Chapter 13
Case Study: SecureVote

Taking a Dapp from MVP to Production

with Max Kaye and Nathan Spataro

'Democratise the world.'
— SecureVote's Massive Transformative Purpose (MTP)

13.1 Introduction and Background

Voting seems simple enough. With paper, voters just fill out a ballot sheet and put it in a box. To count the votes, the box is emptied and the ballots are counted in public. However, there are many underlying complications. How do we know extra votes have not been added? How do we know each voter has voted at most once, or exactly once? If a voter claims their vote *was not* in the final tally, how could we check? How do we know the count is accurate, especially if it can vary every time votes are counted?

Solving all of these problems can be hard, even with paper systems. With electronic voting systems, some things become easier. For example, there may be a public electorate-wide list of voters, and we could ensure each vote has some kind of verifiable cryptographic authentication. This can help us check to who did not vote. And, of course, tallying votes is fast and reliable in an electronic system. However, electronic systems present us with other problems. How can we anonymize votes in an electronic system? How do voters know whether their vote was included without revealing who they voted for? How can we decide which votes are valid without a privileged role?

The potential utility of blockchain technology for voting was identified early,[1] and blockchain can help to solve some of these problems. However, using a blockchain alone is not enough. The exact blockchain chosen, the consensus mechanisms used, and the architecture of the voting platform are all important design decisions

[1] At least by December 2010: https://bitcointalk.org/index.php?topic=2299.msg31851#msg31851.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_13

that impact system capabilities. While this chapter does not describe solutions to all of the hard problems with online voting, it does describe architectural concerns for robust, upgradable, multi-user smart contracts used by SecureVote to address these problems.

This case study concerns the development of *Tokenvote*, a general purpose, multichain governance system for blockchain-based tokens developed by *SecureVote*. Tokenvote uses the Solidity language and is deployed to Ethereum.[2] Tokenvote supports arbitrarily complex append-only voting systems and has been designed to be modular, upgradable, and configurable. SecureVote was founded in 2016 to provide affordable, turn-key voting systems of all types and at all scales.

In this chapter we will cover many of the challenges encountered and trade-off decisions made while taking this smart contract-based voting solution from a minimum viable product (MVP) to production. This journey spanned 4 months, numerous redesigns, and all of the ecosystem issues mentioned above. We will not describe the voting functionality in detail—the principles are outlined in the sidebar below—but rather focus on how the architecture and overall design changed over development. The discussion is in terms of the Solidity language on Ethereum, but many of the architectural issues apply across smart contract languages and platforms.

Principles of Anonymous Voting Using Blockchain

Through a combination of public-private key cryptography and peer-to-peer shuffling, SecureVote achieves that voters can vote anonymously and later verify and confirm that their vote has been recorded correctly. Voters cannot prove which vote was theirs, and no one else can find out how they voted.

To achieve these goals, a ballot is prepared with an electoral roll, containing all addresses that are allowed to vote. For this ballot, voters first create and anonymize an *ephemeral* voting key pair, which is discarded after the ballot completes. The voters use this key pair to anonymously sign their actual vote.

Two rounds of shuffling are necessary: the first one to create the ephemeral anonymized key pairs and the second one for the actual voting. In each round, the shuffling is done *off-chain*, in a peer-to-peer fashion but relying on *on-chain* information like the electoral roll; and the results of the round are published *on-chain*. After each round completes, each voter confirms that the result is well-formed and that their vote/ephemeral public key was recorded correctly, by signing the result.

(continued)

[2] Links to the Tokenvote source code and Ethereum documentation appear in Section 13.6 at the conclusion of this chapter.

Say there are 50 voters. The ballot ensures that each voter can vote at most once and only voters on the anonymized electoral roll can vote. If all 50 voters confirm that their individual vote was recorded correctly, then we know that *all* votes were recorded correctly. In other words, the number of signatures on the ballot must match the number of voters exactly.

More details can be found at https://gitlab.com/exo-one/svst-docker/blob/master/svst-docs/secure.vote.white.napkin.md.

13.2 The MVP Prototype

In late 2017, SecureVote implemented a small MVP to facilitate early governance for the US-based *Swarm Fund*, a blockchain-based organization facilitating the creation of securitized tokens. Although Swarm Fund's security tokens live on a Stellar-based blockchain, their organization-wide token (SWM) is an ERC20 token on Ethereum.

