analyzing and selecting electoral systems.
- *Blockchain Technologies*, which examines Bitcoin and Ethereum, their internal data structures, algorithms, network topology, architecture, and properties thereof.

2.1 Elections

Elections are perhaps the most obvious and intuitive means of allowing members of a population to express their will and take part in their governance process. An election is the generalized process of allowing individuals to express their preference, typically by way of vote, as a means to come to consensus as a group. The

generalized goal of elections is to reach a consensus which accurately and fairly reflects the preferences of the participating voters. This, at first glance, seems like a trivial problem, and generally is in circumstances where the number of voters and choices is small in number. However, elections become complex as the number of actors, choices, or election cycles increase. There are social, mathematical, and practical engineering constraints which all voting systems are bound by, subjected to, and forced to address.

Electoral Systems

An electoral system, is the combination of rules, norms, and procedures which define how a final result, and ultimately consensus, will be determined during an election. Electoral systems can be considered as a composition of three components: ballot, choices, and tallying algorithm. These three components can be woven together to produce a wide spectrum of electoral systems with varying characteristics and properties. The choice and implementation of electoral system has a direct and profound impact on the ways in which democratic systems can and will operate as well as on the perceived legitimacy of the governance model. The decisions concerning the implementation details of electoral systems are among the most important decisions that any democratic organization will make; the choice of design impacts all future

decision making processes and shapes the future of the governance model itself. A poorly designed electoral system can have disastrous effects on the health and perceived legitimacy of a democratic organization both in the immediate and long-term future. Further, once chosen, an electoral system can be difficult to amend as political interests respond to and solidify around the incentives presented to them which are inherent to the electoral system chosen. [39]

Choices A electoral system's choices are the set of options that a voting actor can select from; they are the *who* or *what* being decided in an election. The choices available may be determined by primary vote, polling, write-in, debate, etc. or some other methodology or combination of methodologies.

District Magnitude The number of choices that will be selected as winners is of great importance in an election. In representative democracies the measurement of seats (choices) available is known as district magnitude. An election where the district magnitude is one, i.e., where a single candidate or choice is to be elected or selected, is known as a single-member district (SMD) or single-winner district. An election where the district magnitude is greater than one, i.e., multiple candidates are to be elected, is known as a multi-member district (MMD) or multi-winner district.

Ballot A ballot is the means through which voting actors express their vote. The structure of the ballot influences exactly *how* the voter can express their preferences for a choice or choices, e.g., how many votes an individual is able to cast. This is directly influenced by the tallying method. Ballots, in some texts, are regarded as a process of voting; here the term is used to describe both the medium through which a voting actor marks their choice (e.g., paper, punch card, or electronic machine), as well as the rules regarding how they can mark said medium. The ballot might be thought of as the data structure used to support the tallying algorithm.

Preference Marking Preference marking is a ballot marking process where the electorate is given an opportunity to rank choices: cardinally, ordinally, or by approval. This expression of preference offers greater insights and details into the will and desires of the electorate and can be used by the tallying method and election officials to more optimally determine winners in an election. Preference marking may be mandatory or optional depending on the design of the electoral system.

Electoral Formula The electoral formula is the process used to translate ballots into winning decisions. The electoral formula is best characterized by its tallying method, and in multi-member districts, its proportionality.

Tallying Method The tallying method, or tallying algorithm, is closely tied to how a voter is allowed to mark a ballot and affects how a marked ballot is ultimately counted in the final tally of an election. Different processes of varying complexities exist to count votes. Naturally, the process through which votes are tallied will have a direct and significant influence on the outcome of an election.

Proportionality Proportionality characterizes how closely votes cast translate into choices won. The significance of proportionality becomes especially relevant as the district magnitude increases. The closer in proportion that winning choices are to the votes cast for them, the greater the system expresses proportionality. Analyzing the votes cast and how they map to winning choices and “wasted votes” is the easiest method to determine the proportionality of an electoral formula and is commonly expressed as the index of disproportionality.

2.1.1 A Taxonomy of Electoral Systems

Electoral systems are generally broken into three broad families based primarily on the properties described above. The three families are plurality/majority, proportional representation, and mixed. An overview various electoral systems is offered in Table 2.1 to help visualize the landscape of electoral systems available and where they fall.

Table 2.1: Electoral systems organized by families and characteristics. [39]

Electoral System		Proportional Representation	Plurjority			Mixed
			Plurality	Majority	MMD	
STV	Single Transferable Vote	•				
	List PR	•				
FPTP	First Past The Post		•			
BV	Block Vote		•		•	
PBV	Party Block Vote		•		•	
	Preferential Block Vote			•	•	
AV	Alternative Vote			•		
RV	Range Vote			•		
TRS	Two-Round System		•	•		
	Parallel	•	•	•	•	•
MMP	Mixed-Member Proportional	•	•	•	•	•

MMD: Multi-Member District, a district where multiple seats are available for candidates to hold or where multiple choices will be determined winners.

2.1.1.1 *Plurality/Majority*

Plurality and majority electoral systems are simple in principle although not necessarily in design. After votes have been tallied the choices with the most resulting votes are the decided winners.

Plurality Voting A plurality voting system is one where the winning choices are the ones with the greatest number of votes *but not necessarily an absolute majority*. The United States uses plurality electoral systems almost exclusively.

First-Past-The-Post First-Past-the-Post (FPTP) is a plurality single-member district electoral system used widely across the world and almost exclusively in the United States of America. FPTP is one of the simplest electoral systems to understand; the voter is presented with a set of choices on a ballot and is able to select *one and only one* of the choices. The ballots are then collected and tallied by counting the number of votes cast for each choice; the choice with the most votes wins.

Majority Voting A majority voting system is characterized such that a choice will only be considered the winner with an absolute majority of votes, that is, *greater than 50% of the votes*. Most majority voting systems take advantage of preference marking or multiple election cycles, where less-popular choices are removed in each cycle, to form a majority.

Range Vote Range Vote (RV), also known as Score Voting (SV), is a single-member district majority electoral system. It takes advantage of cardinal preference marked ballots to determine a majority choice. This form of voting allows voters to express a preference between choices and also the degree of preference between

choices.

A range voting ballot allows voters to select a non-negative integer, up to some maximum (usually 9 or 99), which expresses some degree of approval for a particular option, e.g., 0 for least preferred or 9 for most preferred. Some range voting systems also allow for disapproval voting, down to some minimum, to express disapproval for a particular option, but this is less common. The system may also support a “No Opinion” mark that can be cast if a voter is ignorant or indifferent to some choice; this is usually considered the default mark if no mark is made for a particular option and does not count for or against the option. Once all ballots are cast and collected they are tallied. The tallying process is as follows: votes for an option are summed together and divided by the number of ballots that actively voted for that option (marked anything other than “No Opinion”). The option with the highest average score wins.

An approval voting system is the simplest and most restricted kind of range voting, the range of votes is restricted to either 0 or 1. A voter can approve as many options as they want.

2.1.1.2 Proportional Representation

The objective of a Proportional Representation (PR) electoral system is to produce winners from an election that accurately reflect the will of the people and the votes they cast by reducing the disparity between shares of votes cast for choices and shares of winning choices; this is especially relevant when considering seats won in a representative governance models. PR operates by providing a cross section of winners in an election which map proportionally to the votes cast for choices. For example, if a quarter of voters in an election desire some outcome then the election results should reflect that by producing winners in proportion to those voters' desires; i.e., a quarter of the winners of the election should be that of the voters' desired outcome.

There are a number of benefits associated with PR systems:

- Votes translate into choices won with greater proportionality.
- Results in fewer wasted votes.
- Offers minority groups greater representation.
- Restricts regional fiefdoms.
- Promotes greater long-term political health and negotiation.
- Results in greater voter participation and perceived legitimacy.
- Supports a more inclusive cross-section of representatives.

There are also several disadvantages associated with PR systems:

- Quick and coherent policies can be difficult to pass.
- Legislative gridlock can occur when factions are formed.
- Fragmentation of strong parties can occur in cases of extreme pluralism.
- Minority parties can hold parties ransom in coalition negotiation, thus offering smaller parties greater authority than perhaps deserved.
- It is difficult to enforce accountability by throwing out parties or candidates.
- Potentially difficult for electorate to understand or administrators to implement.

Single Transferable Vote Single Transferable Vote (STV) is a PR multi-member electoral system that operates similar to the Alternative Vote (AV). Like AV, STV leverages preference marked ballots. STV operates as follows. Electorate ordinally rank options on their ballots. Ballots are collected and tallied. While tallying, if any option reaches the quota, i.e., receives the minimum number of votes required, they are immediately declared a winner in the election and awarded a seat for the district. Surplus votes for a winning candidate are redistributed to other candidates based on ballot-preference. If the quota cannot be reached by any candidate then the least popular candidate is eliminated and votes are redistributed based on ballot-preference. [46]

There are many quota algorithm implementations which can be used in STV elections. The quota algorithm used will determine the minimum number of votes required to win a seat in an election. One such quota algorithm, perhaps the most popular, is the Droop quota: [39]

$$Quota_{Droop} = \frac{votes}{seats+1} + 1$$

If the quota for winning is met surplus vote allocation occurs, surplus votes are redistributed to second choices. STV implementations can differ here. It would not necessarily be fair to select only some people to have their votes redistributed, since voters can have different second choices. However, it might be acceptable if surplus ballots were randomly selected for second choices. In large enough elections with large enough surpluses the second choices should be statistically proportional. However, non-deterministic winners are not ideal and can lead to infinite recursion in some cases. Instead, everyone's votes (from ballots whose candidate won) can be redistributed to their second choices (or third, fourth, etc. if their second has already won) as a fraction of a vote; this algorithm will usually be based on surplus votes, total votes, previous votes, etc. There are many surplus allocation algorithms. [39] There are also surplus vote allocation algorithms for subsequent surplus vote allocation, for when a second choice has already won. [39,46]

When no option can win excluded candidate vote allocation occurs, the weak-

est options are eliminated and their votes reallocated. There are also a number of algorithms available when deciding how votes should be redistributed which options are eliminated. [39]

2.1.2 Evaluating Electoral Systems

Given the wide range of electoral systems that exist, or could conceivably exist, it becomes necessary to introduce techniques that can be used to objectively analyze and evaluate the various characteristics of electoral systems. Social choice theory provides tools which one can use to examine and categorize voting systems: by their advantages, their disadvantages, and caveats.

2.1.2.1 Criteria

It can be difficult to objectively judge and select an electoral system; the choice of electoral system will have impacts on the groups, ideologies, and candidates that are likely to succeed in a given governance model, and the consequences of an electoral system are not always immediately obvious. In order to compare and contrast electoral systems more objectively, there are criteria that exist to express and describe some characteristic of an electoral system. Table 2.2 provides a summary of criteria compliance across various electoral systems to help visualize the unique

range of compliance across various electoral systems. The criteria fall into four major categories:

1. *Absolute Result Criteria*,
2. *Relative Result Criteria*,
3. *Ballot Counting Criteria*, and
4. *Strategy Criteria*.

Table 2.2: Electoral Criteria Compliance [44,39,20,31,36]

Electoral Criterion	Electoral System											
	<i>Approval</i>	<i>Borda Count</i>	<i>Copeland</i>	<i>IRV (AV)</i>	<i>Kemeny-Young</i>	<i>FPTP</i>	<i>Range Voting</i>	<i>Ranked Pairs</i>	<i>Runoff Voting</i>	<i>Schulze</i>	<i>Sortition</i>	<i>Random Ballot</i>
Majority	Rated	✗	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗
Majority Loser	✗	✓	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗
Mutual Majority	✗	✗	✓	✓	✓	✗	✗	✓	✗	✓	✗	✗
Condorcet	✗	✗	✓	✗	✓	✗	✗	✓	✗	✓	✗	✗
Condorcet Loser	✗	✓	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗
Smith/IDSA	✗	✗	✓	✗	✓	✗	✗	✓	✗	✓	✗	✗
LIIA	✓	✗	✗	✗	✓	✗	✓	✓	✗	✗	✓	✓
IIA	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓
Cloneproof	✓	✗ ⁺	✗ ⁺	✓	✗ ⁺	✗ ⁺	✓	✓	✗ ⁺	✓	✗	✓
Monotone	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓
Consistency	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓
Participation	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓
Reversal Symmetry	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓	✓

Table 2.2: Electoral Criteria Compliance [44,39,20,31,36]

Electoral Criterion	Electoral System											
	<i>Approval</i>	<i>Borda Count</i>	<i>Copeland</i>	<i>IRV (AV)</i>	<i>Kemeny-Young</i>	<i>FPTP</i>	<i>Range Voting</i>	<i>Ranked Pairs</i>	<i>Runoff Voting</i>	<i>Schulze</i>	<i>Sortition</i>	<i>Random Ballot</i>
Polytime	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n!)$	$O(n)$	$O(n)$	$O(n^3)$	$O(n)$	$O(n^3)$	$O(1)$	$O(n)$
Resolvable	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗
Summable	$O(n)$	$O(n)$	$O(n^2)$	$O(n!)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	$O(n)$
Later-no-Harm	✗	✗	✗	✓	✗	—	✗	✗	✓	✗	✓	✓
Later-no-Help	✓	✓	✗	✓	✗	—	✓	✗	✓	✗	✓	✓
No Favorite Betrayal	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✓
Ballot Type	☑	☒	☒	☒	☒	☑	☒	☒	☑	☒	—	☑
Ranks =	✓	✗	✓	✗	✓	—	✓	✓	—	✓	—	—
Ranks >2	✗	✓	✓	✓	✓	✗	✓	✓	✗	✓	—	✗

✓ Criterion is supported.

✗ Criterion is unsupported.

— Criterion is not applicable.

? Criterion is supported under some conditions.

☒ Choices are cardinally ranked by preference.

☒ Choices are ordinally ranked by preference.

☒ Vulnerable to spoilers.

☒ Vulnerable to teams.

☒ Vulnerable to crowds.

☑ Multiple yes/no choice-selections supported.

☑ Single yes/no choice-selection supported.

Absolute Result Criteria The absolute criteria express whether a candidate must or must not win given the state of some ballots. [37,20,21]

Majority Criterion The Majority Criterion (MC) states that a candidate who is preferred by a majority of voters must win. This is expressed in two flavors:

1. *Ranked*, when a choice is preferred by a majority of voters, the choice must win.
2. *Rated*, when a choice is given a perfect score by a majority of voters, the choice must win.

In a ranked electoral systems these two majority criteria are identical in nature.

Mutual Majority Criterion The Mutual Majority Criterion (MMC) states that if a subset, S , of candidates is strictly preferred over every candidate in the absolute complement of subset S , then the winner must come from subset S .

Condorcet Criterion The Condorcet criterion states that a choice who beats out every other choice in a pairwise comparison will win.

Condorcet Loser Criterion The Condorcet loser criterion states that a choice who loses to every other choice in a pairwise comparison will always lose.

Relative Result Criteria The relative result criteria express when a candidate should or should not win given a win in a similar circumstance. [20,21,37]

Independence of Smith-dominated Alternatives Independence of Smith-dominated Alternatives (ISDA) states that an added or removed Smith-dominated

choice, one which would lose in direct pairwise competition with every other choice, will not affect the result of the contest.

Independence of Irrelevant Alternatives Independence of Irrelevant Alternatives (IIA) is a criterion which states that adding or removing a non-winning candidate should not impact the end result.

Local Independence of Irrelevant Alternatives Local Independence of Irrelevant Alternatives (LIIA) is a criterion which states that removing a candidate will not disrupt the transitive ordering.

Independence of Clone Alternatives The independence of clone alternatives, cloneproof, states that the outcome will not change if non-winning candidates similar to an existing candidate are added as choices. There are three flavors which fail independence of clones:

1. *Spoilers*, which are clone negative choices that decrease the chance of a similar choice winning by spreading votes across multiple choices.
2. *Teams*, which are clone positive similar choices that together increase the chance of their winning.
3. *Crowds*, which are non-winning choices that when cloned change the winner without themselves becoming the winner.

Monotonicity Criterion The monotonicity criterion, monotone, states that ranking a winning choice higher will not impact the end result.

Consistency Criterion The Consistency Criterion (CC) states that a winning choice in two complement sets of ballots should remain the winner in a final tally combining the two sets.

Participation Criterion The Participation Criterion (PC) states that voting honestly will always produce better results than not voting at all.

Reversal Symmetry The reversal symmetry criterion states that when individual voter preferences are universally inverted the original winner will never win.

Ballot Counting Criteria Ballot counting criteria concern the process of vote tallying and winner determination. These criteria are especially important with respect to the practical implementation of an electoral system. [44,39,6]

Polynomial Time The polynomial time criterion, polytime, states that the electoral system can calculate the winner in linear time with respect to the number of voters and in polynomial time with respect to the number of candidates.

Resolvable A resolvable electoral system is one where determining a winner should be entirely deterministic, i.e., the electoral system should not depend on random processes such as coin flipping. This criterion is less important in large elections where ties are unlikely to occur.

Summability Summability is a criterion used to express how computationally complex it is to store vote data in a compressed format, and consequentially, how difficult it is to pre-tally votes at individual polling stations and transmit those tallied results to a central counting authority for final counting. Votes are expected to be mapped to a summable array which can be used to determine the winner. The summability criterion is considered k^{th} -order summable if we can map n candidates to a matrix of size n^k . If no k exists the electoral system is considered non-summable.

Strategy Criteria The strategy criteria relate to the incentives and ability for voters to vote using some strategy to produce a desired election result. [44,5,39]

Later-no-Harm Criterion This criterion states that ranking a preference later on a ballot will not harm a choice already listed.

Later-no-Help Criterion This criterion states that ranking a preference later on a ballot will not help a choice already listed.

No Favorite Betrayal No Favorite Betrayal (NFB) states that ranking a choice above your preferred choice will not produce a more desirable or preferred result.

Ballot Format The ballot formats define how a voter is able to express themselves on a ballot. [39]

Ballot Type The ballot type defines how a voter is permitted to mark their ballot. Popular ballot types include single mark, approval, ranked (ordinal), and scored (cardinal).

Equal Ranks A ballot that allows a voter to express equal support for multiple candidates is said to support equal ranks.

Over 2 Ranks A ballot which allows a voter to express interest for a choice in non-binary terms is said to support over 2 ranks, e.g., ordinal and cardinal ballots.

2.2 Blockchain Technologies

This research considers the utility of blockchain technologies with respect to election systems. A blockchain is a distributed transactional database system that tracks transactions in ever-growing linked blocks. Blockchain technologies have been

used to create immutable public ledgers for tracking currency, assets, and rights data. The most notable technology blockchain technology in use today is Bitcoin.

Blockchain technology exists to provide a solution to the Byzantine Generals' Problem in a distributed computing environment.¹ It allows parties on an untrusted and unreliable network to build a trusted source of truth for transaction history. Blockchains use various algorithms to score different versions of transaction history over time to reach consensus within their network. Blockchains might be best described as a tool used to commodify trust.

There are interesting overlaps and parallels to be drawn from the consensus mechanisms used in these, and other computer networks, and the consensus mechanisms leveraged to make decisions as a group in democratic governance systems.

¹The Byzantine Generals' Problem is a thought experiment which helps to illustrate some of the difficulties regarding coordination over untrusted networks. In short, two generals wish to coordinate an attack on an enemy fortress. They must attack simultaneously to succeed in taking the fortress. However, the general's only means of coordination is via courier, and the only routes available for the couriers to travel necessitate travel through the fortress. How do the generals coordinate their attack considering that their couriers might be intercepted, or worse still, replaced with dishonest spies?

2.2.1 Bitcoin

In 2008 the seminal white paper, *Bitcoin: A Peer-to-Peer Electronic Cash System*, was published under the moniker Satoshi Nakamoto. [35] This white paper outlined ideas for a new form of currency, Bitcoin. Bitcoin promised to be the first of its kind; it would become the world’s first decentralized digital currency that would require no trust to authenticate timestamped transactions. It would do this by combining cryptography, a proof-of-work system, and “miners” to create a revolutionary new concept that is now known as blockchain technology. In short, blockchain technology enables individuals who do not trust one another to reach consensus via a trustless platform. More concretely, Bitcoin can be used by Alice and Bob to send money from one to the other over a decentralized network.

2.2.1.1 Network Topology

The Bitcoin network is structured as a peer-to-peer network composed of a variety of node types. Each node supports varying functionalities and features based on their use case. Common functionalities and features required include: [2]

- Mining functionality, to support creating new blocks by solving proof-of-work problems.
- A wallet, to offer users a way to manage their keys plus send and receive

transactions.

- A copy of the full blockchain, which allows nodes to verify transactions independent of other nodes in the network.
- Network routing capabilities, which allows nodes to propagate transactions and discover new nodes.

Node Types The most common node types, classified by their corresponding functionalities, are as follows: [2]

- A *reference client* contains wallet, miner, a full copy of the blockchain, and network routing functionality.
- A *full node* contains a full copy of the blockchain and network routing functionality.
- A *mining node* contains a miner, a full copy of the blockchain, and network routing functionality.
- A *lightweight wallet* contains a wallet and network routing functionality. These nodes depend on *full nodes* to verify transactions for them. These nodes are usually found on mobile devices where storage is limited and the size of the blockchain is inhibitive.

2.2.1.2 Fundamental Concepts

The Bitcoin blockchain is structured as a linked list of blocks, as seen in Figure 2.1. Each block contains a set of transactions and a reference to the previous block in the chain. Blocks are identified by the SHA-256 hash of its header. It is helpful to imagine the blockchain as being blocks stacked vertically, each additional block helping to secure the previous blocks laid before it.

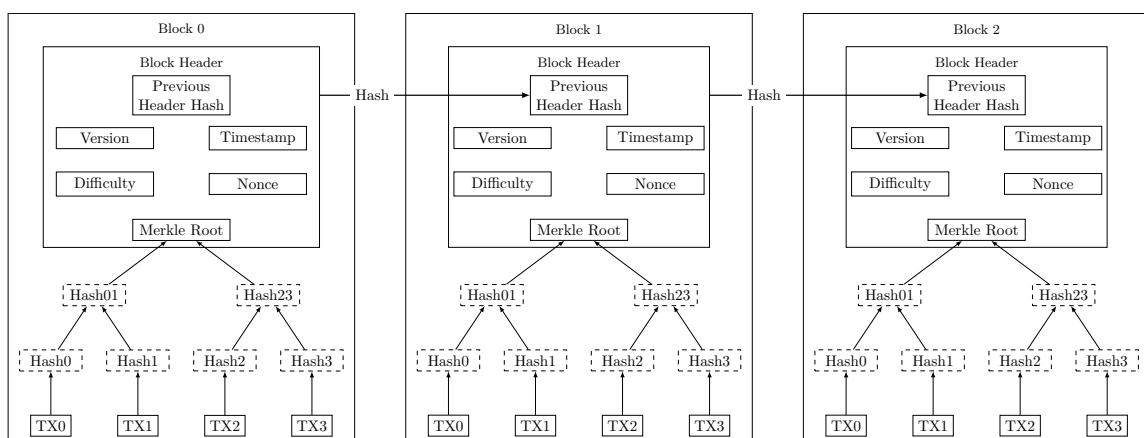


Figure 2.1: A chain of linked Bitcoin blocks. [35]

Blocks The block is the Bitcoin blockchain primitive. Blocks serve as a time-stamping tool for transactions within the network and also prove that some amount of work (computation) occurred. The contents of a block are listed in Table 2.3. Figure 2.2 illustrates the layout of a block.

Table 2.3: The contents of a block [2].

Size	Field	Description
4 bytes	Block Size	The size of the block, in bytes, following this field
80 bytes	Block Header	Several fields form the block header
1–9 bytes (VarInt)	Transaction Counter	How many transactions follow
Variable	Transactions	The transactions recorded in this block

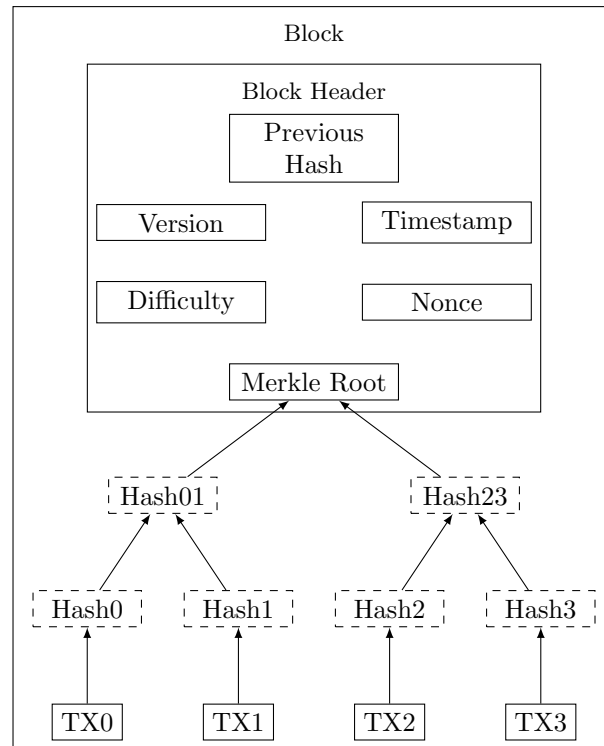


Figure 2.2: A single Bitcoin block and its internal representation. [35]

Block Header The block header is responsible for tracking the metadata of the Bitcoin block. It is also used to represent the block as a whole by SHA-256

hashing the contents of it. The contents of the block header are as seen in Table 2.4.

