BlockVote: A Case Study in Practical Security Of Smart Contract Applications for a Use-Case Where Security is Paramount.

Timothy Muscat -
40341959

Submitted in partial fulfilment of
the requirements of Edinburgh Napier University
for the Degree of
MSc. Advanced Security & Digital Forensics

School of Computing

November 13, 2020

Authorship Declaration

I, Timothy Muscat, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained informed consent from all people I have involved in the work in this dissertation following the School's ethical guidelines.

Signed:

Date: 14/11/2020

Matriculation no: 40341959

General Data Protection Regulation Declaration

Under the General Data Protection Regulation (GDPR) (EU) 2016/679, the University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below one of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

The purpose of this project was to implement a Single Transferrable Vote (STV) system on the Ethereum blockchain using Smart Contracts written in Solidity. The STV system implemented was modelled on that used by the Republic Of Malta for national as well as European Parliamentary elections. Considerable focus was placed on security of the distributed application; security being paramount to a hypothetical decentralised voting system in production. Also investigated was the feasability of deploying to the public Ethereum network rather than a consortium (private) blockchain. The functionality of the system having been verified using unit tests, the smart contract code was then subject to security analysis using both manual and automated methods and found to be satisfactorily secure versus known attack vectors used against smart contracts. Evaluated also were possible attack vectors that could lead to compromise of the system through subverting the blockchain network itself.

Whilst the system was deemed to meet functional requirements and adjudicated to be secure, the smart contracts were deemed to be unfeasible to run on the public Ethereum network owing to the excessively high Ether cost. Nevertheless, it is believed to serve as an adequate proof of concept that may be used as the foundation for a production-grade STV system in future work.

Contents

List of Figures

1 Introduction, Motivation & Theory

1.1 Aims & Research Questions

This section shall seek to describe and provide motivation for the development of the final deliverable software artifact as well as justify the use of blockchain technology in its implementation.

The scope of this dissertation being a case study on secure smart contract development practises, the particular case of a distributed voting system for national elections was chosen as it is a use-case for which absolutely no vulnerabilities or bugs are acceptable. This is both due to the catastrophic consequences such a failure would have, as well as the fact that it is sure to be the target of attacks by politically motivated and/or state-sponsored hackers as well as malicious actors merely looking to cause mischief.

Whilst the concept of online, and even specifically blockchain-based voting systems for democratic elections is hardly a new idea, and code for several proof-of-concept applications already exists, this application shall be tailored to the rather particular voting system used in the Republic of Malta. The system is based on a Single Transferable Vote (STV) system with several caveats, for which no existing code is sufficient. Whilst the focus shall be on the security of the underlying smart contract application on the Ethereum blockchain (the "back-end"), care shall be also given to the security of the dApp [1] as a whole, ensuring that no attacks by means of exploiting the front-end or any "out-of-band" attack vector that would compromise the criteria that a democratic voting system must meet; namely (Hjálmarsson, Hreiarsson, Hamdaqa, & Hjálmtỳsson, 2018):

- An election system should not enable coerced voting.
- An election system should allow a method of secure authentication via an identity verification service.
- An election system should not allow traceability from votes to respective voters.
- An election system should provide transparency, in the form of a verifyable assurance to each voter that their vote was counted, correctly, and without risking the voter's privacy.
- An election system should prevent any third party from tampering with any vote.

[1] dApp: Distributed Application

- An election system should not afford any single entity control over tallying votes and determining the result of an election.
- An election system should only allow eligible individuals to vote in an election

Concisely, the main research questions are:

- What current blockchain voting solutions already exist ? What are their drawbacks? How can we improve on them ?
- What are the main security issues pertaining to online voting systems and how can blockchain remedy them ?
- What relevant advantages does Ethereum offer over other blockchain technologies and what makes it more appropriate for this particular application ?
- How can blockchain help an online voting solution meet the criteria that a free and fair election system must meet ? (*see above*)
- What are the known security issues affecting blockchain applications and how can they be addressed in this application ?
- What are the implications of deploying the solution to the public Ethereum blockchain vs. a private one ?
- In the case of deployment to the public network; what are the projected financial costs of running a real-world election using it and would it be economically feasible ?

1.2 Suitability Of Blockchain For Votation Systems

Despite the enormous leaps in technology over the past few decades, voting in real-world elections for public office remains largely done through in-person voting via paper ballots filled in with pen or pencil and placed in receptacles. Whilst voting by post has been adopted in certain locations in a limited fashion, largely for "absentee" ballots (ie: those who for one reason or another are unable to vote in person), this remains a paper-based, manually counted voting system. Resistance to moving to a fully-electronic, online voting system, aside from inertia, is largely based on lack of confidence in the integrity of such a voting system and concerns about security flaws leading to possible sabotage of elections (Kobie, 2015). The problem of having an online remote voting system based around a centralised infrastructure, that is, a typical

web application based around the monolithic structure of a single database and a back-end server, is that voter confidence and transparency are not guaranteed and problematic to ensure. Voter confidence essentially would be predicated on the public's trust in the persons who build, maintain and run the system. Questions about their impartiality and accusations of tampering both from the inside as well as external attackers may arise. Furthermore, a centralised e-voting structure would create a single point of failure to be targeted by DDoS, exploitation or even physical vandalism. Given recent controversies about alleged meddling by foreign actors in elections and more recently, allegations of vote tampering in the most recent (2020) presidential election in the United States [2] (Committee et al., 2019), this is an especially pertinent issue. A study (Norden & Famighetti, 2015) by New York University School of Law discovered that the electronic voting machines used at many polling stations in the USA are highly vulnerable. Clearly there is still a long way to go before electronic voting becomes trustworthy. The suitability of blockchain technology to this application is in its ability to allow mutually mistrusting third-parties with potentially conflicting interests, in this case voters (who, naturally, would like to see their preferred candidate/s elected and not others) to perform transactions and/or transmit, store and modify information without the need for a trusted third party, whilst also allowing all pertinent data to be examined and audited.

> *"... using [...] blockchain only makes sense when multiple mutually mistrusting entities want to interact and change the state of a system, and are not willing to agree on an online trusted third party" (Wüst & Gervais, 2018)*

[2] The author wishes to clarify that they make no assertions, implied or otherwise, as to whether these allegations are true or not.

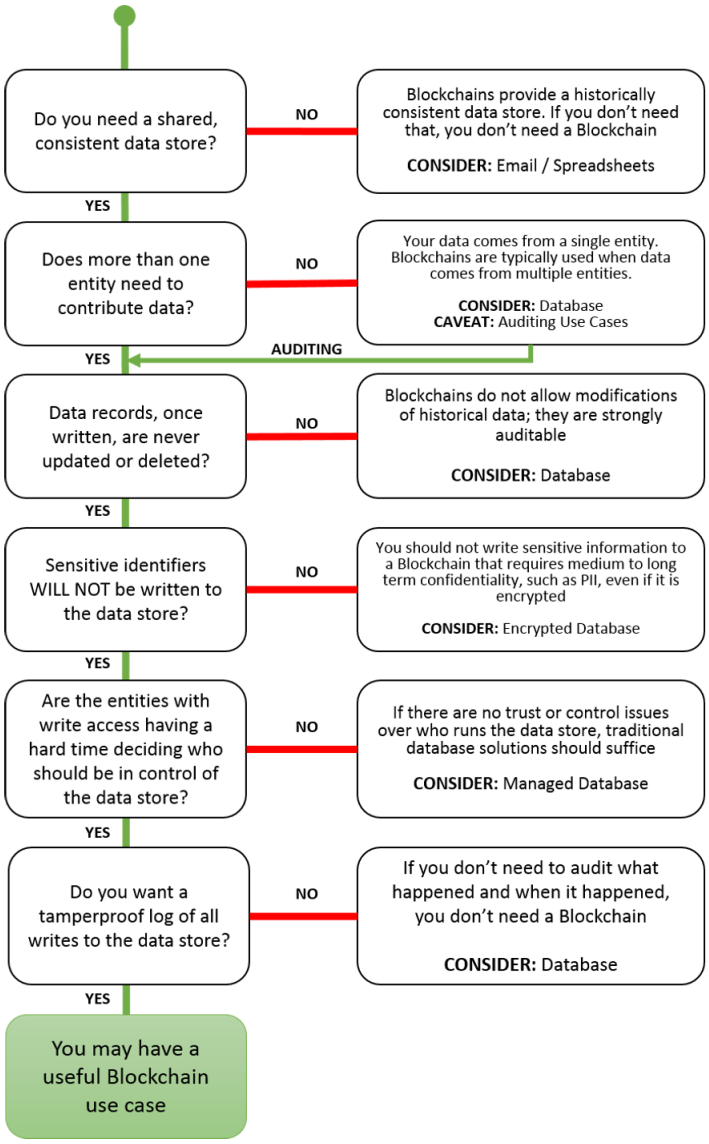

Figure 1: Is this a Blockchain Use-Case ?
(Yaga, Mell, Roby, & Scarfone, 2019)

Going through the above figure 1 from NIST for our particular use-case:

1. **Do you need a shared, consistent data store ?** Vote tallies must be accessible and (within obvious limits) modifiable by participants in the process. The store must necessarily be consistent and, after closing of votation, immutable.

2. **Does more than one entity need to contribute data ?**. All eligible voters must be able to "contribute data", ie: vote. Each of these entities must not be privileged over others.

3. **Data Records once written, are never updated or deleted ?** Votes that are cast must not be alterable by anybody.

4. **Sensitive Identifiers will not be written to a data store.** The secret ballot is a fundamental feature of any modern voting system. All data on the blockchain is publicly accessible, therefore care must be taken that no identifying information pertaining to voters is recorded tying them to their voting selection.

5. **Are the entities with write access having a hard time deciding who should be in control of the data store ?** No entity or entities should have unilateral control over the stored voting data.

6. **Do you want a tamperproof log of all writes to the data store ?** The ability to audit the number of votes cast and by whom to ensure that the election is not only sound but can be readily shown to be sound, is important to ensure public trust in the process.

By the above criteria, it is clear that this is indeed a valid blockchain use-case. We shall now examine potential drawbacks and challenges introduced by the use of blockchain for our application.

1.2.1 Challenges of Electronic Voting Solutions.

Transparency and privacy are both integral to any hypothetical e-voting system, however guaranteeing one without compromising the other can prove problematic. In a conventional centralised system, privacy is easily ensured, while transparency and public verifiability may be less easy to ensure. Transparency and public verifiability are to an extent built into the operation of blockchain-based systems, with transactions on public blockchains such as Ethereum and Bitcoin being viewable by anyone, and the data stored inside Ethereum smart contracts is exposed (even if the variable is marked as *private* in the smart contract code). Ensuring privacy, and thus, the secrecy of voting could easiest be achieved through engineering the application in such a way as to avoid storing any identifying information on the blockchain which could be used to tie a voter to their voting choice, whilst also ensuring the validity of all votes cast. This shall be one of the principle design challenges in developing the application that this project concerns. A previous case

study (Kaspersky, 2016) carried out on the viability of blockchain voting highlighted several problems that would need to be overcome; namely:

- Authentication of users is daunting.
- Voter's personal computer could be compromised
- Denial of Service attacks could be used to prevent voting.

This and other studies have thus suggested blockchain voting systems consist of a traditional physical polling station with voting machines or paper ballots, with the results being recorded on a private blockchain. Attacks on the blockchain itself are of course a possible attack vector; with a potential attacker possibly using the well-known 51% attack whereby they would take control of over 50% of the processing power in the network in order to forge a longer blockchain with their own version of transaction records. Mitigations for this problem could involve use of a private blockchain as used by the VoteWatcher(*VoteWatcher - The World's Most Transparent Voting Machine*, n.d.) project or use of a permissioned blockchain as proposed by the VoteBook(Kirby, Masi, & Maymi, 2016) project whereby the only nodes allowed on the chain would be voting stations (though these would require an actual physical polling station rather than actual online voting). In our implementation, it is expected that the use of a pre-existing large and well-established blockchain, Ethereum, should render 51% attacks, as well as attacks on the general integrity of the blockchain unfeasible.

2 Background & Theory

2.1 General Overview Of Blockchain

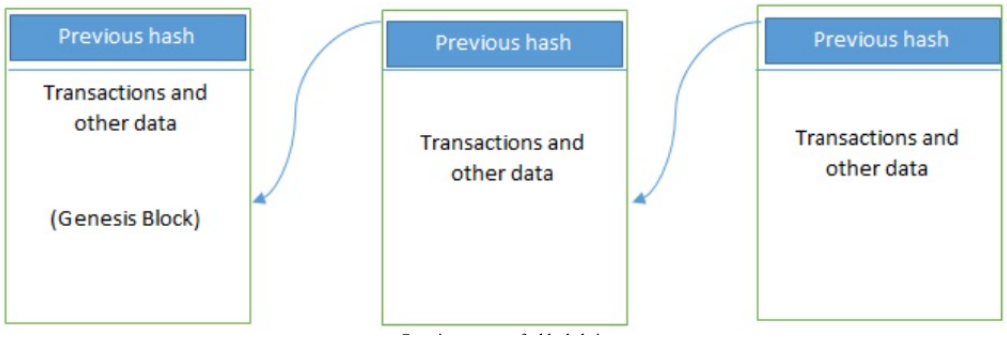

Figure 2: Generic Structure Of A Blockchain

(Bashir, 2017)

A blockchain network, is composed of a peer-to-peer network of nodes where each node has equal status and no central controlling nodes exists. This peer-to-peer network is used to implement a distributed ledger, meaning simply a ledger where each node has its own copy, and transactions are performed by a node through broadcasting the transaction to all the different nodes, which then check the transaction's validity (that is, basically, that the sender of the transaction has enough funds to perform this transaction). The "chain" portion of blockchain is in that transactions are grouped into collections known as blocks, with these blocks being linked to each other through each containing a hash of the contents of teahe previous block. A subset of the nodes in the network act as "miners" or "verifiers" whose job it is to check transactions for validity, collect them into a new block and append it to the chain. Every node keeps a copy of the entire blockchain. An imporant feature of a blockchain as a ledger is that it is strictly append only; new blocks can be added but previous blocks cannot be modified or removed. The immutability of the chain is enforced through the use of a proof-of-work function; in order for a miner to create a new block, it must add a random "nonce value" to the contents of the new block and compute the hash, repeating this process with a different nonce value until the resultant hash meets certain criteria determined by the network, such as the hash ending in a certain number of 0s. This process is referred to as "mining", and the miner node that is

first to compute this hash function and create a valid new block receives compensation in cryptocurrency. The miner then broadcasts a copy of the blockchain with the new block they have just mined to the other nodes, that will check the validity of the transactions and choose to accept it accordingly. This way, a node that attempted to alter a past transaction by tampering with a block would have to recompute new hashes for each of the subsequent blocks in the chain and then broadcast it to the network hoping it is accepted. When 2 or more differing versions of the chain are broadcast, the network will by default choose the longest one as the valid chain. Thus, the only way for a malicious node to tamper with the chain would be by having more computing power than the rest of the network, and thus being able to mine blocks faster and compute a longer chain. This "proof-of-work" algorithm is known as a consensus algorithm; that is, it is the means by which the network is able to reach consensus on what transactions to append to the distributed ledger (the chain) in the absence of a centralised authority. Other consensus algorithms exist, such as *proof-of-stake* and *proof-of-authority*. At the time of writing, proof-of-work is still the dominant consensus algorithm for public blockchain networks.

Generically, a blockchain network consists of the following fundamental components: (Bashir, 2017)

- **Addresses** An address is a unique identifier used to denote senders and recipients on the network. This is usually a public key, or a value derived from one.
- **Transaction** A transfer of value from one address to another.
- **Block** A collection of transactions along with metadata such as the hash of the previous block in the chain, timestamp, and the aforementioned nonce value.
- **Peer-To-Peer (p2p) Network** a network in which all nodes have the same importance without a central authority and can communicate with each other.
- **State Machine** A blockchain can be envisioned as a state transaction function whereby a state is modified from its initial form to a next one as the result of a transaction.
- **Node** Participants in the p2p network. Nodes can be simple participants that send and receive transactions or they may be "mining" nodes that validate transactions and create new blocks.

The blue print for the first blockchain network was defined by Satashi Nakamoto (an alias) in the Bitcoin whitepaper (Nakamoto, 2019), with Bitcoin, the first cryptocurrency facilitated by blockchain being launched sometime later. Since then innumerable other blockchains, both public and private have been launched. A private blockchain being one where the number of nodes in the network is restricted, usually within a certain organisation. Most of these blockchains implement some sort of cryptocurrency, but blockchain technology is also being used for alternate applications such as tracking the ownership of real-world assets, patents and trademarks and distributed execution of code. This latter application is of greatest concern for this project.

2.2 Overview Of Ethereum & Tokens

(Buterin, Wood, et al., 2014)(Wood, 2014)t

2.2.1 Terminology And Components of The Ethereum Network

- **Ether.** The currency of the Ethereum network. It is traded as a normal cryptocurrency like Bitcoin but its main purpose is to allow for payment for execution of smart contract code.
- **Miners** Mining nodes, aside from verifying transactions via solving a proof-of-work hash puzzle in order to mine a block as in Bitcoin, in Ethereum mining nodes also execute smart contract code called by users or other contracts, receiving Ether as compensation.
- **EVM** The Ethereum Virtual Machine. A stack-based virtual machine that executes programs written in the Ethereum bytecode language.
- **Smart Contract** A program written in a language that compiles down to EVM bytecode to be executed by nodes on the Ethereum network. A smart contract program, once launched on the network, is immutable, executes independently and cannot be stopped by anyone unless the contract code explicitly allows it to be under certain conditions. Thus the code, even if it does turn out to have bugs, cannot be modified. This is effectively analogous to a semantic loophole in the wording of a traditional contract that may be exploited in ways not originally expected or intended when the contract was drawn up.
- **Solidity** The first and most common language used for coding smart contracts on Ethereum. Is object oriented and statically typed. Compiles down to EVM bytecode. Other languages exist for writing smart

contracts but at the time of writing do not yet have very large market share.

- **Instance** A particular copy of a smart contract deployed to the Ethereum network.
- **Contract Account** An account on the Ethereum network that identifies an instance of smart contract deployed to the blockchain.
- **Externally Owned Account (EAO)** An account not belonging to a smart contract.
- **Gas** A quantity "purchased" for a specified price in Ether when calling a contract function. Units of gas are consumed by computational steps in the contract bytecode, with certain operations costing more than others (Wood, 2014), with Ether being transferred to the account of the mining node that executes them according to the price in Ether per gas unit. Specifying a higher gas price in Ether will result in the transaction being picked up by a mining node faster whereas a gas price too low may result in the transaction never being picked up at all.
- **Off-chain** In the context of blockchain apps, used to describe an application or activity performed using a traditional program. That is; not using a smart contract program on the blockchain.
- **OpenZeppelin** Not a part of Ethereum, but is an integral part of most practical smart contract development. A library of pre-tested and verified secure Ethereum contracts that can be used by programmers as components or templates (via inheritance) to create their own smart contract applications.
- **Truffle** A suite of applications used to streamline development of smart contract applications, includes (among many other things) a compiler, the ability to run unit tests and boiler plate code to allow easy deployment of contracts.
- **dApp** Short for Distributed application. A dApp is distinct from a smart contract in that a dApp encompasses every component of the application including the off-chain front end that talks to the backend on the blockchain which may be composed of multiple smart contracts.

A complete description of Ethereum and its inner workings is obviously out of scope here. However this section was written to provide a concise summary of the ways in which Ethereum facilitates the execution of distributed

programs in order to provide motivation and justification for its use in this application. The notion of using the underlying blockchain technology for applications other than implementing a currency existed long before Ethereum, with Namecoin, a decentralised database for registration of domain names and identities (among other things) being forked from Bitcoin all the way back in 2010 (Loibl & Naab, 2014) and an idea of a distributed database to store ownership of property titles having been proposed in 1998 (Szabo, 1998), long before the technology to facilitate it existed. In fact, Bitcoin itself does facilitate a weak version of a kind of "proto-" smart contract through its in built scripting language. This language however is insufficient for writing smart contract functionality as it is not considered Turing complete and has other limitations such as not being able to access blockchain metadata data such as the block timestamp.
Ethereum was thus created based on a similar concept to Bitcoin though also significantly different in its implementation in order to facilitate the writing of smart contracts. The main crucial difference between Ethereum and Bitcoin, is the inclusion of a true Turing-complete language programming language allowing programmers to create arbitrary programs that will be executed on the blockchain network by miners. Just as consensus on transactions is achieved through verification by multiple nodes, so is consensus on the execution state of smart contract programs. The design philosophy of Ethereum is based around a number of principles, namely:

- *Simplicity.* The Ethereum protocol is to be as simple as possible, even at the cost of inefficiencies in time or data storage.
- *Universality.* As part of Ethereum's design philosophy, it does not include features to perform particular functions. Ethereum just provides the scripting language and the platform to execute it, with the possibilities being limited only by the programmer's ambition and the Ether needed to run the contracts they code.
- *Modularity.* The components that make up the Ethereum protocol are designed to be modular and independent from each other.
- *Agility.* Details of the implementation of the Ethereum protocol are not set in stone and the implementation of such features as the virtual machine executing the code and the architecture of the protocol as a whole are subject to change.
- *Non-Censorship.* Ethereum does not restrict or prohibit specific uses of the network.