Swarm (unlike many ERC20-based organizations) were proactive about governance from the start. In their whitepaper they described the first version of their *Liquid Democracy Voting Module* (LDVM), a system designed to support the governance of both the foundation and the investment opportunities offered via their platform.[3] There are two important aspects of their design that are common in systems of distributed governance: delegation and stake-weighted votes.

- *Stake-weighted votes*: In many ballots, not every vote is weighted equally, or some parties may have an unequal number of votes. The most common example of stake-weighted voting is by shareholders at a company's annual general meeting (AGM). Each shareholder votes with a weighting proportional to the number of shares they own: 1 share, 1 vote; 2000 shares, 2000 votes. For similar reasons, most token-based communities choose to use stake-weighted votes.
- *Delegation*: Voters can choose another party to act on their behalf. On a blockchain, this could be another account they own, for example, allowing voters to delegate voting power from tokens they own in a cold wallet to a 'voting-only' account in a hot wallet. Or, the delegate could be someone else's account, for example, a prominent community member. Delegation is a common feature of modern digital governance systems. It is similar to the idea of a representative in government but can be done on a per-voter basis. In some systems multiple delegations can be chained together. The original voter can always stop the

[3]Swarm's LDVM design uses fairly standard patterns, and the requirements are currently met by a subset of Tokenvote's capabilities.

delegation and vote directly; only if a voter *does not* vote does the delegate inherit the voter's weighting.

SecureVote was responsible for the initial implementation of Swarm's governance framework. This initial deployment had only a few requirements:

R1.1 Facilitate an open ballot for all SWM token holders and all delegates.
R1.2 Support optional delegation to arbitrary Ethereum addresses.
R1.3 Stake-weight votes according to voters' SWM balances and delegations.
R1.4 Support the deterministic audit of the ballot by arbitrary actors.
R1.5 Support the encryption of votes such that the result is unavailable until the respective secret key has been published.

These requirements seem simple but are practically impossible to meet using *only* smart contract platforms like Ethereum and *on-chain* computation, where all storage, auditing, and delegation resolution occurs within the blockchain's virtual machine. There are two primary reasons for this:

- Historical access: a naive voting system might check a voter's balance at the time the vote is cast. However, this approach introduces multiple race conditions and makes handling delegation difficult. The correct approach is to use snapshots at the start and end of the voting period to retrieve balances and delegations, respectively. Ethereum does not support this kind of arbitrary historical access.
- Transaction cost: with fee-per-operation blockchain platforms, like Ethereum, it is prohibitively expensive to repeatedly load items from storage (like balances, delegations, and votes) and run tightly looped algorithms such as recursive delegation resolution or vote counting. As an example: a well-tuned smart contract could process a maximum of around 400 votes per Ethereum block, or 1600 votes per minute, based on a gas limit of 8 million gas. Processing greater volumes requires splitting the operation across multiple transactions, a tactic which adds overhead and code complexity. Some of the more interesting features, like on-chain decryption, are simply untenable under fee-per-operation models.[4]

SecureVote has previously argued that secure voting (be that on paper or online) is impossible at scale without the use of a well-constructed blockchain. This is due to three goals:

- Immutability: the voting record must be append-only and cannot be changed (even if individual votes can be replaced).
- Censorship resistance: no actor should be capable of preventing a voter from submitting their vote, except through violence. This requirement precludes purely proof-of-stake chains in most cases.

[4]Note: computations like on-chain decryption can be more practical with protocol-layer optimizations such as the `ecrecover` function supported by Ethereum.

- Consensus: voters and auditors must agree on which votes are to be counted *both during and after* the voting period, and these rules must be non-authoritarian (due to the need for censorship resistance), objective, and non-discriminatory.

All centralized systems (including recent end-to-end verifiable designs such as *Prêt à Voter*) fail at least one of these requirements (usually censorship resistance) and are thus not fully secure.

Although Requirement R1.4 (deterministic audit) requires a blockchain to *store* vote data, it does not require that the audit itself is performed on-chain. The lack of historical access available to smart contracts[5] meant SecureVote needed to audit the ballot off-chain. Given this, we opted to move as much processing and functionality off-chain as possible without compromising the platform's integrity. Decryption of votes was done every time an audit was run.