Table 2.4: The contents of a block header [2].

Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the Merkle tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)
4 bytes	Difficulty Target	The proof-of-work algorithm difficulty target for this block
4 bytes	Nonce	A counter used for the proof-of-work algorithm

Merkle Trees A Merkle tree is a data structure that allows one to efficiently verify the contents of a large amount of data. Merkle trees are used extensively in peer-to-peer networks to ensure that blocks of data arrive unaltered and undamaged. The root of a Merkle tree is called a Merkle root. Merkle trees are composed of nodes of hashes. They have the unique property of allowing the verification of the existence of a hash in the tree in $O(\log(n))$ time.

Transactions In Satoshi's original white paper, a coin was defined as tokens transferred via a chain of digital signatures.

“We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous

transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.” [35]

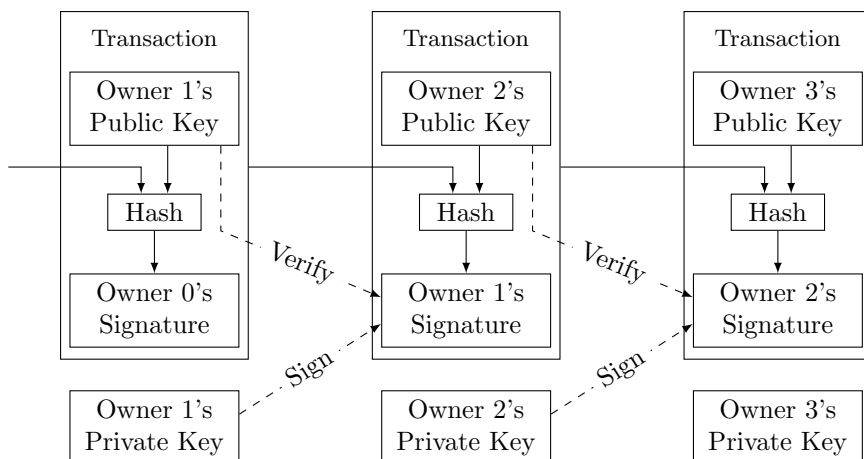


Figure 2.3: A chain of signatures representing an electronic coin. [35]

Figure 2.3 demonstrates Owner 1 hashing the transaction that gave her ownership of the coin with the public key of the new owner (Owner 2), then signing that hash with her private key and publishing that as a new transaction. Owner 2 repeats the process to send his coins to Owner 3.

Proof-of-Work The bitcoin protocol uses a *Proof-Of-Work (POW)* algorithm similar to hashcash. [4,35] The goal of this proof-of-work algorithm is to create a problem that is easy to verify for correctness but difficult to solve for. The proof-of-work algorithm provides a means for mining nodes to be pseudorandomly selected to build

a block of transactions. The probability that a miner will be selected is directly tied to the amount of *work* a miner does.

Algorithm The proof-of-work algorithm depends on the *SHA-256* cryptographic hashing function, a member of the *Secure Hash Algorithm 2 (SHA-2)* family, which produces a 256-bit, 32-byte, hash result.

The primary requirements which a *cryptographic hash function* must fulfill are that it be:

- deterministic, i.e., the same input will always produce the same output,
- quick to compute,
- infeasible to determine the input from the output, i.e., a small change in the input will produce a major, seemingly random, change to the output, and
- infeasible to find a collision in resulting hashes.

These properties of cryptographic hash functions provide *collision resistance*, meaning it is computationally infeasible to find an input that produces a randomly selected hash output. This property of collision resistance is leveraged to build the pseudorandom selection process that determines which node is able to build the next block of transactions.

Difficulty The Bitcoin network has a global *difficulty* — a 256-bit, 32-byte, value — that is recalculated every 2016 blocks. The value is recalculated such that the pseudorandom “mining” process to mint a new block will take approximately 10 minutes to complete for each block minted.

While it is helpful to describe miners as being “selected” to build, or mint, new blocks, it is also inaccurate; in reality miners have a chance of building a correct block which can be measured as a probability, i.e., the miner’s hash rate relative to the total network hash rate: $\frac{\text{miner_hashrate}}{\text{network_hashrate}}$.

Miners repeatedly compute SHA-256 hashes, combining the previous block’s header hash, the current block data, and a *nonce* (a pseudorandom shifting value selected by the miner). The miner’s objective is to find a nonce which produces a hash less than the current difficulty. Due to the properties of the SHA-256 hash this process is, practically speaking, a brute-force processes of trial and error, (the “work” in Proof-of-Work). Once a miner has discovered a valid hash they have generated a valid block.

Network Once a miner generates a valid block it will propagate its solution into the network. Other nodes will then verify that the block is correct and will append it to their chain. One of the transactions which miners will include in their block is a coinbase transaction, a transaction containing reward paid to the miner in

the form of newly minted bitcoin.

Security These properties, combined with the incentive of coinbase rewards, provide security in the form of cryptography, electricity, and hardware. Attacks that would threaten this security depend on breaking the cryptographic primitives in action, finding ways of reducing electricity/hardware costs to outperform the rest of the network in the PoW algorithm, attacking nodes in the network, or colluding with other nodes in the network to outperform the remainder of the network.

2.2.1.3 Example Transaction: Alice to Bob

As an example, we might imagine a circumstance where *Alice wants to send Bob 0.100 BTC*. Assume Alice controls two addresses:

1. `1PwDUn9Vxn9CyaRkZfJrTzYRg6QNCygALY`, which contains 6900000 satoshis (0.069 BTC).
2. `1Ah1CWQ1Zxax2yzg3EZBmSmZTKDoHTuUA1`, which contains 7900000 satoshis (0.079 BTC).

Alice does not have enough bitcoin in either address independently to send Bob the 0.100 BTC. To resolve this Alice creates a new transaction which takes as an input both of her addresses. Alice proves she controls both addresses by signing each transaction address and publishing her public key for each respectively. This process allows a node on the bitcoin network to verify both that public key is associated with the address (through a series of cryptographic hashes) and that the person publishing

this transaction controls the private keys (by verifying the signature). Next Alice sets an output address to Bob’s address, an amount to send to Bob (0.100 BTC), and the rules Bob must fulfill to access the bitcoin she is sending to him (typically signing the transaction and publishing the associated public key). Alice could now publish this transaction to the bitcoin network; however, in doing so she would sacrifice 0.048 BTC, the remainder from her two addresses, which the miners would claim as a transaction fee. To avoid this Alice can set a second transaction output address to an address she controls, sending any additional coins there that she does not want to offer as a transaction fee.

2.2.2 Ethereum

Ethereum is a blockchain-based system initially proposed in late 2013, post-Bitcoin, and released in 2015. [9] Ethereum introduced a few novel features and functionalities with its blockchain network which are not available in the Bitcoin network. Most notably Ethereum offers a distributed decentralized Turing-complete computing platform via the Ethereum Virtual Machine (EVM), which aims to provide an application layer which “[runs] exactly as programmed without any possibility of downtime, censorship, fraud or third party interference.” [17,51]

2.2.2.1 Fundamental Concepts

Currency Like Bitcoin, Ethereum’s blockchain defines and leverages its own token, **ether** — also referred to as **eth** or **ETH**, and sometimes denoted using the Greek symbol Xi, Ξ , the uppercase Old English letter Eth, \mathfrak{D} , or, more rarely, \blacklozenge — which acts as the underlying currency of its blockchain protocol. The smallest unit of **ether** is *wei*. [3,51] The denominations of **ether** are broken down as follows:

Table 2.5: Ether Denominations [3]

Value (in wei)	Exponent	Common Name	SI Name
1	10^0	wei	Wei
1,000	10^3	Babbage	Kilowei or femtoether
1,000,000	10^6	Lovelace	Megawei or picoether
1,000,000,000	10^9	Shannon	Gigawei or nanoether
1,000,000,000,000	10^{12}	Szabo	Microether or micro
1,000,000,000,000,000	10^{15}	Finney	Milliether or milli
1,000,000,000,000,000,000	10^{18}	Ether	Ether
1,000,000,000,000,000,000,000	10^{21}	Grand	Kiloether
1,000,000,000,000,000,000,000,000	10^{24}		Megaether

Accounts Accounts are an Ethereum primitive which provide an abstraction over the Bitcoin equivalent signature chain process; this abstraction helps to both simplify the concept of token ownership as well as extend the idea of what a token is, what a token can be, and how blockchain state can be managed and organized.

Accounts are identified by a 160-bit code, their **address**, and are internally represented by four properties:

1. **nonce**, a monotonically increasing counter which represents the number of transactions sent from the account.
2. **balance**, the amount of **ether**, expressed in wei, which is owned by the account.
3. **storageRoot**, a 256-bit hash of the root node of a Merkle Patricia tree which encodes the storage contents of the account.
4. **codeHash**, an immutable hash of the EVM code corresponding to the account.

Although all accounts are structurally identical it is useful to distinguish between the two practical kinds of accounts which one is likely to encounter and interact with on the Ethereum blockchain, *external accounts* and *internal accounts*:

External Accounts — also referred to as simple accounts, non-contract accounts, externally owned accounts (EOA), and sometimes user accounts — are defined as accounts whose **codeHash** value is the Keccak-256 hash of an empty string; i.e., the account contains no code.

Internal Accounts — also referred to as contract accounts — are those accounts which are not external accounts; i.e., the account contains code.

Both kinds of accounts have the ability to send and receive **ether** as well as interact with **contracts** which have been deployed to the Ethereum network. However, there are some key differences between the two kinds of accounts which are worth highlighting: [18]

External Accounts

- External accounts are managed by public-key cryptography.
- External account creation costs no **ether**.
- Only external accounts can *initiate* transactions.
- Transactions between external accounts can only transact **ether**.

Internal Accounts

- Internal accounts are managed by code.
- Internal account creation costs **ether** which reflects the cost of storing code on the Ethereum network.
- Internal accounts can execute code via the EVM upon receiving transactions, enabling a wide range network functionalities.
- Internal accounts can only send transactions in response to receiving transactions.

Transactions A transaction is a cryptographically-signed instruction constructed by an external actor and submitted to the Ethereum network. There are two kinds of transactions worth distinguishing, contract creation transactions and message call transactions; both kinds of transactions share the following common properties:

1. `to`, a 160-bit **address** representing the *recipient's* account. This value is omitted when building a contract creation transaction.
2. `from`, a signature which identifies the *sender* of the transaction by account **address**.²
3. `value`, the amount of **ether**, expressed in wei, which is to be transferred to the recipient.
4. `nonce`, a value equal to the number of transactions which have been sent by the *sender*.
5. `gasPrice`, a value representing the amount of wei to be paid per unit of gas (expanded on below).
6. `gasLimit`, a value representing the maximum amount of gas which should be used executing the transaction.

Contract creation transactions include the following additional property in

²This field does not *technically* exist, in actuality the signature is represented as three distinct fields (v , r , s), which can be used to determine the **address** representing the sender of the transaction.

the transaction:

1. `init`, an EVM-code fragment which is executed only once and discarded thereafter; it returns the `body`, a second fragment of code that executes each time the account receives a message call, which can occur either by transaction or internal execution of code.

In contrast, a message call transaction includes the following additional property:

1. `data`, an unlimited size byte array which contains the input data of the message call.

Ethereum Virtual Machine The Ethereum Virtual Machine (EVM) is the execution environment which Ethereum code is processed in. The EVM processes low-level bytecode and takes actions against the state of the Ethereum blockchain in response: reading, processing, and writing data. The Ethereum Virtual Machine Specification introduces a low-level instruction set which defines the available operations which an EVM implementation should support: the opcodes, their inputs, outputs, and various other implementation details. The EVM can be described as a *state transition function*, $Y(S, T) = S'$; given a set of transactions, T , and an initial state, S , the state transition function, $Y(S, T)$, will produce a new output state, S' . [51] Several EVM implementations exist which have been written in various lan-

guages, e.g., Go, Python, and C++.

Language Support Several higher-level languages exist which target the Ethereum Virtual Machine and can be used to build “smart contracts;” e.g., Solidity which draws inspiration from C++ and JavaScript, and Vyper which describes itself as a “Pythonic Smart Contract Language.” These languages ship with compilers which can be used to translate their code into low-level EVM bytecode; Solidity, for example, is compiled using `solc`, the Solidity compiler.

2.2.2.2 *Network Topology*

Like Bitcoin, Ethereum exists as a network of nodes, each node supporting differing functionalities, which work collectively to construct the Ethereum blockchain and support the surrounding ecosystem. These nodes can be classified by the functionalities which they support.

Node Types There exists several implementations of the Ethereum protocol, i.e., Ethereum clients, which have been written in various languages, e.g., Geth in Go, Parity in Rust, and pyethereum in Python. Some of these clients support running in different modes.

- A *full node* is a complete implementation of the Ethereum protocol. A full

nodes processes and validates all transactions which have been added to the Ethereum blockchain, thus helping to support the resiliency and reliability of the network. To support this functionality, a full node must maintain a complete copy of the blockchain. Full nodes are also capable of deploying and interacting with contracts, support mining and wallet functionality, and are able to route transactions throughout the network.

- A *remote client* supports a subset of the functionality that a full node supports, generally wallet functionality and the ability to broadcast transactions. Other more complex functionalities generally require interacting with with a full node or other remote services which are capable of fulfilling requests on a remote client's behalf.

The Blockchain Like Bitcoin, the Ethereum blockchain is constructed by leveraging a Proof-of-Work (PoW) algorithm to reach consensus throughout the network. The proof-of-work algorithm leveraged by Ethereum, Ethash, helps to build trust and reliability throughout the network while also securing the blockchain and ensuring that EVM code execution has been processed and that the results produced from said execution are as expected. Ethereum offers cryptoeconomic incentivization in the form of **ether** to promote participation in the proof-of-work process. [51]

Blocks The Ethereum protocol groups collections of transactions into *blocks*. Blocks are linked to previous blocks, via cryptographic hash, which reflect the prior states of the blockchain. When processed collectively these blocks reflect the current state of the Ethereum blockchain.

Proof-of-Work Ethereum's proof-of-work algorithm, Ethash, is a proof-of-work algorithm which was initially inspired by the Hashimoto and Dagger algorithms. The primary motivation behind the Ethash algorithm was to produce a PoW algorithm which would be resistant to application-specific integrated circuits (ASICs). The primary mechanism leveraged to achieve ASIC-resistance lies in the algorithm's memory-bound nature: a significant amount of memory, in addition to computation, is required to correctly compute a proof-of-work solution. By requiring large amounts of memory-bound operations the algorithm makes itself resistant to most kinds of specialized memories and caches. Additionally, the memory requirements are designed to grow and shift over time such that building rapid static caches would become prohibitively expensive. In some sense the Ethash algorithm might be better described as Proof-of-Memory. [10,14,8]

The Ethereum network has been attempting to migrate away from Ethash to a lower-cost consensus algorithm operating through Proof-of-Stake (PoS) but has yet to complete the transition.

Gas In order to validate that EVM code has been executed, and executed as expected, each full node on the network must recompute all transactions and whatever EVM code those transactions have triggered when validating blocks. It is not difficult to imagine how this might cause serious problems and introduce room for exploitation within the network, `while true { expensiveOperation() }`. In order to address this the Ethereum specification introduces an abstraction, *gas*, which has a market-based value and must be paid by the transaction-sender up-front when generating a transaction. Each operation computed by the EVM has an associated gas-price and that gas price is “paid” to the node who successfully mints a block. If an insufficient amount of gas is provided by the transaction-sender then the EVM will throw an out-of-gas (OOG) exception: execution halts, the blockchain state is restored, and all gas submitted is forfeited to the node. If an excess of gas is provided then any gas remaining after transaction execution is refunded to the sender. When generating a transaction on the Ethereum network the creator of the transaction has the choice of defining how much wei they are willing to pay per unit of gas. Nodes are incentivized to process transactions which offer more wei per unit of gas relative to the rest of the transactions available for processing on the network. Table 2.6 introduces gas costs as defined by the Ethereum Yellow Paper.

Table 2.6: Fee Schedule [51]

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero} .
G_{base}	2	Amount of gas to pay for operations of the set W_{base} .
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low} .
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid} .
G_{high}	10	Amount of gas to pay for operations of the set W_{high} .
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lfloor \log_{256}(exponent) \rfloor$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the <i>Homestead transition</i> .

Name	Value	Description*
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdata nonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.

$W_{zero} = \{\text{STOP, RETURN}\}$	CALLDATASIZE, CODESIZE, GASPRICE, COINBASE TIMES-
$W_{low} = \{\text{MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND}\}$	TAMP, NUMBER, DIFFICULTY, GASLIMIT, POP, PC, MSIZE,
$W_{mid} = \{\text{ADDMOD, MULMOD, JUMP}\}$	GAS}
$W_{high} = \{\text{JUMPI}\}$	$W_{verylow} = \{\text{ADD, SUB, NOT, LT, GT, SLT, SGT, EQ,}$
$W_{extcode} = \{\text{EXTCODESIZE}\}$	ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD,
$W_{base} = \{\text{ADDRESS, ORIGIN, CALLER, CALLVALUE,}$	MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}

Chapter 3 – Literature Review

This chapter is broken into two major sections:

- *Internet Voting*, which introduces Internet voting: procedures and concepts, and offers a survey of significant Internet voting systems, projects, pilots, and experiments.
- *End-to-End Verifiability*, which introduces the concept of end-to-end verifiable (E2E-V) voting systems, reviews the functionalities required to support end-to-end verifiability (technical and non-functional), the architectural options available, existing E2E-V systems, and common cryptographic techniques available for use in such systems.

3.1 Internet Voting

Internet voting, sometimes referred to as *remote electronic voting*, is a system of voting where voters are able to cast their votes over the Internet.

Procedure The U.S. Vote Foundation describes the typical phases involved in conducting an Internet based election as follows: [16]

1. *Setup*, during which election officials gather voter information, identify election issues and races, design ballots, etc.
2. *Distribution*, during which election materials are distributed to voters: ballots, credentials, voting instructions, etc.
3. *Voting*, during which voters mark their ballots.
4. *Casting*, where voters finalize and submit their ballots and election officials receive said ballots.
5. *Tallying*, where election officials count votes, tabulate results, and announce winners.
6. *Auditing*, where (as necessary) vote results are evaluated for incorrect results.

Requirements The requirements for an Internet-based voting system are the same as those for any other voting system; it must be *secure*, *correct*, and *private*.

3.1.1 Survey of Internet Voting Systems

In September of 2011 the *Election Assistance Commission (EAC)* published “*A Survey of Internet Voting*,” which offered a broad review of various Internet voting systems used between the years of 2000 and 2011. [49] In total, the survey

reviews 30 Internet voting systems used for elections and primaries, by various parties and governments, at various levels of government: national, state, and local.

3.1.1.1 Voting Over the Internet (VOI) — 2000

In 1986, the *Uniformed and Overseas Citizens Absentee Voting Act (UOCAVA)* was passed to render services to merchant marines, uniformed services, and other overseas civilians. Broadly, UOCAVA mandates that overseas and military voters be able to remotely register and vote in federal elections, and designates the Secretary of Defense as the executive agent responsible for implementing its provisions. Thus, under the *Department of Defense (DoD)*, the *Federal Voting Assistance Program (FVAP)* was established to provide voter assistance, tools, and education to overseas voters with the goal of enabling overseas voters to vote from anywhere in the world. In pursuit of this goal, FVAP established procedures for delivering election materials through domestic, military, and foreign postal systems.

In an effort to eliminate some of the weaknesses inherent in postal systems, primarily transit times and unreliable delivery guarantees, FVAP established the *Voting Over the Internet (VOI)* project.

“The pilot project was designed to examine the feasibility of using the Internet for remote registration and voting in an effort to overcome the

time and distance barriers faced by UOCAVA voters. ‘This was the first time that binding votes were cast over the Internet for federal, state, and local offices, including the President and Members of Congress.’” [49,22]

The VOI architecture consisted of 3 major components, as illustrated in Figure 3.1:

- Citizen infrastructure: any workstation or personal computer with Internet access that was available to the voter.
- FVAP infrastructure: VOI systems maintained by FVAP, namely the FVAP server, which hosted the server-side VOI software, various intrusion detection systems, networking infrastructure, and administrative workstations.
- Local Election Official (LEO) infrastructure: systems managed by LEOs, namely servers running VOI software and workstations to interact with said software.

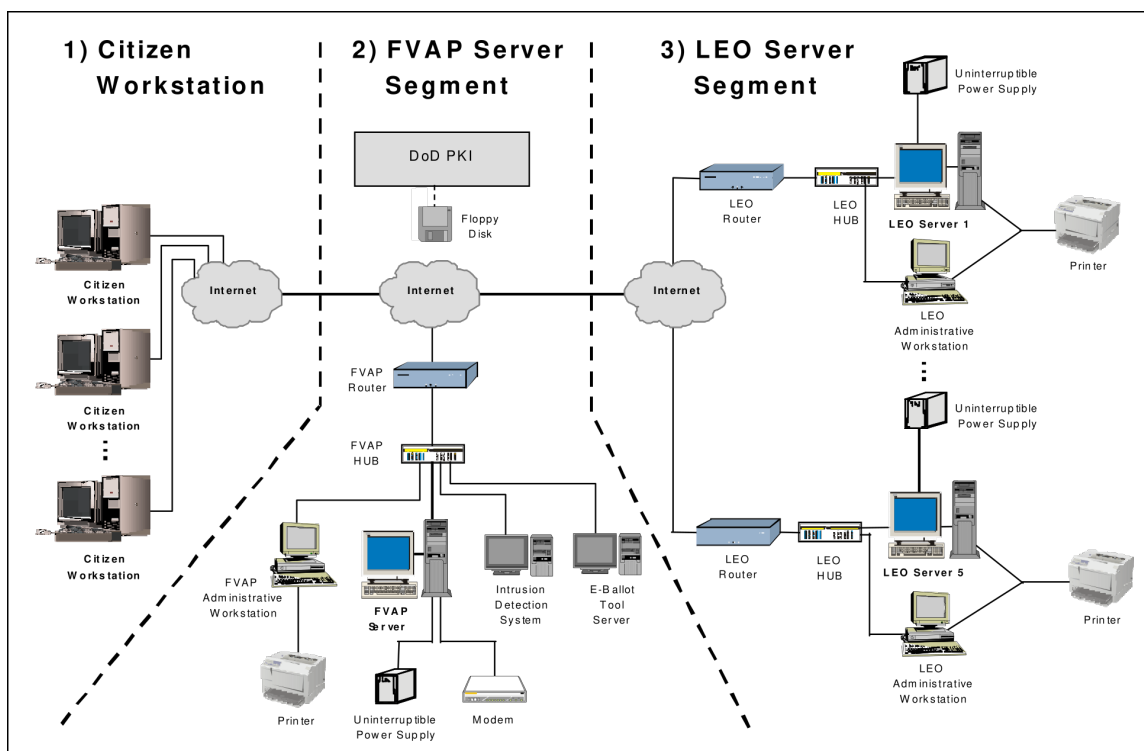


Figure 3.1: VOI System Architecture [22]

Citizen Infrastructure FVAP mailed a CD-ROM to participants containing VOI software, this came in the form of a Netscape Navigator browser plug-in which provided a graphical user interface to cast ballots and communicate with the VOI server-side components.

FVAP Infrastructure The FVAP infrastructure consisted of the FVAP server hosting the VOI software, various networking and power redundancy components to support the server, two network intrusion detection systems, a server hosting

software for creating electronic ballots, and an administrative workstation for interacting with the FVAP server. The collective functionalities which this infrastructure provided included voter authentication, ballot routing to LEO infrastructure, and ballot creation.

Local Election Official Infrastructure Each LEO site managed a server running VOI software which connected over the Internet with the FVAP-maintained VOI server and a workstation which allowed LEOs to perform administrative operations with the server.

The VOI pilot successfully served 84 volunteers across 4 states. Administrators did not detect any intrusions into the system during its operation. However, the DoD acknowledged in their assessment report that one of the major shortcomings of the pilot was its small sample size, and, that the incentive to attack such a system would increase as the number of participants increased. [22] A future security panel criticized the VOI system for taking the position that, “the citizen’s workstation is outside the security perimeter of the system,” noting that it effectively ignores some of the most serious kinds of attacks which the system is vulnerable to. [30] On the topic of remote internet voting the DoD assessment report expressed the following:

“[remote internet voting] is subject to the same security concerns as the current VOI System. For this reason, we cannot recommend this alter-

native as an immediate follow-on development to the VOI Pilot. [22]”

3.1.1.2 *Secure Electronic Registration and Voting Experiment (SERVE) — 2004*

In 2002, following the VOI project, Congress instructed the DoD to carry out a larger demonstration project, “under which absent uniformed services voters are permitted to cast ballots in the regularly scheduled general election for Federal office.” [47] To fulfill this mandate FVAP contracted Accenture to build *The Secure Electronic Registration and Voting Experiment (SERVE)*.