A blockchain typically includes a notion of "state". This simply means the current status quo of the network, in a distributed ledger such as Bitcoin this would be simply which accounts own which units of the cryptocurrency. A blockchain would also define a state transition function by which the state would be changed, taking the current state and a group of transactions as parameters. The function itself is more of an abstraction rather than an actual piece of code and the details of said function will naturally vary from one chain implementation to another (Figure 3).

$$State' = StateTransition(State, Transactions)$$

Figure 3: State Transition Function
(Nakamoto, 2019)

In Ethereum the state is made up of accounts which may belong to a smart contract or else be what is known as an *Externally Owned Account (EAO)*.[3] Each account has a 160-bit address, an Ether balance and a counter variable called a "nonce" which increases with each transaction sent by the account as well as an array of contract bytecode and storage if the account belongs to a smart contract. That an account has a field to track its Ether balance may seem obvious, but is in fact in stark contrast to Bitcoin and many other blockchains which track user ownership through keeping track of "unspent transaction outputs", called UTXOs in a distributed database[4] (Nakamoto, 2019). Transactions in Ethereum are data structures consisting of:

- The Recipient Address
- A signature identifying the sender.
- The amount of Ether to transfer.
- *The fields above are present in all transaction, those below are present only in calls to contract functions, which are also considered transactions in Ethereum.*
- Data field (may be empty)
- a gasLimit or "startgas" value indicating the maximum amount of gas the caller of a contract function is willing to spend on the transaction.

[3] See terminology subsection above.
[4] A description of how this works is not in scope as it does not pertain to Ethereum.

- Gas Price. The amount of Ether the caller is willing to spend per unit of gas. If the product of the Gas price and the Gas limit results in an Ether amount higher than the sender's Ether balance, the transaction will fail.

Ethereum also possesses the concept of "Messages" with the same structure. Messages are distinct from transactions only in that they are sent from one contract to another rather than from an EAO.

2.2.2 Tokens

Typically a token will simply be a small contract that will include a ledger in the form of a hash map to map token holders' account address' to their token balance, and functions to facilitate transferring tokens from one holder to another, creating new tokens or destroying existing ones. The concept of tokens in fact predates Ethereum; the notion of "Colored" coins having been proposed for the Bitcoin network a number of years before the Ethereum network even entered development (Rosenfeld, 2012). The notion of "colored" tokens involved attaching metadata to UTXOs on Bitcoin or similar networks to denote their "color", or rather which token they formed part of. This way wallet software would be able to track ownership of such UTXOs distinct from normal Bitcoin UTXOs. Such "hacky" methods are not needed on Ethereum as tokens are implemented as smart contracts. All these operations are done by simply updating the relevant balances in the ledger inside the contract. Tokens in actual fact are nothing but ledgers implemented in a contract and are not a special feature of Ethereum. Recall that the Ethereum philosophy eschews building features directly into the network and instead opts to merely create a robust framework with which users can implement whatever functionality they like. Typically, when developing a new token, developers will use a pre-existing design pattern providing all the basic functionality such as ERC20 *(See literature review 3.4.2)* and then add and remove functionality as they see fit. OpenZeppelin makes this easier by providing inheritable and pre-audited implementations of popular standards such as ERC20 and ERC721, so developers need only concern themselves with the features particular to their own token and not waste time "reinventing the wheel". Tokens in Ethereum typically represent some real world asset, though they could also be used as their own parallel currency. These assets would typically be shares in a company, voting rights in an organisation, pre-order for pre-release sales of some asset or could even be used to track ownership of property.

2.3 Brief Overview of Maltese Electoral System

In order to understand the motivation behind the structure of the application, a rudimentary understanding of the Maltese electoral system is necessary. Malta is a parliamentary democracy based on a unicameral version of the Westminster system, and is divided into 13 electoral districts, each from which 5 parliamentary candidates are elected. Voting is done through Single Transferable Voting (STV), an uncommon system used only in Malta and the Republic of Ireland, whereby a voter marks candidates by order of preference (*1,2,3,4....n*). In this system, candidates must receive a quota of cast votes on the district in which they are running in order to be awarded one of the 5 seats chosen out of the district. The quota of votes is a function of the number of votes actually cast on that district and the number of seats. The formula being: $(votes/(seats + 1)) + 1$. Votes are counted on a round-based system: In the first round of voting, 1st preference votes are counted and allocated to the candidate's vote count. At the end of each round of counting, if no candidate has reached the quota needed to be elected in that round, the candidate with the lowest number of total votes is eliminated, and their votes transferred to the candidate of next preference. If, by the time a given vote has been counted in the nth round of counting, the candidate selected has either already reached the quota to be elected or been eliminated, then the vote is passed on the next round of voting, with the vote going to the candidate selected by the voter as their n + 1 th preference. (*How Malta Votes: An Overview*, n.d.). This is a basic STV voting system; there are however a few caveats particular to Maltese general elections that should be taken into account:

1. *Proportionality Clause*. As a result of a 1981 constitutional amendment, the party that ultimately wins the election, regardless of the parliamentary seats allocated through the basic STV process, is the party receiving the most 1st count votes. Furthermore, the proportion of seats in parliament allocated to a given party must be equal to their proportion of total 1st count votes obtained by their candidates. Through this mechanism therefore, a party may be allocated more seats whether they won or not. These extra seats are then filled by runner-up canidates; the candidates from that party that got the most votes out of those that did not receive enough votes to be elected during the basic STV stage of the election process.

2. Candidates can, and often do, run on 2 districts rather than just 1. Candidates that obtain the quota on both of the districts they are contesting give up one of them, with the "extra seat" then being filled

through a casual election.

3. *Casual Elections* When a candidate wins on two districts, the seat they vacate is filled by recounting the votes the canidate ultimately received, and passing them to the next preference candidate on each that did NOT receive the votes necessary to be elected, ignoring candidates that have already been elected. The candidate receiving the most votes is elected to the vacated seat.

3 Literature Review

3.1 Introduction

The contents of this review shall be split into 2 main sections, each treating a different facet of smart contract application security. The first shall treat proper coding practices in the writing of smart contracts to prevent exploitation of logical loopholes in the code, past exploited vulnerabilities as well as methods by which to verify the security of blockchain applications. The second shall be a comparative study of ERC20 tokens in contrast with other token models. The purpose of investigating these areas shall be to establish the most appropriate token interface to base our own token on, and to establish a checklist of known vulnerable code patterns to avoid and secure coding practices to use, as well as analysis tools and techniques to use in order to evaluate the security of my own contract code. This shall not exclude the possibility of:

- Deciding that no existing token standard is optimal in its "out-of-the-box" form for our purposes, and developing our own based on existing standards or even from scratch.
- The analysis of our own code through nuanced manual analysis and adversarial testing ("pen-testing") in order to discover previously undocumented vulnerabilities or attack vectors unique to my particular application

3.2 Exploits & Mitigation

Given that smart contracts are merely programs distributed on the Ethereum blockchain and executed by mining nodes, they must be exhaustively debugged and tested to make sure they are free of logical errors that could be exploited by an attacker. This especially in light of the fact that smart contracts are immutable and irreversible; that is, once they are on the blockchain they can neither be rescinded nor modified. Presented here are some common identified potential bugs in smart contracts and possible solutions.

3.2.1 Transaction-Ordering Dependence

Given two or more transactions sent to a smart contract at roughly the same time where one or more of the transactions results in the state of the contract changing, a race condition prevails. If a contract specifies an Ether reward for a correct answer to some computational puzzle, the owner of the contract

may listen for a correct answer sent to the contract, and then send his own transaction instructing the contract to lower the reward to 0. If they are both included in the next block, the owner's transaction may be executed first, thus they get out of paying the reward. A potential solution may be adding a condition that miners must process transactions to a contract marked as having a mutable execution state in the same block in the order they were sent, however this would not stop the case where the malicious actor assigns a higher "Gas Price" to his own transaction so it is included in an earlier block and thus processed first. A solution could be including a "Guard" condition with transactions sent to the contract; in the example above for example, the answer could be sent as a tuple *(Answer, [Reward== x])*, whereby the second element is a list of conditions. The contract will only return the reward if and only if the conditions evaluate to true at the time of execution. (Luu, Chu, Olickel, Saxena, & Hobor, 2016)

3.2.2 Timestamp Dependence

Some contracts may use the timestamp of the block they are run in as input to a function. Miners typically are able to alter the timestamp of a block within certain margins and still have it be accepted. If the contract had a pseudorandom function giving a financial reward to certain wallets with the block timestamp being used as a seed, and the miner knew the nature of the pseudorandom function, they could alter the value of the timestamp to bias the results towards wallet addresses controlled by them. A mitigation could be to simply discourage such functions using user-mutable data such as this, with the programmer merely accepting the consequences of poor programming practices. Other measures may include narrowing the timestamp margin from that of the last block that a new block may have and still be eligible to be accepted which would decrease (but not eliminate) the potential for such manipulation. (Luu et al., 2016)

3.2.3 Exception Mishandling

Failing to check the results of a called function is poor practice hardly unique to blockchain programming. A contract may call another contract, which may throw an exception if this contract say, has run out of gas or attempts to enact a transaction from an address with insufficient balance. Thus, a contract that may transfer ownership of some asset after sending Ether to a wallet address, may end up having transferred the asset without paying the receiver of funds if for some reason the transaction does not complete successfully and the contract does not check before it transfers the asset. The

only possible mitigation here is merely to enforce and encourage good programming standards; the scripting language used to write a smart contract may be simply made to not allow uncaught exceptions to avoid this. (Luu et al., 2016)

3.2.4 Re-entrant Code

When an Ethereum contract calls another contract, the calling contract's execution is suspended until the call completes (as in most programming environments unless the called function is specifically marked as asynchronous). If a contract were to call another contract to send a withdrawal to a user by sending the amount via calling a contract representing the user's account, the user could modify their own contract to call withdraw on the calling contract again recursively before the calling contract has been able to update the user's balance, allowing them to withdraw more than they have in their balance. A similar bug was exploited to steal over 50 million USD worth of Ether in the DAO (Decentralised Autonomous Organisation) hack. A remedy here would be to use some kind of mutex lock inside a static variable which would be set when the contract makes a function call with potential for reentrant code exploitation. If this lock is set, the function would be written to return before it can execute any code. (Dika & Nowostawski, 2018) suggests use of the check-effects-interactions design pattern in order to counter similar vulnerabilities. (Luu et al., 2016)

3.2.5 tx.origin

The identifier tx.origin in the Solidity programming language denotes the identity of the user that started a chain of interactions between smart contracts. This however, is unsuitable for use as authentication as an attacker can easily spoof it. (Dika & Nowostawski, 2018)

3.2.6 Call-stack Depth Exception/"Overflow"

The Solidity Virtual Machine has a maximum call stack depth of 1024 frames, after which further functions called will throw an exception. A malicious actor may manipulate a blockchain application to exhaust the available stack frames as a Denial of Service attack. Adequate exception handling for such an event should therefore be implemented. (Dika & Nowostawski, 2018)

3.2.7 Unchecked-send bug

"Unchecked-send bug" in the context of the Solidity programming language refers to the behaviour of the .send() function used to send a quantity of Ether present in every "address" object, representing a user on the Ethereum blockchain. If this function fails, it does not throw an exception but rather simply returns a boolean value of "false". If the correct return value (true) is not checked for, and the contract code continues executing in the assumption that the send function succeeded, it could result in the recipient losing the Ether they were meant to receive. Similar bugs could result from calling functions on external contracts that never return because the external contract has run out of gas. (Dika & Nowostawski, 2018)

3.2.8 External Calls and Denial of Service

"External calls" refers broadly to execution of any function on another contract. Since attackers could potentially execute malicious code on an external contract, it is important to check the return values of functions executed on an external contract function. When a conditional statement such as an if, while or for condition within a contract depends on the result of a function called on another contract, and the callee permanently either fails or does not return at all, the calling contract will be prevented from completing its execution. A malicious attacker may also deliberately cause denial of service by supplying large amounts of data that is very expensive to process (in terms of Gas). This can be mitigated by writing code to avoid looping behaviour. (Dika & Nowostawski, 2018)

3.2.9 Improper Visibility Modifiers

Whilst not a security vulnerability particular to smart contracts or Solidity, such a vulnerability has led to one of the more serious hacks of smart contracts on the Ethereum blockchain. "Parity" is an open-source smart contract used to facilitate a multi-sig Ether wallet (Multi-sig meaning, that the private keys of multiple users are required to disburse the funds from the wallet). A grievous, (and rather basic) vulnerability was prevalent in the code whereby the function (shown below) used to set the owners of the wallet, whose signatures would be needed to transfer funds from it, did not have its visibility changed from the default, which in Solidity is public. This error allowed malicious users to call the function with an array containing their own wallet addresses, and a low daily limit, thus taking ownership of the entire wallet and allowing them to drain the funds. (Praitheeshan, Pan, Yu, Liu, & Doss, 2019)

3.2.10 Integer Over/Underflow attacks

Use of unsigned integers to store user balances in smart contracts can be exploited by a malicious user requesting to withdraw Ether or tokens when their balance is 0 or less than the withdrawn amount. If the contract does not properly check, an underflow bug can occur whereby the user's balance is reset to $2^{256} - N$ where N is the number of Ether or tokens subtracted from the balance after the balance has already reached 0. (Praitheeshan et al., 2019)

3.2.11 Approve Attack

In ERC20, an address may transfer tokens on behalf of another address by first approving the address a fixed amount of tokens using the approve function, with the latter address then transferring the tokens using the transferFrom function. Given a scenario in which 3 addresses A, B and C, where the owner of B also controls C, an exploit is possible.

1. A approves X amount of tokens to account B
2. A changes the approval amount to Y tokens via another call to the approve function.
3. B sends an allowed amount X to C before the second call to approve gets mined with a high gas value so that it gets mined first.
4. When A's second call to approve Y tokens gets mined, B has already transferred X tokens to C
5. B can now transfer an additional Y number of tokens, for a total of X + Y

Mitigation for this exploit is done by adding a 3rd parameter to the ERC20 approve function:

```
function approve(address spenderAddress, uint256
    currentValueOfAllowance, uint256
    ChangedValueOfAllowanceValue) returns bool success
```

If the current allowance for the passed address is equal to the number of tokens passed as currentValueOfAllowance then change it to ChangedValueOfAllowanceValue and return "true", otherwise do nothing and return "false". (Shirole, Darisi, & Bhirud, 2020)

3.3 Security Evaluation Methods

(Liu & Liu, 2019) and (Praitheeshan et al., 2019) separate security evaluation methods for Solidity smart contracts into 3 categories: Static Analysis, Dynamic Analysis and Formal Verification and lists several presently existing tools for the carrying out of each as well as the analysis methods used by them.

3.3.1 Static Analysis

Static analysis entails analysing code without executing it to look for known vulnerable code patterns.

Tool	Method	Description	Vulnerabilities Detected
OYENTE	Control Flow Graph	A graph is formed where each node represents an atomic execution block, and each edge represents a jump from one block to another. Static analysis is performed by performing symbolic execution through traversing the graph.	Re-entrancy Integer overflow/underflow Transaction order dependence Timestamp dependence Callstack Depth Parity Multisig bug
Zeus	Rule-based Analysis	A security policy is written in a given language and inserted into the contract bytecode as assert statements.	Re-entrancy, Unchecked send, Failed Send, over/underflow, transaction state dependence, block state dependence, transaction order dependence.
Ethir	Control-Flow Graph/Rule-Based	Based on OYENTE; uses rule-based representations of the control-flow graphs generated by it.	Same as OYENTE.
Mythril	Symbolic analysis, fuzzing, source code analysis	Well-developed and maintained tool able to detect unsafe byte code as well as bad coding practices such as lax authorisation and control flow leading to infinite loops.	See: (*https://mythx.io/detectors/*, n.d.)
Securify	Semantic Based/Symbolic execution.	Available as an online tool. (https://securify.ch). Works on patterns of compliance and violation of coding practices. (Tsankov et al., 2018)	Rather than detect specific vulnerabilities, this tool uses compliance patterns to detect code that doesn't comply which may result in vulnerabilities.
SmartCheck	Lexical/Syntactical analysis	Static analysis tool that generates an XML representation as an intermediate representation and identifies vulnerable code paths thence. (Tikhomirov et al., 2018)	See: (*https://tool.smartdec.net/knowledge*, n.d.)
Slither	CFG-based/hybrid	Static analysis tool designed to detect both security vulnerabilities and possible optimisations. Written in Python. Uses own intermediate representation called SlithIR. (Feist, Grieco, & Groce, 2019)	Shadowing, uninitialised variables, reentrancy. (Feist et al., 2019)

3.3.2 Dynamic Analysis

Dynamic analysis refers to analysis done on a running program, simulating a real-world attack.

Tool	Description	Vulnerabilities Detected
MAIAN	Problematic smart contracts are divided into 3 categories. Greedy: contract accepts Ether with unreachable or non-existent commands Prodigal: releases received funds to addresses other than their legitimate owners Suicidal: When a contract may be killed by an address other than its creator. EVM bytecode is symbolically executed across all possible execution traces to check for vulnerabilities.	Call stack depth limitation Destroyable/Suicidal contract Unsecured balance Greedy contracts Prodigal contracts
GASPER	Checks execution paths of contract for costly (in terms of gas) execution patterns.	Gas wastage, infinite loops.

3.3.3 Formal Verification

Formal verification methods use formal proofs using discrete mathematics to show functional correctness and safety at run-time of smart contracts. F*

Framework (Bhargavan et al., 2016) starts by compiling Solidity bytecode into F*, a functional programming language, OR decompiling EVM bytecode into F* if the source code is not available. Contracts translated into F* are then checked for the presence of language constructs that are deemed to be potentially vulnerable.

3.4 Tokens

(Blenkinsop, 2019) (Sameeh, 2019)

3.4.1 Definitions

- *Token*: a unit of exchange whose creation and transfer is facilitated by a smart contract on the blockchain. may denote, a unit of currency, shares in a venture, voting rights, title deeds to some real-world asset etc. May be **fungible** or **non-fungible**.
- *Utility Token:* digital assets designed for spending within a given blockchain ecosystem, such as for paying for services from a given company, redeeming for discounts on consumer goods, virtual currency within an online game etc. Value in "real-world" currencies or cryptocurrency may fluctuate based on supply and demand. Companies may sell tokens to raise funding in "Initial Coin Offerings", with the tokens then being able to be used by owners in the companies blockchain system. Utility tokens do not necessarily have to be tied to the value or success of the issuing company, and are largely unregulated.
- *Security Token:* Represent legal ownership of some asset, such as shares in a company, deed of ownership of real world tangibles such as a house. Usually heavily regulated in most jurisdictions.
- *Fungible vs. Non-Fungible Tokens:* a fungible asset, such as currency (crypto or otherwise), is not unique and can be subdivided into units ex: a 5 £ note is not unique and is as good as any other 5 £ notes. A non-fungible asset is one which is intrinsically unique, is not interchangeable with any other asset, and may not be subdivided. Examples of such assets may include the title deed to a house, an airline ticket or a birth certificate. Utility tokens are typically fungible, whereas Security tokens may be non-fungible (ex. a deed to a house), partially fungible (some contracts allow for partial ownership of an asset) or fully fungible (ex. shares in a company).

3.4.2 Token Standards

- ***ERC20:*** (Shirole et al., 2020) An open source standard defining an interface to be implemented by the developer. ERC20 is the most widespread of fungible token standards. Template includes the following methods to be implemented:
  - *totalSupply*: the total number of ERC20 tokens possessed by the contract.
  - *balanceOf*: Returns the number of tokens possessed by a given user.
  - *transfer*: Transfers a given amount of tokens from the balance of the contract calling the function to a given recipient.
  - *approve*: Allows a user to allow another user to spend a given amount of tokens on their behalf.
  - *transferFrom*: Allows a user to create a transfer of a given number of tokens from a given user to another given user, as long as the former has approved the caller of the function to spend that given number on their behalf with the approve function.
  - *allowance*: Gives the number of tokens a given user has been allowed to spend on behalf of another user.
- ***ERC223:****(dexaran@ethereumclassic.org, 2017)* A suggested improvement to the ERC20 token standard, intended to solve a bug (or rather , an oversight) in the ERC20 specification whereby if tokens are transferred to an address belonging to a smart contract rather than a user wallet address, the tokens may become lost if the recipient contract has no way of actually then transferring or otherwise reacting to the receipt of the transferred tokens. This flaw in the ERC20 standards has already resulted in the loss of millions worth of tokens across several DApps. ERC223 is a proposed extension of the ERC20 standard rather than a complete reworking of it and is designed for backward compatibility with it. In order for a token adhering to the ERC20 token standard to implement the ERC223 standard, its **transfer** function must, after verifying that the recipient address is a smart contract and not a user wallet, attempt to call a function **transferFallback** on the recipient contract having the following signature:

```
1 function tokenFallback(address _from, uint _value, bytes
    _data)
```

If this function is not implemented on the recipient contract, the *transfer* function on the sending contract should fail and the transaction should be reverted.