The initial MVP was incredibly simple, with only three components:

- A small smart contract of around 100 lines of Solidity code, to securely deliver ballot details and store votes
- A rudimentary auditor to authenticate voters, decrypt votes, allocate the appropriate weighting, and resolve delegations
- A user interface

At this stage, the MVP was unable to handle multiple ballots or communities, and an individual smart contract had to be deployed for each ballot, costing around 800,000 gas at a minimum. Although this rudimentary system was quite capable of handling Swarm's needs for the next few months, it was unsuitable for general use and required costly manual attention for every deployment.

13.3 Building Tokenvote

Although the MVP was functional and satisfied basic requirements for one-off ballots, it was not a fully fledged product. Prior to February 2018, SecureVote intended to launch their platform via a custom, separate blockchain they had been developing since June 2017. Although development had been progressing steadily (two prototypes existed at this stage), it was not progressing quickly. In order to launch a viable platform in the shortest period of time, they made the decision to pause development of their custom chain, and pivoted to building out the MVP into a general software-as-a-service (SaaS) platform: Tokenvote. This was to reduce development time and support most of the features of their custom chain, albeit with reduced capacity.

This section covers many of the problems SecureVote encountered while building Tokenvote based on the MVP described above. For each problem we will look at one

[5]Ethereum smart contracts have access to the past 256 states only (corresponding to the past 256 blocks), a period of approximately 1 h.

or more potential solutions and discuss compromises. Some simplified Solidity code is used in the presentation.[6] The simplifications are made to keep the examples as short as possible, so best practices are sometimes ignored.

As a more generic platform, there were additional requirements:

R2.1 Centrally manage and track groups (democracies), including Ether payments and permissions.

R2.2 Allow group administrators to create new ballots, and control permissions around ballots from community members.

R2.3 Extensibility and maintainability: any component can be upgraded, new components can be added, and Tokenvote must support migration to another platform in the future.

R2.4 Browser compatible: the whole stack should be able to be run in a browser, excluding the Ethereum nodes themselves, without compromising the security model.

The goal was for Tokenvote to facilitate everything the Swarm prototype did and more, but to cater for many groups, each with many ballots, without needing any interaction with SecureVote staff.

13.3.1 Tokenvote Architecture Overview

The initial, planned architecture for Tokenvote is shown in Fig. 13.1. After numerous iterations the final architecture is as shown in Fig. 13.3. Each is discussed below.

Planned Architecture

In the initial architecture of Fig. 13.1, administrators interact with an on-chain component that serves as a central hub, which SecureVote call the `Index`. This component is responsible for all administrative functions, including payment of fees for holding a ballot. The `Index` also keeps track of groups of voters, called *Democracies*. If fees are paid, a ballot is set up for a Democracy through a factory contract, the `Ballot Box Factory`. See Section 7.4.4 for a general discussion of the factory contract pattern. This factory contract can create a ballot by deploying a new `Ballot Box` smart contract, through which voters can cast their votes.

For the reasons explained around the MVP above, tallying and weighting of the votes is done offline through an `Auditor` component. This component is available to any voter, so that independent auditing is possible. The `Auditor` also queries the relevant ERC20 contract for token holdings and other details as required, including

[6]Simplifications include omitting keywords like `view` or `pure` on function declarations, and declaring functions `public` instead of `external`.