SERVE was built under the United States’ Department of Defense’s (DoD) Federal Voting Assistance Program (FVAP) to be deployed for the 2002 or 2004 elections. Broadly, the motivations behind SERVE were to produce an Internet-based voting system to reduced barriers to voting for Americans living overseas; specifically the objectives of the project were to:

1. “assess whether the use of electronic voting technology could improve the voting participation success rate for UOCAVA citizens,” [23] and
2. “assess the potential impact on state and local election administration of an automated alternative to the conventional by-mail process of absentee registration and voting.” [23]

Fifty counties covering 7 states were targeted for participation and the system

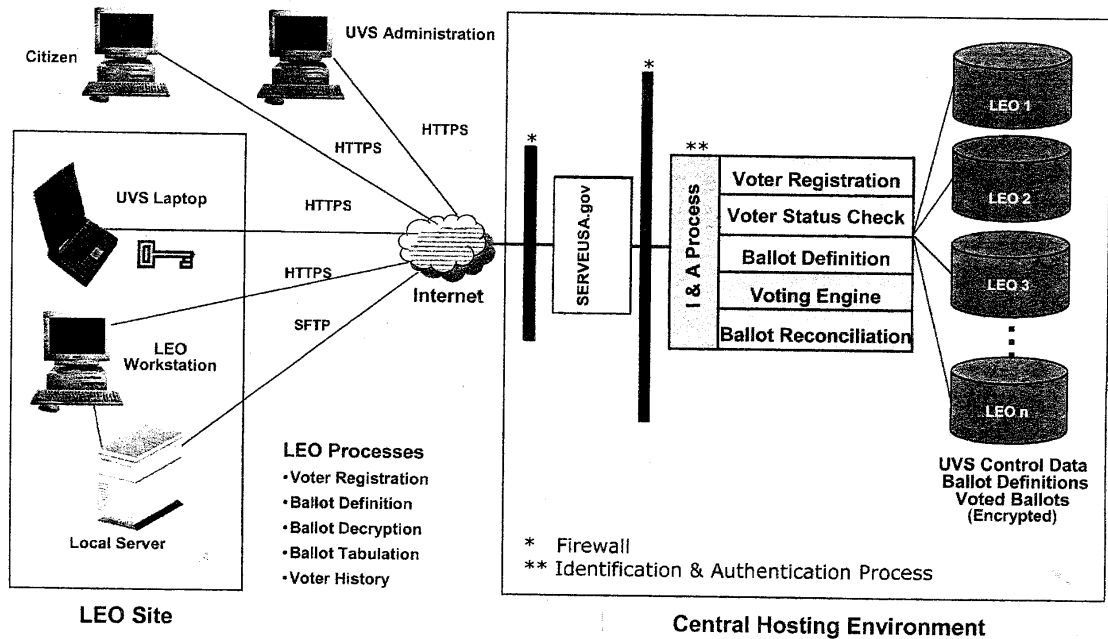
was designed to handle both the registration and voting process.

Architecture SERVE shared many architectural-similarities to VOI which are reflected in the SERVE architecture diagram seen in Figure 3.2:

- SERVE was designed as a web-based service which a voter could connect to via web browser.
- LEOs managed a local server which could be used to interact with the central SERVE system.
- The central SERVE system, which performed the bulk of the system processing, was maintained by FVAP and stored voter information until the appropriate LEO server downloaded it.

The system was described as consisting of eight integrated subsystems: Identification and Authentication; Common Services; Voter Registration; Election Administration; Ballot Definition; Voting; Download and Decryption; and Tabulation.

Architecture Overview



UVS = UOCAVA Voting System

Figure 3.2: SERVE System Architecture [49]

To participate one had to have a military ID (a Common Access Card), or could enroll in the SERVE system by presenting face-to-face proof of citizenship to a SERVE official. Once enrolled and registered, a participant could vote via a web browser through the SERVE site. Figure 3.3 outlines the protocol used for casting a ballot through the SERVE web application.

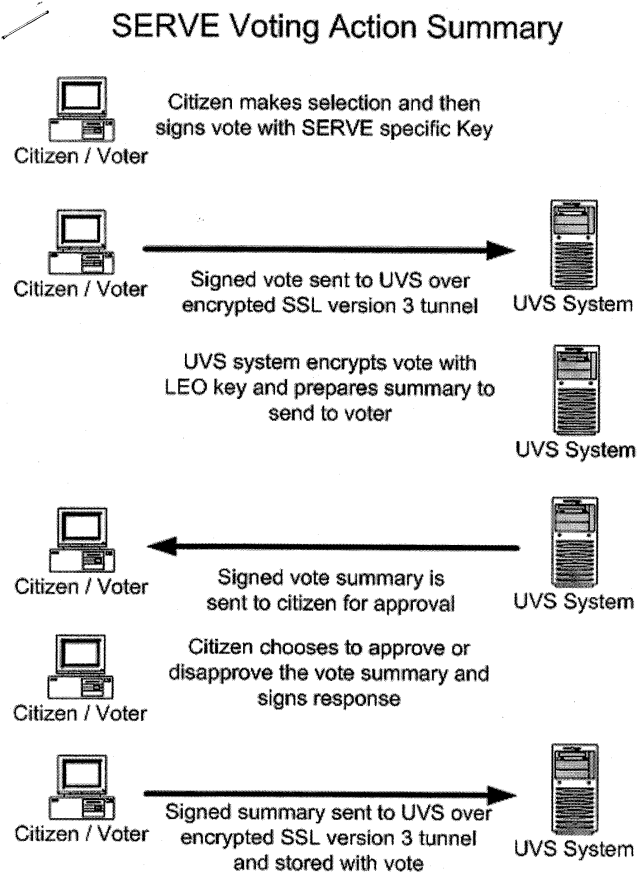


Figure 3.3: SERVE Voting Protocol [49]

SERVE received harsh criticism from independent system reviewers, members of the *Security Peer Review Group (SPRG)*, academics and industry professionals

who were assembled by FVAP to evaluate the system. A group of these members independently publicized concerns regarding the security of the system and Internet-based voting systems more broadly. [30]

The report published notes that SERVE suffers from a number of vulnerabilities and goes into great detail regarding the risks these vulnerabilities pose and the complexity of performing various attacks to take advantage of these vulnerabilities. The report noted: [30]

1. Lack of voter-verified audit trails and vulnerabilities to insider attacks. Vulnerabilities in software are difficult to find and intentionally obfuscated vulnerabilities are even more so. The essentially unauditible nature of electronic voting systems necessitate some form of voter-verified audit trail.
2. Privacy. Several system design issues were identified which would allow LEOs or SERVE administrators to tie a ballot to a voter's identity.
3. Vote Buying/Selling. The nature of Internet voting makes selling credentials for voting systems a very real possibility.
4. Intimidation. Voter intimidation is a problem which all remote voting systems must contend with, this problem extends to Internet voting systems.
5. Large-Scale Impact. Electronic voting machines, if compromised, might enable attackers to modify or damage tens or hundreds of thousands of ballots.

Internet voting systems face this same issue, except on a much larger scale; the entire system essentially acts as a single electronic voting machine, significantly increasing the scale of impact if compromised. Paper-based systems do not face these same exposures.

6. Too Many Potential Attacks. Electronic systems present a large attack surface, exposure to the Internet presents even more. Mitigation of all of the kinds of attacks possible would not be feasibility.
7. Many Sources of Attacks. Elections held over the Internet are vulnerable to attacks from around the globe; nation-state entities, terrorists, individual hackers and more would all have the ability to attack system.
8. Undetectable Attacks. Electronic systems make detecting attacks extremely difficult and the lack of a detected attack on a system does not prove that no attack occurred.
9. On-screen Electioneering. Many states prevent campaigning within some distance of a polling place; however, no such laws exist to prevent ISPs, web browsers, or other entities from displaying ads to voters.

In addition the report had this to say about future attempts at building Internet voting systems:

“Like the proponents of SERVE, we believe that there should be better

support for voting for our military overseas. Still, we regret that we are forced to conclude that the best course is not to field the SERVE system at all. Because the danger of successful, large-scale attacks is so great, we reluctantly recommend shutting down the development of SERVE immediately and not attempting anything like it in the future until both the Internet and the world's home computer infrastructure have been fundamentally redesigned, or some other unforeseen security breakthroughs appear.” [30]

In response to these criticisms and concerns documented in this report, the then-Deputy Secretary of Defense Paul Wolfowitz decided that the SERVE project would not go forward as planned for the 2004 election, effectively killing the project. [23] Three years later the DoD published a report which downplayed the criticisms and concerns published by the SPRG members. [23] In reaction to this DoD report, the members of the SPRG independently publicized a response which criticized the report for downplaying the concerns laid out in their initial security analysis of SERVE, and further reiterated their concerns regarding Internet voting. The members of the SPRG noted again that the issues faced by SERVE are ones which are not capable of being fixed by a better design or architecture of Internet voting systems, because the fundamental issues are ones which could only be fixed by redesigning both the

Internet and personal computers. [29]

3.1.1.3 *D.C. Digital Vote-by-Mail System (DVBM) — 2010*

In 2009 the *Military and Overseas Voter Empowerment Act (MOVE)* was passed. This act amended UOCAVA and other statutes to provide further protections to eligible citizens. Specifically the act aimed to reduce the number of ballots which are not counted due to late receipt. MOVE accomplishes this by requiring that states send absentee ballots no later than 45 days prior to election day. MOVE goes further by requiring that all registration material and blank ballots be available electronically and removes requirements regarding notarization on voting applications and ballots. [48]

In search of a solution to improve their compliance with the MOVE act, Washington's District of Columbia Board of Elections and Ethics (DCBOEE/BOEE) planned to launch an Internet voting system, the D.C. Digital Vote-by-Mail (DVBM) system, for use in the November 2010 general election. The project was developed in partnership with the Open Source Digital Voting (OSDV) Foundation's Trust-TheVote project, who viewed the project as a mostly academic effort. [34] The system was slated to be operational in time for the November 2010 general election and aimed to provide two primary functionalities: [49,50]

1. allow voters to electronically access voting materials
2. allow voters to optionally cast their ballot over the internet

The DVBM architecture, illustrated in Figure 3.4, was developed as a web application using the Ruby on Rails framework; was hosted using the Apache web server; used MySQL as its database technology, which stored the global election state (voters' names, addresses, etc.); and used the underlying (Linux) filesystem to store encrypted ballots cast by voters. When the voting phase of the election was complete, election officials would transfer the encrypted ballots to an air-gapped computer for decryption and printing. Printed ballots would be counted alongside other mail-in absentee ballots. [50]

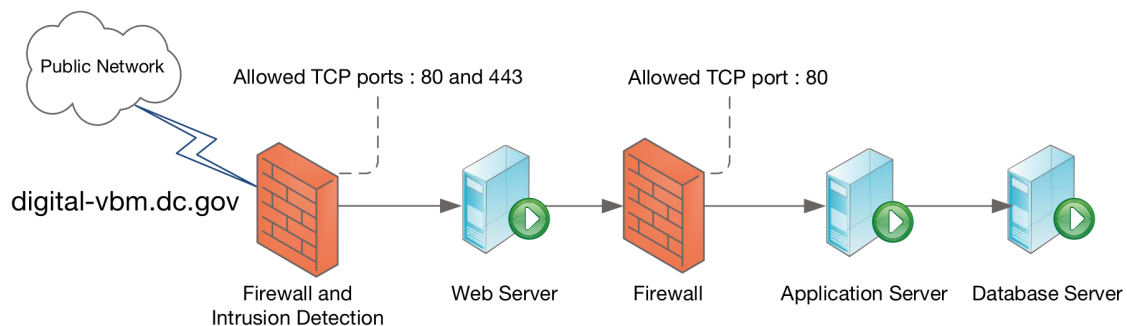


Figure 3.4: DVBM System Architecture [50]

Prior to the official launch of the system the BOEE opted to conduct a mock election where members of the public, researchers, and hackers alike were invited to to test the functionality of the system, discover vulnerabilities, and attempt to

compromise the security, reliability, and availability of the system. [49,50,34]

Within 48 hours of the system going live researchers from The University of Michigan, playing the role of an attacker, demonstrated a number of vulnerabilities and attacks on the system and managed to gain near-complete control of the election server. Their intrusion was not detected for nearly 2 days. [50]

Demonstrated vulnerabilities included being able to:

- penetrate the network of the election software
- determine voter's identities
- locate unencrypted ballots, thus mapping voter's identities to their personal votes
- modify ballots
- cast fake ballots
- modify the election system software itself

Once election officials became aware of the attack, the mock election was suspended, five days ahead of schedule, citing "usability issues." Election officials later confirmed that they were unable to detect the attack or network presence using their intrusion detection system. Due to the test results, the portion of the system which allowed for ballots to be submitted over the Internet was not used. [50]

3.1.1.4 Estonia — 2005

Estonia began using Internet voting in 2005 [49]; in the 2015 Estonian parliamentary elections, 30.5% of all voters voted over the Internet. Estonia maintains what is likely the most advanced national identification cards in the world. Estonian IDs are part of a *Public Key Infrastructure (PKI)* where IDs serve as smart cards which possess two RSA key pairs: one for signing and one for authentication; these cryptographic functions are performed directly on the card. The signatures produced by these IDs are used extensively throughout the country and are considered legally binding. These cryptographic IDs allow Estonia to provide voter authentication capabilities that cannot be reproduced in the US. Despite the advanced authentication capabilities that Estonia is able to achieve, researchers in 2014 devised a number of attacks that could be performed on the Estonian voting system to spoil ballots, damage ballot secrecy, and steal or drop votes. The researchers also criticized the transparency and operational security of the system, noting that videos of the administration processes — provided for transparency purposes — recorded administrators entering root passwords, revealed network credentials which had been posted on a wall, and showed administrators using USB drives containing personal files to move sensitive election materials between systems. [45]

3.2 End-to-End Verifiability

The vulnerabilities and weaknesses of electronic voting systems demonstrate a clear need for a stronger kind of system design. *End-to-End Verifiable (E2E-V)* voting systems are systems which aim to provide such a design by providing the following features for voters:

1. allows voters to check that the system recorded their votes correctly
2. allows voters to check that the system included their votes in the final tally
3. allows voters to count the recorded votes and double-check the announced outcome of the election.

An *End-to-End Verifiable Internet Voting (E2E-VIV)* is an E2E-V voting system that supports voting over the Internet.

3.2.1 Requirements

In 2015, the U.S. Vote Foundation published a specification and feasibility assessment study which laid out requirements for building an end-to-end verifiable internet voting system. The study broke these requirements into two categories: [16]

- *Technical Requirements*, the set of requirements which can be directly addressed by system design and architecture
- *Non-Functional Requirements*, the set of requirements which must be imposed

on a system by external entities

3.2.1.1 *Technical Requirements*

The ten categories of technical requirements, those which should be addressed by the design and architecture of the systems itself, are: [16]

1. functional requirements
2. accessibility requirements
3. usability requirements
4. security requirements
5. authentication requirements
6. auditing requirements
7. system operational requirements
8. reliability requirements
9. interoperability requirements
10. certification requirements

The *functional requirements* deal primarily with casting and recording of ballots. *Receipt freedom* is one such functional requirement. An electoral system which expresses receipt freedom is said to make it impossible for a voter to prove to anyone how they voted. Others functional requirements include:

- ensuring that a voter cast a ballot if such an act is recorded
- data retention in case of failure
- multi-vote functionality to overwrite previous votes
- maintaining voter anonymity

Usability is mostly concerned with user experience and confirmation guarantees. For example, voters should be confident that their vote was cast by being provided a confirmation screen. The voting process should be both intuitive and guide the voter through the process. Presentations such as the butterfly ballot should be avoided at all costs.

Digital voting systems have the potential to provide wider *accessibility* guarantees than traditional paper ballots for voters with disabilities. To provide these guarantees developers must involve voters throughout the development process to identify accessibility issues and implement solutions.

Security is an integral property and requirement which voting systems must maintain. Included in this requirement is that:

- no data can be permanently lost
- integrity of voters, candidates, ballot information, cast ballots, and other critical information must be maintained
- accurate timing information is critical for auditing

- voting equipment must be protected
- the system must perform regular health checks

Authentication is the process of ascertaining the validity of a claimed identity. Authentication ensures that the voting system can enforce privacy and prevent multi-voting, Sybil attacks, and vote theft. All individuals must be identified uniquely. The system must allow access to services only to authorized users, e.g., only allow election officials to load ballot info.

The property of *auditability* means that a voting system is capable of comprehensive examination. Auditability must exist at all stages and levels of the voting process. The system must keep auditable logs of all relevant activity and the logs must be public and write only. Furthermore, the logs cannot leak any data regarding voters or the way any ballot was cast. Privacy must always be the foremost concern.

The auditing system must actively report issues and information in real-time. At least the following events should be recorded: [16]

- “all voting-related information, including the number of eligible voters and votes cast, the number of invalid votes, count and recount results, etc.”
- “any detected attacks on the operation of the system or its communication infrastructure”
- “any system failures, malfunctions, or other detected threats to proper system

operation.”

The system should provide auditing features which support the ability to: [16]

- “cross-check and verify the correct operation of the voting system and the accuracy of the election results”
- “detect voter fraud”
- “prove that all counted votes are legitimate and that all ballots have been counted”

Finally, auditability must extend to the source code, actions performed, and the documentation itself.

System operational requirements are those that enforce and regulate transparency, accountability, system configuration, and updates. Logs, software, configurations, versions, updates, etc., must all be managed and produced to audit for tampering. Protocols should be in place to guard sensitive equipment at all times and handle system failures. Officials managing these systems and the procedures themselves must be scrutinized closely to prevent insider attacks and election fraud.

Reliability is the property of a system behaving reasonably and as expected under both normal conditions and while under attack. During an election period a system should be highly available. 99.9% availability is a minimum for voting systems. The system must also be able to recover from any failure within 10 minutes,

with the exception for failure caused by natural disaster or malicious attack. The system should have redundant backup systems for critical components of the system.

Internet voting systems are compelling targets for Distributed Denial of Service (DDoS) attacks, therefore it is important that an E2E-VIV system be hardened to such attacks and be able to continue operation with full correctness during a sustained DDoS attack.

An E2E-VIV system must use open standards for *interoperability* between components, services, and other E2E-VIV systems. Logs and documentation of such standards must be published so that anyone can download, inspect, and publish analysis and concerns.

Finally, there should be *certification* and test procedures involved for every functional requirement; these tests should be able to be run on demand. Formal proofs of security and correctness should be provided wherever possible, and third-parties should be hired to conduct an independent review, audit, and test of the system.

3.2.1.2 *Non-functional Requirements*

The five non-functional requirements defined, those which must be fulfilled by external entities, e.g., operators and administrators, are: [16]

1. operational requirements
2. procedural requirements
3. legal requirements
4. assurance requirements
5. maintenance/evolvability requirements

The specification describes several *operational* requirements: election and registration timing, maintaining voter registration and candidate nomination lists, providing receipt freedom, voter assistance, election integrity, and openness.

Voters must be well-informed on how to register, vote, and protect their privacy in the voting system. Clear instruction on when voting and registration occurs should be announced far in advance for the voter's benefit. When multiple forms of remote voting take place, votes cast over the Internet should not be accepted after other forms of remote voting end. E2E-VIV systems must publish a voter register that is regularly updated. Voters should be able to check that information in the register is accurate and request corrections. The ballot presented to voters must be consistent, fair, unbiased, and free from any superfluous information about candidates/choices.

Operational receipt freedom represents two different requirements depending on whether a voter is voting from a supervised or unsupervised location. In a super-

vised location receipt freedom requires that the voting terminal clear all indication of how a ballot was cast and ensure that no paper trail representing how the ballot was cast is able to leave the polling place (except by official means). In an unsupervised location any visual proof of vote should not be able to be used to determine how a vote was cast or will be tallied.

If test ballots are capable of being submitted then those ballots must be clearly marked as a test ballot with instruction on how to cast a real ballot. The voting system should not disclose any results to any person until after the voting period has ended, including alternative forms of voting. Tallying should be done as soon as possible afterwards and the tallying process should be transparent, recorded, and be able to be replayed. Any irregularities which affect the integrity of votes should be recorded.

An E2E-VIV system must be open and function properly regardless of the hardware or software being used to run the voting software. The system must be available for auditing by external actors, especially when considering components which are expected to be run on external systems or voter's machines.

Procedural requirements define the processes required to deploy and run the E2E-VIV system.

- Procedures should be published regarding provisioning, certification, mainte-

nance, availability, and use. For example, when updates occur, election officials must call upon an independent body to perform verifications of performance and certification of intent.

- Procedures should be in place to teach voters the voting process.
- Election officials should have maintenance and security procedures to ensure that voting equipment is operating nominally and has not been tampered with. For example, conducting sensitive operations should require teams of at least two people.
- As much as possible there should be procedures in place to allow observers to watch election procedures.
- Procedures should be in place to update results in the event that a voter proves that their vote was not accurately received or counted.

The *legal* requirements include national, state, and local laws that apply to voting systems, e.g., accessibility, anonymity, and availability guarantees. Any deployed E2E-VIV system must comply with these laws. For example, election officials must ensure that only one ballot by each voter is tallied when multiple means of voting exist, e.g., remote and traditional polling place.

“There must be no impediments to interested parties who want to study the E2E-VIV system. In particular, no nondisclosure agreement or con-

tract of any kind may be required for download and study of, or for building, testing and publishing test results for, the E2E-VIV system.” [16]

To meet *assurance* requirements, client-side software must be functional and free of bugs across a wide range of hardware and software stack combinations. There must be strong security with respect to authentication such that voter credentials cannot be forged or invalidated without breaking underlying cryptographic protocols.

The entirety of the voting system — e.g., software, documentation, design, architecture, algorithms, build scripts, issue tracking system, etc. — must be free, open, and public. All available resources should be up to date, certified, and released under license that permits anyone to download, build, test, or modify the source.

To meet *maintenance and evolvability requirements* election officials must have the right and ability to update the election system to conform to law, technology, or threat independent of the original vendor.

3.2.2 Architecture

The study, “The Future of Voting: End-to-End Verifiable Internet Voting,” provides an architectural feature model, seen in Figure 3.5, which defines over 127,000 possible architectural variants. [16]

```

1  -- This diagram shows the various dimensions of an E2EVIV architecture
2  static_diagram E2EVIV_Architecture_Dimensions
3  component
4    class E2EVIV_ARCHITECTURE
5    feature
6      authority_distribution: SET[VALUE]
7        ensure 0 < Result.count;
8        for_all v: VALUE such_that v member_of Result
9          it_holds v member_of { Centralized, Distributed };
10     end
11    crypto_protocols: SET[VALUE]
12      ensure 0 < Result.count;
13      for_all v: VALUE such_that v member_of Result
14        it_holds v member_of { On_Paper, Mechanized, Verified, Generated };
15    end
16    correctness_evidence: SET[VALUE]
17      ensure 0 < Result.count;
18      for_all v: VALUE such_that v member_of Result
19        it_holds v member_of { Process_Based, Assertions };
20    end
21    implementation_type: SET[VALUE]
22      ensure 0 < Result.count;
23      for_all v: VALUE such_that v member_of Result
24        it_holds v member_of { Golden_Implementation, Open_Protocols_and_Specs };
25    end
26    key_distribution_method: SET[VALUE]
27      ensure 0 < Result.count;
28      for_all v: VALUE such_that v member_of Result
29        it_holds v member_of { Public_Ceremony, Threshold_Cryptography, PKI, Web_of_Trust };
30    end
31    deployment_style: SET[VALUE]
32      ensure 0 < Result.count;
33      for_all v: VALUE such_that v member_of Result
34        it_holds v member_of { Trusted_Servers, Public_Cloud, Peer_to_Peer };
35    end
36    client_technology: SET[VALUE]
37      ensure 0 < Result.count;
38      for_all v: VALUE such_that v member_of Result
39        it_holds v member_of { Custom_App, Web_Based };
40    end
41  end
42 end

```

Figure 3.5: A specification of the possible variants for an E2E-VIV system. [16]

Most of the variability available when constructing an E2E-VIV system stems from the cryptographic techniques and tools available to select from when designing

the system.

3.2.2.1 Cryptographic Techniques and Tools

The cryptographic techniques and cryptosystems available for use in E2E-V electoral systems are presented in this section; this is not a comprehensive list of cryptographic techniques and tools available, but includes some of the most common techniques and tools leveraged in E2E-VIV systems.