- ***ERC777:*** *(Dafflon & Baylina, 2017)* Another suggested improvement to the ERC20 standard, intending to alleviate the inefficiency (in terms of Gas usage) caused by one having to call the approve and transferFrom functions in order to debit an amount of tokens rather than a single function. The ERC777 function seeks to make sending of tokens more like sending of Ether, it thus replaces the approve and transferFrom combo with a single send() function. ERC777 tokens differ from ERC20/223 tokens mainly in the following two functions, for which it defines the concept of operators and holders, where holders are the addresses owning the tokens and the operators are addresses authorised to conduct token transactions on behalf of the holders:

```
function send(address to, uint256 amount, bytes calldata data) external
    - sends tokens to an address, from the address calling the function

function operatorSend(address from, address to, uint256 amount, bytes calldata data, bytes calldata operatorData) external
    - sends tokens to an address on behalf of a holder address; the calling contract address must be defined as an operator for the given holder address.
```

Another problem that ERC777 aims to address is that contracts whose token balances are updated are not "notified" when their balances are updated, meaning that the contracts cannot "react", in the same way that they can when Ether is transferred to them. In order to facilitate this, contracts that work with an ERC777 token should implement the following interfaces:

```
function tokensToSend(address operator, address from, address to, uint256 amount, bytes calldata userData, bytes calldata operatorData) external

function tokensReceived(address operator, address from, address to, uint256 amount, bytes calldata data, bytes calldata operatorData) external
```

These functions will be called by the token contract on the "holder" address from which the tokens are sent and on the receiving contract.

- ***ERC721:*** *(William Entriken, Evans, Shirley, & Sachs, 2018)* A token standard used for defining an interface to allow for the creation, management and trading of non-fungible tokens. The function tokenMetadata renders the tokens non-fungible by creating a unique set of attributes. An ERC721 compliant token must implement the following interface as well as the ERC165 interface.

```
function name() constant returns (string name)
    - returns name of contract.

function symbol() constant returns (string symbol)
    - returns shorthand name of token

function totalSupply() constant returns (uint256 totalSupply);
function balanceOf(address _owner) constant returns (uint balance);

function ownerOf(uint256 _tokenId) constant returns (address owner);
    - returns owner of a token of given ID

function approve(address _to, uint256 _tokenId);
    - grants permission from calling user to transfer a token of given ID on their behalf

function takeOwnership(uint256 _tokenId);
    - given that the calling user has been granted permission to take a given token via the approve function, a user may call this function to take ownership of a token of given id

function transfer(address _to, uint256 _tokenId);
    - given that the calling user has been granted permission to take a given token via the approve function, a user may call this function to transfer a token of given id to another user or contract.

function tokenOfOwnerByIndex(address _owner, uint256 _index) constant returns (uint tokenId);
    - Given that an owner's tokens are tracked by an array, returns the id of a token given the address of the owner and the index of the token to retrieve.

function tokenMetadata(uint256 _tokenId) constant returns (string infoUrl);
```

```
26      - Returns the metadata making an NFT token unique
            given its unique ID as a parameter. This metadata
            will naturally vary from application to
            application but will generally include an HTTP url
             giving more information about the token.
27
28  event Transfer(address indexed _from, address indexed _to
        , uint256 _tokenId);
29      - Fires whenever a token changes ownership.
30  event Approval(address indexed _owner, address indexed
        _approved, uint256 _tokenId);
31      - Fires whenever a user approves another user to
            handle a token on their behalf.
```

- ***ERC 1155:*** *(Witek Radomski, Cooke, Castonguay, Therien, & Binet, 2018)* ERC1155 is a standard intended to be able to be used for both fungible and non-fungible assets, with a stated use-case being fungible assets with classes such as event tickets which may have different classes such as Economy, Premium, VIP etc which may be fungible within their own class but not fungible as a whole. The main aim for this standard was to allow creation of multiple tokens, both fungible and non-fungible within the same contract rather than having to create a different contract for each token, thus a very likely use-case is in online games where a player may possess several different types of virtual assets which may be fungible (ex. ingame currency) or non-fungible (ex. ingame collectibles, otherwise, a different contract would have to be created for each item type, obviously infeasible in an online game which may have tens of thousands of different types of items or a trading platform where thousands of assets of different types such as futures contracts, stocks and bonds may be traded.

- ***ERC 1400:*** *(Dossa, Ruiz, Vogelsteller, & Gosselin, 2018a)* ERC1400 is a collection of token standards that together create a security token standard designed to track the exchange of real-world securities on the Ethereum platform while complying with the corresponding regulatory requirements. The ERC1400 standard is an umbrella term for the following standard:

- ***ERC 1594:*** *(Dossa, Ruiz, Vogelsteller, & Gosselin, 2018b)* The core standard for security tokens; allows injection of off-chain data into transactions and allows for the issuance of and managing of securities on the blockchain by specifying a standard interface.

- ***ERC 1410:*** *(Ruiz, Vogelsteller, Dossa, & Gosselin, 2018)* Allows for the partitioning of a token holder's balances into different balances.
- ***ERC 1643:*** *(Dossa, Ruiz, Gosselin, & Vogelsteller, 2018)* Allows for off-chain documents to be attached to transactions of securities, and for these documents to be retrieved and removed.
- ***ERC 1644:*** *(Gosselin, Dossa, Vogelsteller, & Ruiz, 2018)* Allows for forced transfers of tokens, say, to facilitate seizure of assets.

3.5 Blockchain Voting

This section will discuss existing work both theoretical and practical pertaining to both blockchain and general electronic voting solutions.

(Moura & Gomes, 2017) cites the lack of transparency created by electronic voting machines such as those used in Brasil; the voter has no guarantee that their vote has truly been cast and that it will be counted correctly. The authors also cite speculation that Russian hackers may have tampered with electronic voting machines in the USA during the 2016 presidential election. Whether this is true or not is beside the point; the point being that the lack of transparency and centralisation means that the public cannot know for sure.

(Yavuz, Koç, Çabuk, & Dalkılıç, 2018) cites Estonia as a relative success story, having been using electronic voting as its main way of conducting public elections since 2003 through a government web portal, with citizens authenticating themselves through the use of electronic ID cards inserted into government-provided card readers. Whilst being a success story relatively, such a system still suffers from having a single-point of failure, vulnerable to DDoS and cyberattacks, as well as providing limited transparency as the tallying of votes is ultimately in control of a single entity, leading to the possibility of tampering by internal personnel. The paper also cites scalability as a concern. Given the rather small population of Estonia, particularly given the need to manufacture and distribute card readers to eligible voters at likely considerable cost, this system may not be feasible for larger countries. The paper goes on to describe an Ethereum-based e-voting solution, with the caveat that no solution is provided to the problem of voter authentication, and how this is to be reconciled with voter anonymity, banking on the simple fact that votes would be tied to an Ethereum account address rather than a person's name or ID card number to provide anonymity. Moreover, the

presented solution would be impractical for deployment on any large scale as it would require the entire voting population to be in possession of an Ethereum account address, and know how to set up and use wallet software. This is of course impractical given that a significant proportion of people in any given country require assistance to perform even basic tasks on a computer and would likely result in many people being unable to vote on election day. Additionally, the transactions needed to vote would be launched directly from the voter's wallet address so would need to be paid for by them in Ether.

(Kshetri & Voas, 2018) propose a partial solution to the aforementioned problem in their solution, whereby voters would be issued a fresh wallet address "pre-loaded" with a single "coin" that they could spend in order to vote. The problems from the previously discussed solution still present being that it would still presumably require voters to install some kind of wallet software, and voter anonymity would be guaranteed only if no data linking the voter to their issued wallet address existed. The paper also appears to imply that a custom blockchain would be created for the express purpose of holding the election, which appears both redundant and insecure given the verifiable security and robustness of Ethereum.

(Hjálmarsson et al., 2018) similarly, also proposes a voting system based on a private or "permissioned" blockchain, with each voting district hosting its own node. In this case it is an Ethereum-based network where the elections are implemented as smart contracts. In their proposed system, voters are issued a wallet address in order to be able to vote for each election which will be pre-generated by the election authority. The voter would authenticate themselves on a central government system and be pointed to the smart contract pertaining to their electoral district; each district having their own smart contract that conducts its own tally. The voter would be issued a transaction ID after casting their vote, in theory enabling them to verify their vote has been cast using a blockchain explorer. Drawbacks of this system include:

- It is significantly centralised and therefore vulnerable to attack, as it uses a private blockchain consisting of a small number of nodes controlled by a single organisation.
- It requires voters to be able to use wallet software.
- Wallet addresses are generated for each voter prior to the election. This creates a point of failure whereby a malicious government employee or

a hacker could create wallet addresses for non-existent voters to "stuff" the ballot.

Another proposed system is *Votebook (Kirby et al., 2016)*, which propose a system that would be superficially (that is, from voters' perspective) similar to current electronic voting machines. Under this system, the nodes in the blockchain network would be simply the voting machines themselves, with the keys being assigned to the machines by a central server. The system assumes a single selection voting system whereby transactions in a block consist of hashes of a voter's id concatenated with their selection of candidate. Meanwhile another blockchain runs in parallel that stores hashes of just the voter IDs, in order to ensure voters cannot vote more than once. The system proposed by Votebook relies on isolating the system as much as possible; the blockchains will be completely private and on a network of computers not connected to the internet, and thus is effectively centralised. This was in fact by design as the designers of the system felt remote voting posed too much of a risk of voter suppression through DDoS attacks and voter coercion. Advantages of this approach include no need for behavioural change from voters and ensures accessibility to voters without internet access, but ultimately it still remains a highly centralised system. (Osgood, 2016) in its description of VoteWatcher as another blockchain based voting solutions describes a similar setup; private blockchain, voters do not use their own devices to vote.

3.6 Criticism of Blockchain As a Voting Solution

In this article (Jefferson, 2018) for U.S. Vote Foundation, the author criticises the notion of blockchain voting on issues that, while not intrinsic to blockchain technology, must be considered. The first of the author's criticisms is that, as pointed out above, most voting solutions being presently proposed are not truly distributed and are in fact private blockchains composed of nodes under the control of a single organisation. Thus, they offer little incentive in terms of accountability over a simple centralised database. The author goes on to cite voter identification and authentication as a practical problem, citing (in the context of the USA) how authentication of voters would be easily compromised given that most available forms of identification such as social security numbers and driving license numbers have been compromised in data breaches as well as the possibility of DDoS attacks to prevent certain segments of the population from voting, and the possibility of malware on voters' devices interfering with the voting. The author goes

on to (somewhat perplexingly) claim that blockchain technology lacks transparency and auditability and is susceptible to the same hacking threats that traditional centralised services are. Whether the author is speaking about the centralised private blockchain election solutions previously talked about or public blockchains such as Ethereum is not made clear; but the author's assertions are reasonably valid in the former case but completely unfounded in the case of the latter. Another article, (Mearian, 2019), published in *ComputerWorld* highlights many of the same concerns (again, appearing to operate on the assumption of a private blockchain), concentrating particularly on the blockchain-enabled mobile phone voting application "Voatz" used by the U.S. Military for some of its members stationed abroad. The article cites among other things its questionable authentication procedure which makes use of biometrics; namely thumb prints and facial recognition via mobile phone camera. (Jefferson, Buell, Skoglund, Kiniry, & Greenbaum, 2019) lists several known concerns with this "Voatz" app, namely its use of a permissioned private blockchain and its use of biometrics with known high-error rates coupled with machine learning algorithms for facial comparison, with the company not divulging the data set used to train said algorithms. Criticisms of Voatz largely revolve around its lack of transparency in terms of the details of its implementation, which it has never divulged in detail. (Suwito & Dutta, 2019) Emphasises the need for resilience of a hypothetical e-voting system against voter coercion and vote selling that may be rendered possible as a result of the inherent verifiability of blockchain transactions. While oft cited as an inherent advantage of blockchain-based voting, verifiability of transactions conducted on blockchain, that is, in this case the transfer of a vote to a candidate from a voter, presents some problems. Whilst e-voting systems that provide some kind of receipt or allow the voter to verify what candidate or party their vote ultimately went to may be appreciated by voters for the peace of mind they provide, it introduces possible problems. In a traditional ballot system, selling of votes by dishonest voters and coercing voters is impractical as a 3rd party has no way of knowing how the voter ultimately casts their vote; the fraudulent or coerced voter could simply agree to vote for a given candidate and then vote for another. The paper makes a distinction between *weak* verifiability and *strong* verifiability; the latter being a situation where it is possible to verify which candidate the vote ended up at whereas the former is where a voter may only verify that the vote has been cast. Clearly, a voting system with strong verifiability induces concerns about coercion and vote-selling.

Conclusion To summarise:

- Present blockchain voting solutions make use of private, permissioned blockchains rather than public, proven networks such as Ethereum.
- This leads to centralisation and lack of inherent verifiability and transparency due to the closed nature of the system and only the personnel of the organisation running the system being able to access the blockchain.
- Systems such as these offer little inherent advantage over traditional server structures.
- These systems very often use custom-built blockchains for which security is not verified.
- Many concerns about blockchain voting systems could be easily alleviated by making use of an established public blockchain that are known to be practically secure and easily auditable. Ethereum most certainly fits this criteria, and additionally supports the easy writing of bespoke code to accomodate the various nuances of different elections in different jurisdictions.
- None of these solutions implement an STV voting system.

Additionally, it is felt that the ERC721 token standard adequately meets the requirements for use in implementing the vote token for this application and it will not be necessary to code one from scratch or modify it. Based on the findings here, the next chapter shall present the design and implementation of a system intended to address the identified shortcomings in the discussed solutions.

4 Design & Implementation

4.1 High-Level Overview Of Application Structure

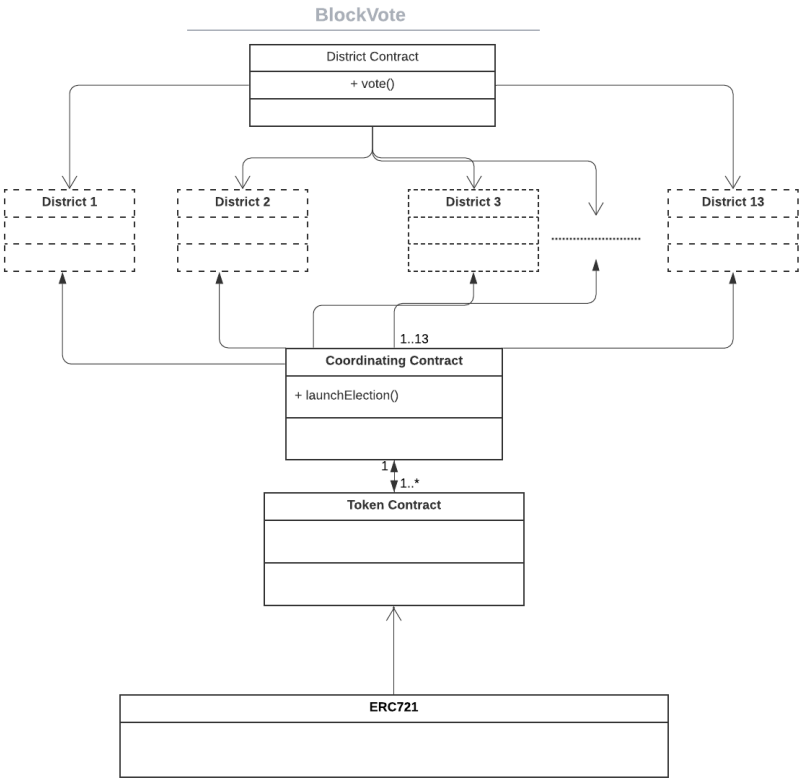

Figure 4: High-Level Overview Of Distributed Application Structure

4.1.1 Components of Applications

In figure 4, one may find a basic outline of how the application is to be structured and how the smart contracts that make up the distributed application are to relate to one another.

The coordinating, or "controller" contract shall be permanent, in that it is to be reusable across different elections and stores results of individual elections in its own storage on the blockchain.

The district contracts will be responsible for tallying the votes, with a separate one for each electoral district. District contracts will not be reused

between elections and will be created for each election. This is to account for changes in the number of eligible voters in each district between elections as well as possible future changes in the electoral divisions. They will not however, be torn down after an election and instead be left on the blockchain in order so that election results stored therein may be verified at any point in the future.

A new token, used to represent votes, is created for each election cycle and retired afterwards. As previously mentioned, this token will be based on the ERC721 standard.

4.1.2 Choice of Token Standard

The token shall be an ERC721 token extended with some extra functionality particular to the application. The choice of the ERC721 standard (discussed in Section *3.4.21* of the literature review) as the basis for the voting token was made owing to ERC721's built-in facilitation of adding metadata to tokens, a feature ERC20 and other token standards thence derived lack, owing to them being designed to facilitate fungible tokens. The decision to conceive of a vote as a non-fungible token is admittedly a somewhat peculiar one and requires some justification:
Whilst every vote is obviously to be considered of equal value, and a vote that has been cast must not carry any data that could identify the person who casts it, the nature of the STV vote necessitate the addition of some metadata, the metadata consisting of the order of preferences given by the voter, and also data to record the transfer of the vote from one candidate to another during counting. The former is necessary to facilitate the counting of votes in a robust way whilst the latter will help in ensuring ease of auditability. This metadata will also be needed in the event of a casual election (see above). Ultimately, the votes in this system are not true NFTs, and our implementation deliberately removes or restricts use of much of the functionality of the underlying ERC721 technology (by overriding unnecessary functions and replacing them with empty functions) that is not needed for our purposes. The use of ERC721 here is therefore very much an "off-label" application. The decision to make use of its infrastructure was driven largely by security concerns; whilst the resulting contract is far larger than it needs to be as much of the ERC721 is not being used at all and this will naturally make it more costly to deploy, this is balanced out by the guaranteed security given by using the infrastructure of a pre-audited and widely used token framework rather than building a new one from scratch. Whilst an explanation of the entirety of ERC721 is out of scope, the full body of the

T. Muscat

implemented token along with comments explaining the functionality it adds to, and removes from, the standard ERC721 contract it inherits from may be found in the relevant entry in Appendix A.

4.2 Lifecycle of an Election

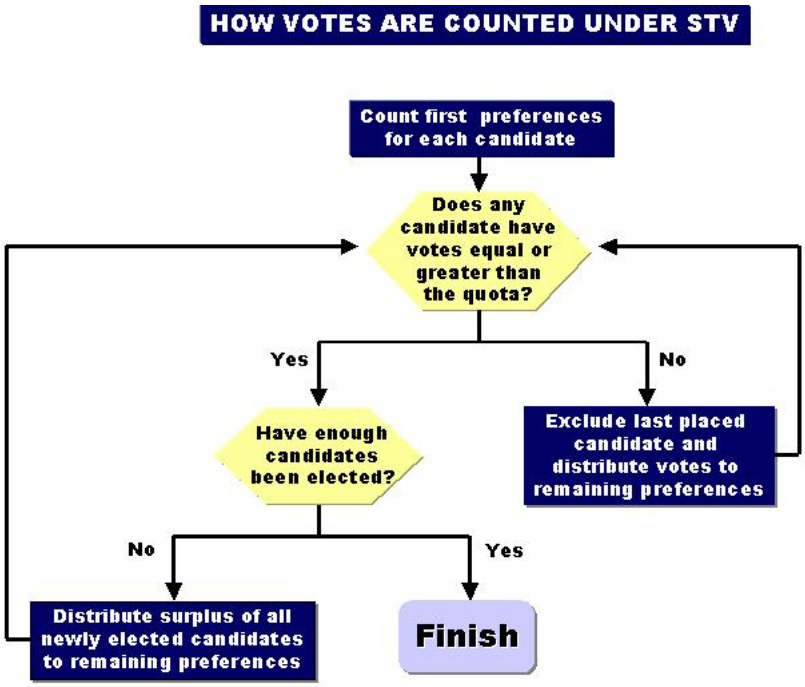

Figure 5: STV Vote Counting Procedure, for illustration of the algorithm. (*All about STV and FPP*, n.d.)

1. Coordinating contract launches a new token with timestamps indicating the time at which polls are to open and at which time they will close. Before and after this time window the token will be in a paused state and no transactions (ie: voting) will be possible

2. One token will be minted for each eligible voter, with one single token being added to the balance of the controlling contract[5]. Each token unit will be initialised with a district number as metadata, ensuring it can only be "spent" on a candidate on the electoral district in which

[5]See subsection 2.4 for explanation

the voter lives. Important to note here is that candidates, being themselves eligible voters will simply have an account address same as other citizens and will be themselves assigned a token with which to vote.

3. District contracts are launched and initialised with the list of candidates contesting on that district and their corresponding account numbers.

4. Voting opens, and the token is taken out of the paused state.

5. Voters cast their votes, with their list of preferences being added to the metadata of the token unit.

6. The voter's token is transferred to the district contract, which then increments the 1st count votes of the party their first preference candidate belongs to. The token is then transferred to the voter's first selected preference if that candidate has not yet reached the quota to be elected. If they have, then the vote is transferred to their 2nd preference candidate unless they also have already reached the quota and so on. If all of the voter's selected preferences have already reached the quota, then the vote is deemed non-transferable and sent to a special address reserved for such votes. (effectively, it is discarded)

7. Voting closes, token returns to paused state.

8. Counting begins, only transactions between district contract accounts and candidates being permitted.

9. Each district contract will iterate through its list of candidates and mark all that obtained the quota as "elected", and placing all those that were not in a storage array (storage meaning, it persists in the contract's data).

10. If all seats were filled, counting is finished.

11. Else, having sorted the list of unelected candidates in ascending order of votes obtained, the candidate with least votes is marked as eliminated. Their votes are then reallocated to their next preference that has not either already been elected or eliminated. If the voter has marked no subsequent preferences matching these criteria, the vote is discarded as previous.

12. This step is repeated for the unelected candidate with 2nd,3rd....nth least number of votes until all seats are filled.

13. The coordinating contract tallies the first count votes and declares which party has won.

14. The proportions of first count votes obtained by both parties are calculated and additional seats S assigned by the proportionality principle.

15. For a given number S seats assigned to a given party P, the S candidates from party P with the highest number of votes that were not elected are chosen to fill the seats and their final status updated to "elected"

16. Data pertaining to the election is stored in the coordinating contract, such that results are persistent and can be verified.

The computation for the elimination and reallocations steps being expensive and repetitive, it is performed largely off-chain via a script interacting with the smart contract when needed to retrieve and update data. This should not effect the transparency of the process as all transactions sent by this script can be verified on the blockchain. All other computation is performed on-chain.

4.3 Explanation Of Vote "Minting" and User Authentication.

In the previous section it was said that uncast votes are first stored in the balance of the controlling smart contract, rather than the voter's wallet address. The reason for this is simply that it is unfeasible to expect an entire country's population to set-up wallet software on their own computers in order to vote, since many will not be able to. At the time of writing, the number of people familiar with the technology is low enough that such a system would have serious concerns about excluding large sections of the voting population who are not able to set up the software correctly, such as the elderly. Additionally, there are concerns that it may compromise voting anonymity since all transactions on the blockchain are transparent and one would be able to see how a person voted if they knew their Ethereum account address. One could mitigate this by instructing voters to create a new account address with which to vote but there would be no way to enforce compliance. We therefore propose the following system for the creation and use of vote tokens:

1. A 256-bit vote token ID is generated by catenating the voter's government ID card number to a randomly generated nonce value and

computing a 256-bit hash of the resulting string. The particular hashing algorithm is relatively unimportant, as long as the message digest is exactly 256 bits long. However, choosing a slow hashing algorithm such as *scrypt* may alleviate concerns about brute force hash collision attacks.

2. A new vote token unit is generated with the resultant token ID and added to the balance of the controlling contract.

3. The nonce value is communicated to the voter (in a minimally cumbersome-to-type fashion such as base64) in such a way that is NOT saved on any government system or elsewhere.

4. Upon logging into the voting portal, the voter will enter the nonce value when requested. The system (off-chain) will then catenate the entered nonce value to the user's ID card number and recompute the hash, thus obtaining the token ID of their assigned vote token.

The logistics of the first step in the above procedure could proceed in two different fashions, both of which have drawbacks:

- The 256-bit token IDs could be generated and the tokens minted in a single batch, with the nonce values then being communicated to all eligible voters. The drawback here is that there is no intrinsic voter anonymity; voters would simply have to trust that the government or contractor didn't simply save the nonce values and use them to derive the token IDs of voters and see how they've voted. 6

- Alternatively, voters could be required to log into a government portal to generate their nonce values which they would later save to be used on polling day. The drawback here is that it adds an extra step which may confuse certain voters and may hinder acceptance of the new system. Complaints from voters who are unable to vote because they didn't save or somehow lost their given nonce value will be inevitable. 7

The application is able to function independent of which of these approaches is chosen.

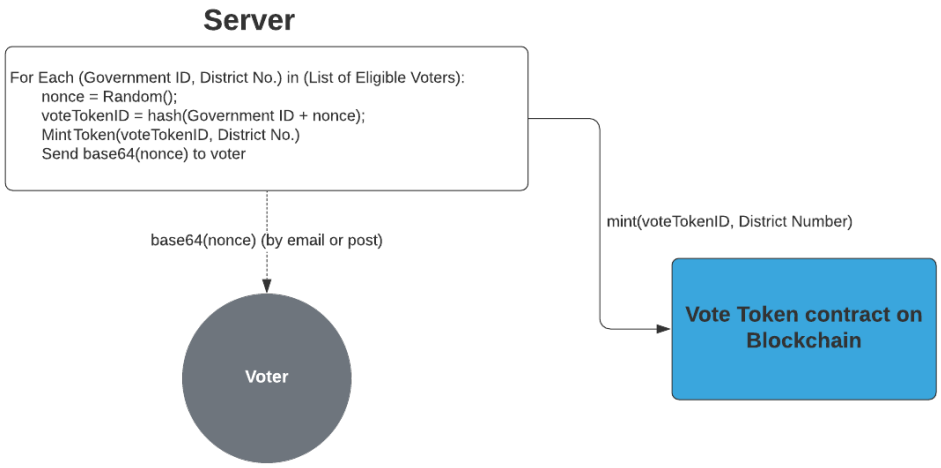

Figure 6: Minting Procedure 1

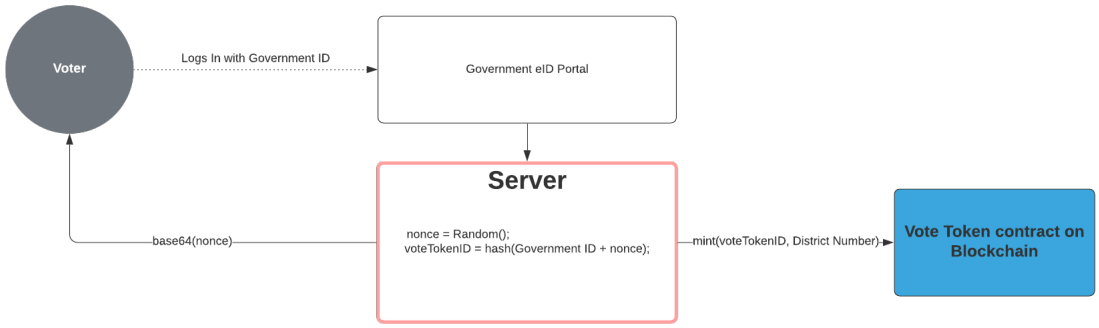

Figure 7: Minting Procedure 2

4.4 Voting Procedure

1. The voter would log into a government portal using their electronic ID account[6].

2. The existence of a mapping between a voter's electronic ID number and an Ethereum address pre-generated stored off-chain in a relational

[6]Such an infrastructure is already in place in the Republic of Malta, one's electronic identity username is simply one's national ID card number.

database is assumed.

3. A token is minted for the voter, with a token ID consisting of a pseudo-randomly generated 256-bit number [7] number and the voter's district stored as metadata. This token ID will not be stored anywhere, and the voter must note it down. This token ID will, from the user-perspective, effectively function as a one-time password for when the voter casts their vote. This step is analogous to collecting a physical one-time voting document used to vote at in-person elections.

4. When voting opens, the voter will again log into a purpose-built government portal with their electronic ID, enter their token ID and fill in their ballot through an on-screen interface.

5. The application will ensure that the token ID is owned by the address corresponding to the voter; ie that it does not belong to someone else or it has already been cast.

6. The application will check the token's metadata to ensure that the voter is not attempting to cast their vote in a district other than their own[8].

7. The voter will receive a prompt asking them to confirm their selection.

8. The vote is cast; ie: the ownership of the voter's token is transferred to the contract account corresponding to their district.

4.5 Pseudocode

Below is pseudocode detailing the logic of the major functionality of the application. (8 10 9 11)

[7] Tokens in the ERC721 standard use 256-bit unsigned integers to uniquely identify tokens.

[8] Of course, the front-end application would not allow this under normal operation. This is a fail-safe to ensure exploitation of the back-end is not possible if the front-end application is compromised.

```
Election = New Election(Name, VoteStart,
    VoteEnd, NumberOfPartiesContesting,
    CandidatesContesting[DistrictNumber][])

ElectionsArray.Push(Election)

for i := 1 to NumberOfDistricts:
    Election.Districts.Push(New District(i, CandidatesContesting[i-1]))
```

Figure 8: Election Controller Launching Election

This pseudocode 8 outlines the procedure by which the Election Controller contract launches a new election and stores a data structure representing it within the contract itself.

```
Function vote(voteTokenID, preferencesArrayOfAddresses[])
    //Within District Contract
    Require:
        VoteTokenContract.getTokenMetadata(voteTokenID).District = ThisDistrict
        Revert transaction otherwise

    Transfer(vote,from: sender,to: this district contract)

    VoteToken.setTokenMetadata(voteTokenID,preferencesArrayOfAddresses)

    Cast Votes For this District += 1

    Increment 1st count votes for 1st pref. candidate's party.

    i:=1
    while(preferences[i] has vote balance >= quota AND preferences[i] exists)
        i++

    if(preferences[i] exists)
        Transfer(vote,from: this district contract, to: preferences[i)
    else
        Discard Vote
```

Figure 9: Voting Procedure Pseudocode

This pseudocode 9 shows the logic by which individual district contracts receive a voter's vote token and distribute it to the correct candidate, based on the voter's set preferences.

```
    For candidate In Candidates:
        If(candidate.votes > quota):
            candidate.elected := True
            seatsRemaining--
        Else:
            notElected.push(candidate)

    if(seatsRemaining > 0):
        sort(notElected)
        For Each candidate in notElected:
            redistribute votes to next pref.
            If(seatsRemaining == 0)
                break
```

Figure 10: District Counting Procedure Pseudocode

This pseudocode 10 details the logic by which each individual District smart contract checks how many candidates have reached the quota needed to be elected, and thus how many seats have been filled. It also shows the logic by which, if the number of filled seats is less than the total number of seats for this district, the candidates with the least votes are eliminated and their votes transferred to the next preference candidate until all seats have been filled. This function will be called from the Election Controller contract on each District contract before the logic shown in figure 11 is performed.

```
For Each District Contract in Election.DistrictContracts :
    Await: District.CountVotes()

    Require District -> Counting Succesful:
    If Not Throw Error.

For Each Party in Election.Parties:
    Party.1stCountVotes += District.1stCountVotes[Party]

Election.Winner := MostFirstCountVotes(Election.Parties)
Election.Concluded := True
```

Figure 11: Election Controller Tallying.

This pseudocode 11 details the logic through which the Election Controller contract tallies the 1st count votes from each District contract and decides which party is the overall winner of an election. The logic for *District.CountVotes()* is shown in figure 10.

5 Evaluation

5.1 Unit Functionality Testing

(See appendix B.1 for code listings of unit tests) In this sub-section, the functionality of the underlying smart contracts shall be evaluated using Javascript unit tests facilitated by the Truffle framework. Functionality here is understood to mean that the components of the application as well as the system as a whole meet the specifications and work as expected. Functionality in terms of adequacy of the entire application to its stated purpose will be analysed in a later section [9].

5.1.1 Testing the Vote Token

The following criteria were used to verify the functionality of the Vote Token contract. It should:

1. The Contract should properly store the votation start and end times as well as the "name" value passed to the contract constructor.
2. The "mint" function should increase the token balance of the passed address by 1.
3. The minted token should have the correct district number stored as metadata.
4. Should revert token transfers called from non-allowed addresses.
5. Should not revert token transfers from allowed addresses.
6. The "setVotePreferences(tokenID,address[] preferences)" function should map a given tokenID to another mapping of integers to addresses created from the passed array of addresses in such a way that integers from 1 to preferences.length map to addresses that equal the addresses at 0 to preferences.length-1 in the passed array.

5.1.2 Testing The District Contract

(Code listing: B.1.1)

[9] See Appendices for listings of the unit test code as well as contract code.

1. "vote(tokenID, candidateAddresses)" function should succeed if called by the owner of the token and the district number stored as metadata corresponding to the tokenID matches the number of the district contract the vote function is called from.

2. "vote(tokenID, candidateAddresses)" function should fail if called by the owner of the token and the district number stored as metadata corresponding to the tokenID does not match the number of the district contract the vote function is called from.

3. "vote(tokenID, candidateAddresses)" function should store voting preferences correctly.

4. The function "getQuota()" should calculate the correct quota value from the number of cast votes and seats up for election. That is, output should agree with the formula (castVotes/(seats+1)) + 1 (Results are always integers, digits after decimal point are truncated.)

5. The District contract should be able to transfer all tokens from a holder (an eliminated candidate) back to itself provided the tokens held by the user have the same district number as itself, and fail otherwise.

6. Given a "toy" example votation consisting of 5 cast votes (and a corresponding quota of 1) each selecting the same 5 candidates from the same party, each candidate should end up with 1 vote and be marked as elected. Party should end up with 5 1st count votes, and the other party with 0.'

5.1.3 Testing the Election Controller contract.

(Code Listing: B.1.2)

1. "function launchElection (name, voteStart, voteEnd, numberOfPartiesContesting, candidates[][])" should create an instance of an Election Struct having the parameters passed in the function and store it in an array inside contract storage along with an instance of the VoteToken contract and 13 instances of the District contract.

2. "function mintVote(address to, uint256 tokenId, uint8 districtNo, uint32 electionNo)" should call the voteToken.mint function corresponding to the the election identifier passed as electionNo correctly.

3. Election controller "endElection" function should tally votes with correct results.

5.2 Cost Analysis

	Gas Used	**Price In ETH**	**Price In Euro**
Contract Deployment	9948517	0.8	244
Mint Vote Token	247325	0.02 (x305556 = 6111.2)	1,863,892
Cast Vote	436220	0.035 (x305556 = 10694.46)	3,261,810
Launch Election	52106727	4.17	1272
Calculating & Storing Election results	4220301	0.72	220
Total		**71811.35**	**21,902,462**

Prices based on average price of Ether and Gas at the time of writing (26/09/2020), 305 Euros and 80 gwei respectively. The multiplicand 305556 was the number of votes cast in the 2013 general election. Here used as a benchmark. The transaction costs for all operations needed to conduct the election were calculated. For the case of "Cast Vote", the transaction cost will vary depending on how far down the voter's listed preferences the candidate to which the vote must be transferred is, thus an average was taken. The cost to launch an election will also vary according to how many candidates are contesting, thus an average over the last 9 elections was taken. Gas costs for other transactions are constant. As can be seen above, while other costs are moderate, the price of minting vote tokens for the entire voting population and, at voting time, transferring them to the appropriate candidates, grows to unmanageable levels.

5.3 System Requirements Analysis

(Code Listing: B.1.2) Taking (Hjálmarsson et al., 2018) as the basic system requirements for the solution presented here, this section will detail the ways in which the solution is able to fulfil them.

1. **An election system should allow a method of secure authentication via an identity verification service.** In production, the system would make use of existing government electronic ID infrastructure used by citizens of the Republic of Malta to access government services using their ID card document number as their login name.

2. **An election system should not allow traceability from votes to respective voters.** The ID by which an ERC721 token is uniquely

identified is a 256-bit unsigned integer. A vote token ID shall be generated (off-chain) for every voter through taking a Bcrypt-256 hash of the voter's ID number catenated with a random nonce value, thus ensuring its uniqueness. This nonce value will be communicated to every eligible voter in the form of a base64 encoded number which will serve as a one-time password; it will not be stored anywhere on-chain or off. At the time of voting the voter will simply input their one-time password, and the system will calculate the vote token ID, allowing them to cast their vote. In this manner, there should be no way to associate a vote token ID to any voter once the vote is cast.

3. **An election system should provide transparency, in the form of a verifyable assurance to each voter that their vote was counted, correctly,and without risking the voter's privacy.** The nature of a public blockchain such as Ethereum ensures that all transactions therein are able to be viewed and verified by outside observers; it is simply built into the technology and little extra effort to ensure transparency and auditability is needed on the part of the developer. However, in order to aid transparency and improve voter confidence in the system, both the Solidity source code of the application and the corresponding EVM bytecode should be made open source.

4. **An election system should prevent any third party from tampering withany vote.** As previously stated, vote token IDs are generated as $scrypt(VoterIDCardNumber + NonceValue)$ at the time the vote token is minted, and voters prove their identity by providing the nonce value given to them as a one-time password. There exists of course the theoretical possibility of a malicious actor attempting to brute force a valid vote token through trying to create a hash collision. In order to mitigate this possibility, the "vote" function in the ElectionController is only callable by the owner of the contract, which will only do so through the web interface, which should be set up with flood protection to prevent bruteforcing. Through only allowing the vote function to be called by the owner of the contract, we prevent hash collision attempts through direct interaction with the contract code bypassing the web interface. Even absent such measures, the large size of a 256-bit hash space and the slowness of the scrypt hashing algorithm ensures bruteforce attempts would be infeasible; searching just half the hash space using Hashcat on Amazon Cloud with 36 cores would take approximately $1.079x10^{28}$ years.

5. **An election system should not afford any single entity control**

over tallying votes and determining the result of an election. Necessarily, the main coordinating contract must have its operations restricted to it's owning address, in this case, the wallet address that deployed it. The persons in control of this account address, ie: those in possession of its corresponding private key are therefore responsible for initialising an election and for calling the function/s used to tally the votes. However, beyond this no person will have any control over the tallying of votes and the determination of the outcome as the decentralised nature of smart contracts and the immutability of the blockchain means that no one will be able to tamper with the results. Providing there is consensus on the conditions of victory prior to the coding of the contract, and agreement to accept the results it computes, the interpretation of the results including the allocation of seats and declaration of an overall winner can also be carried out automatically by the contract.

6. **An election system should only allow eligible individuals to vote in an election.** As said above, there are measures in place to mitigate attempts to forge votes through guessing a legitimate vote token ID which should be more than sufficient. However, interfacing with the system being done using the governments pre-existing e-id system means that that same system could be a potential weak link. Depending on how the random nonces/one time passwords are delivered to the voters, compromise of the government portal could potentially lead to an attacker generating vote tokens for non-existent or ineligible ID card numbers and using them to vote illegally. This could potentially be a problem if the system requires voters to log in prior to election day to generate their vote token and receive their one time password to be used later. A possible mitigation therefore is to pre-generate vote tokens for all eligible voters and send the corresponding one time passwords to the voters by post or email. This may however compromise points number 2 and 4 on this list, as one would not have any intrinsic guarantee someone did not secretly store the one time passwords along with the corresponding ID card numbers. A malicious actor that did this could then be able to see how a certain person voted or even vote using their vote token instead of them.

7. **An election system should not enable coerced voting.** Coerced voting is not considered to be a widespread issue among modern democracies. However the novelty of any electronic voting system means that persons who are not computer literate will require assistance to vote,

they may be intentionally prevented from voting or deceived into voting in ways different from their intention by others. This is however, not a concern particular to this implementation.

5.4 Security Analysis

This section will be an audit of the security of the underlying smart contracts of the application. The audit shall proceed in two parts: a manual check of the code for known vulnerable code patterns and analysis using automated tools previously discussed.

5.4.1 Manual Analysis

Taking the list of known vulnerable code patterns described in (Ma, Gorzny, Mack, & Zulkoski, 2019a) as reference, this section shall give a brief explanation of each and how they were avoided or mitigated in our code.

- **Integer Over/Underflow.** Solidity has no automated checks against integer over and underflow. The default (and maximum) length for unsigned integers in Solidity is 256 bits though shorter bit-lengths may be used. If given two integers, $n > m$ and a value is assigned such that $m = m - n$, the value assigned to m will be $max(m) - (m - n)$. This is an underflow; overflows are also possible naturally but given the large defaulr width of solidity integers underflow bugs are far more common. These bugs can be avoided using the OpenZeppelin *SafeMath* library in place of the default math operations. OpenZeppelin's implementation of ERC721 already uses this SafeMath library in order to prevent exploitation using underflows. Our own code features no function in which callers may pass arbitrary values, and most of the functionality in our contracts is restricted to the contract owner. The use of SafeMath is thus of diminished importance here. Nonetheless, for the sake of absolute assurance, the SafeMath library was used for arithmetic operations that modify the contract state.12

- **Race Conditions.** *Race conditions have been previously described in the Literature Review. Section: 3.4.2* The code is not vulnerable to race conditions. Freshly minted vote tokens are all initially owned by the controller contract. When an election is launched, new District contracts are created, each given authorisation to transfer all tokens from the controller contract. This may seem insecure, but it is far more gas efficient than authorising each token individually when a person

```
15    // Decreases the allowance for the spender to
16    // withdraw tokens from the account of msg.sender by value n
17    function decreaseAllowance(address spender, uint256 n) public
          returns(bool) {
18      uint256 approvedNow = approvals[msg.sender][spender];
19      approvals[msg.sender][spender] = approvedNow - n;
20      return true;
21    }
22    ...
23
24  }
```

Figure 12: Example of code vulnerable to underflow attacks through passing arbitrary values without bounds checking.