Asymmetric cryptography, also known as public-key cryptography, uses pairs of keys to securely encrypt and decrypt messages. There are many different public-key based cryptosystems available for use.

Homomorphic cryptographic schemes allow one to perform basic arithmetic operations on ciphertexts without requiring decryption of the ciphertext. This property has a number of uses in E2E-VIV systems; for example, an E2E-VIV system might leverage this property to tally a collection of encrypted ballots without decrypting any individual's ballots.

Additive homomorphic encryption schemes enable processing of ciphertexts by way of addition. The Pallier and Benaloh cryptosystems both support additive homomorphic encryption. [1]

Multiplicative homomorphic encryption schemes enable processing cipher-

texts by way of multiplication. The ElGamal cryptosystem supports multiplicative homomorphic encryption.

Most E2E-VIV systems depend on an append-only web/public bulletin board. [27] The append-only public bulletin board is a publicly-visible secure location where election operations and ballot data are submitted, logged, and made available to support auditing requirements. [16,1,7,40,42,33] Blockchains fulfill most of the requirements of an append-only public bulletin board.

Secret sharing and threshold schemes allow a collection of actors to cooperate to produce “shares” of a secret; each participant is responsible for managing their share of the secret and some threshold of shares must come together to recover the complete secret or perform cryptographic operations. [32,11]

A zero-knowledge proof (ZKP) is a probabilistic method which allows one party to prove knowledge of some secret without revealing any information about the secret itself. A zero-knowledge proof satisfies the following properties: [16,25]

1. Completeness, an honest verifier will be convinced by an honest prover.
2. Soundness, an honest verifier will not be convinced by a dishonest prover.
3. Zero-knowledge, the verifier will not learn any information regarding the secret itself.

A useful form of zero-knowledge proofs are non-interactive zero-knowledge proofs

— also known as NIZKs, zk-SNARKs, or zkSTARKs — which are zero-knowledge proofs which require no interaction between the prover and verifier for the verifier to be convinced of correctness. [1,7,33,42]

Mixnet schemes are used to provide anonymity. Mix networks are operated by a set of trusted nodes, mix-servers, which consume messages — typically encrypted ballot data in the case of E2E-VIV systems — from a set of network-participants to produce a random permutation of the input messages. Each node performs a “mix” operation on the incoming messages in such a way that the output cannot be unscrambled and tied back to a network-participant except by the node itself which is performing the mixing operation. Therefore, as long as any single mix-server in the mix network is acting honestly, the anonymity of the participants will be maintained. [42,33]

A decryption mixnet operates by encrypting a message in multiple layers with each mix server’s public key. To decrypt the message, the message layers are decrypted in the opposite order they were encrypted in by each node. Each node forwards its decrypted results to the next node in the mix network. So long as a single node does not reveal the source of the message then the message will become untraceable (assuming no information is leaked by the message itself). [42]

A re-encryption mixnet works by leveraging the re-encryption properties of

the underlying encryption scheme. Certain cryptosystems make it possible to change a ciphertext without modifying the underlying message. In this way a set of nodes can shuffle and re-encrypt a ciphertext then pass them to the next mixnet node to repeat the process. So long as a single node does not reveal the shuffling process the anonymity offered by the mixnet will be maintained. [42,7]

Blind signature schemes separate the voter authentication, authorization, and signing components from the vote tallying, shuffling, and decryption components. A voter will encrypt a ballot (blind it) then send it to a signing authority who will blind-sign the encrypted ballot after it has verified that the voter is qualified to vote. Once a voter has acquired a blind signed ballot they can strip their identifying data, unblind the ballot, and submit the signed ballot through an anonymous channel. The underlying cryptosystem makes it such that the blind signed ballot is equivalent to a signed unblinded ballot. [16]

Chapter 4 – Methods

This work designs, builds, and analyzes various electoral systems, electoral features, and voter authentication models. Implementations are provided as smart contracts which are used to determine their efficacy, feasibility, and usability as tools to aid in on-chain decision-making processes.

It is undeniable that the underlying infrastructure and properties thereof, which are intrinsic to blockchain technologies, present risks and weaknesses with regard to voting; however, these properties also present unique opportunities to explore non-traditional and novel approaches to governance models, mechanisms, and electoral systems. This research aims to lean into the advantages of the intrinsic properties made available by blockchain technologies, while attempting to minimize the risks presented and weaknesses exposed which are derived alongside them.

4.1 Requirements

Borrowing from the requirements introduced and defined by, “The Future of Voting: End-to-End Verifiable Internet Voting” [16] — which were outlined in Chapter 3, *Literature Review*, and grouped into two major categories: technical and non-functional — a set of requirements are identified and targeted for satisfaction. These requirements provide a framework to analyze the results produced by this research. Table 4.1 summarizes the requirements targeted for fulfillment.

Table 4.1: Requirements targeted for fulfillment.

Requirements	Targeted			Untargeted
	Fully	Mostly	Partially	
<i>Technical</i>				
Authentication	•			
System Operational	•			
Reliability		•		
Security		•		
Auditing		•		
Functional		•		
Interoperability			•	
Certification			•	
Accessibility				•
Usability				•

Table 4.1: Requirements targeted for fulfillment.

Requirements	Targeted			Untargeted
	Fully	Mostly	Partially	
<i>Non-Functional</i>				
Maintenance/Evolvability			•	
Assurance			•	
Operational				•
Procedural				•
Legal				•

The requirements identified in “The Future of Voting: End-to-End Verifiable Internet Voting” have rightfully demanding and difficult to meet criteria for fulfillment; however, the expectations laid out for a system of the kind imagined in the study are far beyond the scope of this research. Therefore, a subset of the technical and non-functional requirements are identified which are considered achievable, relevant, and within the scope of this research. The requirements are individually assessed and categorized based on whether their criteria for satisfaction are considered feasible to fully, mostly, or partially satisfy. The remaining requirements are considered either outside of the scope of this research or not feasible to otherwise fulfill and are therefore not targeted. Several requirements are identified which are considered partially or mostly satisfied by virtue of intrinsic properties made available through

the underlying blockchain technology.

4.1.1 Technical Requirements

The technical requirements are requirements which the study asserts should be fulfilled through the design and architecture of the system. This research attempts to fully satisfy the authentication and system operational requirements; considers the accessibility and usability requirements beyond the scope of this research, and therefore makes no effort to satisfy them; and targets partial satisfaction of the remaining requirements. Requirements which are targeted to be partially satisfied are reviewed in further detail.

Reliability and Security Requirements Reliability and security requirements are considered *mostly* satisfiable due to the inherent properties made available by Ethereum and its decentralized structure; however, there are some notable potential attacks on the network which are worth considering; these are discussed in further detail in Chapter 5, *Results*.

Auditability Requirements Ethereum inherently offers a tremendously detailed log through its blockchain structure which provides a wide range of opportunities for detailed auditing and validation. Furthermore, auditing and validation functionality

are inherent features of the blockchain. However, the auditability which Ethereum provides is not considered a privacy preserving feature and therefore falls short of the criteria presented.

Functional Requirements A subset of the functional requirements are deemed relevant to this research:

- ✓ Multi-vote functionality is targeted to be investigated and fully satisfied in all electoral system implementations.
- ✚ Ensuring that “a voter’s ballot, and the act of them casting a ballot, is recorded and retained as expected;” is considered *mostly* fulfilled through intrinsic properties of the Ethereum blockchain. Where this property falls short is when a voter casts a ballot, broadcasts a transaction to the network, which miners fail to include in any blocks. This is discussed further in Chapter 5, *Results*.
- ✚ Maintaining voter anonymity is considered *partially* fulfilled through intrinsic properties of the Ethereum blockchain. While it is *technically* possible to broadcast privacy-maintaining transactions, those which preserve anonymity on the Ethereum network, it is likely beyond the capabilities of most voting actors and would be difficult to guarantee in any meaningful way. Pseudo-anonymity is a better profile of the kind of anonymity which Ethereum offers, This is discussed further in Chapter 5, *Results*.

4.1.2 Non-Functional Requirements

The non-functional requirements are those which the study asserts must be fulfilled by entities external to the system itself. These requirements are regarded as being mostly beyond the scope and context of this research, however, the existence of this document does itself at least *partially* satisfy some of the categories of requirements, e.g., assurance, maintenance, and evolvability. Detailed documentation regarding design, architecture and implementation is provided in Appendix A, *Software Documentation*.

4.2 Assumptions and Objectives

Given the nascent nature of blockchain technologies, it is difficult to imagine the demands and needs of the organizations and communities which will come to exist in these cyberspace environments; and therefore difficult to predict which governance models and electoral systems would best support them. However, it seems likely that the needs of these organizations will vary as dramatically by organization as the organizations which have come to exist in meatspace environments. It therefore follows that a wide range of kinds of electoral systems should exist which would serve to support the various governance models and decision-making processes that are likely to form over time. Given these assumptions, and in compliance with the

aforementioned requirements, *the objectives of this research are defined as follows:*

1. This research aims to explore various electoral systems and features, using the content reviewed in Section 2.1 as a basis for exploration.
 - (a) Provide a range of kinds of electoral contracts using Table 2.1 and Table 2.2 as the primary resource for electoral system and feature selection.
 - i. Support single-winner elections and provide plurality and majority electoral system in fulfillment of that objective.
 - A. Implement first-past-the-post to fulfill the targeted single-winner plurality electoral system, due mostly to the fact that it is extremely simple, well-understood, and widely-adopted.
 - B. Implement range vote to fulfill the targeted single-winner majority electoral system, due to its effectiveness and simplicity.
 - ii. Support multi-winner elections and provide an electoral system which offers proportional representation; Single transferable vote is selected as the electoral system implementation to fulfill both objectives due to its wide-adoption, prevalence, and effectiveness as a PR system.
 - (b) Explore the design, implementation, and efficacy of various ballot types across the various electoral systems selected.
2. This research aims to explore voter authentication, registration, and access control

patterns for managing voter registration.

3. This research aims to explore delegative or liquid governance models.

4.3 Tooling

As mentioned in Section 2.2.2.1, Ethereum offers a code execution environment via the Ethereum Virtual Machine. The EVM is responsible for executing a set of low-level instructions which can be used to update the state of the network. [51] A number of higher-level languages exist that are capable of being compiled into the low-level bytecode representation required by the EVM for execution; this research uses the Solidity language. In order for the EVM to execute bytecode, it must first exist on the Ethereum blockchain. Deployment of bytecode to the Ethereum blockchain is performed by constructing, and submitting to the Ethereum network, an account creation transaction which includes the EVM bytecode. The account creation transaction instructs the EVM to create a new account and to store the included bytecode within it. Upon receiving the account creation transaction the EVM will execute a special `CREATE` instruction (see Table 2.6) which will eventually result in the creation of an “internal account.” This internal account will contain the EVM bytecode, submitted with the create account transaction, and is commonly referred to as a “smart contract.” Once a smart contract exists on Ethereum blockchain, the

bytecode stored within it can be executed by constructing and submitting another transaction to the network; this time instructing the EVM to execute the bytecode stored within the smart contract. Beyond the bytecode stored in them when created, smart contracts also contain and maintain their own internal state on the network, which they can use as storage.

4.3.1 Frameworks

Given the complex process involved in compiling and deploying smart contracts to the Ethereum network, a framework is typically leveraged to simplify development and manage the smart contract deployment life-cycle. This research leveraged two different software frameworks to aid in the development of smart contracts: Embark and Truffle. Both of these frameworks aid in the compilation of Solidity code, deployment of EVM bytecode, provisioning of test networks, and testing of contract functionality.

4.3.1.1 Embark

Initially, smart contract implementations were built using the Embark framework. The Embark framework is designed to aid in the construction and deployment of Decentralized Applications (DApps); it supports functionality to provision test

networks, build and deploy contract code, and execute tests against deployed smart contracts. The Embark framework is capable of managing, and deploying contracts to, various other Ethereum blockchains; e.g., the testnet, private net, and livenet. The Embark framework supports several useful development features. For example, Embark can be configured to detect changes to contract code during development, then trigger a recompilation and redeployment of modified contract code to the appropriate Ethereum networks.

To build Solidity code into EVM bytecode, the Embark framework leverages `solcjs`, which offers JavaScript bindings for the Solidity compiler. For expedited development and testing, Embark exposes the Ethereum JavaScript `testrpc`. The `testrpc` simulates a fully functioning Ethereum client, features an API for programmatic interaction (a useful feature during test execution), and is capable of transaction execution speeds that complete near-instantaneously (speeds which could not be matched by a live Ethereum network). Embark simplifies interaction with Ethereum contracts by generating contract-equivalent (promise-based) JavaScript functions that can be called to execute contract code. JavaScript support is provided by wrapping the `web3js` library which implements the generic Ethereum JSON RPC spec.

The Embark framework also boasts support for other tools useful for build-

ing decentralized applications; decentralized storage support via IPFS, decentralized messaging functionality via Whisper and Orbit, and a curses-like dashboard which exposes logs, environment configuration, contract state, service availability, the status of application components and dependencies, and features an interactive console for executing commands and querying application state. A screenshot of the Embark dashboard is shown in Figure 4.1.

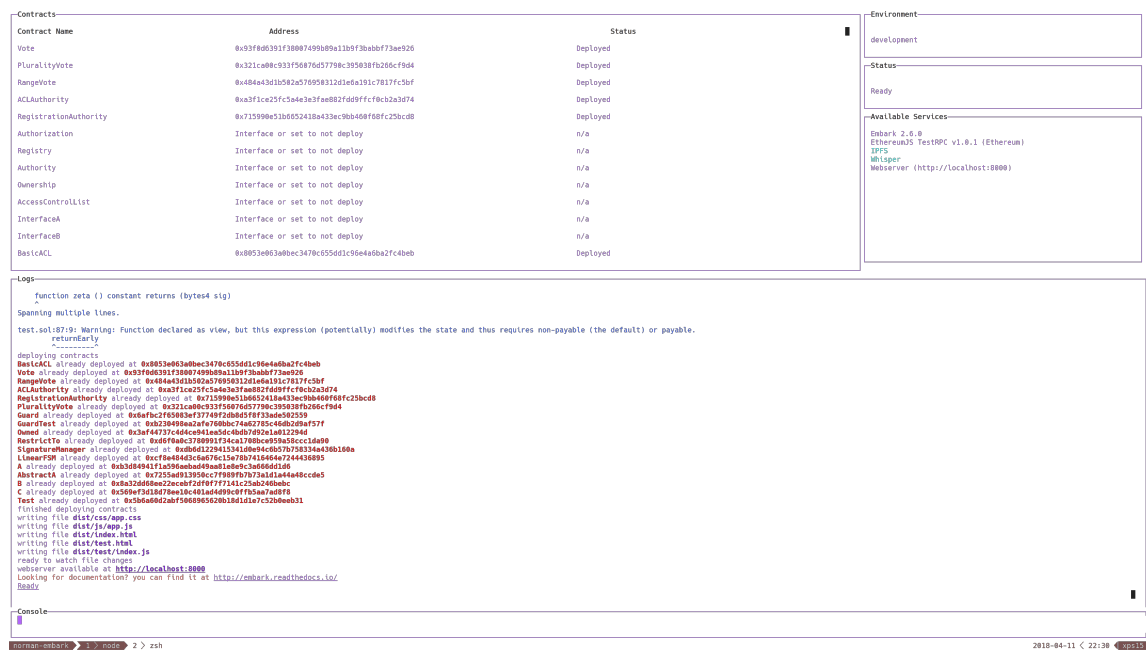


Figure 4.1: The Embark dashboard and application state.

4.3.1.2 Truffle

Later smart contract development was completed using the Truffle framework. When compared to the contract development features of the Embark Framework,

the Truffle framework can be viewed as an almost feature-identical framework — i.e., smart contract lifecycle management, testing functionality, JavaScript bindings, and network management — however, unlike the Embark framework, the Truffle framework offers essentially no features related to DApp development. The Truffle framework’s more targeted focus with regards to smart contract development and management produced a cleaner development experience during the pursuit of this research, which is concerned strictly with smart contract design and implementation and not DApp development. Furthermore, the Truffle framework supports executing workflows and tests in TypeScript, and can be used in conjunction with `typeChain` to generate TypeScript types for use with the smart contract bindings generated by Truffle. A screenshot displaying the execution of a contract test via the Truffle framework can be seen in Figure 4.2.

```

~/projects/election-contracts
> yarn test --stacktrace-extra ./test/unicast-fptp.ts
yarn run v1.22.10
$ yarn truffle test --stacktrace-extra ./test/unicast-fptp.ts
$ /home/nate/projects/election-contracts/node_modules/.bin/truffle test --stacktrace-extra ./test/unicast-fptp.ts
Using network 'test'.

Compiling your contracts...
=====
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/multicast-range-vote.sol
> Compiling ./contracts/unicast-fptp.sol
> Compiling ./contracts/unicast-range-vote.sol
> Compiling ./contracts/vote.sol
> Compilation warnings encountered:

Warning: Visibility for constructor is ignored. If you want the contract to be non-deployable, making it "abstract" is sufficient.
--> /home/nate/projects/election-contracts/contracts/Migrations.sol:12:3:
12 |   constructor() public {
    |     ^ (Relevant source part starts here and spans across multiple lines).

> Artifacts written to /tmp/test--2345455-vFKwau2DzLWD
> Compiled successfully using:
   - solc: 0.8.3+commit.8d00100c.Emscripten.clang

Elections
First-Past-the-Post
Contract: UniCastFtp
Initialization
  ✓ Create new contract (178ms)
  ✓ Contract choices match configuration
Vote
  ✓ First voter casts ballot (43ms)
  ✓ Second voter casts ballot (45ms)
  ✓ Second voter cannot cast second ballot (482ms)
  ✓ Third voter casts ballot (63ms)
  ✓ Contract records 3 voters
Tally
  ✓ Tally ballots (44ms)
  ✓ First choice tally results correct (44ms)
  ✓ Second choice tally results correct
  ✓ Third choice tally results correct
Results
  ✓ Compute results (122ms)
  ✓ First choice is winner
  ✓ Other choices are not winners

14 passing (1s)

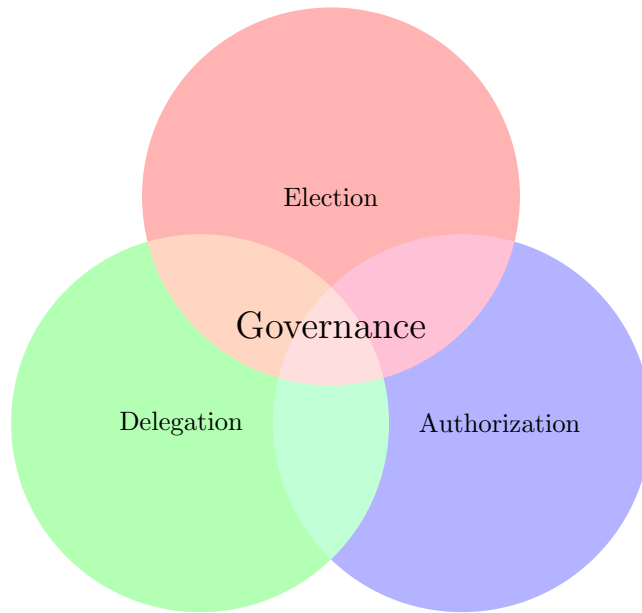
Done in 12.58s.

```

Figure 4.2: Testing an election contract using the Truffle framework.

4.4 Architecture and Design

The design of the system is approached in three phases: authorization, election, and delegation. Each phase considers different components of the overall system architecture and each component is responsible for managing a different set of responsibilities within the architecture.



4.4.1 Authorization Components

The authorization components are designed to provide access control: to build and maintain a set of eligible voters, administrators, and whatever other roles might be necessary to implement the electoral systems. Ultimately, the authorization components are responsible for restricting access to sensitive contract function calls that mutate the state of the contract, e.g., casting ballots and configuration elections. The design must be flexible enough to facilitate the varying and evolving needs of different organizations and communities but consistent enough to support them all through a common interface. Concepts are borrowed from traditional operating system access control schemes/models and authorization mechanisms which provide

guidance for system design. The underlying authentication features are handled via the asymmetric cryptography provided by the underlying blockchain infrastructure.

4.4.1.1 Access Control

Access control schemes exist to authorize access to data and resources; they are responsible for managing and defining the relationships between permissions, operations, objects, and subjects. Several access control model schemes were considered in this phase; among them were: access control lists, discretionary access control, mandatory access control, and role-based access control.

Discretionary Access Control Discretionary Access Control (DAC) is a form of access control where the owner of some resource/object can dictate the operations and permissions that other subjects can take on the resource/object. Additionally, the owner of the resource/object can pass ownership to some other subject. You can see a form of this in POSIX file systems where ownership over files is granted and transferred through commands like `chown` and `chmod`.

Access Control Lists An Access Control List (ACL) is a collection of *subject*, *resource*, and *permission* relationships which can be understood as a matrix, where each cell, indexed by *subject* and *resource*, reflects the *permissions* available for the

subject to access *resource*.

Role-Based Access Control Role-Based Access Control (RBAC) is a form of access control where collections of permissions are assigned to roles; roles are then assigned to users. In RBAC roles are hierarchical, thus roles can be inherited from parent roles.

4.4.1.2 *Design*

The fundamental resources exposed by smart contracts are function calls, thus the security implementation and access control model must revolve around that. We define an interface, `Authority`, which defines the `canCall` function. Any contract wishing to implement access control on functions can leverage a contract that implements the `Authority` interface to authorize access to those functions.

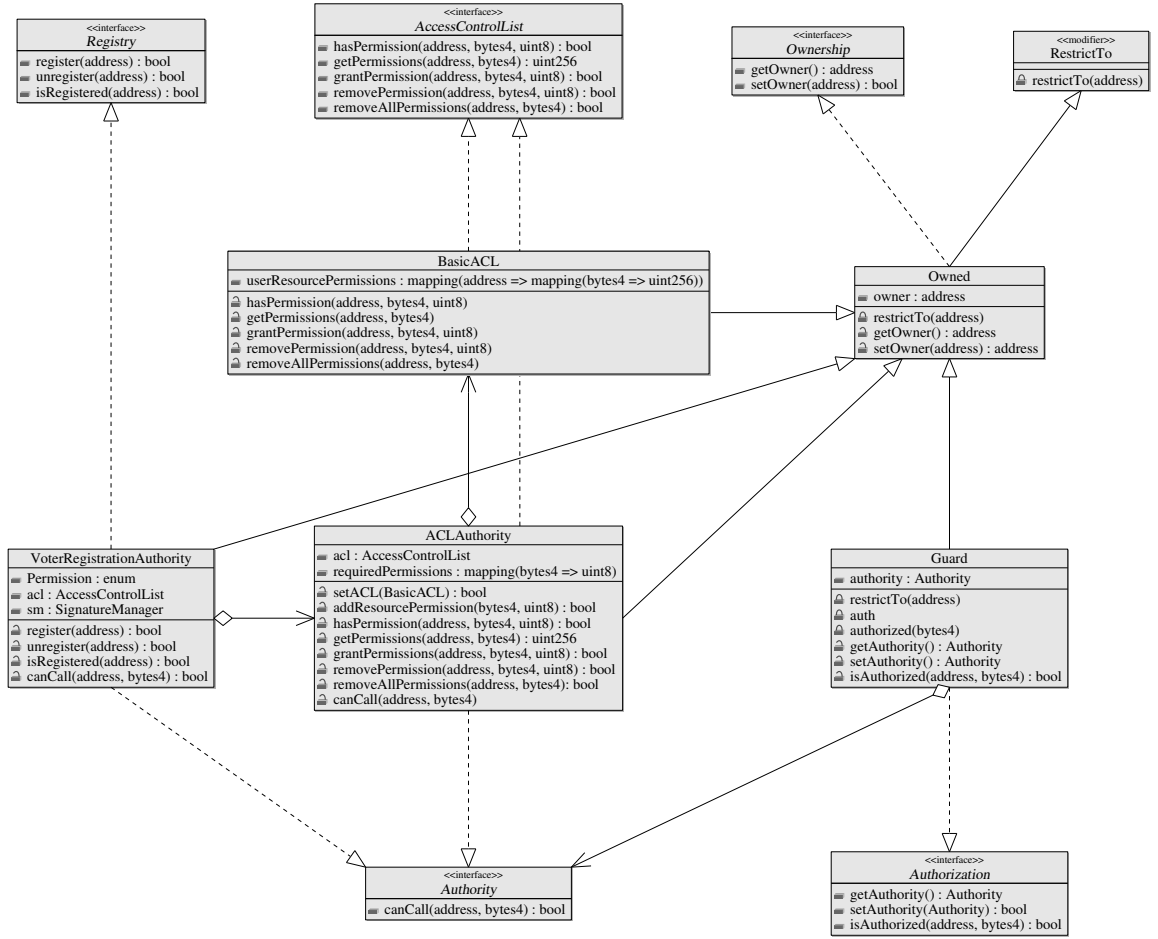


Figure 4.3: Authorization dependency graph modeling.