(Ma et al., 2019a)

votes, and there is no way the District contract can be made to transfer these tokens except at preprogrammed times.

```
1         //ElectionController.sol
2         function mintVote(uint256 _tokenId, uint8
              _districtNo, uint32 _electionNo) public
              onlyOwner {
3          elections[_electionNo].voteToken.mint(address(
               this), _tokenId, _districtNo);
4
5          }
6         ////////////////////
7          function launchElection (...) public onlyOwner{
8
9         ////////////////////
10         for(uint8 i = 1 ; i <= districtsNo; i++){
11             District d = districtFactory.create(i,
                   candidates[i-1]);
12             elections[elections.length-1].voteToken.
                   setApprovalForAll(address(d),true);
13             elections[elections.length-1].
                   districtContracts[i] = d;
14        ////////////////////
15        function vote(uint8 _electionNo, uint256
              _voteTokenID, address[] memory _preferences)
              public onlyOwner {
16         ////////////////////
17         District _district = elections[_electionNo].
               districtContracts[districtNo];
18         _district.vote(_voteTokenID,_preferences);
19        }
```

```
20          ////////////////////
21          //District.sol
22          function vote(uint256 tokenId, address[] memory
               _preferences) public onlyOwner {
23           ///////////////
24           voteToken.transferFrom(msg.sender,address(this),
                tokenId);
25           ///////////////
26           }
```

In the above code snippets, the process of authorising the transfer of and using vote tokens is atomic and can only be done in pre-determined, automatic ways allowed by the code and is dependent on user input only in the ElectionController/vote function, which is callable only by the contract's owning address.

- **Re-Entrancy.** *Re-entrancy vulnerabilities have been explained previously.* The contract code features no code subject to re-entrant code vulnerabilities. The application makes no use of contract fallback functions and does not at any point make or allow for any transfers of Ether to or from any of its contracts. All externally callable functions (excluding views) are restricted to owners.

- **Transaction-ordering.** Transaction ordering refers to the ability of miners to arbitrarily order the transactions they process when creating a new block. It was found that, while it may not necessarily impact the security of the contract, the execution of the application is intrinsically influenced by the order of transactions. This is investigated in detail in *Section* 5.6.3 .

- **Timestamp Dependence.** The program is dependent on block timestamps where it checks whether the voting period is open or closed before performing certain operations. This is investigated in detail in *Section* 5.5.4.

- **Denial Of Service: Exceeding Block Gas Limit.** In a smart contract function performing iteration for a number of times determined at runtime, there exists the possibility that the array may grow so large that the function will be unable to execute as it will consume enough gas to exceed the block gas limit (averaging around 12,500,000 gas at the time of writing). One such iterative structure exists in the District contracts, where the number of iterations of the *count* function is dependent on the number of candidates and the number of votes cast on that district. The loop's iterations are hard limited by the number of

candidates on the district, and the loop is thus not vulnerable to having its size increased arbitrarily in order to make it too large to execute. Whilst there may be no risk of the counting loop growing too large to execute as a result of malicious manipulation, there does exist the possibility, in theory, that the loop may grow too large as a result of there being too many candidates to iterate through. This is not presently a concern, given the small number of candidates contesting on a given district in Malta, but if the system were adapted to a jurisdiction with larger numbers of candidates it may become a concern.

- **Improper Visibility Modifiers.** As a rule; all functionality that can change any aspect of the state of any of the contracts was restricted to *onlyOwner* or *onlyAllowed*, where the former restricts the calling to only the contract's owner and the latter restricts it to only a predetermined number of authorised addresses. Furthermore, functionality that needn't be exposed is set to internal or *private* as visibility.

- **tx.origin** *(See literature review for description.)* The *tx.origin* construct is avoided. As previously stated, functions able to modify the state of the contract are restricted to authorised contracts; mainly by the *onlyOwner* modifier. Functions within the Election Controller contract will revert if the caller is not the owning address, whilst the vote token and district contracts, being created by and consequently "owned" by the controller contract will revert if called from anywhere other than the controller contract. Thus, for the latter two contracts, dependence on the address initiating the transaction is avoided in favour of dependence on the address of the intermediate calling contract. To clarify: "ownership", as defined by the OpenZeppelin "Ownable" implementation, is not a transitive property (*Access Control*, n.d.).

5.4.2 Automated Analysis

Automated analysis was carried out using the tools *Mythril*, *Slither* and *SmartCheck*, talked about previously *(refer to literature review section)*. Other automated analysis tools mentioned previously were not used as they were not appropriate to the application under test. *OYENTE* and *EthIR* do not support Solidity compiler versions 0.5 and above (this project was built using compiler version 0.6.2) whereas *Securify* does not support contracts using import statements. Whilst none of the tools raised any vulnerabilities, all raised concern about the fact that the *count* and *vote* functions within the District contract feature loops that iterate for a number of iterations not

known at compile time. This may signal the need for possible refactoring in future revisions of the application.

5.5 Absolute Limits on Security & Future Threats

It is unfortunately possible that even having written hermetically secure smart contract code, the application could still be compromised in ways not necessarily in the contract writers' control. This section shall deal with certain nuances and flaws inherent in the Ethereum network and blockchain in general which could potentially be exploited.

5.5.1 51% Attacks

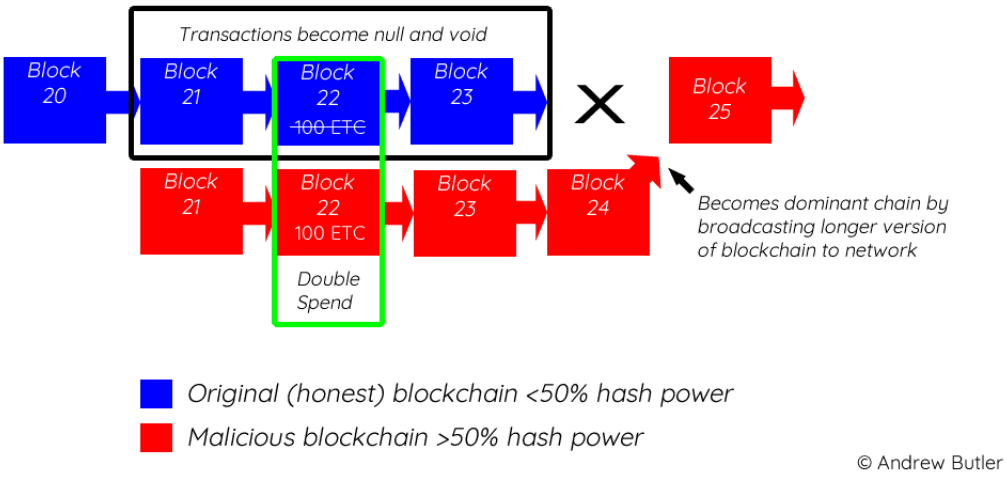

Figure 13: Illustration of 51% Attack

(Butler, 2019)

Previously it was said that deploying the application to the public Ethereum network rather than a local or private blockchain as other blockchain voting solutions have done in the past would ensure both security and transparency as it is virtually impossible to compromise the entire Ethereum network. Whilst this is, for most practical applications true, it is in fact theoretically possible for the main network to be compromised. If an actor wished to tamper with the Ethereum network or any blockchain using Proof-of-work consensus, in order to modify the transactions therein or pass fraudulent new

ones of their own, they would likely do so through a 51%. attack. (Sayeed & Marco-Gisbert, 2019) A 51% attack would entail obtaining control of a majority of the mining power on a given blockchain network that uses Proof Of Work as its method of verification. Once this is done, the malicious actor/s would be able to mine blocks faster than the rest of the network and hence forge a longer chain in the same amount of time. The mining pool in control of the 51% hashing power would normally mine blocks without broadcasting them, until it has forged a sufficiently long chain which it will then broadcast to the network. Since blockchain networks typically accept the longest broadcast chain as the real one, the malicious actors could use their majority of hashing power to compute a longer chain containing fraudulent transactions which would then be accepted by the network over the genuine one, with the subsequent blocks on the legitimate chain becoming "orphaned'. A 51% attack is for most intents and purposes unfeasible for large blockchain networks like Bitcoin and Ethereum, though they have been succesful against smaller blockchains with less hashing power, namely Bitcoin Gold and Verge, the former having been, at the time of writing, already been victim to two separate 51% attacks. This is because few, if any, individuals or groups of individuals have the resources needed to own a majority of hashing power in large networks such as these, and any financial reward derived from tampering with the networks would be dwarfed by the cost of obtaining the hashing power needed to do it. Additionally, once the hack was noticed the fraudulent funds would quickly become worthless. This lack of feasability leads to large networks such as Ethereum and Bitcoin being called *cryptoeconomically secure*(Ma, Gorzny, Mack, & Zulkoski, 2019b). However, if a national election were to be held on such a network, hostile national governments and intelligence agencies may have an interest in tampering with the network in order to manipulate the results rather than for any financial gain. The possibility of a foreign intelligence agency or state-sponsored hacking group acting as a proxy gaining control of enough mining nodes to manipulate the Ethereum network is a remote one, but must be taken into account. Whilst few, if any, powerful foreign actors are likely to have enough interest in the outcome of General Elections in Malta to attempt such an attack, if a blockchain voting system were adopted by a larger country such as the United States of America or the United Kingdom, it may become a very real threat. This especially in light of the present (at the time of writing) geopolitical tensions between certain Western governments and the People's Republic of China, which at present controls a disproportionately large share of Ethereum mining power (McIntosh, 2020). Whilst blockchains are usually touted as an immutable and permanent ledger unable to be modified by any central authority barring attacks described above, there is one well-known instance where a legitimate

central authority did in fact decide to modify transactions on the chain and were able to. The Distribured Autonomous Organisation or DAO as it's popularly known was a smart contract allowing investors to pool money in the form of Ether and receive tokens to denote their stake. Infamously, using a reentrant code bug in the contract, an unknown attacker was able to steal millions worth of Ether. Facing the likely ruin of the platform, there were many who thought that they should just hard fork the chain in order to reverse the transactions and give the investors back their Ether, effectively, running a 51 % attack on their own network. Others were opposed to this as they felt that this violated the philosophy of Ethereum and smart contracts. The code is meant to be the law, and all the attacker did was run the code in ways the programmers never intended, akin to finding a loophole in a traditional contract. Ultimately the former, perhaps more pragmatic group prevailed and enough mining nodes agreed to go forward with the hard fork. Those not in agreement kept mining blocks on the original branch of the chain where the tokens were stolen and formed what is now known as Ethereum Classic. It is important to note that the hack was purely the result of a bug in the contract code; the Ethereum network itself was never compromised and the contract ran *exactly* as it was written, merely not as it was intended. Far more troubling than the actual hack was the action taken after as it set a worrisome precedent: If those controlling a majority of the mining power in the network find the result of certain transactions unacceptable, they can and should tamper with the distributed ledger in order to reverse them (Mehar et al., 2019). This leads to speculation of what could happen if a majority of otherwise usually legitimate mining nodes decided that the result of an election held using the blockchain was unacceptable.

5.5.2 EVM Flaws

The Ethereum Virtual Machine (EVM) has been said to be ''well tested but not necessarily well documented" (Ma, Gorzny, Mack, & Zulkoski, 2019c). The first paper describing the actual opcodes of the EVM was the so-called Ethereum "Yellow Paper" (Wood, 2014), and this was later found to contain errors and omissions (Ma et al., 2019c). Whilst none of these discovered errors that have thus far resulted in exploitation, the EVM remains relatively poorly understood, with no consensus on a formal specification for it, (though competing models presently exist (Hildenbrandt et al., 2017) (Grishchenko, Maffei, & Schneidewind, 2018)) thus it remains to be seen if in the future attacks exploiting the EVM itself will be discovered. While this is a purely hypothetical scenario, if a country is to depend on the Ethereum network to hold and decide its elections, all possible attack vectors must be taken into

account, even those not presently considered feasible.

5.5.3 Malicious Miners & Transaction Ordering

Transactions in the Ethereum network are aggregated into blocks and executed by miners. Miners ultimately decide which transactions to pick up and include in the block they are currently mining. Normally this is decided by the gas price assigned to the transaction; transactions whose gas price is too low will take longer to be picked up or never be picked up at all as other transactions will take precedence. The concern however, is that a malicious actor or actors may be able to control or influence a large enough pool of miners to not accept the transactions needed to launch the necessary smart contracts or perform the necessary operations in order to sabotage the election process. As mentioned above, the malicious actors in this case would likely be state or state-sponsored agents. Another way in which miners under the influence of malicious actors could hinder or tamper with the election process is through transaction ordering. When miners aggregate transactions into blocks, there is no restriction on the ordering in which these transactions are placed in the block and subsequently called. If a transaction affects the state of a contract that is called by another transaction in the block, a race condition arises. If the miner in question has a vested interest in seeing one transaction executed before another, this can prove problematic as they are free to manipulate the order in which they execute the transactions on their own node. An obvious example of this would be a lottery contract where 2 accounts both hold a winning "ticket" and the first one to claim it gets the full amount of Ether. If a miner has 2 transactions in their block to claim the prize, one from an account they control and one from another that they don't, the miner will naturally order the transactions in such a way that theirs will be executed first. The ways in which this may effect a voting system are not obvious, and if the contracts are written well it shouldn't, not for a system where voters cast ballots for a single candidate anyway. Indeed, the order in which ballots are cast and counted in a simple 50% +1 voting setup is naturally irrelevant, however it is important in an STV voting system such as the one that this application implements. Recall the previous discussion of how counting works under STV; voters select candidates according to preference, with their vote going to their nth preference candidate in the nth round of counting if that candidate has not yet reached the quota of votes needed to be elected when the vote is counted, or transferred to their n+1 th preference in the next round of counting if they have (or they've been eliminated). The simple example below should illustrate how the order in which votes are counted can result in different outcomes:

Scenario 1	**Preference**	1	2	3	4
Order Of Counting					
0		Alice	Bob	Carl	Dan
1		Alice	Dan	Carl	Bob
2		Alice	Carl	Bob	Dan
3		Alice	Carl	Dan	Bob

Scenario 2	**Preference**	1	2	3	4
Order Of Counting					
0		Alice	Carl	Bob	Dan
1		Alice	Carl	Dan	Bob
2		Alice	Bob	Carl	Dan
3		Alice	Dan	Carl	Bob

Assuming the quota to be elected is 2 votes; the result of the counting after 4 rounds in both scenarios would be:

Scenario 1	Votes Obtained	Elected
Alice	2	Yes
Bob	0	No
Carl	2	Yes
Dan	0	No

Scenario 2	Votes Obtained	Elected
Alice	2	Yes
Bob	1	No
Carl	0	No
Dan	1	No

As can be plainly seen the result of counting certain votes before others can significantly impact the final results of the counting. This is not a bug or a flaw in the system; it is merely the nature of the STV counting process. Clearly then, a malicious miner could reorder voting transactions to make sure votes with certain preferences are counted first, and thus influence the outcome of the election. However, even absent any malicious miners manipulating the order of transactions, certain quirks of the ERC721 standard in fact lead to votes not being counted quite in the way one might expect (see next section).

5.5.4 Timestamp Manipulation

When mining a block, miners include a timestamp which will apply to all transactions within that block. Block timestamps can be to a certain degree manipulated by miners, with the Ethereum network allowing a tolerance of up to +/- 15 minutes (Wood, 2014). There is precisely one point in the application where the block timestamp affects execution:14

```
1 function _beforeTokenTransfer(address from, address to,
    uint256 tokenId) internal override {
2     require((now >= voteStart && now <= voteEnd) || (
          _msgSender() == owner()),"Voting Not Open.");
3     require(now <= voteEnd,"Election is over.");
4 }
```

Figure 14: Timestamp Dependence via the now keyword.

In Solidity, the keyword *now* represents merely the timestamp of the block the code is executing in. This function executes before every token transfer and ensure that voting has started before the transfer is performed. Hypothetically a malicious miner could modify the timestamp in blocks containing voting transactions near within 15 minutes of the end and beginning of the voting period in order to prevent those trying to vote within these short windows from doing so. The practical consequences of this are unlikely to be devastating, as those prevented from voting near the beginning of the voting period would likely just try again later beyond the timeframe in which manipulation of the timestamp would be possible[10] whilst most people will have already voted well before the 15 minute time window at the end of the voting period when a malicious miner could cause their voting transactions to be rejected. However, such an attack could potentially still shake hard-won confidence in the system if is reported on and may cause frustration of users.

5.5.5 Quantum Computing

As in all sectors dependent on strong cryptography for their functioning (our entire IT infrastructure effectively) the future advent of Quantum computing poses a potentially grave threat to blockchain. In the current situation, the 256-bit keys used by Ethereum as well as Bitcoin are projected to be secure

[10] In such a way that it would be accepted by the Ethereum network that is. Attempting to modify the block timestamp beyond the 15 minute tolerance will cause the block to be rejected.

Security level	Symmetric	ECC	DSA/RSA	Protects to year
80	80	160	1024	2010
112	112	224	2048	2030
128	128	256	3072	2040
192	192	384	7680	2080
256	256	512	15360	2120

Figure 15: Comparable Key Sizes

(Brown, 2009)

until at least the year 204015, if one takes into account only "classical" (non-quantum) computers.

Whilst presently, quantum computers are still considered a highly theoretical area and are not at a stage where they could pose a serious threat to security, there presently already exist two quantum algorithms that could potentially be used to render present asymmetric cryptography and hashing methods obsolete. Namely *Shor's Algorithm* for prime factorisation that could be able to crack asymmetric keys presently considered secure in very small amounts of time ((Aggarwal, Brennen, Lee, Santha, & Tomamichel, 2017) projects an estimated 10 minutes to crack current EC keys used by Bitcoin) ((Proos & Zalka, 2003)). *Grover's Algorithm* is a quantum method that could potentially improve the efficiency of hash cracking, however not as dramatically. The threats to blockchain from quantum computing are twofold ((*Binance*, 2018)); the first is obviously that whilst a classical computer may take decades or even millenia to crack a private key through bruteforce methods, a quantum computer is theorised to be able to do this in a fraction of that time, meaning individual accounts would no longer be secure should the technology become widespread in the future. Another less obvious implication is that if a single miner were able to acquire a quantum computer before others, they could use it to quickly acquire ¿51% of the hashing power and commit the attacks talked about previously. Of course, if the blockchain were to survive the advent of quantum computing long enough for all miners to switch to quantum computers, this concern would be moot. The speed of ASIC miners currently in use is considered to be comparable to the estimated hashing speed of quantum computers likely to become available in the

coming years, so quantum computing is not as large a threat to the PoW aspect of blockchain as it is to the transaction signing aspect ((Aggarwal et al., 2017)). As with other attacks considered practically "unfeasible", the threat of quantum computing-powered attacks may not be so unfeasible with state actors in the mix. It is not outside the realm of possibility that states may be carrying out quantum computing research that is not being disclosed, in a sort of cryptographic "arms race". Certainly, for the reasons mentioned above an intelligence agency equipped with quantum computing technology capable of cracking current encryptions would have an immense advantage over their rivals. If a blockchain voting solution were adopted by a country, a well-resourced state actor in possession of secret quantum computing technology could very well be able to tamper with the election process.

5.6 Drawbacks & Criticism

5.6.1 Ether Cost

As clearly shown above (5.2), the system in its present state is unreasonably costly to run on the public Ethereum network, even for a small voting population such as the Republic Of Malta. The system's functionality having been verified however, it could of course be simply deployed to a private Ethereum network where this would not be a problem. Recall however, that a central aim of this work was to create a votation system that could be used on the public Ethereum network. As previously explained, a blockchain voting system implemented on a private blockchain partially defeats the purpose of using a blockchain in the first place as opposed to a traditional centralised system.

5.6.2 Centralisation

In our motivation for this application's development, it was said that blockchain would be ideal for a voting system as no single entity would have control over the counting process. This is broadly true in the sense that counting is conducted automatically by immutable smart contract code, and once the process is begun it cannot be stopped, nor can anyone tamper with the results. However, a significant degree of centralisation still exists in terms of who is able to launch an election and trigger counting and with how users interact with the smart contracts.

```
1    import "@openzeppelin/contracts/access/Ownable.sol";
2    contract ElectionController is Ownable{...
3    /////Owned by address that deploys the system contracts.
4    import "@openzeppelin/contracts/access/Ownable.sol";
5    contract District is Ownable{...
6    /////Owned by ElectionController contract.
7    import "@openzeppelin/contracts/access/Ownable.sol";
8    contract VoteToken is ERC721, Ownable{...
9    /////Owned By Owned by ElectionController contract.
```

Figure 16: Ownership Of Contracts

Naturally, running transactions on the contract cannot be permitted for everyone. All contracts [11] in the application inherit from the Ownable contract from OpenZeppelin (*Access Control*, n.d.), which, by default, sets the owner of any contract that inherits from it to the address which created it. This contract allows contract writers to mark contract functions with the custom modifier *onlyOwner* which will cause any function called by any address other than the one currently set as the owner to revert. The owner for the District and VoteToken contracts would simply be the ElectionController contract as it is the contract responsible for deploying them automatically when an election is launched. In each of the contracts, all functions that are able to modify the state of the contract [12] are marked as onlyOwner to ensure no one can tamper with the data stored in the contract or launch unauthorised operations (in contrast, functions used to retrieve data are able to be called by everyone, to ensure transparency). However, the fact remains that the person/s with access to the account that launched the Election Controller contract are the sole people with the ability to launch and end elections. The contract is written in such a way that even the owner is not able to destroy any data and the actual potential for misuse if the owning account were ever compromised is quite low. The most that a malicious actor who was able to get access to the owning account would be able to do would be to launch a bogus election, which would do nothing except waste the Ether on the account in gas costs, as the public would not accept the results in any case. The owner cannot for example arbitrarily transfer tokens or destroy the contract; the code simply does not allow this. They would not even be able to prematurely end an election since that function will revert even for the contract owner if the voting period has not ended. Rather, the concern lies in what would happen if access to the account was somehow

[11] save for the Factory contracts. See Appendices.

[12] Barring the unnecessary ERC721 functions overridden with empty functions in VoteToken.

lost to the account. As previously said, the Election Controller contract is designed to be a persistent hub for both holding elections and storing their results; serving as a distributed database of election results to be reused on subsequent elections. If access were lost to the owning account, one would lose the ability to retrieve and verify previous election results (though technically the data could still be read directly on the blockchain, this would be extremely cumbersome). This then also begs the question who should be allowed access to the private key needed to access the controlling account ? It could perhaps be placed in control of the president of the republic in the case of Malta, since this is the person who is vested with the constitutional power to dissolve parliament and call an election.

As previously said, voters will not in fact receive their vote tokens on their own Ethereum wallets for practical reasons. Rather a vote token will be minted for them by means of an off-chain program running on a server minting the token for them and placing it in the balance of the Election Controller contract, with the voters merely using data that only they will know to prove their "ownership" of the vote token and authorising the contract to use it on their behalf in order to vote. Therefore, voters are in fact completely reliant on a centralised traditional web application in order to both obtain and use their votes, and their votes will always belong not to their own addresses but to those of contracts they do not control. In practice it would appear that there is a fairly large degree of centralisation in this supposedly decentralised application; to summarise:

- A single account address has sole ownership of the central controlling contract and is solely capable of launching and ending elections.
- Minting of vote tokens and voting is done through a centralised web application; thus the application has a single point of failure vulnerable to DDoS attacks.
- Voters are never actually in direct possession of their vote tokens; the single monolithic contract holds and uses them on their behalf.

5.6.3 Deterministic Counting Order

The current (at the time of writing) OpenZeppelin ERC721 implementation allows smart contracts an interface with which they may access all the tokens owned by an address as a "list". This is done by first retrieving the balance of a given address, and then, using a for loop and the "tokenOfOwnerByIndex" function, accessing the tokens in the order that they were added to the address' balance (C, Figure: 17). In reality this is not quite a simple list or any such linear data structure, as the operations required to shift the elements when a token at index n $(0 < n < length - 1)$ is spent or otherwise removed would be expensive. Instead a self-shifting linear data structure is simulated through use of mappings using Openzeppelin's EnumerableSet library (OpenZeppelin, 2020a). A somewhat peculiar and not immediately obvious way in which this pseudo-list operates is when it comes to adjusting itself after a token has been removed from a user's balance. When a token is removed via the "transfer" function of ERC721(C, Figure: 17), the remove and add functions in the EnumerableSet library are called with the sender and recipient (respectively) as parameters along with the 256-bit ID of the token. The add function code is relatively uninteresting (C, Figure: 19), as it

simply pushes the value to the end of an internal array as one would expect. Therefore, one might expect that since when a vote is cast this vote token is merely appended to the end of an array, that the votes would always be counted in the order they were cast. The remove (C, Figure: 20) function however has some perhaps unexpected behaviour that will affect the way in which tokens are counted. When a token is removed from an address' balance, the removed token ID inside the internal array storing all of the address' tokens is replaced with the value at last index of the array, with the element at the last index of the array then being popped off, ensuring the array always has length equal to the number of tokens owned by the address. The previously last vote token in the array will thus then be counted next. That votes are counted in the order they are cast is of course neither a requirement nor even a desirable property of an STV voting system, as this would allow manipulation of the election, however all this is to say that whilst votes are not in fact counted in the order they are cast, even absent manipulation of the order of transactions, there does in fact exist a deterministic relationship between the order in which votes are cast, and this is an inherent flaw in the system, which could potentially lead to strategic voting in order to reduce or increase the chances of a certain candidate getting a seat. Recall that in our implementation, votes are immediately transferred to the specified candidates when they are cast rather than being counted as a batch so this is not a concern in that regard. However, recall also that batch counting of votes is used when reallocating votes from eliminated candidates.

6 Conclusion

Notwithstanding the caveats and drawbacks of the system discussed above, it is felt that the implemented solution satisfied many of the criteria to a reasonable degree. That is:

- A voting solution in the form of a distributed application on an Ethereum network was built.
- The implemented solution was shown to be able to collect and correctly count votes as per the rules of Single Transferable Voting in a district-based system.
- The application was analysed and shown to be robust against common exploitation vectors and free of known insecure code patterns and bugs leading to unintended operation.
- The implemented application was tested using quantities of data reflective of a real-world scenario and was found to be able to run to completion correctly.

It did not, however fulfill the crucial criterion of being able to be deployed on the public Ethereum network.
Potential future work could include:

- Investigating whether it would be possible to refactor the code in such a way as to make it more efficient in terms of gas costs in order to bring the financial costs down to more reasonable levels.
- Pursuant to the above; it may be worth investigating whether any of the other languages in development for Ethereum smart contract development such as Vyper could lead to more efficient code.
- Investigating whether it would be possible to address the centralisation issues previously talked about in such a fashion as to still provide a practical application.
- Further auditing and penetration testing.
- Development of a full-featured suite of front-end apps to enable simple management of the application by authorised personnel as well as use by the public.

- Smart contracts do not presently perform calculation of additional seats allocated as per proportionality principle, nor do they perform reallocation of a candidates votes after counting when that candidate has won a seat on two districts and must renounce one. This is not to mean that the functionality of the system is incomplete; rather that it implements a standard STV system without the amendments particular to the electoral system of Malta or any other jurisdiction. Future iterations of this application could include this and other functionality added in a bespoke fashion particular to the jurisdiction where it is to be used.

In conclusion, it is felt that the application serves as an adequate proof of concept. In comparison to other e-voting solutions discussed *(Literature Review, Section 3.5)*, both blockchain and centralised, the system developed here offers innovation in that it:

- Implements an STV voting system rather than a simple first-past-the-post voting system.
- Solves the problem of voter anonymity vs. voter authentication in a robust way that is applicable in a real-world application.
- Designed to allow voting without the voter having to set up or use any kind of wallet software or know how to do anything beyond log into a web page.
- Does not make use of any special hardware, such as the e-voting system used in Estonia. (Yavuz et al., 2018)

Regarding the demonstrated financial unfeasibility of deployment on a public network (barring a colossal drop in the price of Ether), it serves as an illustration of the practical limitations of Ethereum for large-scale applications, at least as far as the public network is concerned. Given the excessive costs of conducting operations involving relatively large numbers of transactions such as this on the public network, an item of potential future discussion could be the trade-off between the relatively low financial cost of using a private blockchain network and the inherently lesser assurances of transparency, accountability and security, contrasted with the high financial cost but guaranteed transparency, accountability and security that comes with use of a public blockchain network.

References

Access control. (n.d.). OpenZeppelin. Retrieved from https://docs.openzeppelin.com/contracts/2.x/access-control

Aggarwal, D., Brennen, G. K., Lee, T., Santha, M., & Tomamichel, M. (2017). Quantum attacks on bitcoin, and how to protect against them. *arXiv preprint arXiv:1710.10377*.

All about stv and fpp. (n.d.). Retrieved from http://www.localcouncils.govt.nz/lgip.nsf/wpg_URL/About-Local-Government-Participate-in-Local-Government-All-about-STV-and-FPP

Bashir, I. (2017). *Mastering blockchain.* Packt Publishing Ltd.

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., ... others (2016). Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 acm workshop on programming languages and analysis for security* (pp. 91–96).

Binance. (2018, May). Binance. Retrieved from https://www.binance.vision/blockchain/quantum-computers-and-cryptocurrencies

Blenkinsop, C. (2019, Jul). *Nonfungible tokens, explained.* Retrieved from https://cointelegraph.com/explained/non-fungible-tokens-explained

Brown, D. (2009). Standards for efficient cryptography, sec 1: elliptic curve cryptography. *Released Standard Version*, *1*.

Buterin, V., Wood, G., et al. (2014). *Ethereum whitepaper.* Ethereum Foundation. Retrieved from https://ethereum.org/en/whitepaper/

Butler, A. (2019, Jan). *Ethereum classic attacked! how does the 51% attack occur?* Retrieved from https://hackernoon.com/ethereum-classic-attacked-how-does-the-51-attack-occur-a5f3fa5d852e

Committee, U. S. S. I., et al. (2019). Russian efforts against election infrastructure. *edited by United States Senate Intelligence Committee*.

Dafflon, J., & Baylina, J. (2017, Nov). *Eip-777: Erc777 token standard.* Retrieved from https://eips.ethereum.org/EIPS/eip-777

dexaran@ethereumclassic.org, D. (2017, Mar). *Erc223 token standard · issue 223 · ethereum/eips.* Retrieved from https://github.com/ethereum/eips/issues/223

Dika, A., & Nowostawski, M. (2018). Security vulnerabilities in ethereum smart contracts. In *2018 ieee international conference on internet of things (ithings) and ieee green computing and communications (greencom) and ieee cyber, physical and social computing (cpscom) and ieee smart data (smartdata)* (pp. 955–962).

Dossa, A., Ruiz, P., Gosselin, S., & Vogelsteller, F. (2018, Dec). *Erc-1643:*

Document management standard · issue 1643 · ethereum/eips. Retrieved from https://github.com/ethereum/EIPs/issues/1643

Dossa, A., Ruiz, P., Vogelsteller, F., & Gosselin, S. (2018a, Sep). *Erc 1400: Security token standard.* Retrieved from https://github.com/ethereum/EIPs/issues/1411

Dossa, A., Ruiz, P., Vogelsteller, F., & Gosselin, S. (2018b, Nov). *Erc 1594: Core security token standard · issue 1594 · ethereum/eips.* Retrieved from https://github.com/ethereum/eips/issues/1594

Feist, J., Grieco, G., & Groce, A. (2019). Slither: A static analysis framework for smart contracts. In *2019 ieee/acm 2nd international workshop on emerging trends in software engineering for blockchain (wetseb)* (p. 8-15).

Gosselin, S., Dossa, A., Vogelsteller, F., & Ruiz, P. (2018, Dec). *Erc-1644: Controller token operation standard · issue 1644 · ethereum/eips.* Retrieved from https://github.com/ethereum/EIPs/issues/1644

Grishchenko, I., Maffei, M., & Schneidewind, C. (2018). A semantic framework for the security analysis of ethereum smart contracts. In *International conference on principles of security and trust* (pp. 243–269).

Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., & Rosu, G. (2017). *Kevm: A complete semantics of the ethereum virtual machine* (Tech. Rep.).

Hjálmarsson, F. ., Hreiarsson, G. K., Hamdaqa, M., & Hjálmtỳsson, G. (2018). Blockchain-based e-voting system. In *2018 ieee 11th international conference on cloud computing (cloud)* (pp. 983–986).

How malta votes: An overview. (n.d.). Retrieved from https://www.um.edu.mt/projects/maltaelections/stvsystem/howmaltavotes

(n.d.). Retrieved from https://mythx.io/detectors/

(n.d.). Retrieved from https://tool.smartdec.net/knowledge

Jefferson, D. (2018). The myth of "secure" blockchain voting. *Verified Voting*.

Jefferson, D., Buell, D., Skoglund, K., Kiniry, J., & Greenbaum, J. (2019). *What we don't know about the voatz "blockchain" internet voting system.*

Kaspersky, E. (2016). Cyber security case study competition-kaspersky. *The Economist*, *15*.

Kirby, K., Masi, A., & Maymi, F. (2016). Votebook. a proposal for a blockchain-based electronic voting system. *The Economist*, *6*.

Kobie, N. (2015, March). Why electronic voting isn't secure – but may be safe enough. *The Guardian*. Retrieved from https://www.theguardian.com/technology/2015/mar/30/why-electronic-voting-is-not-secure

Kshetri, N., & Voas, J. (2018). Blockchain-enabled e-voting. *IEEE Software*, *35*(4), 95–99.

Liu, J., & Liu, Z. (2019). A survey on security verification of blockchain smart contracts. *IEEE Access*, *7*, 77894–77904.

Loibl, A., & Naab, J. (2014). Namecoin. *namecoin. info*.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 acm sigsac conference on computer and communications security* (pp. 254–269).

Ma, R., Gorzny, J., Mack, O., & Zulkoski, E. (2019a). *Fundamentals of smart contract security (kindle edition)*. Momentum Press, LLC. Retrieved from https://books.google.pl/books?id=k-wevgEACAAJ

Ma, R., Gorzny, J., Mack, O., & Zulkoski, E. (2019b). *Fundamentals of smart contract security (kindle edition)*. Momentum Press, LLC. Retrieved from https://books.google.pl/books?id=k-wevgEACAAJ

Ma, R., Gorzny, J., Mack, O., & Zulkoski, E. (2019c). *Fundamentals of smart contract security (kindle edition)*. Momentum Press, LLC. Retrieved from https://books.google.pl/books?id=k-wevgEACAAJ

McIntosh, R. (2020, Mar). *Does china control bitcoin and ethereum?: Finance magnates.* Retrieved from https://www.financemagnates.com/cryptocurrency/news/does-china-control-bitcoin-and-ethereum/

Mearian, L. (2019, Aug). *Why blockchain-based voting could threaten democracy.* Retrieved from https://www.computerworld.com/article/3430697/why-blockchain-could-be-a-threat-to-democracy.html

Mehar, M. I., Shier, C. L., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., ... Laskowski, M. (2019). Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. *Journal of Cases on Information Technology (JCIT)*, *21*(1), 19–32.

Moura, T., & Gomes, A. (2017). Blockchain voting and its effects on election transparency and voter confidence. In *Proceedings of the 18th annual international conference on digital government research* (pp. 574–575).

Nakamoto, S. (2019). *Bitcoin: A peer-to-peer electronic cash system* (Tech. Rep.). Manubot.

Norden, L. D., & Famighetti, C. (2015). *America's voting machines at risk*. Brennan Center for Justice at New York University School of Law.

OpenZeppelin. (2020a, Sep). *Openzeppelin/openzeppelin-contracts.* Retrieved from https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/EnumerableSet.sol

OpenZeppelin. (2020b, Sep). *Openzeppelin/openzeppelin-contracts.* Retrieved from https://github.com/OpenZeppelin/openzeppelin

-contracts/blob/master/contracts/token/ERC721/ERC721.sol

Osgood, R. (2016). The future of democracy: Blockchain voting. *COMP116: Information security*, 1–21.

Praitheeshan, P., Pan, L., Yu, J., Liu, J., & Doss, R. (2019). Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*.

Proos, J., & Zalka, C. (2003). Shor's discrete logarithm quantum algorithm for elliptic curves. *arXiv preprint quant-ph/0301141*.

Rosenfeld, M. (2012). Overview of colored coins. *White paper, bitcoil. co. il*, *41*, 94.

Ruiz, P., Vogelsteller, F., Dossa, A., & Gosselin, S. (2018, Sep). *Erc 1410: Partially fungible token standard · issue 1410 · ethereum/eips.* Retrieved from https://github.com/ethereum/eips/issues/1410

Sameeh, T. (2019, Nov). *Ico basics - the difference between security tokens and utility tokens.* Retrieved from https://www.cointelligence.com/content/ico-basics-security-tokens-vs-utility-tokens

Sayeed, S., & Marco-Gisbert, H. (2019). Assessing blockchain consensus and security mechanisms against the 51% attack. *Applied Sciences*, *9*(9), 1788.

Shirole, M., Darisi, M., & Bhirud, S. (2020). Cryptocurrency token: An overview. *IC-BCT 2019*, 133–140.

Suwito, M. H., & Dutta, S. (2019). Verifiable e-voting with resistance against physical forced abstention attack. In *2019 international workshop on big data and information security (iwbis)* (pp. 85–90).

Szabo, N. (1998). Secure property titles with owner authority. *Online at http://szabo. best. vwh. net/securetitle. html*.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018). Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain* (pp. 9–16).

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., & Vechev, M. (2018). Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 acm sigsac conference on computer and communications security* (p. 67–82). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3243734.3243780 doi: 10.1145/3243734.3243780

Votewatcher - the world's most transparent voting machine. (n.d.). Retrieved from http://votewatcher.com/

William Entriken, D. S., Evans, J., Shirley, D., & Sachs, N. (2018, Jan). *Eip-721: Erc-721 non-fungible token standard.* Retrieved from https://eips.ethereum.org/EIPS/eip-721

Witek Radomski, A. C., Cooke, A., Castonguay, P., Therien, J., & Binet, E. (2018, Jun). *Eip-1155: Erc-1155 multi token standard.* Retrieved from https://eips.ethereum.org/EIPS/eip-1155

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*.

Wüst, K., & Gervais, A. (2018). Do you need a blockchain? In *2018 crypto valley conference on blockchain technology (cvcbt)* (pp. 45–54).

Yaga, D., Mell, P., Roby, N., & Scarfone, K. (2019). Blockchain technology overview. *arXiv preprint arXiv:1906.11078*.

Yavuz, E., Koç, A. K., Çabuk, U. C., & Dalkılıç, G. (2018). Towards secure e-voting using ethereum blockchain. In *2018 6th international symposium on digital forensic and security (isdfs)* (pp. 1–7).