We define the `Authorization` interface which offers mechanisms for interacting with an `Authority`. This interface defines `getAuthority`, `setAuthority`, and most importantly, `isAuthorized`. A contract that realizes the `Authorization` interface is expected to aggregate an `Authority` but does not necessarily need one externally; for example, a contract that provides `Authorization` may also be its own `Authority`. Likewise,

an `Authority` can provide its own `Authorization`. The `Authority` and `Authorization` interfaces together form our access control primitives.

The `Guard` contract realizes the `Authorization` interface and provides the modifier functions `auth` and `authorized`. These can be applied to other functions to easily restrict function-call access to accounts that the `Authority` has approved, e.g.,

</> Guard Usage Example

Listing 4.1

```
1 function sensitive () public auth returns (bool _success) {
2   // The `auth` modifier prevents this function from being
3   // called until the Authority has confirmed that the
4   // the message sender has the proper privileges.
5   return true;
6 }
```

This implementation conforms to a number of design principles: separation of concerns, dependency inversion principle, open/closed principle, interface segregation, substitution principle, and single responsibility principle. Ultimately it provides the flexibility and resiliency necessary to build out a wide variety of electoral systems.

Ownership Access control often starts before our `Authority` and `Authorization` realizations. Most contracts offer a primitive form of access control through `Ownership`. All contracts and most contract functions are open and public by design; thus, it is important to lock down sensitive contract functions from deployment. A common pattern is to store the `address` of the creator of a contract as an owner in the contract state and to use that address to restrict access to sensitive public-facing

functions. More complicated ownership mechanisms can be built using this pattern by leveraging contracts which delegate ownership to another contract which can then define more complex access control and management mechanisms. To facilitate in this and many other useful patterns we introduce a simple `Ownership` interface. The `Ownership` interface defines functions `getOwner` and `setOwner` which get and set the account address of a contract's owners respectively.

To facilitate in restricting access of functions to a single account we also define a contract `RestrictTo` that provides a modifier function `restrictTo`. The `restrictTo` modifier can be applied to any other function and takes an account address argument; if the address of the account calling a `restrictTo` modified function does not match the argument provided to `restrictTo` (often the owner), then the function will immediately exit and revert any changes to the contract. Finally, we realize and extend these two contracts to form the concrete `Owned` contract. This pattern is so ubiquitous that most, if not all, concrete contracts in our governance ecosystem will extend this contract.

Authority We need to provide access control beyond interfaces, standards, and single account address restrictions. Our access control implementation leans towards access control lists as a primitive that can be extended to provide other access control mechanisms such as role-based access control. Alternatively, a contract can realize

the `Authority` interface itself if ACLs are not a useful abstraction.

Access Control List We first define an ACL interface, `AccessControllist`, which defines basic ACL functionality: `hasPermission`, `getPermissions`, `grantPermission`, `removePermission`, and `removeAllPermissions`. The design supports assigning up to 256 unique permissions per contract signature.

Basic ACL Next we realize a concrete implementation of the `AccessControllist` interface named `BasicACL`. `BasicACL` is primarily a database contract which creates a mapping from account address to a mapping of functions (the first 4 bytes of their signature) to an unsigned 256-bit integer that represents the permissions. This implementation is similar to what can be found in POSIX compliant operating systems. In Java one might represent this access control model as `HashMap<Account, HashMap<Function,Permission>>`.

Naturally `BasicACL` provides implementations for the functions defined by `AccessControllist`: `hasPermission`, `getPermissions`, `grantPermission`, `removePermission`, and `removeAllPermissions`. These work using bitwise NOT, OR, and AND operations to modify the 256-bit unsigned integer representing permissions for a particular (account, function) pair.

ACL Authority We can easily build an `ACLAuthority` out of our `BasicACL` using the adapter pattern and aggregation. We simply aggregate an instance of a `BasicACL` and realize the `Authority` and `AccessControllist` interfaces. Finally, we lean on the `hasPermission` function to implement our `canCall` function which presumably forwards our request to a `BasicACL` instance and returns the result. This structure also allows us to implement a composite pattern and aggregate many trusted authorities into one.

Voter Registration Authority Our final step is to use a facade to hide the more complex implementation details from outside contracts. We introduce a `VoterRegistrationAuthority` which realizes the `Registry` and `Authority` interfaces. The `Registry` interface provides some nice abstractions: `register`, `unregister`, and `isRegistered`. We presume the contract aggregates some number of `AccessControllist`s as a foundation for its `Registry` function implementations, but does not need to necessarily. Finally, the `canCall` function leans on the `Registry` functions, specifically `isRegistered` to grant access to functions.

The end result is a clean, public, and composable voter registration authority.

4.4.2 Election Components

The election components are responsible for constructing and operating election processes. These responsibilities include maintaining the integrity and security of the election, handling votes, tallying ballots, and determining election winners.

The phases typically involved in conducting an Internet-based election, described in Section 3.1, are: Setup, Distribution, Voting, Casting, Tallying, and Auditing. Responsibilities pertaining to voter registration, which are typically handled in the Setup phase, are expected to be managed by Authorization components. The Distribution phase, which mostly pertains to processes such as mailing election materials to voters, is considered outside the scope of this research and therefore ignored. The Auditing phase, where election integrity and results are scrutinized, is assumed to be handled and provided by the underlying features made available by Ethereum.

4.4.2.1 Design

Each electoral system implementation is unique, due mostly to the unique requirements necessitated by their underlying tallying algorithms and ballot structures.

There are important electoral criteria worth considering when analyzing the implementation feasibility of these electoral systems, namely, the ballot counting

criteria: summability, polynomial time, and resolvable. These electoral criteria are significant because they impact the gas costs associated with operating the electoral system. Table 2.2 is a useful resource when considering these electoral criteria. Both the first-past-the-post and range vote electoral systems are a 1st-order summable; i.e., ballots can be tallied using a linear amount of space with respect to the number of choices available without losing information required to complete the tallying processes. Both also have polytime ballot counting implementations that are linear in the number of candidates and voters. STV is not listed in the table; however, its single-winner equivalent (IRV) is; which implies that STV has, at best, a tallying complexity of $O(n^2)$ and a summability complexity of $O(n!)$ with respect to the number candidates.

Gas costs must be considered when implementing, deploying, and running contract functions on the Ethereum Virtual Machine (EVM). If gas costs are too high then it may become infeasible for an external account to afford to execute a contract function, e.g., the network fees become too high to vote. Another concern to consider is that the cost to execute a contract function may become too high to ever complete in a single block, therefore never actually complete. The most expensive operations, by no small margin, are operations involving storage. Therefore it is important to minimize storage usage.

Finite State Machine Several of the voting system designs use a phased approach with timed transitions between election states: Configuration, Frozen, Vote, and Tally. These phases can be modeled as a finite state machine where calls to various functions trigger transitions to different stages. A modifier function can be used to restrict access of functions to only be callable during their appropriate state. Most of the transitions from state to state can be designed as timed transitions which occur lazily, i.e., the check to confirm that it is time to transition from one state to another occurs when a function is called. The rationale for this approach is due to the fact that Ethereum offers no mechanism to trigger code execution *from within the blockchain itself*; therefore all function calls, even if called from another function or even a function in a different contract, must ultimately have originated from a transaction created by an externally owned account. The product of this timed transition implementation implies that the state restriction modifier needs to occur after the lazily evaluated timed transition modifier; i.e., first update the current state if it should be, then validate if the function should be executed. On the other hand, if transitions are manually updated during a function's execution then the opposite behavior should occur; i.e., first validate if the function should be run, then update the state as required.

Configuration The Configuration phase, as implied by the name, exists to provide election administrators an opportunity to configure the election contract: choices, election start/end time, etc. After the configuration is complete the administrators can freeze the contract, preventing it from being modified further.

Frozen Transitioning to the Frozen phase could simply set a boolean `frozen` to true and update the phase to Frozen. The boolean `frozen` should be checked at the start of every administrative function's execution and should cause the immediate cease of code execution if the value is true. This serves to prevent potentially malicious election administrators from modifying an election contract after the voting process has started; it also serves to notify external contracts and users that the election is configured and is ready or waiting to move into the Vote phase.

Vote The Vote phase is the phase during which various kinds of voting can occur for a particular election contest. A `vote` function should be defined. For flexibility, a `vote (uint8[], uint8[])` function signature is considered which is capable of being leveraged by most kinds of electoral systems and ballot structures. For example, as a sparse vector, where both arrays are of equal length; values in the first array act as a choice index and values from the second array act as a choice value. This could be used to marginally reduce the size of transactions for many

kinds electoral systems.

Tally The Tally phase is the final phase of the election process that takes place after the end of the Voting phase. The exact tallying process will depend on the electoral system itself.

First-Past-the-Post The process for a first-past-the-post election is as follows:

1. Configuration

- (a) Election administrators add each contest choice:

```
addChoice(bytes32 _choice).
```

- (b) Election administrators set the voting start time:

```
setVoteStartTime(uint _voteStartTime).
```

- (c) Election administrators set the voting end time:

```
setVoteEndTime(uint _voteStartTime).
```

- (d) Election administrators freeze the electoral contract, preventing further contract configuration mutations: `freeze()`.

2. Frozen

- (a) All functions are disabled until start time is met.

3. Vote

- (a) Authorized voters cast votes by calling the `vote(uint8 _choice)` function,

where the value provided is the choice the voter supports. Votes are immediately summed into the struct of the appropriate choice. This is feasible because the first-past-the-post electoral system is summable in linear time and with linear space consumption.

4. Tally

- (a) When the voting end time is reached the election moves into the Tally phase. At this point only the tally function can be called. The tally function, in this case, will check the number of votes for each candidate and set the winner to the choice that has received the most support. The election administrator is expected to run this contract function but any account can execute this function without any consequence to the result of the contract.

Range Voting The process for a range vote is very similar to the process for first-past-the-post. It is as follows:

1. Configuration

- (a) Election administrators add each contest choice:

```
addChoice(bytes32 _choice).
```

- (b) Election administrators set the voting start time:

```
setVoteStartTime(uint _voteStartTime).
```

- (c) Election administrators set the voting end time:

```
setVoteEndTime(uint _voteStartTime).
```

- (d) Election administrators set the max score (≤ 100) a choice can receive:

```
setMaxScore(uint8 _maxScore).
```

- (e) Election administrators freeze the electoral contract, preventing further contract configuration mutations: `freeze()`.

2. Frozen

- (a) All functions are disabled until start time is met.

3. Vote

- (a) Authorized voters cast votes by calling the `vote(uint8[] _choices, uint8[] _scores)` function. The parameters act as a sparse vector where the value provided in `_choice` indicates the choice being scored and the corresponding value provided in `_scores` indicates the score of the choice. Scores are immediately summed into the struct of the appropriate choice and a `voteCount` member is incremented. This is feasible because the range vote electoral system is summable in linear time and with linear space consumption.

4. Tally

- (a) When the voting end time is reached the election moves into the Tally phase. At this point only the tally function can be called. The tally func-

tion for this electoral system will, for each choice, multiply the summed score by 10^p where p is `precision` then divide the result by `voteCount` to find an average score for the choice. The winner is set to the choice with the highest average score. The score is multiplied by 10^p because the EVM does not have floating point functionality and a greater precision makes ties less likely. The election administrator is expected to run this contract function but any account can execute this function without any consequence to the result of the contract.

4.4.3 Delegation Components

The delegation components offer mechanisms for the electorate to vest votes to delegates who may vote on their behalf. A delegation hierarchy lends itself to a graph representation. [24] Specifically a directed acyclic graph (DAG) forest-like structure where pendant and isolated vertices represent voters and all other vertices represent delegates. A sink vertex represents a delegate who has not delegated their vote further. Directed edges represent delegations. The total weight of a voter, their voting power, can be measured by recursively calculating the total number of incoming edges for each vertex. We represent edges in the graph as follows:

```
1 struct Voter {  
2     uint40 weight; // 5 bytes  
3     address delegate; // 32 bytes  
4 }
```

Each time a delegation occurs a depth first traversal should be performed to ensure that no cycles are created by the delegation; doing so maintains the acyclic invariant required of the graph to support voter delegation. A simple graph coloring algorithm is considered, beginning with white vertices and coloring vertices black as they are traversed; traversal starts from the vertex representing the voter who is delegating their vote and ends at some sink vertex. If no cycle is detected the algorithm should check to see if the voter has already delegated their vote; if so, it should traverse down their current delegation path, decreasing the weight of each vertex visited by the weight of the voter or delegate delegating their vote. Finally, the algorithm should record the new delegate address and perform a final traversal, following the same path originally traversed while performing cycle detection, increasing the weight of each vertex visited along the way by the weight of the voter or delegate delegating their vote. More briefly:

1. Confirm that there are no cycles within the new delegation chain.
2. Decrease the weight of the delegates in the old delegation chain.
3. Increase the weights of the delegates in the new delegation chain.

A complete implementation of this algorithm is as follows:¹

</> Vote Delegation

Listing 4.3

```
1 mapping (address ⇒ Voter) voters;
2
3 function delegateVote (address delegate) public auth returns (bool _success) {
4     // The `auth` modifier prevents this function from being
5     // called until the Authority has confirmed that the
6     // the message sender has the proper privileges.
7     Voter cursor;
8     uint40 weight = voters[msg.sender].weight;
9
10    mapping (address ⇒ bool) visited;
11    visited[msg.sender] = true;
12    visited[delegate] = true;
13
14    // Cycle Detection
15    cursor = voters[delegate];
16    while (cursor.delegate) {
17        address newDelegate = cursor.delegate;
18        if (visited[newDelegate]) return false;
19        cursor = voters[newDelegate];
20        visited[newDelegate] = true;
21    }
22
23    // Decrement weights of old delegate chain.
24    cursor = voters[msg.sender];
25    while (cursor.delegate) {
26        address newDelegate = cursor.delegate;
27        cursor = voters[newDelegate];
28        cursor.weight -= weight;
29    }
30
31    // Increment weights of new delegate chain.
32    cursor = voters[msg.sender];
33    cursor.delegate = delegate;
34    while (cursor.delegate) {
```

¹Here we assume that every voter has their voting weight initialized to 1. This algorithm might be improved by combining the cycle detection and weight accumulation steps while leveraging the `revert()` function to revert the state of the contract if a cycle is detected. We opt not to do that for now because return values are not currently available with the `revert()` functionality.


```
35     address newDelegate = cursor.delegate;
36     cursor = voters[newDelegate];
37     cursor.weight += weight;
38 }
39
40 return true;
41 }
```

Chapter 5 – Results

This chapter outlines the results of this research. This chapter is divided into two major sections:

1. *Test Results*, which reviews significant benchmarks, test results, and operational costs discovered in the design and implementation of this research.
2. *Analysis*, which uses the requirements and criteria, targeted in Section 4.1, and objectives targeted in Section 4.2, to measure the shortcoming and results of this research and also reviews insights and unexpected challenges encountered during the implementation phase.

5.1 Test Results

Testing Ethereum contracts can be approached in several ways, but the choices one can make with regards to testing mostly align themselves into one of two categories: the driver which is executing the contract being tested and the network environment the contract is being tested within.

1. The test driver generally comes in one of two shapes:
 - (a) Contract-to-contract unit tests, which use tests written as smart contracts to drive contract execution. These tests provide confidence that inter-contract communication will function as expected.
 - (b) Client-to-contract unit tests, which use transactions generated by external accounts to drive contract execution. These tests provide confidence that external accounts can run contract code as expected.
2. The network environment comes in several shapes, listed from least realistic to most:
 - (a) The `testrpc`, now deprecated, is a Node.js-based Ethereum client which simulates a full client and network behavior. It is free to execute and fast, but lacks the complexities of a real node.
 - (b) Ganache, which replaces `testrpc` and is functionally identical from a testing perspective.
 - (c) One can run a local instance of Ethereum. This is free and accurately represents an Ethereum node, but lacks real-world network complexities.
 - (d) There are several “testnets,” which are test networks designed for actual contract deployments. Testnets provide a close-to-reality network environment while still providing a free means of acquiring funds for testing.

- (e) The most accurate environment to test in is the actual Ethereum network; this is not free, but offers a perfect real-world environment to test contracts in.

The tests built in this research leverage the `testrpc` and Ganache as development and testing environments, generally launched as either `ganache-cli --accounts 100 --networkId 7357` OR `testrpc --gasLimit 0xffffffff --port 8546`. Tests were driven through JavaScript and TypeScript contract bindings using the Embark and Truffle frameworks respectively.

5.1.1 Election Components

Several election contracts were implemented with various modifications to their implementations based on their underlying electoral system and features being targeted. Each election contract accepts a configuration object containing the choices available for the election along with various other properties based on the requirements of the underlying electoral system. Additional details regarding implementation are available in Appendix A.

5.1.1.1 Election Simulation

To generate estimates regarding the cost of execution and storage required to support features of electoral systems, several sets of elections were simulated using various configurations of contract implementation, choice availability, ballot structure, and voter selection. The general process required producing minimal implementations of each electoral system and modifying them as necessary to explore the advantages and disadvantages of various design choices. Each electoral system implementation then had minimal tests written to validate that basic functionality was present, then simulations implemented. Each simulation leveraged hundreds of accounts which were generated for our test network using Ganache. Each account, to the extent supported by the underlying electoral system, made random voting choices: selecting and scoring candidates through pseudorandom number generation.

Table 5.1: Simulated gas consumption by contract function.

Contract	Operation			Gas Consumption	
	Function	Call	Voters	Total	Average
UniCastFptp	vote	1	100	8,007,700	80,077
UniCastRangeVote	vote	1	100	17,417,564	174,175
MultiCastFptp	vote	1	100	7,447,632	74,476
MultiCastFptp	vote	2	100	3,208,496	32,084
MultiCastRangeVote	vote	1	100	23,266,645	232,666

Table 5.1: Simulated gas consumption by contract function.

Contract	Operation			Gas Consumption	
	Function	Call	Voters	Total	Average
MultiCastRangeVote	vote	2	100	9,525,039	95,250
UniCastFptp	tally	1	1	0	0
UniCastRangeVote	tally	1	1	149,356	149,356
MultiCastFptp	tally	1	1	917,790	917,790
MultiCastRangeVote	tally	1	1	5,342,859	5,342,859
UniCastFptp	computeResults	1	1	360,516	360,516
UniCastRangeVote	computeResults	1	1	47,416	47,416
MultiCastFptp	computeResults	1	1	47,438	47,438
MultiCastRangeVote	computeResults	1	1	47,394	47,394

Table 5.1 documents the result of contract simulations, mapping gas consumption by election contract function. These values were generated by measuring the total gas consumption observed over several simulation runs and computing the average cost per contract function. Simulations were configured using 100 voters in each run, with 10 available choices to select from in each election. A random choice selection and evaluation process was used to generate ballots used to vote. The tally and results of each election were validated independently alongside the contract’s results.

Ballot Storage and Tallying Algorithms Section 2.1 introduced concepts regarding ballot structure and tallying algorithm. Section 2.1.2 further-refined these ideas while introducing the ballot counting criteria: polytime, resolvable, and summability. To measure the significance of these criteria and the impact of various ballot formats on our overall design, several implementations of FPTP and range vote were implemented and simulated.

1. `UniCastFtp` is an implementation of first-past-the-post which was designed to permit a single ballot to be cast per voter. This implementation takes advantage of the summability criterion to avoid storing each individual voter's ballot and instead maintains a continuous tally of the ballots. The space required is therefore constant with respect to the number of voters and linear with respect to the number of candidates whose `vote_counts` are being tallied.
2. `UniCastRangeVote`, like `UniCastFtp`, is an implementation of range vote designed to only allow a single ballot to be cast per voter. This implementation again takes advantage of the summability criterion to avoid storing each individual voter's ballot and instead maintains a continuous tally of the ballots. The range vote implementation needs to store slightly more information to support its tallying algorithm, but shares the same storage complexity as the `UniCastFtp` implementation.

The `UniCastFtp` implementation demonstrated a *very* consistent gas consumption across its function calls through the exercised simulations. The `UniCastRangeVote` implementation maintained a slightly less consistent gas consumption across function calls, but stayed respectably stable.

Multi-Vote Functionality and Tallying Algorithms The functional requirements laid out in Section 4.1 and Table 4.1 demand multi-vote support. Two contracts were written to demonstrate this functionality.

1. `MultiCastFtp`, an implementation of first-past-the-post designed to permit a voter to cast multiple ballots.
2. `MultiUniCastRangeVote`, like `MultiCastFtp`, is an implementation of range vote designed to allow multiple ballots to be cast per voter.

It is clear that a voter’s ability to cast multiple ballots in an election could only be considered fair if the principle of “one person, one vote” were adhered to. That is to say, any ballot which has been submitted and had some partial tally applied based on its marks *must* have its computations undone before a newer ballot’s marks can be considered for tallying. The ability to cast multiple ballots therefore carries with it an implied requirement of vote reversal. In order to reverse a previous ballot’s impact on a tally, knowledge of the ballot’s marks is required. Therefore, the most recent ballot of each voter must be maintained in the contract’s state to support

vote-reversal in the incident that they cast multiple ballots.

This draws into question the viability and utility of leveraging the summability criterion as a storage-saving feature, as was leveraged in the previous contracts, and introduces questions with regard to the *eager evaluation vs lazy evaluation* of tallying algorithms. The unicast implementation of each electoral system took advantage of eager evaluation, which essentially forced each voter to pay a nominal amount of extra gas to keep the current tally of each choice maintained, this is reflected in Table 5.1 by the low gas cost associated with the unicast `tally` functions (0 in the case of `contract UniCastFtp`!). The more traditional approach to tabulating results in an election is to defer the tallying procedure until the end of the voting period and have an election administrator incur the cost and responsibility of tallying ballots. This was the approach taken in the multicast implementations of these two contracts; however, it is worth noting that a middle-ground exists which would increase the cost to vote by a nominal amount and result in a less-expensive tally cost.

One interesting outcome regarding ballot storage and storage costs is reflected in Table 5.1; there is a dramatic reduction in gas costs incurred when casting a second ballot. The high initial gas cost reflects the high charge incurred for consuming storage space on the blockchain. The lower cost reflected by the second call is a product of the new ballot replacing the old ballot in storage, therefore consuming no

additional space on the blockchain.

Precision Decision and Casting Fake Ballots Each electoral system has advantages and disadvantages. It is the responsibility of those selecting and designing such systems to attempt to balance those advantages and disadvantages in a reasonable way. Two examples of this encountered during this research can be seen in the range vote implementations. The range vote implementation requires the following configuration to be provided during contract construction:

```
</> RangeVoteConfiguration Listing 5.1  
1 struct RangeVoteConfiguration {  
2     string[] choices;  
3     RangeVoteConfigurationFakeBallots fake_ballots;  
4     uint8 min_range;  
5     uint8 max_range;  
6     uint8 tally_precision;  
7 }  
8  
9 struct RangeVoteConfigurationFakeBallots {  
10     uint8 score;  
11     uint40 voter_count;  
12 }
```

The `tally_precision` value is worth addressing. The tallying algorithm used to determine the rank of a choice in a range vote election is computed by taking the summation of all of the scores cast for the choice and dividing that summation by the number of voters who ranked the choice. This produces a score average which is used to determine the winner of an election. One weakness of the EVM is that floating

point operations are not supported; in order to address that a tally precision is used, which multiplies the `score_sum` by $10^{\text{tally_precision}}$ before dividing it to maintain some degree of precision. What an appropriate value to assign to `tally_precision` is will depend on the specifics of the election.

Another value worth addressing is the `struct fake_ballots` property. One weakness of the range vote electoral system is that naive implementations can result in elections where unknown alternatives, being neither voted for or against, win elections again more popular alternatives; imagine voting for oneself in a large election, and being the only person to, therefore winning the election with a perfect score average. One common technique to avoid this kind of result in range vote implementations is for election administrators to agree ahead of time to some number of fake ballots to be cast. These fake ballots will score each choice with the same rank, typically 0, and therefore bias results towards choices which have received enough votes to overcome the low scores received from the fake ballots.

Modeling Finite-State Machine Transitions Election contract's state transitions were tested by creating and conducting mock elections to ensure that the results are as expected. Passage of time in the `testrpc` is simulated using the `evm_increaseTime` RPC method. [19]

- **First-Past-the-Post** The tests for the First-Past-the-Post electoral system

are as follows:

1. Generate a First-Past-the-Post instance.
2. Test that the owner and creation time are properly initialized.
3. Confirm that the phase is `Configuration`.
4. Add two choices to be voted for.
5. Set the voting start time.
6. Set the voting end time.
7. `freeze` the contract so that no further configuration can occur.
8. Confirm that the contract is `Frozen`.
9. Increase the time of the EVM such that the contract's `timedTransitions` will trigger and move the contract into the `Vote` phase on the first ballot submission.
10. Generate a set of Ethereum accounts and vote for the choices.
11. Monitor the cast ballots to confirm that the votes are submitted as expected.
12. Confirm that the phase is `Vote`.
13. Increase the time of the EVM such that the contract's `timedTransitions` will trigger and move the contract into the `Tally` phase.
14. Loop over the choices to find the winner.

15. Confirm that the winner is the expected winner.
16. Confirm that the phase is `Tally`.
- **Range Vote** The tests for the Range Vote electoral system are as follows:
 1. Generate a Range Vote instance.
 2. Test that the owner and creation time are properly initialized.
 3. Confirm that the phase is `Configuration`.
 4. Add three choices to be voted for.
 5. Set the voting start time.
 6. Set the voting end time.
 7. `freeze` the contract so that no further configuration can occur.
 8. Confirm that the contract is `Frozen`.
 9. Increase the time of the EVM such that the contract's `timedTransitions` will trigger and move the contract into the `vote` phase on the first ballot submission.
 10. Generate a set of Ethereum accounts and vote for the choices.
 11. Monitor the cast ballots to confirm that the votes are submitted as expected.
 12. Confirm that the phase is `Vote`.
 13. Increase the time of the EVM such that the contract's `timedTransitions`

will trigger and move the contract into the `Tally` phase.

14. Loop over the choices to find the winner.
15. Confirm that the winner is the expected winner.
16. Confirm that the phase is `Tally`

5.1.2 Authorization Components

There are a number of contracts that need testing that revolve around authorization, the most important being the `VoterRegistrationAuthority`, `Guard`, `ACLAuthority`, and `BasicACL`.

Voter Registration Authority The tests for the `VoterRegistrationAuthority` are as follows:

1. Generate a set of external voting accounts.
2. Deploy an instance of the `VoterRegistrationAuthority`.
3. Register a subset of the voting accounts with the `VoterRegistrationAuthority`.
4. Verify that the registered voting accounts return true when passed to `isRegistered`.
5. Verify that the unregistered voting accounts return false when passed to `isRegistered`.
6. Unregister a subset of the registered voting accounts.
7. Verify that the registered voting accounts return true when passed to `isRegistered`.
8. Verify that the unregistered voting accounts return false when passed to `isRegistered`.

Guard The `Guard` is tested using the previously deployed `VoterRegistrationAuthority` as an `Authority`. A `TestGuard` instance is tested by attaching an `auth` modifier to a dummy `vote` function. The `vote` function is then tested by confirming that each of the verified voting accounts can call the `vote` function. Each unregistered account is also tested to confirm that none of those accounts can call the `vote` function.

5.2 Analysis

Section 4.1 introduced a set of requirements to judge the results of this work and Section 4.2 introduced a set of objectives this research targeted for completion. This section reviews those sections in the context of test results produced for this work.

5.2.1 Requirements

Table 5.2 highlights these requirements targeted from Section 4.1 and provides an overview of what was and was not fulfilled.

Table 5.2: Requirements targeted for fulfillment.

Requirements	Fulfilled			Unfulfilled
	Fully	Mostly	Partially	
<i>Technical</i>				
Authentication		•		
System Operational	•			
Reliability		•		
Security		•		
Auditing		•		
Functional		•		
Interoperability			•	
Certification			•	
Accessibility				•
Usability				•
<i>Non-Functional</i>				
Maintenance/Evolvability			•	
Assurance			•	
Operational				•
Procedural				•
Legal				•

5.2.1.1 *Technical Requirements*

The technical requirements are ones which are meant to be fulfilled through the design and architecture of the system. The system operational requirements are considered fulfilled here. The functional and audibility requirements were fulfilled as expected. The accessibility and usability requirements were not targeted.

Certification The certification requirements assert that proofs of correctness are necessary, which this research falls short of, but the tests and documentation available do fulfill other certification requirements listed.

Authentication This research deeply explored authentication models and techniques for managing access control for use in constructing voter registries. Much of this research focused on building systems which would be reusable and composable across many election and non-election systems; however, the result of these access control models produced a security interface that, although working, is difficult to interact with in a reasonable way. Inter-contract communication is difficult and managing locks and mutexes across them is even more-so. Managing a security interface which is entirely or mostly internal to a single contract is likely a much safer approach to consider in most cases, although probably also a less flexible model in the long-term. However, a majority of the most important authentication functionalities

required are provided through the underlying blockchain infrastructure.

Reliability Requirements The underlying resiliency of the Ethereum network provides strong reliability guarantees; however, the network is not without its issues and has seen attacks on it over time. More recently the network has faced serious scaling issues which have driven gas prices up and made smart contract operation an expensive endeavour.

Security Requirements This work, by virtue of building on an Internet-based network, is accepting risks which make any consideration or application of its results inappropriate for use in large-scale high-risk elections. Specifically, the intrinsically electronic and remote properties, which are inherent in all publicly-available “off-the-shelf” blockchain technologies, present so large an attack surface for would-be attackers that it becomes an untenable foundation for any voting systems being considered for use in elections where a high-risk threat model is maintained. Correctness, security, integrity, verifiability, and efficacy are the primary concerns of the systems designed and considered in this research; however, it is worth noting that the privacy constraints offered by traditional voting systems are severely loosened within the scope of this research: no notion of receipt-freedom is assumed and privacy is only guaranteed insofar as the pseudo-anonymity provided by the Ethereum

blockchain is maintained.

It is feasible that blockchain technologies could demonstrate utility in large-scale voting systems if incorporated using well-established and known-safe patterns: e.g., voter verifiable paper audit trails, private networks, requiring transactions be signed by authorized sources, using voting machines which have been certified by a federal commission, which are being hosted in traditional polling locations, and are leveraging the appropriate cryptographic techniques to ensure privacy; e.g., homomorphic encryption, blind signing, and mix networks.

5.2.1.2 Non-Functional Requirements

The non-functional requirements as described in Section 4.1 are only fulfilled insofar as this document and source code exists. Strong efforts were not otherwise made to fulfill these requirements.

5.2.2 Objectives

Section 4.2 introduced a set of objectives this research targeted for completion. This research succeeded in exploring electoral system design and implementation and produced several contracts demonstrating single-winner electoral systems. However, this failed to implement a multi-winner electoral system supporting proportional

representation. STV is often described as a complex to implement electoral system with difficult to handle edge-cases. The finite storage and computational resources made available by the EVM make it a poor candidate for exploration. There are other proportional representation systems with much lower storage and execution complexities which are probably better candidates for resource-constrained environments like smart contracts.

This research thoroughly explored and designed many voter authentication, registration, and access control patterns for voter registration. However, as previously stated, inter-contract communication is difficult to manage, especially when multiple contracts might be deferring responsibility to a single contract managing state. This issue, combined with the resource constraints of smart contracts, is ultimately what made further research into delegative or liquid governance models infeasible. There are too many inter-contract communication issues to manage when attempting to compute delegations across contracts and there does not appear to be any clear way of designing smart contract dependencies in a way which enabled them to be open to state changes while still providing consistent values during sensitive contract execution procedures like ballot tabulation or delegation analysis.

Chapter 6 – Conclusion

The issues faced by governments and societies — especially while wrestling with issues such as the COVID-19 pandemic, state-sponsored election interference, and claims of election fraud — demonstrate a clear need for improvements with regards to electoral systems, voting systems, and governance models. The concept of decentralized organizations, democratically operating and existing on-chain, offers a provocative and alluring alternative approach which might have the potential to support more egalitarian and meritocratic social structures and governance models. Section 2.1 introduced elections, electoral systems, and the tools available for analyzing them. Section 2.2 introduced blockchain technologies, their design, and the properties thereof which might prove useful in a number of systems, such as voting, where a high degree of verifiability and confidence of correctness is required.

The literature reviewed in Section 3.1 demonstrates the difficulty of building Internet-based voting systems, and reviews a number of theoretical and real-world attacks that have been demonstrated in the various attempts to build such systems. As seen in Section 3.2, the tensions present in the security, privacy, and authenti-

cation requirements of voting systems are challenging to resolve; however, although still not without risks, there is promise in Internet voting if systems can be designed which achieve end-to-end verifiability.

Section 4.4 explored several election, authorization, and delegation designs for use on the Ethereum blockchain. However, although Chapter 5 demonstrated the possibility implementing such systems, the issues outlined in Section 5.2 make clear that the Ethereum blockchain still requires many improvements before it could be considered capable of supporting the kind of next-generation elections and electoral systems this research sought to investigate.

Among other things, the semantics available for building such systems are complex and unintuitive, an inevitable source of bugs. Further, tools and techniques, which are currently unavailable, are necessary to support large-scale systems such as voting; for example, inter-contract communication and inter-block computations should be possible with consistent states available across contract boundaries from start to end of computation.

More importantly, the computation and storage requirements demanded for electoral system operation is too high to make their use on-chain practical in most scenarios. Measuring generously, the least expensive vote operation listed in Table 5.1 would cost the equivalent of \$7.87, the average of these vote operations (across elec-

toral systems implementations) measures in at \$28.18 per vote, and in the worst case a single vote operation might cost upwards of \$57.11. The cost of executing the most-expensive tallying operation is significantly worse, costing the equivalent of over \$1,300.00 to tally the ballots of an election simulation with just 100 voters.¹ These prices are clearly far too high for serious consideration, and at best they exclude from consideration a large number of popular electoral systems which have much higher computation and storage demands than the ones selected.

These costs appear even worse when viewed in light of the fact that the measurements made in Table 5.1 were the product of electoral system implementations which were specifically designed to accurately measure the raw costs of the electoral systems' computational and storage requirements, and therefore included no security or state management systems beyond that. Inclusion of such systems which would certainly have made these operations more costly. Still, these results should not be viewed in vacuum; some estimates put the cost of election administration in the US at around \$8.00 per voter, so the lower bounds of these results suggest that there is potential here, especially with advancements to the Ethereum network and if some scaling issues can be resolved.

¹The gas price at the time of writing is approximately 120 gwei.

Summary of Accomplishments

- Designed authorization components for managing voter registration and providing access control in a generalized way.
- Implemented election contracts, tests, and simulations to benchmark and analyze electoral system performance and cost.
- Identified electoral system criteria relevant to Ethereum-based implementations of electoral systems.
- Identified techniques for improving some performance characteristics of electoral systems.
- Identified features which damage performance characteristics of electoral systems.

Future Work

- Unless scaling issues are resolved, it is likely that more reasonable approaches to on-chain electoral system design would involve using dedicated or permissioned blockchain implementations and leveraging the more traditional approaches to E2E-VIV security as outlined in Section 3.2 and Section 5.2.
- If scaling issues are resolved then the following items seem like worthwhile areas

of research:

- A set of hardened electoral system contracts and libraries.
- Electoral system contracts which provide support for multi-winner elections and support proportional representation.
- A standardized election interface. This seems difficult to produce given the unique inputs and outputs required for different kinds of elections.
- An visual interface for marking and casting ballots to election contracts.
- Approaches and semantics for achieving consistency with inter-contract computation, especially with regards to inter-block computation.

Bibliography

- [1] Ben Adida. 2008. Helios: web-based open-audit voting. In *USENIX security symposium*. Volume 17, 335–348.
- [2] A.M. Antonopoulos. 2014. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media. ISBN: 9781491902646. <https://books.google.com/books?id=IXmrBQAAQBAJ>.
- [3] A.M. Antonopoulos and G.W.P. D. 2018. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media. ISBN: 9781491971918. <https://books.google.com/books?id=nJJ5DwAAQBAJ>.
- [4] Adam Back. 2002. Hashcash - A Denial of Service Counter-Measure. Technical report.
- [5] John J. Bartholdi and James B. Orlin. 1991. Single transferable vote resists strategic voting. *Social Choice and Welfare*, 8, 4, (October 1991), 341–354. ISSN: 0176-1714, 1432-217X. DOI: 10.1007/BF00183045. <http://link.springer.com/10.1007/BF00183045>.

- [6] Nadja Betzler, Arkadii Slinko, and Johannes Uhlmann. 2013. On the computation of fully proportional representation. *Journal of Artificial Intelligence Research*, 47, 475–519.
- [7] Dan Boneh and Philippe Golle. 2002. Almost entirely correct mixing with applications to voting. In *Proceedings of the 9th ACM conference on Computer and communications security*, 68–77.
- [8] Vitalik Buterin. 2013. Dagger: A Memory-Hard to Compute, Memory-Easy to Verify Script Alternative. Technical report. <http://www.hashcash.org/papers/dagger.html>.
- [9] Vitalik Buterin et al. 2014. A Next-Generation Smart Contract and Decentralized Application Platform. Technical report 37.
- [10] Vitalik Buterin. 2014. Dagger Hashimoto. eth.wiki. <https://eth.wiki/concepts/dagger-hashimoto>.
- [11] John Canny and Stephen Sorkin. 2004. Practical large-scale distributed key generation. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 138–152.
- [12] Jimmy Carter, Gerald R. Ford, Lloyd N. Cutler, Robert H. Michel, and Philip Zelikow. 2002. *To Assure Pride and Confidence in the Electoral Process: Re-*

port of the National Commission on Federal Election Reform. Brookings Institution Press. ISBN: 9780815706311. <http://www.jstor.org/stable/10.7864/j.ctv80cdn3>.

- [13] Nikos Chondros, Bingsheng Zhang, Thomas Zacharias, Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Alex Delis, Aggelos Kiayias, and Mema Roussopoulos. 2016. D-DEMOS: a distributed, end-to-end verifiable, internet voting system. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 711–720.
- [14] Thaddeus Dryja. 2009. Hashimoto: I/O Bound Proof-of-Work. Technical report.
- [15] Quinn DuPont. 2017. Experiments in algorithmic governance: a history and ethnography of “the DAO,” a failed decentralized autonomous organization. *Bitcoin and beyond*, 157–177.
- [16] Susan Dzieduszycka-Suinat, Judy Murray, Joseph R Kiniry, Daniel M Zimmerman, Daniel Wagner, Philip Robinson, Adam Foltzer, and Shpatar Morina. 2015. The Future of Voting: End-to-End Verifiable Internet Voting. Specification and Feasibility Assessment Study. U.S. Vote Foundation, (July 1, 2015), 136. https://usvotefoundation-drupal.s3.amazonaws.com/prod/E2EVIV_full_report.pdf.

- [17] Ethereum Foundation. 2018. Ethereum. ethereum.org. <https://ethereum.org>.
- [18] Ethereum Foundation. 2021. Ethereum accounts. [ethereum.org](https://ethereum.org/en/developers/docs/accounts/). <https://ethereum.org/en/developers/docs/accounts/>.
- [19] ethereumjs. 2017. ethereumjs-testrpc. npm package documentation. (2017). <https://www.npmjs.com/package/ethereumjs-testrpc>.
- [20] FairVote.org. 2021. Single winner voting method comparison chart. FairVote. <https://www.fairvote.org/alternatives>.
- [21] FairVote.org. 2021. Single winner voting method comparison chart (archived). FairVote. <https://archive3.fairvote.org/reforms/instant-runoff-voting/irv-and-the-status-quo/irv-versus-alternative-reforms/single-winner-voting-method-comparison-chart/>.
- [22] Federal Voting Assistance Program. 2001. Voting Over the Internet Pilot Project Assessment Report. Technical report. Department of Defense Washington Headquarters Services, (June 1, 2001), 93. <https://www.fvap.gov/uploads/FVAP/Reports/voi.pdf>.
- [23] Federal Voting Assistance Program. 2007. Expanding the Use of Electronic Voting Technology for UOCAVA Citizens. Technical report. Department of

Defense. https://www.eac.gov/sites/default/files/eac_assets/1/6/Exhibit%20A%20-%20L.pdf.

- [24] Bryan Alexander Ford. 2002. Delegative Democracy. Technical report.
- [25] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1991. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38, 3, 690–728.
- [26] Andrew Gumbel. 2005. *Steal This Vote*. Nation Books, (July 2005). ISBN: 9781560256762.
- [27] James Heather and David Lundin. 2008. The append-only web bulletin board. In *International Workshop on Formal Aspects in Security and Trust*. Springer, 242–256.
- [28] Nathan Hernandez. 2021. Nathanph/election-contracts. GitHub. <https://github.com/nathanph/election-contracts>.
- [29] David Jefferson, Avi Rubin, and Barbara Simons. 2007. A comment on the May 2007 DoD report on Voting Technologies for UOCAVA Citizens. Technical report. https://graphics8.nytimes.com/packages/pdf/national/2007_voting_comment.pdf.

- [30] David Jefferson, Aviel D Rubin, Barbara Simons, and David Wagner. 2004. A Security Analysis of the Secure Electronic Registration and Voting Experiment (SERVE). Technical report.
- [31] Chunheng Jiang, Sujoy Sikdar, Jun Wang, Lirong Xia, and Zhibing Zhao. 2017. Practical algorithms for computing stv and other multi-round voting rules. In *EXPLORE-2017: The 4th Workshop on Exploring Beyond the Worst Case in Computational Social Choice*.
- [32] Aniket Kate and Ian Goldberg. 2009. Distributed key generation for the internet. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 119–128.
- [33] Aggelos Kiayias and Moti Yung. 2004. The vector-ballot e-voting approach. In *International Conference on Financial Cryptography*. Springer, 72–89.
- [34] Gregory Miller. 2010. The d.c. pilot project: facts vs. fictions from our viewpoint. TrustTheVote. (July 6, 2010). <https://trustthevote.org/blog/2010/07/06/dc-pilot-project-facts-vs-fictions-osdv-viewpoint/>.
- [35] Satoshi Nakamoto. 2009. Bitcoin: a peer-to-peer electronic cash system. (2009). <http://www.bitcoin.org/bitcoin.pdf>.

- [36] Mike Ossipoff and Warren Smith. 2021. RangeVoting.org - survey of FBC (favorite-betrayal criterion). <https://rangevoting.org/FBCsurvey.html>.
- [37] 2021. RangeVoting.org - criteria obeyed and disobeyed by score voting. <https://www.rangevoting.org/Criteria.html>.
- [38] 2021. RangeVoting.org - single winner voting system comparison chart. <https://www.rangevoting.org/CompChart.html#properties>.
- [39] Andrew Reynolds. 2005. *Electoral system design: The New International IDEA Handbook*. International Institute for Democracy and Electoral Assistance, Stockholm, Sweden. ISBN: 91-85391-18-2.
- [40] Peter YA Ryan and Vanessa Teague. 2009. Pretty good democracy. In *International Workshop on Security Protocols*. Springer, 111–130.
- [41] Donald G Saari. 2008. Mathematics and voting. *Notices of the AMS*, 55, 4, 448–455.
- [42] Krishna Sampigethaya and Radha Poovendran. 2006. A survey on mix networks and their secure applications. *Proceedings of the IEEE*, 94, 12, 2142–2181.

- [43] Dominik Schiener. 2016. Liquid democracy: true democracy for the 21st century. Medium. (March 11, 2016). <https://medium.com/organizer-sandbox/liquid-democracy-true-democracy-for-the-21st-century-7c66f5e53b6f>.
- [44] Warren D Smith. 2006. Comparative Survey of Multiwinner Election Methods. Technical report, 16.
- [45] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J Alex Halderman. 2014. Security analysis of the estonian internet voting system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 703–715.
- [46] Nicolaus Tideman. 1995. The single transferable vote. *Journal of Economic Perspectives*, 9, 1, 27–38.
- [47] U.S. Congress. 2001. Electronic voting demonstration project. (2001). <https://uscode.house.gov/statviewer.htm?volume=115&page=1277>.
- [48] U.S. Congress. 2010. National defense authorization act for fiscal year 2010. (2010). <https://uscode.house.gov/statviewer.htm?volume=123&page=2190>.
- [49] U.S. Election Assistance Commission. 2011. A Survey of Internet Voting. Technical report. U.S. Election Assistance Commission, (September 14, 2011), 149. https://www.eac.gov/sites/default/files/eac_assets/1/28/SIV-FINAL.pdf.

- [50] Scott Wolchok, Eric Wustrow, Dawn Isabel, and J Alex Halderman. 2012. Attacking the washington, dc internet voting system. In *International Conference on Financial Cryptography and Data Security*. Springer, 114–128.
- [51] Gavin Wood et al. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Technical report 2014, 1–32.

Appendix A – Software Documentation

The source code, architecture, design, and documentation thereof are presented, reviewed, and provided here. The code reviewed in this section is not guaranteed to represent the most accurate or recent state of the software and is abbreviated throughout for readability. The most recent state of the software can be found on GitHub. [28] This appendix is divided into three major sections:

- *Authorization Components*, which introduces the design and implementation of authorization-related components.
- *Election Components*, which introduces the design and implementation of election-related components.
- *Delegation Components*, which introduces the design and implementation of delegation-related components.

A.1 Authorization Components

The authorization components are those which are responsible for managing access control for **contracts**. This section builds varying complexities of access control beginning with:

1. *Primitive Contract Ownership*, which reviews basic access control mechanisms; then
2. *Generalized Access Control*, which introduces components for guarding access to resources; next
3. *Access Control Lists*, which demonstrate components for constructing and managing access control lists and integrating them with the components introduced for generalized access control; and finally,
4. *Registries*, which demonstrates voter registry components leveraging the aforementioned components to build and manage voter databases.

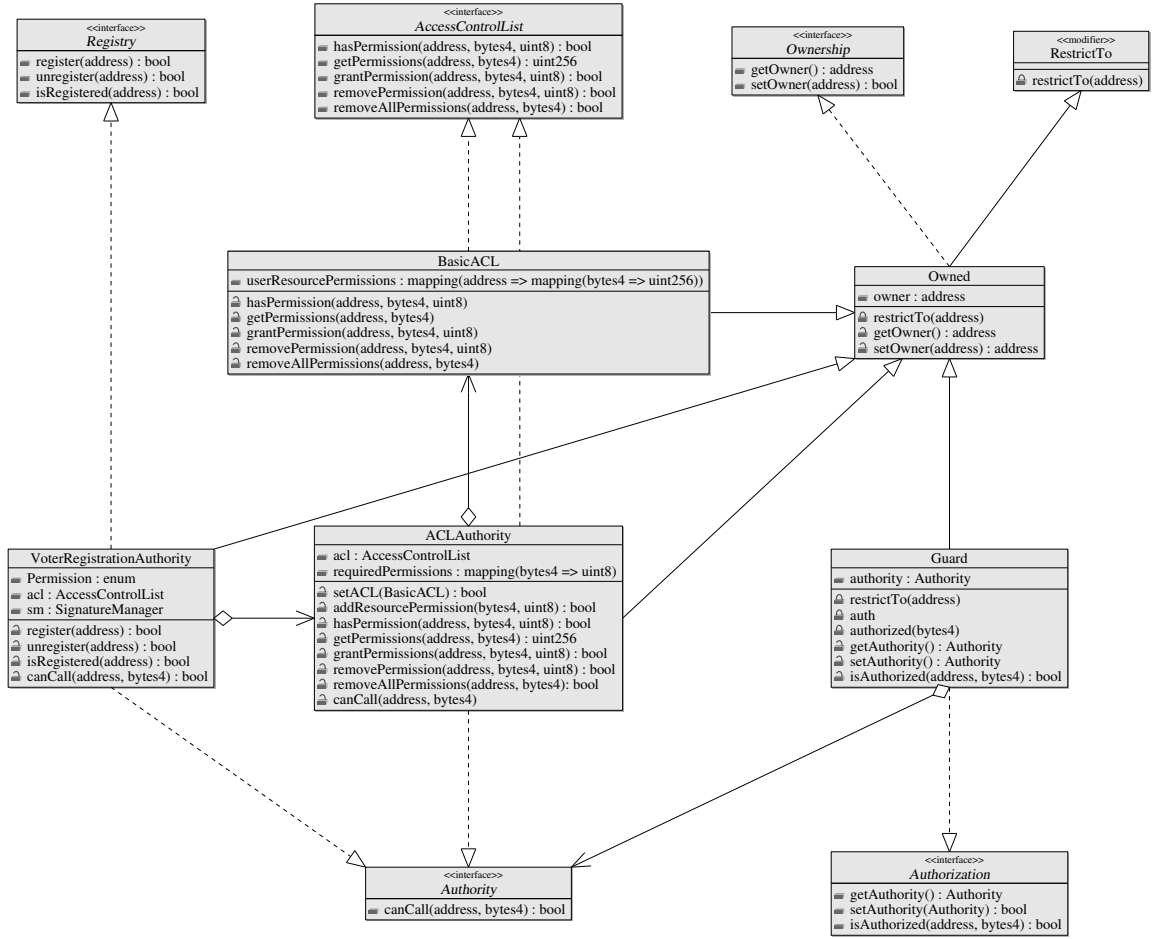


Figure A.1: Authorization dependency graph modeling.

A.1.1 Primitive Contract Ownership

Managing **contract** ownership is one of the most basic forms of access control in the Ethereum ecosystem. Here we introduce:

1. **interface Ownership**, which defines **functions** to express **contract** ownership,

2. `contract RestrictTo`, which defines a `modifier` for restricting access to `function` calls based on an `address`, and
3. `contract Owned`, a convenience `contract`, which provides an implementation of `interface Ownership` and leverages `contract RestrictTo`.