Appendices

A Smart Contract Code

```
1  //ElectionController.sol
2  pragma solidity ^0.6.0;
3  pragma experimental ABIEncoderV2;
4
5  import "@openzeppelin/contracts/access/Ownable.sol";
6  import "./VoteToken.sol";
7  import "./Shared.sol";
8  import "./District.sol";
9  import "./DistrictFactory.sol";
10 import "./VoteTokenFactory.sol";
11 import "@openzeppelin/contracts/math/SafeMath.sol";
12
13
14 contract ElectionController is Ownable{
15     using SafeMath for uint64;
16
17     Shared.Election[] public elections;
18     uint8 public districtsNo;
19     DistrictFactory internal districtFactory;
20     VoteTokenFactory internal voteTokenFactory;
21
22     constructor(uint8 _districtsNo, DistrictFactory
           _districtFactory, VoteTokenFactory _voteTokenFactory)
           Ownable() public {
23         districtsNo = _districtsNo;
24         districtFactory = _districtFactory;
25         voteTokenFactory = _voteTokenFactory;
26     }
27
28     function endElection(uint8 _electionNo) onlyOwner
           external{
29         //Shared.Election storage election = elections[
               _electionNo]; //storage in order to copy by
               reference.
30         require(now > elections[_electionNo].voteEnd,"Voting
               Not Over Yet");
31         uint8 partyWithMostVotesIndex = 254;
32         uint64 mostVotes = 0;
33
34         for(uint8 i = 1; i <= districtsNo; i++){
35
36             require(elections[_electionNo].districtContracts[
                   i].counted() == true, string(abi.encodePacked(
```

```
                "Counting not yet done for district: ",i)));
37
38              for(uint8 j = 0; j < elections[_electionNo].
                    numberOfPartiesContesting ;j++){
39
40                  uint64 _votes = elections[_electionNo].
                        districtContracts[i].party1stCountVotes(j)
                        ;
41
42                  elections[_electionNo].party1stCountVotes[j]
                        = uint64(elections[_electionNo].
                        party1stCountVotes[j].add(_votes));
43                  if(elections[_electionNo].party1stCountVotes[
                        j] > mostVotes){
44                      mostVotes = elections[_electionNo].
                            party1stCountVotes[j];
45                      partyWithMostVotesIndex = j;
46                  }
47              }
48          }
49
50          elections[_electionNo].partyWithMostVotesIndex =
                partyWithMostVotesIndex;
51          elections[_electionNo].concluded = true;
52      }
53
54      function setCounted(uint8 _electionNo, uint8 _districtNo,
            bool _counted) public onlyOwner{
55          require(elections[_electionNo].districtContracts[
                _districtNo].countVotesRan(),"Counting not yet
                done.");
56          elections[_electionNo].districtContracts[_districtNo
                ].setCounted(_counted);
57      }
58
59      function endDistrict(uint8 _electionNo,uint8 _districtNo)
            onlyOwner external{
60          elections[_electionNo].districtContracts[_districtNo
                ].countVotes();
61      }
62
63      function launchDistrict(uint8 _electionNo,uint8
            _districtNo, Shared.Candidate[] memory candidates)
            public onlyOwner{
64          District d = districtFactory.create(_districtNo,
                candidates);
65          elections[_electionNo].voteToken.setApprovalForAll(
                address(d),true);
66          elections[_electionNo].districtContracts[_districtNo]
```

```
            = d;
67          elections[_electionNo].voteToken.setAllowed(address(d
                ));
68          elections[_electionNo].districtContracts[_districtNo
                ].setVoteToken(elections[_electionNo].voteToken);
69
70      }
71
72      event ElectionLaunched(uint electionNo, uint time, uint
            voteStart, uint voteEnd, uint8
            numberOfPartiesContesting, string name);
73      function launchElection (
74              string memory _name,
75              uint _voteStart,
76              uint _voteEnd,
77              uint8 _numberOfPartiesContesting
78              ) public onlyOwner{
79
80          VoteToken _voteToken = voteTokenFactory.create(_name,
                _voteStart,_voteEnd);
81          elections.push(Shared.Election({name: _name,
                voteStart: _voteStart, voteEnd: _voteEnd,
                voteToken: _voteToken, concluded:false,
                partyWithMostVotesIndex:255,
                numberOfPartiesContesting:
                _numberOfPartiesContesting}));
82
83          require(elections.length > 0,"Election Launch Failed.
                ");
84
85          emit ElectionLaunched(elections.length-1, now,
                _voteStart, _voteEnd, _numberOfPartiesContesting,
                _name);
86      }
87
88      function vote(uint8 _electionNo, uint256 _voteTokenID,
            address[] memory _preferences) public onlyOwner {
89          uint8 districtNo = elections[_electionNo].voteToken.
                district(_voteTokenID);
90          require(districtNo != 0,"Invalid Vote Token.");
91          District _district = elections[_electionNo].
                districtContracts[districtNo];
92
93          _district.vote(_voteTokenID,_preferences);
94      }
95
96      function mintVote(uint256 _tokenId, uint8 _districtNo,
            uint32 _electionNo) public onlyOwner {
97          elections[_electionNo].voteToken.mint(address(this),
```

```
            _tokenId, _districtNo);
98          elections[_electionNo].districtContracts[_districtNo
            ].incrementCastVotes();
99      }
100
101     function balanceOf(address _address, uint16 _election)
            public view returns(uint256){
102         return elections[_election].voteToken.balanceOf(
            _address);
103     }
104
105     function getDistrictCandidateAddresses(uint8 _election,
            uint8 _districtNo)  public view returns(address[]
            memory){
106         address[] memory addresses = elections[_election].
            districtContracts[_districtNo].
            getCandidateAddresses();
107         return addresses;
108     }
109
110     function getDistrictContract(uint8 _election, uint8
            _districtNo) public view returns(District){
111         return elections[_election].districtContracts[
            _districtNo];
112     }
113
114 }
```

```
1  //District.sol
2  pragma solidity ^0.6.0;
3  pragma experimental ABIEncoderV2;
4
5  import "./VoteToken.sol";
6  import "./Shared.sol";
7  import "@openzeppelin/contracts/access/Ownable.sol";
8  import "@openzeppelin/contracts/math/SafeMath.sol";
9  import "@openzeppelin/contracts/token/ERC721/ERC721Holder.sol
       ";
10 import "./Untransferable.sol";
11
12 contract District is Ownable,ERC721Holder{
13     VoteToken public voteToken;
14     Untransferable public untransferable;
15     uint8 public districtNumber;
16     uint public castVotes;
17     uint8 constant public seats = 5;
18     uint8 public seatsRemaining = 5 ;
19     uint32 public quota;
20     uint8 public candidatesCount;
```

```
21
22      using SafeMath for uint64;
23
24      mapping(address => Shared.Candidate) public candidates;
25      address[] public candidateAddresses;
26
27      mapping(uint8 => uint64) public party1stCountVotes;
28
29      bool public counted = false;
30
31      address[] notElected;
32      address[] elected;
33
34      event Vote(address _voter, uint8 districtNo, uint
            castVotes);
35
36      constructor(uint8 _districtNumber, Shared.Candidate[]
            memory _candidates) Ownable() ERC721Holder() public{
37          districtNumber = _districtNumber;
38          castVotes = 0;
39          candidatesCount = uint8(_candidates.length);
40          untransferable = new Untransferable();
41
42          for(uint8 i = 0; i < candidatesCount; i++){
43              candidateAddresses.push(_candidates[i]._address);
44              candidates[_candidates[i]._address] = _candidates
                    [i];
45          }
46      }
47
48      function incrementCastVotes() public {
49          castVotes += 1;
50      }
51
52      function setEliminated(address _candidateAddress, bool
            _eliminated) public onlyOwner{
53          candidates[_candidateAddress].eliminated =
                _eliminated;
54      }
55
56      function setCounted(bool _counted) public onlyOwner{
57          counted = _counted;
58      }
59
60      function onERC721Received(address operator, address from,
            uint256 tokenId, bytes memory data) public override
            returns (bytes4) {
61
62          uint8 pref = 1;
```

```
63
64          while(voteToken.preferences(tokenId,pref) != address
                (0)){
65
66              address candidateAddress = voteToken.preferences(
                    tokenId,pref);
67              if(pref==1)
68                  party1stCountVotes[candidates[
                        candidateAddress].party] = uint64(
                        party1stCountVotes[candidates[
                        candidateAddress].party].add(1));
69                  /*
70                  Party 1st count votes balance increases
                        regardless of whether the vote actually
                        ends up at the 1st preference candidate or
71                  is transferred to another candidate,
                        potentially of a different party.
72                  */
73              if(uint32(voteToken.balanceOf(candidateAddress))
                    < getQuota()){
74                  voteToken.transferFrom(address(this),
                        candidateAddress,tokenId);
75                  break;
76              }
77              else
78                  ++pref;
79          }
80
81          if(voteToken.ownerOf(tokenId)==address(this))
82              voteToken.transferFrom(address(this),address(
                    untransferable),tokenId);
83
84          return super.onERC721Received(operator,from,tokenId,
                data);
85
86      }
87
88      function setVoteToken(VoteToken _voteToken) public
            onlyOwner{
89          voteToken = _voteToken;
90      }
91
92      function getCandidateAddresses() external view returns(
            address[] memory){
93          return candidateAddresses;
94      }
95
96      function vote(uint256 tokenId, address[] calldata
            _preferences) external onlyOwner {
```

```
97          require(voteToken.district(tokenId) == districtNumber
                ,"Incorrect District");
98          if(quota==0)
99              quota = getQuota();
100
101         for(uint8 i = 0; i < _preferences.length; i++)
102             require(candidates[_preferences[i]]._address !=
                    address(0), "Voted for candidate not from this
                     district. Vote Rejected.");
103
104         voteToken.setVotePreferences(tokenId, _preferences);
105         voteToken.safeTransferFrom(msg.sender,address(this),
                tokenId);
106
107         emit Vote(msg.sender, districtNumber, castVotes);
108     }
109
110
111     function getQuota() public view returns(uint32){
112         return uint32((castVotes/(seats + 1)) + 1);
113     }
114
115
116     function getAllElected() external view returns(address[]
            memory){
117         require(countVotesRan,"Not Counted Yet");
118         return elected;
119     }
120
121     function getNotElected() external view returns(address[]
            memory){
122         require(countVotesRan,"Not Counted Yet");
123         return notElected;
124     }
125
126     function transferAllBack(address _from) public onlyOwner
            {
127         voteToken.transferAllBack(_from);
128     }
129
130     bool public countVotesRan = false;
131     function countVotes() external onlyOwner{
132
133         for (uint8 j = 0; j < candidatesCount; j++){
134
135             if(voteToken.balanceOf(candidateAddresses[j]) >=
                    quota){
136                 candidates[candidateAddresses[j]].elected =
                        true;
```

```
                elected.push(candidateAddresses[j]);
                --seatsRemaining;

                if(seatsRemaining == 0)
                    break;
            }
            else
                notElected.push(candidateAddresses[j]);
        }

        countVotesRan = true;
    }

    function kill() external onlyOwner {
        selfdestruct(msg.sender);
    }

}
```

```
//VoteToken.sol
pragma solidity ^0.6.0;
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721Holder.sol
    ";
import "./District.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";

contract VoteToken is ERC721, Ownable{
    mapping(uint256 => uint8) public district; //maps token
        ID to District voter it is issued to is on.
    mapping(uint256 => mapping(uint8=>address)) public
        preferences; //maps token ID to the preferences
        selected by the voter.
    mapping(uint256 => mapping(address=>uint8)) public
        addressToPreferences; //maps tokenID to a mapping of a
         selected candidate to the voter's selected preference
        .

    uint public voteStart; //Time (in seconds since epoch)
    uint public voteEnd;
    uint64 public votesTotal;

    using SafeMath for uint64;
    //Much of the ERC721 functionality will be restricted to
        only the District contracts.
    mapping(address => bool) internal allowedContracts;
    modifier OnlyAllowed {
        require(allowedContracts[msg.sender] == true,"Calling
```

```
            Address not in list of allowed contracts.");
23          _;
24      }
25
26      constructor(string memory __name, string memory __symbol,
            uint _voteStart, uint _voteEnd) ERC721(__name,
            __symbol) Ownable() public {
27          voteStart = _voteStart;
28          voteEnd = _voteEnd;
29          allowedContracts[address(this)] = true;
30
31      }
32
33      //Called For Each District Contract Address.
34      function setAllowed(address _address) external onlyOwner
            {
35          allowedContracts[_address] = true;
36      }
37
38      /*
39      Ensures No tokens can be transferred (ie: voting) before
            or after voting period ends
40      */
41      function _beforeTokenTransfer(address from, address to,
            uint256 tokenId) internal override {
42          require((now >= voteStart && now <= voteEnd) || (
                _msgSender() == owner()),"Voting Not Open.");
43          require(now <= voteEnd,"Election is over.");
44      }
45
46      /* Maps a given vote Token ID to a mapping of the voter's
            preferences (1..N) to the addresses of the
            corresponding candidates. */
47      function setVotePreferences(uint256 _tokenId, address[]
            calldata _preferences) external {
48          _isApprovedOrOwner(_msgSender(), _tokenId);
49
50          for(uint8 i = 1; i <= _preferences.length; i++){
51              preferences[_tokenId][i] = _preferences[i-1];
52              addressToPreferences[_tokenId][_preferences[i-1]]
                    = i;
53          }
54      }
55
56      /* Utilises vanilla ERC721 mint functionality but adds
            the restrictions that the mint is rejected if the
            recipient has a balance larger than 0;
57      As every voter may only be issued one vote and that only
            the owner of the VoteToken contract (the Election
```

```
        Controller contract) may call it.
58      Additionally, it also assigns the number of the electoral
            district in which the voter resides as metadata*/
59      function mint(address _to, uint256 _tokenId, uint8
            _districtNo) public onlyOwner{
60          //require(balanceOf(_to) == 0,"Recipient Already has
                a vote. (balance non 0)"); //no one may have more
                than 1 vote.
61
62          _mint(_to , _tokenId);
63          district[_tokenId] = _districtNo;
64          votesTotal++;
65      }
66
67      /*
68      May be called only by District contracts (onlyAllowed).
            Used in order to redistribute votes from candidates
            that are eliminated in the process of counting.
69      Has the additional restriction that each token must "
            belong" to the District that is attempting to seize it
            .
70       */
71      function transferAllBack(address _from) external
            OnlyAllowed {
72          uint8 _districtNo = District(msg.sender).
                districtNumber();
73          uint balance = balanceOf(_from);
74
75          for(uint32 i = 0; i < balance; i++){
76              uint256 tokenId = tokenOfOwnerByIndex(_from,i);
77              require(district[tokenId] == _districtNo,"
                    Attempting to seize token from another
                    district.");
78              transferFrom(_from,msg.sender,tokenId);
79          }
80      }
81
82      /*Proxy for the vanilla transferFrom function of ERC721,
            but restricts it to only allowed callers (District
            contracts))*/
83      function transferFrom(address from, address to, uint256
            tokenId) public OnlyAllowed override {
84          super.transferFrom(from,to,tokenId);
85      }
86
87      /*Proxy for the vanilla safeTransferFrom function of
            ERC721, but restricts it to only allowed callers (
            District contracts))*/
88      function safeTransferFrom(address from, address to,
```

```
        uint256 tokenId) public OnlyAllowed override {
89          super.safeTransferFrom(from,to,tokenId);
90
91      }
92
93
94      function kill() external onlyOwner {
95          selfdestruct(msg.sender);
96      }
97  }
```

```
1   //Shared.sol
2   pragma solidity ^0.6.0;
3   import "./VoteToken.sol";
4   import "./District.sol";
5
6
7   library Shared {
8       uint8 constant parties = 2;
9       struct Candidate{
10          address _address;
11          uint8 party;
12          uint8 districtNo;
13
14          bool elected;
15          bool eliminated;
16      }
17
18      /*
19      struct DistrictResults {
20          uint8 districtNo;
21          uint64[] party1stCountVotes;
22          Candidate[] candidateResults;
23      }*/
24
25
26      struct Election{
27          mapping(uint8 => District) districtContracts;
28          string name;
29          uint voteStart;
30          uint voteEnd;
31          VoteToken voteToken;
32          bool concluded;
33          uint8 partyWithMostVotesIndex;
34          uint8 numberOfPartiesContesting;
35
36          mapping(uint8 => uint64) party1stCountVotes;
37      }
38
```

```
39
40 }
41
42 //DistrictFactory.sol
43 pragma solidity ^0.6.0;
44 pragma experimental ABIEncoderV2;
45
46 import "./District.sol";
47
48 contract DistrictFactory{
49     constructor() public{
50
51     }
52
53     function create(uint8 _districtNumber, Shared.Candidate[]
           calldata _candidates) external returns(District) {
54         District d = new District(_districtNumber,
               _candidates);
55         d.transferOwnership(msg.sender);
56         return d;
57     }
58 }
59
60 //VoteTokenFactory.sol
61 pragma solidity ^0.6.0;
62 pragma experimental ABIEncoderV2;
63
64 import './VoteToken.sol';
65
66 contract VoteTokenFactory{
67
68     constructor() public{
69
70     }
71
72     function create(string calldata _name, uint _voteStart,
           uint _voteEnd) external returns(VoteToken){
73         VoteToken t = new VoteToken(_name,"",_voteStart,
               _voteEnd);
74         t.transferOwnership(msg.sender);
75         return t;
76     }
77 }
78
79 //Untransferable.sol
80 pragma solidity ^0.6.2;
81
82 contract Untransferable{
83 //intentionally empty; used simply to hold nontransferable
```

```
    votes.
}
```

B Test Code

B.1 Unit Tests

B.1.1 Vote Token Test

```
//VoteTokenTest.js
const chai = require("chai");
const expect = chai.expect;
const assert = chai.assert;
const truffleAssert = require('truffle-assertions');
chai.should();

const VoteToken = artifacts.require("./VoteToken.sol");
const District = artifacts.require("./District.sol");

contract("VoteToken", async accounts => {
  var voteStart;
  var voteEnd;
  var name;
  var voteToken;

  beforeEach(async () => {
    voteStart = 0;
    voteEnd = Math.floor(Date.now()/1000) + 9999999999;
    name = "TEST";
    voteToken = await VoteToken.new(name,"",voteStart,voteEnd
        ,{from: accounts[0]});
  });

  it("Should have the attributes passed to it", async ()=>{

    assert.notEqual(voteToken, undefined,voteToken.toString()
        );

    let _name = await voteToken.name();
    assert.equal(_name,name);

    let _voteEnd = await voteToken.voteEnd();

    let _voteStart = await voteToken.voteStart();

    assert.equal(_voteStart, voteStart);
    assert.equal(_voteEnd,voteEnd);
  });
```

```
38
39  it("Minting should increase balance. New token should have
        correct district associated with it. Should be
        impossible to mint new token to address that already has
         one.", async () => {
40
41    let tokenID = 111;
42    let district = 13;
43
44    let balanceBefore = (await voteToken.votesTotal()).
        toNumber();
45    await voteToken.mint(accounts[0],tokenID,district);
46
47    let _balance = await voteToken.balanceOf(accounts[0]);
48    assert.equal(_balance,1,"Balance incorrect; not 1");
49
50    let _district = await voteToken.district(tokenID);
51    assert.equal(_district,district,"District incorrect.");
52    assert.equal(await voteToken.votesTotal(),balanceBefore
        +1);
53
54  });
55
56  it("Should revert all transactions when current time is
        after set voting period but allow owning address to mint
         tokens provided the end time of the election hasn't
        come to pass.", async () => {
57    voteStart = 1;
58    voteEnd = voteStart + 1;
59    name = "TEST";
60
61    let _voteToken = await VoteToken.new(name,"",voteStart,
        voteEnd,{from: accounts[0]});
62    await _voteToken.setAllowed(accounts[0]);
63    await truffleAssert.reverts(_voteToken.mint(accounts
        [0],1,13),"Election is over.");
64
65    voteStart = (new Date(2050,12,30)).getTime()/1000;
66    voteEnd = (new Date(2050,12,31)).getTime()/1000;
67    _voteToken = await VoteToken.new(name,"",voteStart,
        voteEnd,{from: accounts[0]});
68
69    await truffleAssert.passes(_voteToken.mint(accounts
        [1],1,13));
70
71  } );
72
73  it('Should Revert transfers from non-allowed addresses',
        async () => {
```

```
74      voteStart = 1;
75      voteEnd = voteStart + 1;
76      name = "TEST";
77
78      let _voteToken = await VoteToken.new(name,"",voteStart,
            voteEnd,{from: accounts[0]});
79      await truffleAssert.reverts(_voteToken.transferFrom(
            accounts[0],accounts[1],1),reason="Calling Address not
             in list of allowed contracts.");
80  });
81
82  it("Should create a mapping of uint => address from an
        array of n addresses where a given integer  n >= i > 0
        should map to the address in the i-1 th index in the
        passed array. \n This mapping should be in turn
        accessible via the mapping 'preferences' where the
        tokenId passed to the setVotePreferences function should
         map to the mapping described above."
83  , async () => {
84      let _tokenId = 2;
85      await voteToken.mint(accounts[0],_tokenId,13);
86      let _preferences = accounts.slice(1,6);
87      await voteToken.setVotePreferences(_tokenId,_preferences)
            ;
88
89      for(let i = 1; i <= _preferences.length; i++){
90          assert.equal(await voteToken.preferences(_tokenId,i),
                _preferences[i-1],"voteToken(tokenId,i) !=
                _preferences[i-1]");
91      }
92  });
93
94  it('Should not Revert transfers from allowed addresses.',
        async () => {
95      let tokenID = 111;
96      let districtNo = 13;
97
98      var candidateAddresses = [];
99      for(let i = 0; i < 10; i++)
100         candidateAddresses.push(await web3.eth.accounts.
                create().address);
101
102     var candidateStructs = [];
103
104     candidateAddresses.slice(0,5).forEach( (_address) => {
105         candidateStructs.push([
106             _address, // _address
107             0, //party
108             districtNo, //districtNo
```

```
            0, // elected
            0 //eliminated
        ]);
    });

    candidateAddresses.slice(5,10).forEach( (_address) => {
        candidateStructs.push([
            _address, // _address
            1, //party
            districtNo, //districtNo
            0, // elected
            0 //eliminated
        ]);
    });


    let _district = await District.new(districtNo,
       candidateStructs);
    voteToken.setAllowed(_district.address);
    await _district.setVoteToken(voteToken.address);

    await voteToken.mint(accounts[0],tokenID,districtNo);
    await voteToken.setApprovalForAll(_district.address,true)
       ;
    let _preferences = candidateAddresses.slice(1,3);
    await truffleAssert.passes(_district.vote(tokenID,
       _preferences));
    //calls transferFrom function, which checks if caller is
       in allowed list.

  });

});
```

B.1.2 District Unit Test

```
//DistrictTest.js
const chai = require("chai");
const expect = chai.expect;
const assert = chai.assert;
const truffleAssert = require('truffle-assertions');
chai.should();

const VoteToken = artifacts.require("./VoteToken.sol");
const District = artifacts.require("./District.sol");

contract("District" , async accounts => {
    var voteToken;
    var district;
```

```
14      const districtNo = 13;
15      var candidateAddresses;
16      var candidateStructs;
17
18      beforeEach(async () => {
19          let voteStart = 0;
20          let voteEnd = Math.floor(Date.now()/1000) +
                99999999999;
21          let name = "TEST";
22          voteToken = await VoteToken.new(name,"",voteStart,
                voteEnd);
23
24          candidateAddresses = [];
25          for(let i = 0; i < 10; i++)
26              candidateAddresses.push(await web3.eth.accounts.
                    create().address);
27
28          candidateStructs = [];
29
30          candidateAddresses.slice(0,5).forEach( (_address) =>
                {
31              candidateStructs.push([
32                  _address, // _address
33                  0, //party
34                  districtNo, //districtNo
35                  0, // elected
36                  0 //eliminated
37              ]);
38          });
39
40          candidateAddresses.slice(5,10).forEach( (_address) =>
                {
41              candidateStructs.push([
42                  _address, // _address
43                  1, //party
44                  districtNo, //districtNo
45                  0, // elected
46                  0 //eliminated
47              ]);
48          });
49
50          district = await District.new(districtNo,
                candidateStructs);
51          voteToken.setAllowed(district.address);
52          await district.setVoteToken(voteToken.address);
53
54          console.log("Candidates Count: " + await district.
                candidatesCount());
55          console.log(candidateStructs);
```

```
56      });
57
58
59
60      it("Vote function does not revert when attempting to vote
            with a VoteToken instance having the correct district
            number (13). And increases cast votes count. ie:
           voting succeeds", async () => {
61          let tokenId = 1;
62          voteToken.mint(accounts[0], tokenId, 13);
63          voteToken.setApprovalForAll(district.address,true);
64
65          await truffleAssert.passes(district.vote(tokenId,
                candidateAddresses.slice(0,5)), "Reverts vote
                function transaction on receiving a token with
                correct district number.");
66      });
67
68      it("Vote function reverts when attempting to vote with a
           VoteToken instance having the incorrect district
           number (12). And does not increase cast votes count.
           ie: voting fails", async () => {
69          let tokenId = 1;
70          voteToken.mint(accounts[0], tokenId, 12);
71          voteToken.setApprovalForAll(district.address,true);
72
73          await truffleAssert.reverts(district.vote(tokenId,
                candidateAddresses.slice(0,5)),reason="Incorrect
                District." , message="Does not revert vote
                function transaction on receiving a token with
                incorrect district number.");
74      });
75
76      it("Vote function sets voting preferences correctly",
           async () => {
77          let tokenId = 1;
78          voteToken.mint(accounts[0], tokenId, 13);
79          voteToken.setApprovalForAll(district.address,true);
80
81          await truffleAssert.passes(district.vote(tokenId,
                candidateAddresses.slice(0,5)), "Reverts vote
                function transaction on receiving a token with
                correct district number.");
82
83          for(let i =0; i < candidateAddresses.slice(0,5).
                length; i++){
84              let x = await voteToken.preferences(tokenId,i+1);
85              assert.equal(x,candidateAddresses.slice(0,5)[i]);
86          }
```

```
87      });
88
89
90      it("getQuota() returns correct values", async () => {
91          //By mathematical induction
92          //For all castVotes < seats + 1 (seats == 5) quota ==
                1
93           assert.equal(await district.getQuota(),1); //
                castVotes == 0
94
95           let tokenId = 1;
96           await voteToken.mint(accounts[0], tokenId, 13);
97           await voteToken.setApprovalForAll(district.address,
                true);
98           await truffleAssert.passes(district.vote(tokenId,
                candidateAddresses.slice(0,5)));
99           assert.equal(await district.getQuota(),1); //
                castVotes == 1
100
101          tokenId = 2;
102          await voteToken.mint(accounts[0], tokenId, 13);
103          await voteToken.setApprovalForAll(district.address,
                true);
104          await truffleAssert.passes(district.vote(tokenId,
                candidateAddresses.slice(0,5)));
105          assert.equal(await district.getQuota(),1); //
                castVotes == 2
106
107
108     });
109
110     it("Should transfer all tokens back to owner, and revert
            if caller is not District to which that token
            corresponds", async () => {
111          await voteToken.mint(accounts[0],2,13);
112          await voteToken.setApprovalForAll(district.address,
                true);
113
114          let districtBalance = await voteToken.balanceOf(
                district.address);
115          let otherBalance = await voteToken.balanceOf(accounts
                [0]);
116
117          assert.equal(otherBalance,1,"Mint Failed");
118          assert.equal(districtBalance,0);
119
120          await truffleAssert.passes(district.transferAllBack(
                accounts[0]));
121          let _districtBalance = await voteToken.balanceOf(
```

```
            district.address);
122         assert.equal(_districtBalance.toNumber(),
               districtBalance.toNumber() + otherBalance.toNumber
               ());
123
124         await voteToken.mint(accounts[1],3,13);
125         await truffleAssert.reverts(district.transferAllBack(
               accounts[1]),"ERC721: transfer caller is not owner
                nor approved.");
126
127
128
129         await voteToken.mint(accounts[0],4,12);
130         await truffleAssert.reverts(district.transferAllBack(
               accounts[0]),"Attempting to seize token from
               another district.");
131   });
132
133   it('Given 5 cast votes (and a corresponding quota of 1)
        each selecting the same 5 candidates from the same party
         (id 0), each candidate should end up with 1 vote and be
         marked as elected. party\'s index should map to 5 in
        the party1stCountVotes mapping', async () => {
134
135     for(let i = 1; i<=5; i++){
136
137         await voteToken.mint(accounts[0], i, districtNo);
138         await voteToken.setApprovalForAll(district.address,
               true);
139
140         let a = await district.vote(i,candidateAddresses.
               slice(0,5));
141
142         for(let j = 1; j<=5; j++)
143             console.log(await voteToken.preferences(i,j));
144
145     }
146
147     assert.equal(await district.candidatesCount(),10,"
           Candidates count is not 10 before vote counting");
148     await district.countVotes();
149     assert.equal(await district.candidatesCount(),10,"
           Candidates count is not 10 after vote counting");
150
151
152     console.log(candidateAddresses.slice(0,5));
153
154     var i = 0;
155     candidateAddresses.slice(0,5).forEach(async (
```

```
        candidateAddress) =>{
156         let balance = await voteToken.balanceOf(
                candidateAddress);
157
158         assert.equal(1,balance,candidateAddress +" " + i + "
                Incorrect Balance.");
159         i++;
160     });
161
162     console.log(candidateAddresses.slice(5,10));
163
164     i = 0;
165     candidateAddresses.slice(5,10).forEach(async (
            candidateAddress) =>{
166         let balance = await voteToken.balanceOf(
                candidateAddress);
167
168         assert.equal(0,balance,candidateAddress +" " + i + "
                Incorrect Balance.");
169         i++;
170     });
171 });
172
173 it("Should Show correct number of 1st count votes for both
    parties (0 5) after voting",async ()=>{
174     for(let i = 6; i<=10; i++){
175         await voteToken.mint(accounts[0], i, districtNo);
176         district.incrementCastVotes();
177     }
178     await voteToken.setApprovalForAll(district.address,true);
179
180     assert.equal(await district.castVotes(),5)
181     assert.equal(await district.getQuota(),1);
182     for(let i = 6; i<=10; i++){
183         //console.log(i);
184         //castVote(candidateAddresses.slice(0,5), i);
185         console.log(accounts[0]);
186         await district.vote(i,candidateAddresses.slice(5,10))
                ;
187     //console.log(i);
188         for(let j = 1; j<=5; j++)
189             console.log(await voteToken.preferences(i,j));
190
191     }
192
193     {let _01stCount = await district.party1stCountVotes(0);
194     assert.equal(_01stCount,0,"Party index 0 incorrect number
            of votes.");
195     let _11stCount = await district.party1stCountVotes(1);
```

```
196     assert.equal(_11stCount,5,"Party index 0 incorrect number
            of votes.");}
197
198     await district.countVotes();
199     assert.equal(0,await district.seatsRemaining());
200     await district.setCounted(true);
201
202     console.log("QUOTA:" + await district.getQuota());
203     candidateAddresses.slice(5,10).forEach(async (
            candidateAddress) =>{
204         let candidateStruct = await district.candidates(
                candidateAddress);
205         console.log(candidateAddress +" "+candidateStruct.
                elected);
206         assert.equal(candidateStruct.elected,true);
207     });
208 });
209
210 it("Should Show correct number of 1st count votes for both
        parties (5 0) after voting",async ()=>{
211     for(let i = 1; i<=5; i++){
212         await voteToken.mint(accounts[0], i, districtNo);
213         district.incrementCastVotes();
214     }
215     await voteToken.setApprovalForAll(district.address,true);
216
217     assert.equal(await district.castVotes(),5)
218     assert.equal(await district.getQuota(),1);
219     for(let i = 1; i<=5; i++){
220         //console.log(i);
221         //castVote(candidateAddresses.slice(0,5), i);
222         console.log(accounts[0]);
223         await district.vote(i,candidateAddresses.slice(0,5));
224     //console.log(i);
225         for(let j = 1; j<=5; j++)
226             console.log(await voteToken.preferences(i,j));
227
228     }
229
230     let _01stCount = await district.party1stCountVotes(0);
231     assert.equal(_01stCount,5,"Party index 0 incorrect number
            of votes.");
232     let _11stCount = await district.party1stCountVotes(1);
233     assert.equal(_11stCount,0,"Party index 0 incorrect number
            of votes.");
234
235     let a = await district.countVotes();
236     assert.equal(0,await district.seatsRemaining());
237     await district.setCounted(true);
```

```
238
239     console.log("QUOTA:" + await district.getQuota());
240     candidateAddresses.slice(0,5).forEach(async (
            candidateAddress) =>{
241         let candidateStruct = await district.candidates(
                candidateAddress);
242         console.log(candidateAddress +" "+ candidateStruct.
                elected);
243         assert.equal(candidateStruct.elected,true);
244         });
245     });
246 });
```

B.1.3 Election Controller Tests

```
1  //ElectionControllerTest.js
2  const chai = require("chai");
3  const expect = chai.expect;
4  const assert = chai.assert;
5  const truffleAssert = require('truffle-assertions');
6  const { contracts_build_directory } = require("../truffle-
       config");
7  chai.should();
8
9  const ElectionController = artifacts.require("./
       ElectionController.sol");
10 const DistrictFactory = artifacts.require("./DistrictFactory.
       sol");
11 const VoteTokenFactory = artifacts.require("./
       VoteTokenFactory.sol");
12 const District = artifacts.require("./District.sol");
13 const VoteToken = artifacts.require("./VoteToken.sol");
14
15 contract("ElectionController",async (accounts) => {
16     var electionController;
17     var districtFactory;
18     var voteTokenFactory;
19     var candidates;
20     var _name;
21     var _voteStart;
22     var _voteEnd;
23     const districtsNo = 13;
24     const numberOfPartiesContesting = 2;
25
26     beforeEach( async () => {
27
28         districtFactory = await DistrictFactory.new();
29         voteTokenFactory = await VoteTokenFactory.new();
30         candidates = [];
```

```
31          candidateAddresses = [];
32          _name = "TEST";
33          _voteStart = 0;
34          _voteEnd = Math.floor(Date.now()) + 999999999;
35          electionController = await ElectionController.new(
               districtsNo,districtFactory.address,
               voteTokenFactory.address);
36
37          for (let i=0;i<districtsNo;i++){
38              candidates.push([]);
39              candidateAddresses.push([]);
40
41              for(let j=0;j<10;j++){
42                  let a = await web3.eth.accounts.create().
                       address;
43                  candidateAddresses[i].push(a);
44                  candidates[i].push([
45                      a,//await web3.eth.accounts.create(),//
                           address
46                      0, //Party
47                      i+1, //District No
48                      false, //elected
49                      false //eliminated
50                  ]);
51              }
52
53              for(let j=10;j<20;j++){
54                  let a = await web3.eth.accounts.create().
                       address;
55                  candidateAddresses[i].push(a);
56                  candidates[i].push([
57                      a,//await web3.eth.accounts.create(),//
                           address
58                      1, //Party
59                      i+1, //District No
60                      false, //elected
61                      false //eliminated
62                  ]);
63              }
64          }
65
66          let a = await electionController.launchElection(_name
               ,_voteStart,_voteEnd,numberOfPartiesContesting);
67
68          let b;
69          for(let i=1; i<=districtsNo;i++){
70              b = await electionController.launchDistrict(0,i,
                   candidates[i-1]);
71          }
```

```
72      });
73
74      it("Should push an election with the correct parameters",
            async ()=>{
75          let election = await electionController.elections(0);
76          console.log(election);
77          console.log(election.name);
78          assert.equal(_name,election.name);
79          assert.equal(_voteStart,election.voteStart);
80          assert.equal(_voteEnd,election.voteEnd);
81          assert.equal(await electionController.districtsNo()
                ,13);
82
83          let voteToken = await VoteToken.at(election.voteToken
                );
84          assert.equal(_name,await voteToken.name());
85          assert.equal(_voteStart,await voteToken.voteStart());
86          assert.equal(_voteEnd,await voteToken.voteEnd());
87
88          for(let i=1;i<=13;i++){
89              let districtContract = await District.at(await
                    electionController.getDistrictContract(0,i));
90
91              assert.equal(i,await districtContract.
                    districtNumber());
92              assert.equal(election.voteToken,await
                    districtContract.voteToken());
93
94              var districtCandidateAddresses = await
                    electionController.
                    getDistrictCandidateAddresses(0,i);
95              console.log(districtCandidateAddresses);
96              assert.equal(20,districtCandidateAddresses.length
                    );
97
98              for(let j = 0; j<20; j++)
99                  assert.equal(candidates[i-1][j][0],
                        districtCandidateAddresses[j]);
100         }
101     });
102
103     it("mintVote should increase vote balance of controller
            contract. Should revert if address already has vote",
            async ()=>{
104         let address = await web3.eth.accounts.create().
                address;
105         await electionController.mintVote(1,13,0);
106
107         let balance = await electionController.balanceOf(
```

```
            electionController.address,0);
108         //console.log(balance.toNumber());
109         assert.equal(balance,1);
110
111         await truffleAssert.reverts(electionController.
                mintVote(1,13,0),"ERC721: token already minted.");
112
113         let election = await electionController.elections(0);
114         //console.log(election);
115         let voteToken = await VoteToken.at(election.voteToken
                );
116         await truffleAssert.reverts(voteToken.mint(address
                ,1,13),"Ownable: caller is not the owner");
117     });
118
119     it('endElection should fail. Because end of election has
            not passed.', async () => {
120         await truffleAssert.reverts(electionController.
                endElection(0),reason="Voting Not Over Yet",
                message="Voting time check failed.");
121     });
122
123 });
```

```
1  //ElectionControllerTest_2.js
2  const chai = require("chai");
3  const expect = chai.expect;
4  const assert = chai.assert;
5  const truffleAssert = require('truffle-assertions');
6  const { contracts_build_directory } = require("../truffle-
       config");
7  chai.should();
8
9  const ElectionController = artifacts.require("./
       ElectionController.sol");
10 const DistrictFactory = artifacts.require("./DistrictFactory.
       sol");
11 const VoteTokenFactory = artifacts.require("./
       VoteTokenFactory.sol");
12 const District = artifacts.require("./District.sol");
13 const VoteToken = artifacts.require("./VoteToken.sol");
14
15 contract("ElectionController",async (accounts) => {
16
17     var electionController;
18     var districtFactory;
19     var voteTokenFactory;
20     var candidates;
21     var _name;
```

```
22      var _voteStart;
23      var _voteEnd;
24      var districtsNo;
25      var numberOfPartiesContesting;
26
27      beforeEach( async () => {
28          districtsNo = 13;
29          numberOfPartiesContesting = 2;
30          districtFactory = await DistrictFactory.new();
31          voteTokenFactory = await VoteTokenFactory.new();
32          candidates = [];
33          candidateAddresses = [];
34          _name = "TEST";
35          _voteStart = 0;
36          _voteEnd = Math.floor(Date.now()/1000) + 240;
37          electionController = await ElectionController.new(
                districtsNo,districtFactory.address,
                voteTokenFactory.address);
38
39          for (let i=0;i<districtsNo;i++){
40              candidates.push([]);
41              candidateAddresses.push([]);
42
43              for(let j=0;j<5;j++){
44                  let a = await web3.eth.accounts.create().
                        address;
45                  candidateAddresses[i].push(a);
46                  candidates[i].push([
47                      a,//await web3.eth.accounts.create(),//
                            address
48                      0, //Party
49                      i+1, //District No
50                      false, //elected
51                      false //eliminated
52                  ]);
53              }
54
55              for(let j=5;j<10;j++){
56                  let a = await web3.eth.accounts.create().
                        address;
57                  candidateAddresses[i].push(a);
58                  candidates[i].push([
59                      a,//await web3.eth.accounts.create(),//
                            address
60                      1, //Party
61                      i+1, //District No
62                      false, //elected
63                      false //eliminated
64                  ]);
```

```
65              }
66          }
67
68          await electionController.launchElection(_name,
                _voteStart,_voteEnd,numberOfPartiesContesting);
69          for(let k = 1; k<=districtsNo;k++)
70              await electionController.launchDistrict(0,k,
                    candidates[k-1]);
71
72          var election = await electionController.elections(0);
73          var voteToken = await VoteToken.at(election.voteToken
                );
74
75
76          var id = 0;
77          for(let i = 0; i<13; i++){
78              let d = await electionController.
                    getDistrictContract(0,i+1);
79              let district = await District.at(d);
80              console.log(district.address);
81
82              for(let j = 0; j<5;j++){
83                  //10 votes for party 1 on each district, for
                        10 candidates from party 1
84                  await electionController.mintVote(++id, i
                        +1,0);
85                  let a = await electionController.vote(0,id,
                        candidateAddresses[i].slice(5,));
86
87              }
88
89          }
90      });
91
92      it("Election Should End with correct results",async ()=>{
93
94          function timeout(ms) {
95              return new Promise(resolve => setTimeout(resolve,
                    ms));
96          }
97
98          await timeout(240000);
99          //wait for voting period to conclude.
100         for(let k = 1; k<=districtsNo;k++){
101             await electionController.endDistrict(0,k);
102             await electionController.setCounted(0,k,true);
103
104             let district = await District.at(await
                    electionController.getDistrictContract(0,k));
```

```
105             let candidateAddresses = await district.
                    getCandidateAddresses();
106
107             for(let l=5; l<10;l++){
108                 let candidateStruct = await district.
                        candidates(candidateAddresses[l]);
109                 assert.isTrue(candidateStruct.elected,"
                        Candidate not elected.");
110             }
111         }
112
113         election = await electionController.elections(0);
114         console.log(election);
115         assert.notEqual(await electionController.districtsNo
                ),0);
116
117         let z = await electionController.endElection(0);
118         console.log(z.toString());
119
120         election = await electionController.elections(0);
121
122         console.log(election);
123         assert.equal(election.partyWithMostVotesIndex,1);
124         });
125
126 });
```

C Other

```
1  //ERC721.sol
2  ///////
3  import "../../utils/EnumerableSet.sol";
4  ///////
5  // Mapping from holder address to their (enumerable) set of
       owned tokens
6      mapping (address => EnumerableSet.UintSet) private
           _holderTokens;
7  ///////
8  function tokenOfOwnerByIndex(address owner, uint256 index)
       public view override returns (uint256) {
9          return _holderTokens[owner].at(index);
10     }
11 ///////
12 function _transfer(address from, address to, uint256 tokenId)
       internal virtual {
13 ///////
14     _holderTokens[from].remove(tokenId);
15     _holderTokens[to].add(tokenId);
16 ///////
17     }
```

Figure 17: Snippets from ERC721.sol showing the relevant lines.

(OpenZeppelin, 2020b)

```
//EnumerableSet.sol (library)
//////////////////////
    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus
           1 because index 0
        // means a value is not in the set.
        mapping (bytes32 => uint256) _indexes;
    }
    ///////////////
    struct UintSet {
        Set _inner;
    }
    //////////////
    function _at(Set storage set, uint256 index) private view
        returns (bytes32) {
        require(set._values.length > index, "EnumerableSet:
           index out of bounds");
        return set._values[index];
    }
    //////////////
    function at(AddressSet storage set, uint256 index)
       internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }
    ////////////
}
```

Figure 18: Relevant Code From Openzeppelińs EnumerableSet library

(OpenZeppelin, 2020a)

```
1     ////////////
2     function _add(Set storage set, bytes32 value) private
          returns (bool) {
3     if (!_contains(set, value)) {
4         set._values.push(value);
5         // The value is stored at length-1, but we add 1 to
              all indexes
6         // and use 0 as a sentinel value
7         set._indexes[value] = set._values.length;
8         return true;
9     } else {
10        return false;
11        }
12    }
13    ////////////
14    function add(AddressSet storage set, address value)
          internal returns (bool) {
15        return _add(set._inner, bytes32(uint256(value)));
16    }
17    ////////////
```

Figure 19: Relevant Code From Openzeppelin's EnumerableSet library (2)

(OpenZeppelin, 2020a)

```
1         function _remove(Set storage set, bytes32 value)
              private returns (bool) {
2             // We read and store the value's index to prevent
                  multiple reads from the same storage slot
3             uint256 valueIndex = set._indexes[value];
4             if (valueIndex != 0) { // Equivalent to contains(
                  set, value)
5             // To delete an element from the _values array in
                  O(1), we swap the element to delete with the
                  last one in
6             // the array, and then remove the last element (
                  sometimes called as 'swap and pop').
7             // This modifies the order of the array, as noted
                  in {at}.
8             uint256 toDeleteIndex = valueIndex - 1;
9             uint256 lastIndex = set._values.length - 1;
10            // When the value to delete is the last one, the
                  swap operation is unnecessary. However, since
                  this occurs
11            // so rarely, we still do the swap anyway to
                  avoid the gas cost of adding an 'if' statement
                  .
12            bytes32 lastvalue = set._values[lastIndex];
13            // Move the last value to the index where the
                  value to delete is
14            set._values[toDeleteIndex] = lastvalue;
15            // Update the index for the moved value
16            set._indexes[lastvalue] = toDeleteIndex + 1; //
                  All indexes are 1-based
17            // Delete the slot where the moved value was
                  stored
18            set._values.pop();
19            // Delete the index for the deleted slot
20            delete set._indexes[value];
21            return true;
22            } else {
23                return false;}
24            }
25 ////////////
26     function remove(AddressSet storage set, address value)
           internal returns (bool) {
27         return _remove(set._inner, bytes32(uint256(value)));
28     }
29 ////////////
```

Figure 20: Relevant Code From Openzeppelińs EnumerableSet library (3)

(OpenZeppelin, 2020a)

D Project Diary

Date	Entry
19/06/2020	Research Proposal First Draft submitted for review.
23/06/2020	Research Proposal Final Draft finished. As per suggestion, focus was realigned towards security of smart contracts rather than creation of a production-ready application. Also arranged formatting and presentation
10/07/2020	Initial Report and Literature Review First Draft submitted for review.
27/07/2020	Initial Report Final Draft finished. Fixed wording to present a more cohesive narrative. Elaborated content list and work plan.
10/08/2020	Start of Coding of application.
28/08/2020	Start of writing of dissertation
18/09/2020	Dissertation first draft complete
26/09/2020	Dissertation second draft complete Arranged structure of dissertation. Fixed typos and formatting. Improved narrative flow.
13/11/2020	Dissertation final submission complete Arranged index not displaying correctly. Fixed minor typographical errors.