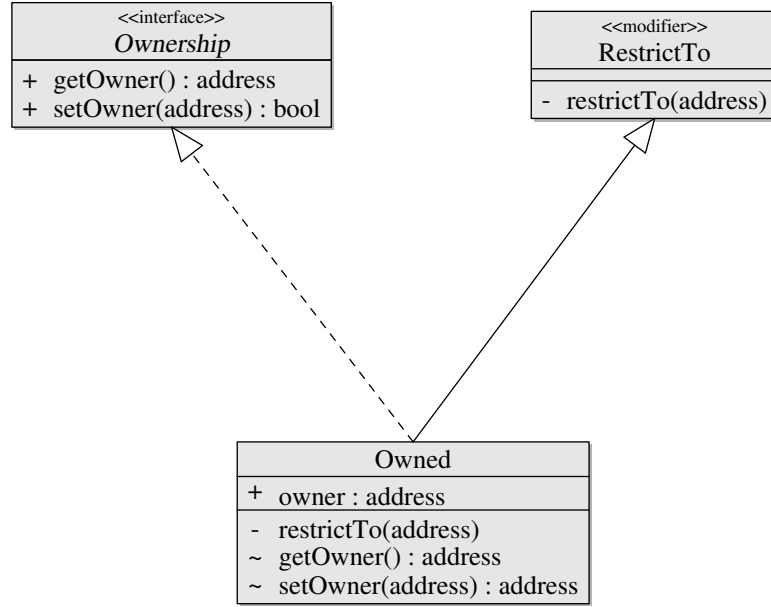


Figure A.2: Contract ownership dependency graph modeling.

A.1.1.1 Contract *RestrictTo*

The `contract`, `contract RestrictTo`, introduces a single `modifier`, `modifier restrictTo`, which requires the caller of the `function`, `msg.sender`, to have the same `address` as the argument, `address _subject`, provided to the `modifier` when called.

```
</> contract RestrictTo
```

Listing A.1

```
1 contract RestrictTo {  
2     modifier restrictTo (address _subject) {  
3         require(msg.sender == _subject);  
4         _;  
5     }  
6 }
```

⚡ Contract Operations

🔑 `modifier restrictTo (address _subject)`, restricts access to `function` calls based on an `address` provided.

Restriction to `functions` is accomplished by comparing the `address` of the `function caller`, `msg.sender`, against the configured `address`, `address _subject`. If the `address` of the `msg.sender` does not match the `address` of the `_subject` then the `require` statement will force the immediate arrest of `contract` evaluation, `revert` the *state* of the `contract`, refund any remaining gas, `gasleft()`, to the *caller*, and exit.¹


➡ `address _subject`, the *subject* who is to be granted access to the `function`.

The `address` of the *subject* may be *any* Ethereum account, including *contract accounts*.

Interface Ownership The `interface`, `interface Ownership`, introduces two `functions` for managing contract ownership:

¹Note that the *caller*, `msg.sender`, of the `function` will not necessarily be an *external account*, e.g., human user; the *caller* of the `contract function` may itself also be a `contract`, i.e., a *contract account*, which is calling the `contract function` from its own `contract function`.

1. `function` `getOwner`, which is expected to return the `address` of the owner of the `contract`, and
2. `function` `setOwner`, which is expected to update the `address` of the owner of the `contract`.

 interface Ownership
Listing A.2

```

1 interface Ownership {
2   function getOwner () public view returns (address _owner);
3   function setOwner (address _owner) public returns (bool _success);
4 }
```

i *Interface Specification*

- ⚙️ `function` `getOwner` (), returns the `address` of the owner of the `contract`.
 - ↩️ `address` `_owner`, the *subject* representing the owner of the `contract`.
- ⚙️ `function` `setOwner` (`address` `_owner`), updates the `address` representing the owner of the `contract`.
 - ➡️ `address` `_owner`, the *subject* which is to be granted ownership of the `contract`.²
 - ↩️ `bool` `_success`, resolves to `true` if the operation was successful; otherwise `false`.

A.1.1.2 *Contract Owned*

A convenience `contract`, `contract` `Owned`, implements `interface` `Ownership` and extends `contract` `RestrictTo`; in doing so, `contract` `Owned` provides a simple mechanism

² The `address` of the *subject* may be *any* Ethereum account, including *contract accounts*.

for expressing **contract** ownership, extending **contract** Owned; e.g., **contract** MyContract **is** Owned {}.

```
</> contract Owned Listing A.3
1  contract Owned is RestrictTo, Ownership {
2    address public owner;
3
4    constructor () {
5        owner = msg.sender;
6    }
7
8    function getOwner () public view returns (address _owner) {
9        return owner;
10   }
11
12   function setOwner (address _owner) public restrictTo(owner) returns (bool
    ↪ _success) {
13       owner = _owner;
14       return true;
15   }
16 }
```

Contract State

🔑 **address** owner, maintains the **address** of the current owner of the **contract**.

⚙️ Contract Operations

⚙️ **constructor** (), upon creation and initialization of this **contract** the **constructor** sets the **address** owner property of the **contract** to the **address** of the **contract** creator, **msg.sender**, i.e., the *subject* submitting the CREATE opcode.

⚙️ **function** getOwner (), returns the **address** of the owner of the **contract**.
↪ **address** _owner, the *subject* representing the owner of the **contract**.

⚙️ **function** setOwner (**address** _owner), updates the **address** representing the owner

of the **contract**, effectively transferring ownership of the **contract**.

🔑 **modifier** `restrictTo (owner)`, restricts access to the **function** such that *only* the current owner of the **contract** can update/transfer ownership of the **contract**.

➡ **address** `_owner`, the *subject* who is to be granted access to the **function**.

↩ **bool** `_success`, the *subject* representing the owner of the **contract**

A.1.2 Generalizing Contract Access Control

In order to generalize **contract** access control we introduce:

1. **interface** `Authority`, which defines **functions** to determine whether some *subject* is permitted to access some *resource*,
2. **contract** `Authorization`, which defines a **modifier** for restricting access to **function** calls based on an **address**, and
3. **contract** `Guard`, a convenience **contract**, which provides an implementation of **interface** `Authorization` by aggregating implementations of **interface** `Authority`.

Together these components allow us to provide a generalized access control model; isolating and deferring the responsibilities of authorization.

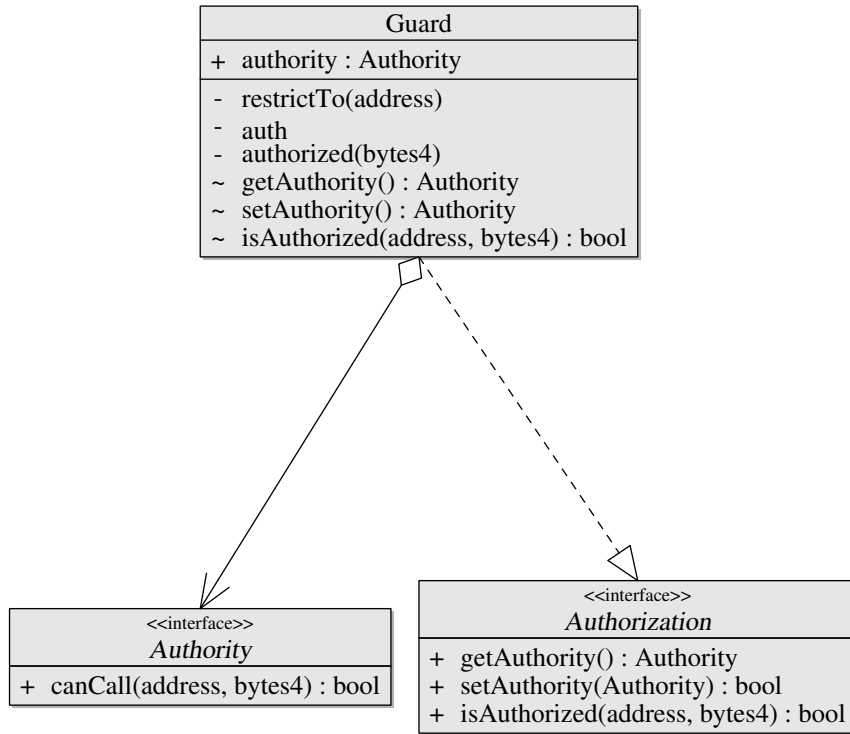


Figure A.3: Generalized contract access control.

A.1.2.1 Interface Authority

The **interface**, **interface Authority**, introduces functionality for managing whether some *subject*, typically an Ethereum account represented by **address**, can access some *resource*, typically an Ethereum **function** represented by **function** signature, **bytes4**. By defining the *resource* by its **function** signature and not by a specific **function** owned by a specific **contract** we leave open the possibility for managing **contract** access control across **contracts**, a functionality which will be necessary to build pseudo-centralized


registration authorities.

interface Authority

Listing A.4

```
1 interface Authority {  
2   function canCall (address _subject, bytes4 _resource) public constant returns  
   ↪ (bool _canCall);  
3 }
```

i Interface Specification

 **function** canCall (**address** _subject, **bytes4** _resource), evaluates whether some *resource*, **contract function**, can be used by some *subject*, Ethereum account.

↪ **address** _subject, the *subject*, Ethereum account, whose permissions are being evaluated.³

↪ **bytes4** _resource, the *resource*, **contract function**, which the *subject's* permissions are being evaluated against.

↪ **bool** _canCall, resolves to **true** if the *subject*, **address** _subject, is permitted to access the *resource*, **bytes4** _resource, otherwise **false**.

A.1.2.2 Interface Authorization

The **interface**, **interface** Authorization, introduces functionality for managing authorities, **function** getAuthority() and **function** setAuthority(), and also functionality similar to that of an Authority, **function** isAuthorized().

³The **address** of the *subject* may be *any* Ethereum account, including *contract accounts*.

```

1 interface Authorization {
2   function getAuthority () public constant returns (address _authority);
3   function setAuthority (address _authority) public returns (bool _success);
4   function isAuthorized (address _subject, bytes4 _resource) public returns
   ↪ (bool _isAuthorized);
5 }

```

i Interface Specification

⚙️ **function** `getAuthority ()`, returns the **address** of a **contract** which implements the **interface** `Authority`.

↪ **address** `_authority`, the **address** of a **contract** which implements the **interface** `Authority`.

⚙️ **function** `setAuthority (address _authority)`, updates the state of the **contract** to reflect the new **contract** which provides an implementation of the **Authority interface**.

↪ **address** `_authority`, the **address** of a **contract** which implements the **interface** `Authority`.

↪ **bool** `_success`, resolves to **true** if the operation was successful; otherwise **false**.

⚙️ **function** `isAuthorized (address _subject, bytes4 _resource)`, evaluates whether some *subject*, Ethereum account, is authorized to access some *resource*, **contract function**.

↪ **address** `_subject`, the *subject*, Ethereum account, whose permissions are being evaluated.⁴

↪ **bytes4** `_resource`, the *resource*, **contract function**, which the *subject's* permissions are being evaluated against.

⁴The **address** of the *subject* may be *any* Ethereum account, including *contract accounts*.

↩ `bool _isAuthorized`, resolves to `true` if the *subject*, `address _subject`, is permitted to access the *resource*, `bytes4 _resource`, otherwise `false`.

A.1.2.3 Contract Guarded

The `contract`, `contract Guarded`, is a `contract` which offers a convenient mechanism for easily integrating generalized `contract` access control functionality; e.g., `contract MyContract is Guarded {}`. `contract Guarded`, by virtue of implementing the `Authorization interface`, supports deferring access control responsibilities to an external `contract` which implements the `Authority interface` while leaving open the possibility for a `contract` to provide its own access control implementation by itself implementing the `Authority interface`.

```
</> contract Guarded Listing A.6
1  contract Guarded is Owned, Authorization {
2      Authority public authority;
3
4      function getAuthority () public constant returns (address _authority) {
5          return address(authority);
6      }
7
8      function setAuthority (address _authority) public auth returns (bool
9          ↪ _success) {
10         authority = Authority(_authority);
11         return true;
12     }
13     function isAuthorized (address _subject, bytes4 _resource) public returns
14         ↪ (bool _isAuthorized) {
```

```

14     if (_subject == address(this)) return true;
15     if (authority == Authority(0)) return false;
16     if (_subject == owner) return true;
17     return authority.canCall(_subject, _resource);
18 }
19
20 modifier auth {
21     assert(isAuthorized(msg.sender, msg.sig));
22     _;
23 }
24
25 modifier authorized (bytes4 _resource) {
26     assert(isAuthorized(msg.sender, _resource));
27     _;
28 }
29 }

```

⚡ Contract Operations

🔑 `modifier auth ()`, restricts access to **function** calls based on an **address** provided.

Restriction to **functions** is accomplished by comparing the **address** of the **function caller**, `msg.sender`, against the configured **address**, `address _subject`.

If the **address** of the `msg.sender` does not match the **address** of the `_subject` then the **require** statement will force the immediately arrest of **contract** evaluation, **revert** the *state* of the **contract**, refund any remaining gas, `gasleft()`, to the *caller*, and exit.⁵

➡ `address _subject`, the *subject* who is to be granted access to the **function**.

The **address** of the *subject* may be *any* Ethereum account, including *contract accounts*.

🔑 `modifier authorized (address _resource)`, restricts access to **function** calls based

⁵Note that the *caller*, `msg.sender`, of the **function** will not necessarily be an *external account*, e.g., human user; the *caller* of the **contract function** may itself also be a **contract**, i.e., a *contract account*, which is calling the **contract function** from its own **contract function**.

on an **address** provided.

Restriction to **functions** is accomplished by comparing the **address** of the **function caller**, `msg.sender`, against the configured **address**, `address _subject`. If the **address** of the `msg.sender` does not match the **address** of the `_subject` then the **require** statement will force the immediately arrest of **contract** evaluation, **revert** the *state* of the **contract**, refund any remaining gas, `gasleft()`, to the *caller*, and exit.⁶

➡ `address _subject`, the *subject* who is to be granted access to the **function**.

The **address** of the *subject* may be *any* Ethereum account, including *contract accounts*.

A.1.3 Access Control Lists

We introduce access control lists to provide a generalized form of access control through a well-understood and common interface:

1. **interface** `AccessControlList`, which defines the basic actions required for an access control list implementation;
 2. **contract** `BasicACL`, which provides a basic implementation of **interface** `AccessControlList`;
- and

⁶Note that the *caller*, `msg.sender`, of the **function** will not necessarily be an *external account*, e.g., human user; the *caller* of the **contract function** may itself also be a **contract**, i.e., a *contract account*, which is calling the **contract function** from its own **contract function**.

3. **contract** `ACLAuthority`, which aggregates an **interface** `AccessControlList` implementation, **contract** `BasicACL` in this case, to back an **interface** `Authority` implementation.

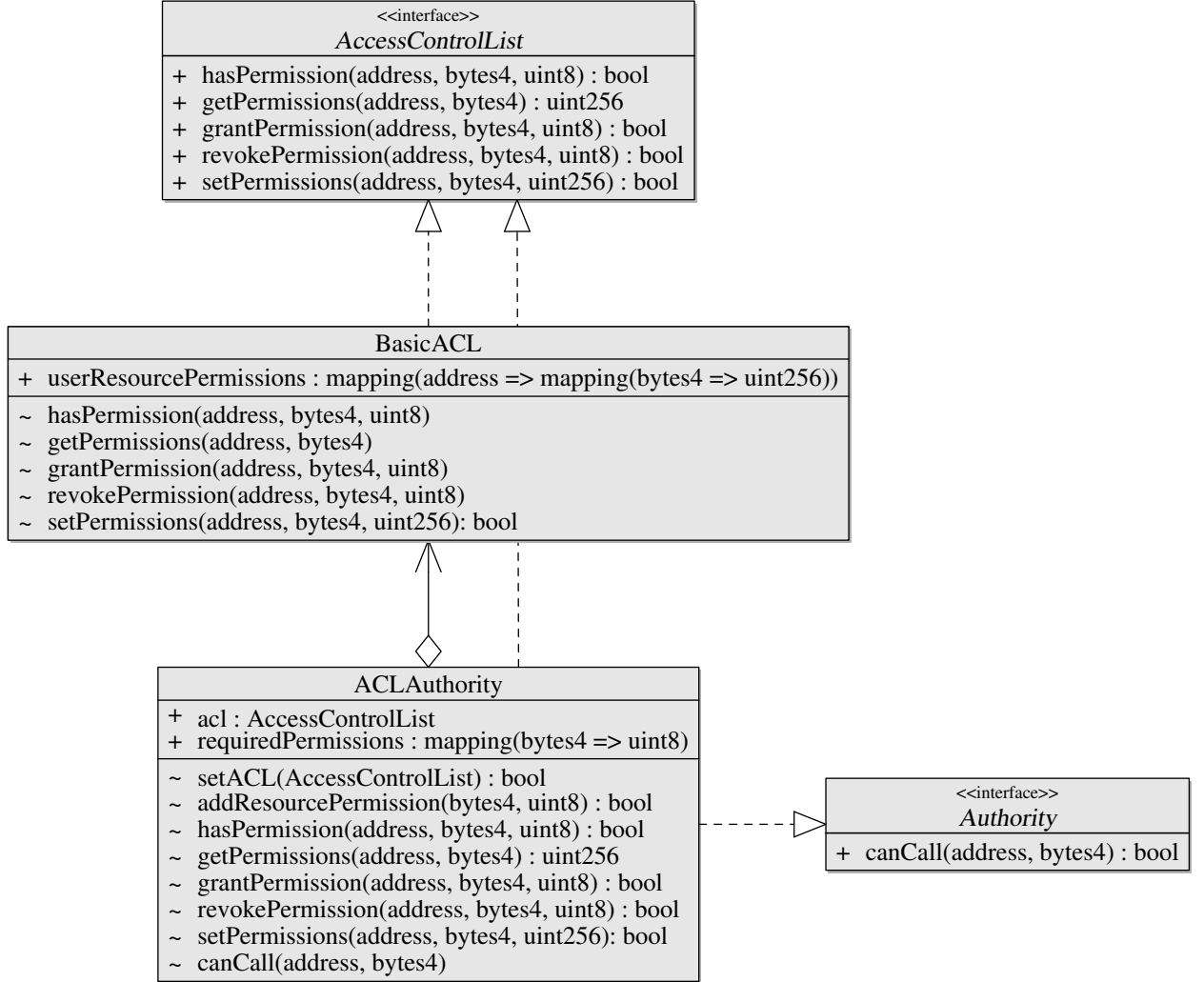


Figure A.4: Generalized contract access control by way of access control lists.

A.1.3.1 Interface *AccessControlList*

The **interface**, `interface AccessControlList`, introduces five **function** definitions to achieve basic access control list functionality:

1. **function** `hasPermission`, which validates that a *subject* has the *permission* required access some *resource*.
2. **function** `getPermissions`, which returns the *permissions* some *subject* has for a given *resource*.
3. **function** `setPermissions`, which updates the *permissions* some *subject* has for a given *resource*.
4. **function** `grantPermission`, which grants a single *permission* for some *subject* with respect to some *resource*.
5. **function** `revokePermission`, which revokes a single *permission* for some *subject* with respect to some *resource*.

 `interface AccessControlList`

Listing A.7

```
1 interface AccessControlList {
2   function hasPermission (address _subject, bytes4 _resource, uint8
    ↪ _permission)
3     public view returns (bool _hasPermission);
4
5   function getPermissions (address _subject, bytes4 _resource)
6     public view returns (uint256 _permissions);
7
8   function setPermissions (address _subject, bytes4 _resource, uint256
    ↪ _permissions)
```

```

9     public returns (bool _success);
10
11    function grantPermission (address _subject, bytes4 _resource, uint8
    ↪ _permission)
12        public returns (bool _success);
13
14    function revokePermission (address _subject, bytes4 _resource, uint8
    ↪ _permission)
15        public returns (bool _success);
16 }

```

i Interface Specification

⚙️ **function** `hasPermission (address _subject, bytes4 _resource, uint8 _permission)`, evaluates whether some *subject* has the requisite *permission* to access some *resource*.

➡️ `address _subject`, the **address** of an account representing the *subject*, i.e., the *user*, to evaluate permissions against.

➡️ `bytes4 _resource`, the *resource*, signature hash of a Solidity **function**, to validate *permissions* against.

➡️ `uint8 _permission`, a *permission* level, or *action*, which is being validated.

↩️ `bool _hasPermission`, returns the **true** if the *subject* has the requisite *permission* to access the *resource*, otherwise **false**.

⚙️ **function** `getPermissions (address _subject, bytes4 _resource)`, retrieves the *permissions*, for a $\langle \text{subject}, \text{resource} \rangle$ pair.

➡️ `address _subject`, the account **address** of the *subject* whose *permissions* are to be retrieved.

➡️ `bytes4 _resource`, the *resource*, **function** signature, which the *permissions* should be retrieved for.

↩️ `uint256 _permissions`, the **uint256** representation of the *subject's permissions* for a given *resource*.

⚙️ **function** setPermissions (**address** _subject, **bytes4** _resource, **uint256** _permissions),

updates a *subject's permissions* for some *resource*.

➡️ **address** _subject, the account **address** of the *subject* whose *permissions* are to be modified.

➡️ **bytes4** _resource, the *resource*, **function** signature, which the *permissions* are to be modified for.

➡️ **uint256** _permissions, the **uint256** representation of the *subject's new permissions* for the *resource*.

↩️ **bool** _success, resolves to **true** if the operation was successful, otherwise **false**.

⚙️ **function** grantPermission (**address** _subject, **bytes4** _resource, **uint8** _permission), grants

a *subject* some *permission* for a given *resource*.

➡️ **address** _subject, the account **address** of the *subject* whose *permissions* are to be modified.

➡️ **bytes4** _resource, the *resource*, **function** signature, which the *permissions* are to be modified for.

➡️ **uint8** _permission, a **uint8** value representing the *permission-bit* to enable for the <*subject, resource*> pair.

↩️ **bool** _success, resolves to **true** if the operation was successful, otherwise **false**.

⚙️ **function** revokePermission (**address** _subject, **bytes4** _resource, **uint8** _permission),

revokes a *subject's permission* for a given *resource*.

➡️ **address** _subject, the account **address** of the *subject* whose *permissions* are to be modified.

➡️ **bytes4** _resource, the *resource*, **function** signature, which the *permissions* are to be modified for.

- ➡ `uint8 _permission`, a `uint8` value representing the *permission-bit* to revoke for the $\langle \textit{subject}, \textit{resource} \rangle$ pair.
- ➡ `bool _success`, resolves to `true` if the operation was successful, otherwise `false`.

A.1.3.2 Contract BasicACL

The `contract`, `contract BasicACL`, implements the ACL `interface`, `interface AccessControllist`, to provide a primitive ACL implementation. The `contract` implementation is backed by a mapping of references, from *subject* to *resource* to *permissions* — as described in Chapter 4, *Methods* — i.e., a nested sparse vector mapping.

</> contract BasicACL
Listing A.8

```

1 contract BasicACL is Owned, AccessControllist {
2   mapping (address ⇒ mapping (bytes4 ⇒ uint256)) subjectResourcePermissions;
3
4   function hasPermission (address _subject, bytes4 _resource, uint8
   ↳ _permission) public constant restrictTo(owner) returns (bool
   ↳ _hasPermission) {
5     uint256 result = subjectResourcePermissions[_subject][_resource] &
   ↳ (uint256(1) << _permission);
6     return (result > 0);
7   }
8
9   function getPermissions (address _subject, bytes4 _resource) public constant
   ↳ restrictTo(owner) returns (uint256 _permissions) {
10    return subjectResourcePermissions[_subject][_resource];
11  }
12
13  function grantPermission (address _subject, bytes4 _resource, uint8
   ↳ _permission) public restrictTo(owner) returns (bool _success) {
14    subjectResourcePermissions[_subject][_resource] |= uint256(1) <<
   ↳ _permission;
15    return true;

```

```

16 }
17
18 function revokePermission (address _subject, bytes4 _resource, uint8
↳ _permission) public restrictTo(owner) returns (bool _success) {
19     subjectResourcePermissions[_subject][_resource] &= ~(uint256(1) <<
↳ _permission);
20     return true;
21 }
22
23 function setPermissions (address _subject, bytes4 _resource, _permissions)
↳ public restrictTo(owner) returns (bool _success) {
24     subjectResourcePermissions[_subject][_resource] = _permissions;
25     return true;
26 }
27 }

```

Contract State

🗄️ `mapping (address ⇒ mapping (bytes4 ⇒ uint256)) subjectResourcePermissions` a nesting mapping used to record the *permissions* which *subjects* have to access various *resources*.

Here *subjects* are represented and identified by their account `address`; *resources* are assumed to be `function` signatures, `bytes4`; and *permissions* are bit vectors, backed by `uint256` values, where bit masks are leveraged to retrieve individual *permission* values.

⚙️ Contract Operations

⚙️ `function hasPermission (address _subject, bytes4 _resource, uint8 _permission)`, evaluates whether some *subject* has the requisite *permission* to access some *resource*.

Permission evaluation occurs by:

1. retrieving the *permissions* bit vector from the `mapping subjectResourcePermissions`,

where the *subject*, `address _subject`, and *resource*, `bytes4 _resource`, are used as keys;

2. creating a *permission* bit mask by left-shifting 1 `_permission` times,

```
1 << _permission;
```

3. evaluating `permissions ^ bit_mask`; and finally,

4. returning `true` if the value resulting from the evaluation is greater than 0, i.e., the *subject* has *permission* to access to the *resource*.

➡ `address _subject`, the `address` of an account representing the *subject*, i.e., the *user*, to evaluate permissions against.

➡ `bytes4 _resource`, the *resource*, signature hash of a Solidity `function`, to validate *permissions* against.

➡ `uint8 _permission`, a *permission* level, or *action*, which is being validated.

➡ `bool _hasPermission`, returns the `true` if the *subject* has the requisite *permission* to access the *resource*, otherwise `false`.

⚙️ `function getPermissions (address _subject, bytes4 _resource)`, retrieves the *permissions*, for a *<subject, resource>* pair.

➡ `address _subject`, the account `address` of the *subject* whose *permissions* are to be retrieved.

➡ `bytes4 _resource`, the *resource*, `function` signature, which the *permissions* should be retrieved for.

➡ `uint256 _permissions`, the `uint256` representation of the *subject's permissions* for a given *resource*.

⚙️ `function setPermissions (address _subject, bytes4 _resource, uint256 _permissions)`,

updates a *subject's permissions* for some *resource*.

🔑 `modifier restrictTo (owner)`, restricts access to the `function` such that *only* the current owner of the `contract` can call it.

➡ `address _subject`, the account `address` of the *subject* whose *permissions* are to be modified.

➡ `bytes4 _resource`, the *resource*, `function` signature, which the *permissions* are to be modified for.

➡ `uint256 _permissions`, the `uint256` representation of the *subject's* new *permissions* for the *resource*.

↩ `bool _success`, resolves to `true` if the operation was successful, otherwise `false`.

⚙️ `function grantPermission (address _subject, bytes4 _resource, uint8 _permission)`, grants a *subject permission* for a given *resource*.

Permission grant occurs by:

1. retrieving the *permissions* bit vector from the `mapping subjectResourcePermissions`, where the *subject*, `address _subject`, and *resource*, `bytes4 _resource`, are used as keys;
2. creating a *permission* bit mask by left-shifting 1 `_permission` times,
`1 << _permission;`
3. evaluating `permissions ∨ bit_mask` to produce a new *permissions* bit vector; and finally,
4. updating the state of the `contract` by storing the resulting *permissions* bit vector back into the `subjectResourcePermissions mapping`.

🔑 `modifier restrictTo (owner)`, restricts access to the `function` such that *only* the current owner of the `contract` can call it.

- ➡ `address _subject`, the account `address` of the *subject* whose *permissions* are to be modified.
- ➡ `bytes4 _resource`, the *resource*, `function` signature, which the *permissions* are to be modified for.
- ➡ `uint8 _permission`, a `uint8` value representing the *permission-bit* to enable for the $\langle \textit{subject}, \textit{resource} \rangle$ pair.
- ➡ `bool _success`, resolves to `true` if the operation was successful, otherwise `false`.

⚙️ `function revokePermission (address _subject, bytes4 _resource, uint8 _permission)`, revokes a *subject's permission* for a given *resource*.

Permission revocation occurs by:

1. retrieving the *permissions* bit vector from the `mapping subjectResourcePermissions`, where the *subject*, `address _subject`, and *resource*, `bytes4 _resource`, are used as keys;
2. creating a *permission* bit mask by left-shifting 1 `_permission` times,
`1 << _permission`;
3. flipping all of the bits of the *permission* bit mask, `~bit_mask`;
4. evaluating `permissions ^ bit_mask` to produce a new *permissions* bit vector; and finally,
5. updating the state of the `contract` by storing the resulting *permissions* bit vector back into the `subjectResourcePermissions mapping`.

🔑 `modifier restrictTo (owner)`, restricts access to the `function` such that *only* the current owner of the `contract` can call it.

- ➡ `address _subject`, the account `address` of the *subject* whose *permissions* are to be modified.

- ➡ `bytes4 _resource`, the *resource*, `function` signature, which the *permissions* are to be modified for.
- ➡ `uint8 _permission`, a `uint8` value representing the *permission-bit* to revoke for the $\langle \textit{subject}, \textit{resource} \rangle$ pair.
- ➡ `bool _success`, resolves to `true` if the operation was successful, otherwise `false`.

A.1.3.3 Contract ACLAuthority

The `contract`, `contract ACLAuthority`, merges the ACL functionality introduced by `interface AccessControllist` with the generalized access control functionality introduced by `interface Authority`.

</> contract ACLAuthority
Listing A.9

```

1 contract ACLAuthority is Owned, Authority, AccessControllist {
2     AccessControllist acl;
3     mapping (bytes4 => uint8) requiredResourcePermission;
4
5     constructor (bool _createACL) {
6         if (_createACL) acl = new BasicACL();
7     }
8
9     function hasPermission (address _subject, bytes4 _resource, uint8
    ↪ _permission) public constant returns (bool _hasPermission) {
10         return acl.hasPermission(_subject, _resource, _permission);
11     }
12
13     function getPermissions (address _subject, bytes4 _resource) public constant
    ↪ returns (uint256 _permissions) {
14         return acl.getPermissions(_subject, _resource);
15     }



```

```

16
17 function grantPermission (address _subject, bytes4 _resource, uint8
    ↪ _permission) public restrictTo(owner) returns (bool _success) {
18     return acl.grantPermission(_subject, _resource, _permission);
19 }
20
21 function revokePermission (address _subject, bytes4 _resource, uint8
    ↪ _permission) public restrictTo(owner) returns (bool _success) {
22     return acl.removePermission(_subject, _resource, _permission);
23 }
24
25 function setPermissions (address _subject, bytes4 _resource, uint256
    ↪ _permissions) public restrictTo(owner) returns (bool _success) {
26     return acl.setPermissions(_subject, _resource, _permissions);
27 }
28
29 function setACL(AccessControlList _acl) public restrictTo(owner) returns
    ↪ (bool _success) {
30     assert(_acl.owner() == address(this));
31     acl = _acl;
32     return true;
33 }
34
35 function setRequiredResourcePermission (bytes4 _resource, uint8 _permission)
    ↪ public restrictTo(owner) returns (bool _success) {
36     requiredResourcePermission[_resource] = _permission;
37     return true;
38 }
39
40 function canCall (address _subject, bytes4 _resource) public constant returns
    ↪ (bool _canCall) {
41     return hasPermission(_subject, _resource,
        ↪ requiredResourcePermission[_sig]);
42 }
43 }

```

Contract State

-  `AccessControlList acl` maintains the **address** of an ACL implementation, a **contract** which implements `interface AccessControlList`.
-  `mapping (bytes4 ⇒ uint8) requiredResourcePermissions` maintains a mapping of **functions**, `bytes4`, to required *permission*, `uint8`.

⌘ Contract Operations

- ⚙ **constructor** (**bool** _createACL), upon creation and initialization of this **contract** the **constructor** can deploy a **contract**, **contract** BasicACL, if the **bool**, **bool** _createACL, is set to **true**.
 - ➡ **bool** _createACL, set to **true** to deploy an ACL implementation, **contract** BasicACL, in addition to this **contract**; otherwise **false**.
- ⚙ **function** setAcl (AccessControllist _acl), updates the **contract** reference which is responsible for managing ACL requests.
 - 🔧 **modifier** restrictTo (owner), restricts access to the **function** such that *only* the current owner of the **contract** can call it.
 - ➡ AccessControllist _acl, the ACL implementation which access control responsibilities are to be delegated to.
 - ➡ **bool** _success, resolves to **true** if the operation was successful, otherwise **false**.
- ⚙ **function** setRequiredResourcePermission (**bytes4** _resource, **uint8** _permission), updates the mapping representing the *permission* required to access some *resource*, **function** signature.
 - 🔧 **modifier** restrictTo (owner), restricts access to the **function** such that *only* the current owner of the **contract** can call it.
 - ➡ **bytes4** _resource, the *resource*, **function** signature, which the *permission* is to be modified for.
 - ➡ **uint8** _permission, a **uint8** value representing the *permission-bit* required for the *resource*.
 - ➡ **bool** _success, resolves to **true** if the operation was successful, otherwise

`false`.

⚙️ `function` `canCall` (`address` `_subject`, `bytes4` `_resource`), evaluates whether a *subject* can access to some *resource* by validating with the ACL implementation that the *subject* has the required *resource permission*.

➡️ `address` `_subject`, the *subject*, Ethereum account, whose *permissions* are being evaluated.

➡️ `bytes4` `_resource`, the *resource*, `contract function`, which the *subject's permissions* are being evaluated against.

↩️ `bool` `_canCall`, resolves to `true` if the *subject*, `address` `_subject`, is permitted to access the *resource*, `bytes4` `_resource`, otherwise `false`.

⚙️ `function` `hasPermission` (`address` `_subject`, `bytes4` `_resource`, `uint8` `_permission`),
`function` `getPermissions` (`address` `_subject`, `bytes4` `_resource`),
`function` `setPermissions` (`address` `_subject`, `bytes4` `_resource`, `uint256` `_permissions`),
`function` `grantPermission` (`address` `_subject`, `bytes4` `_resource`, `uint8` `_permission`),
`function` `revokePermission` (`address` `_subject`, `bytes4` `_resource`, `uint8` `_permission`),
all ACL responsibilities originating from `interface` `AccessControllist` are delegated to the provided ACL implementation stored in `AccessControllist` `acl`.

🔧 `modifier` `restrictTo` (`owner`), restricts access to the `function` such that *only* the current owner of the `contract` can call this `function`; this applies to `functions` `setPermissions`, `grantPermission`, and `revokePermission`.

A.1.4 Registries

Having completed the work of generalizing access control and building access control lists, we now introduce a simplified access control model for managing voter registration during elections.

1. `interface Registry`, which defines the functions for registering voters for an election, and
2. `contract VoterRegistrationAuthority`, which implements and aggregates several `interfaces` and `interface` implementations — namely `interface Registry`, `interface Authority`, and `contract ACLAuthority` — to build a trusted source of registered voters.

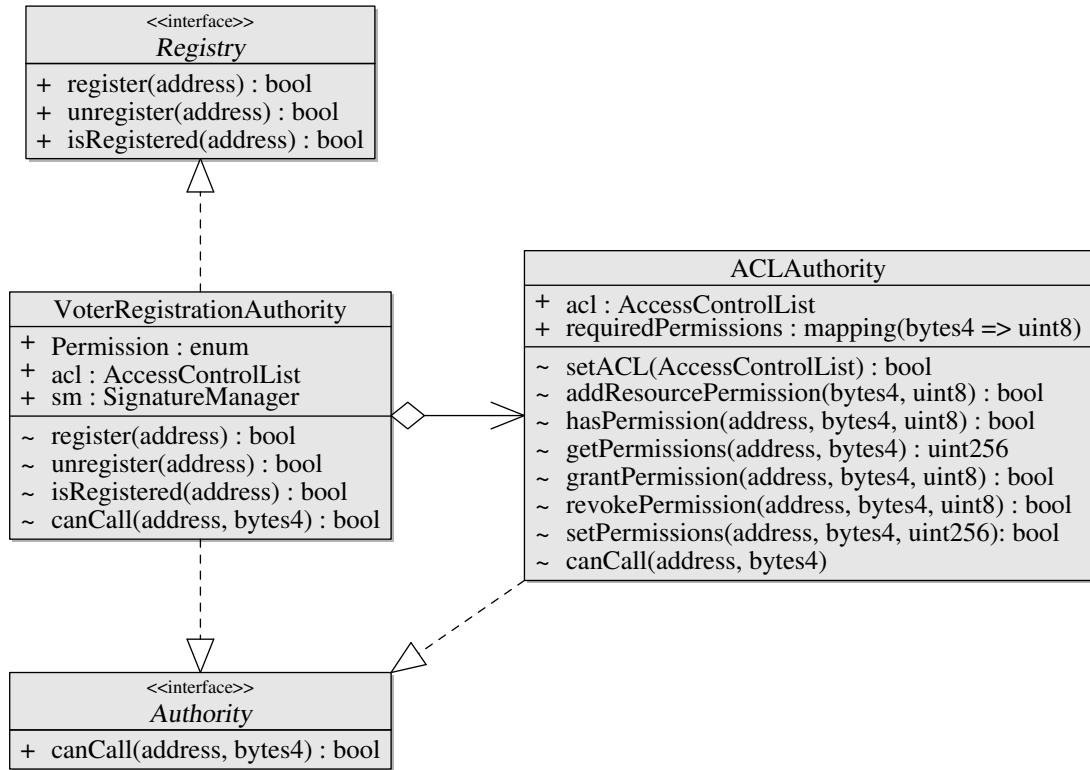


Figure A.5: Simplified election registry.

A.1.4.1 Interface Registry

The **interface**, **interface Registry**, introduces three **function** definitions required to achieve basic registry functionality:

1. **function register**, which *registers* a *subject*.
2. **function unregister**, which *unregisters* a *subject*.
3. **function isRegistered**, which evaluates whether a *subject* is *registered*.

```

1 interface Registry {
2   function register (address _subject) public returns (bool _success);
3   function unregister (address _subject) public returns (bool _success);
4   function isRegistered (address _subject) public constant returns (bool
    ↪ _isRegistered);
5 }

```

i Interface Specification

⚙️ **function** `isRegistered (address _subject)`, evaluates whether some *subject* is *registered*.

➡️ `address _subject`, the **address** of an account representing the *subject*, to evaluate the *registration* of.

⬅️ `bool _isRegistered`, returns the **true** if the *subject* is *registered*, otherwise **false**.

⚙️ **function** `unregister (address _subject)`, *unregisters* a *subject*.

➡️ `address _subject`, the account **address** of the *subject* who is to be *unregistered*.

⬅️ `bool _success`, resolves to **true** if the operation was successful, otherwise **false**.

⚙️ **function** `register (address _subject)`, *registers* a *subject*.

➡️ `address _subject`, the account **address** of the *subject* who is to be *registered*.

⬅️ `bool _success`, resolves to **true** if the operation was successful, otherwise **false**.

A.1.4.2 Contract *VoterRegistrationAuthority*

The **contract**, `contract VoterRegistrationAuthority`, is the final component of our authorization design and is required to construct a generalized voter registration authority capable of managing registered voters and conducting elections.

```
</> contract VoterRegistrationAuthority Listing A.11
1  contract VoterRegistrationAuthority is Owned, Registry, Authority {
2      enum Permissions {
3          Vote
4      }
5
6      AccessControllist acl;
7      mapping (bytes32 => bytes4) resourceSignatures;
8
9      constructor () public {
10         acl = new ACLAuthority(true);
11
12         resourceSignatures['vote'] = bytes4(sha3('vote()));
13
14         acl.setRequiredResourcePermission(
15             resourceSignature('vote'),
16             uint8(Permissions.Vote)
17         );
18     }
19
20     function register (address _voter) public restrictTo(owner) returns (bool
    ↳ _success) {
21         return acl.grantPermission(_voter, resourceSignatures['vote'],
    ↳ uint8(Permissions.Vote));
22     }
23
24     function unregister (address _voter) public restrictTo(owner) returns (bool
    ↳ _success) {
25         return acl.revokePermission(_voter, resourceSignatures['vote'],
    ↳ uint8(Permissions.Vote));
26     }
27
28     function isRegistered (address _voter) public constant returns (bool
    ↳ _isRegistered) {
```

```

29     return acl.hasPermission(_voter, resourceSignatures['vote'],
30                               ↪ uint8(Permissions.Vote));
31 }
32 function canCall (address _subject, bytes4 _resource) public constant returns
33 ↪ (bool _canCall) {
34     return acl.canCall(_subject, _resource);
35 }

```

☰ Contract State

- 🔑 `AccessControllist acl` maintains the **address** of an ACL implementation, a **contract** which implements `interface AccessControllist`.
- 🔑 `mapping (bytes32 ⇒ bytes4) resourceSignatures` maintains a mapping of strings, **bytes32**, to **function** signature, **bytes4**.

⚙️ Contract Operations

- ⚙️ `constructor ()`, upon creation and initialization of this **contract** the **constructor** will deploy a **contract**, `contract BasicACL`.
- ⚙️ `function isRegistered (address _subject)`, evaluates whether some *subject* is *registered*.
 - ➡ `address _subject`, the **address** of an account representing the *subject*, to evaluate the *registration* of.
 - ➡ `bool _isRegistered`, returns the **true** if the *subject* is *registered*, otherwise **false**.

- ⚙️ `function unregister (address _subject)`, *unregisters a subject.*
 - ➡️ `address _subject`, the account `address` of the *subject* who is to be *unregistered*.
 - ⬅️ `bool _success`, resolves to `true` if the operation was successful, otherwise `false`.
- ⚙️ `function register (address _subject)`, *registers a subject.*
 - ➡️ `address _subject`, the account `address` of the *subject* who is to be *registered*.
 - ⬅️ `bool _success`, resolves to `true` if the operation was successful, otherwise `false`.

A.2 Election Components

This section explores the components relating to elections.

A.2.1 Election Contracts

```
</> contract FirstPastThePost Listing A.12  
1  contract FirstPastThePost is Owned, Guard {  
2      uint public creationTime = now;  
3  
4      Choice[] public choices;  
5      mapping(address ⇒ bool) public voted;  
6      Choice public winner;  
7  
8      enum Phase { Configuration, Frozen, Vote, Tally }  
9  
10     struct PhaseProperties {  
11         Phase phase;  
12         uint startTime;  
13         uint endTime;
```

```

14     Phase nextPhase;
15 }
16
17 mapping (Phase => PhaseProperties) phases;
18 Phase public phase = Phase.Configuration;
19
20 constructor () {
21     phases[Phase.Configuration] = PhaseProperties({
22         phase: Phase.Configuration,
23         start_time: now,
24         end_time: 0,
25         next_phase: Phase.Configuration
26     });
27 }
28
29 // This is the current phase.
30 uint public freezeTime;
31
32 // Start and end time for the election.
33 uint public voteStartTime;
34 uint public voteEndTime;
35
36 // This is a type for a single proposal.
37 struct Choice {
38     // Short name (up to 32 bytes).
39     bytes32 name;
40
41     // TODO: choice description?
42     // bytes32 description;
43
44     // Number of accumulated votes.
45     uint40 voteCount;
46 }
47
48 // Restrict calls to _account.
49 modifier restrictTo(address _account) {
50     require(msg.sender == _account);
51     _;
52 }
53
54 // Check current stage.
55 modifier atPhase(Phase _phase) {
56     require(phase == _phase);
57     _;
58 }
59
60 // This modifier goes to the next phase after the function is done.

```

```

61 modifier transitionNextPhase() {
62     _;
63     nextPhase();
64 }
65
66 // Move into next phase.
67 function nextPhase() internal {
68     phase = Phase(uint(phase) + 1);
69 }
70
71 // Perform timed transitions. Be sure to mention this modifier first,
72 // otherwise the guards will not take the new stage into account.
73 modifier timedTransitions() {
74     if (phase == Phase.Frozen && now ≥ voteStartTime)
75         nextPhase();
76     if (phase == Phase.Vote && now ≥ voteEndTime)
77         nextPhase();
78     // The other stages transition by transaction
79     _;
80 }
81
82 /* Election Functions */
83 // Freeze the configuration.
84 function freeze() restrictTo(owner) atPhase(Phase.Configuration)
85     ↪ transitionNextPhase {
86     require(voteStartTime > now);
87     require(choices.length > 1);
88     freezeTime = now;
89     delete owner;
90 }
91
92 // Cast a ballot.
93 function vote(uint8 choice) timedTransitions atPhase(Phase.Vote) {
94     // Each address can only vote once.
95     require(!voted[msg.sender]);
96     voted[msg.sender] = true;
97
98     choices[choice].voteCount += 1;
99 }
100
101 // Tally ballots.
102 function tally() timedTransitions atPhase(Phase.Tally) {
103     winner = choices[0];
104     for (uint8 i = 0; i < choices.length; i++) {
105         if (choices[i].voteCount > winner.voteCount)
106             winner = choices[i];
107     }

```

```

107 }
108
109 /* Configuration Functions */
110 constructor () {}
111
112 function setVoteStartTime(uint _voteStartTime) atPhase(Phase.Configuration)
113 ↪ restrictTo(owner) {
114     voteStartTime = _voteStartTime;
115 }
116
117 function setVoteEndTime(uint _voteEndTime) atPhase(Phase.Configuration)
118 ↪ restrictTo(owner) {
119     voteEndTime = _voteEndTime;
120 }
121
122 function addChoices(bytes32[] _choices) atPhase(Phase.Configuration)
123 ↪ restrictTo(owner) {
124     for (uint i = 0; i < _choices.length; i++) {
125         choices.push(Choice({
126             name: _choices[i],
127             voteCount: 0
128         }));
129     }
130 }
131
132 function addChoice(bytes32 _name) atPhase(Phase.Configuration)
133 ↪ restrictTo(owner) {
134     choices.push(Choice({
135         name: _name,
136         voteCount: 0
137     }));
138 }
139 }

```

A.3 Delegation Components

This section explores the components relating to vote delegation. The delegation components are responsible for managing the delegation of voters' votes to delegates.

A.3.1 Delegation Contracts

</> Vote Delegation

Listing A.13

```
1 mapping (address ⇒ Voter) voters;
2
3 function delegateVote (address delegate) public auth returns (bool _success) {
4     // The `auth` modifier prevents this function from being
5     // called until the Authority has confirmed that the
6     // the message sender has the proper privileges.
7     Voter cursor;
8     uint40 weight = voters[msg.sender].weight;
9
10    // Cycle Detection
11    mapping (address ⇒ bool) visited;
12    visited[msg.sender] = true;
13    visited[delegate] = true;
14
15    cursor = voters[delegate];
16    while (cursor.delegate) {
17        address newDelegate = cursor.delegate;
18        if (visited[newDelegate]) return false;
19        cursor = voters[newDelegate];
20        visited[newDelegate] = true;
21    }
22
23    // Decrement weights of old delegate chain.
24    cursor = voters[msg.sender];
25    while (cursor.delegate) {
26        address newDelegate = cursor.delegate;
27        cursor = voters[newDelegate];
28        cursor.weight -= weight;
29    }
30
31    // Increment weights of new delegate chain.
32    cursor = voters[msg.sender];
33    cursor.delegate = delegate;
34    while (cursor.delegate) {
35        address newDelegate = cursor.delegate;
36        cursor = voters[newDelegate];
37        cursor.weight += weight;
38    }
39
40    return true;
41 }
```

Vita

Nathaniel Patrick Hernandez — born in Memphis, Tennessee, to Alison and Juan Hernandez — developed a passion for technology and software at a young age and has been passionately writing software and exploring the field of computer science ever since. In 2012 Nathan earned his Associate of Science from Coastal Carolina Community College. The next year, with barely the wherewithal to attend a single semester, he began his Bachelor of Science at Appalachian State University. Nathan was fortunate to have been noticed by the faculty of The Department of Computer Science there and presented with opportunities to work with faculty members in a variety of ways.

Dr. Rahman Tashakkori in particular played a significant role in Nathan's undergraduate degree, academic career, and personal life — investing significant amounts of time and energy which came in the shape of mentorship, engaging academic problems, personal challenges, and friendship — helping Nathan to grow personally, academically, and eventually providing the encouragement to pursue a Master's degree. Throughout Nathan's academic career Dr. Tashakkori presented

him with scholarship and research opportunities; allowing him, along with a team of other hand-picked students, to conduct research, build software and hardware systems, publish results, and present findings on the beehive monitoring and analytics systems they built together. This research served as the foundation for Nathan's undergraduate thesis which enabled him to earn his Bachelor of Science.

Outside of academia, Nathan spent just under a year interning with IBM's jStart team, a small group within IBM's Emerging Technologies department responsible for prototyping systems for clients which leveraged new and promising technologies. Nathan spent a majority of his time there immersing himself in the field of big-data analytics: exploring Apache Kafka, analyzing client data, generating insights, and building real-time data-processing pipelines. Having a strong interest in cybersecurity and cryptography, Nathan also spent a summer in San Francisco, California, interning at UnifyID, a startup whose goal is to build advanced security, authentication, and password management solutions. While there Nathan worked on building important components of UnifyID's Google Chrome extension for password management and participated in launching their private beta. A few months after the end of his internship UnifyID went on to win runner-ups in the 2016 TechCrunch Disrupt Battlefield.