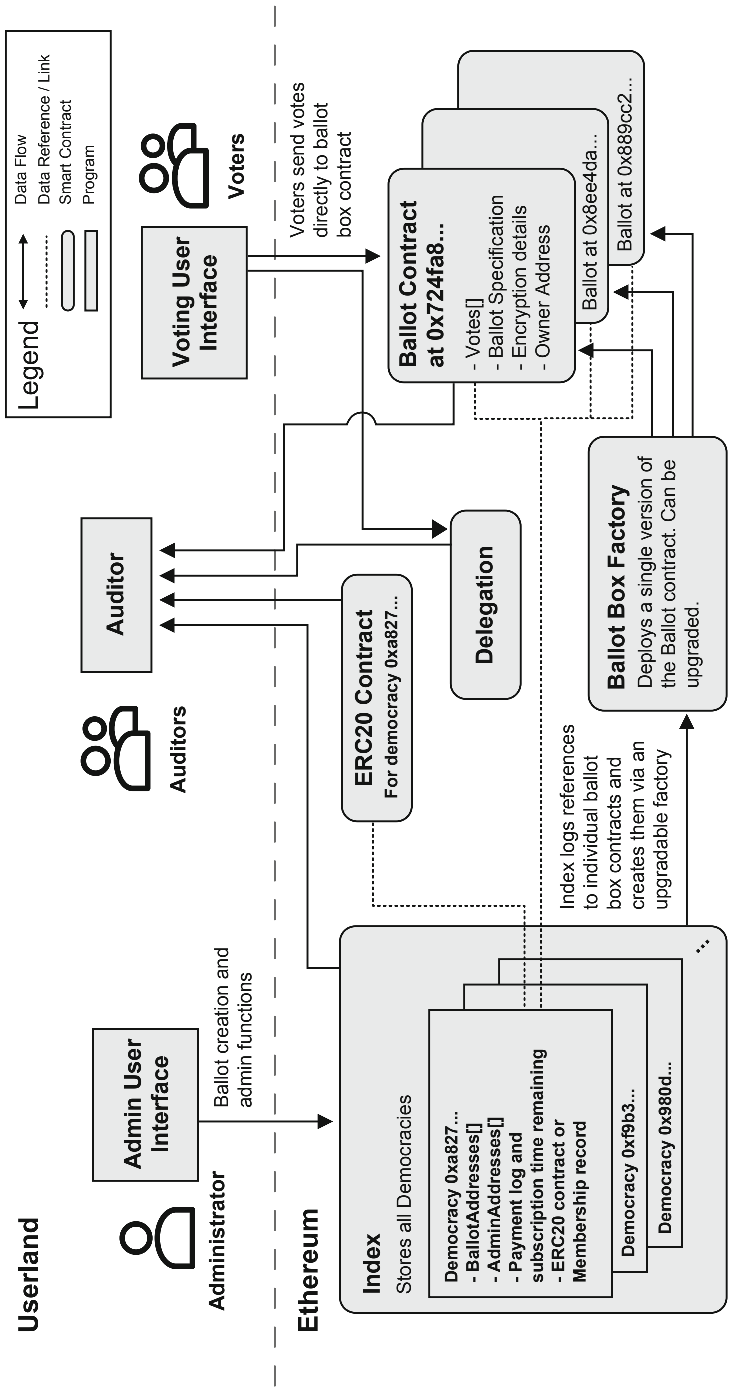

Fig. 13.1 Planned architecture for Tokenvote before development

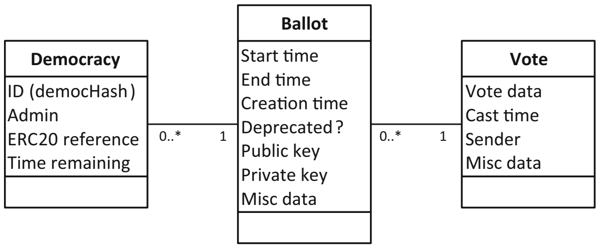

Fig. 13.2 Logical view of the required data structure, as a UML class diagram

delegation and balances on other chains. This allows the auditor to consider tokens across the public Ethereum and Ethereum Classic blockchains, among others.

The logical data structure for this design is depicted in Fig. 13.2. As shown, each democracy can have an arbitrary number of ballots, and each ballot can have many votes. Each vote belongs to one ballot and each ballot to one democracy.

The reference to the relevant ERC20 contract is stored for the democracy. Encrypted ballots can be held by generating a key pair, publishing the public key for all voters to encrypt their votes, and revealing the private (secret) key after the end time of the ballot. This method avoids influence of intermediate results on voters while the ballot is ongoing.

Final Architecture

In the final architecture depicted in Fig. 13.3, the basic components are still present, but there are some significant changes:

- Instead of creating one smart contract per ballot, all ballots that use a particular feature set are stored in the same contract, the `Ballot Box Storage`. This includes ballots from different democracies. It in turn relies on the code outsourced to the `Ballot Box Library` contract, implementing the data contract and library contract patterns from Section 7.4. Collapsing all ballots into few smart contracts is more efficient in terms of gas cost. As discussed in earlier chapters, this results in reduced monetary cost, increased throughput, and reduced danger of network congestion.
- Following the same patterns to achieve upgradability and separation of concerns in the `Index`, data on payments is stored in the `Payments Backend`, and all other data for the `Index` is stored in the `Data Store Backend`. Pricing for community ballots is calculated in the `Community Ballot Payment` contract; adaptive, context-dependent pricing is needed to avoid spamming democracies with too many ballots.
- To allow easy addressing, the Ethereum Name Service, `ENS`, is used. The `ENS Proxy` implements the contract registry pattern (Section 7.4.1). Requesters can look up the reference for the latest version of the `Index` contract.

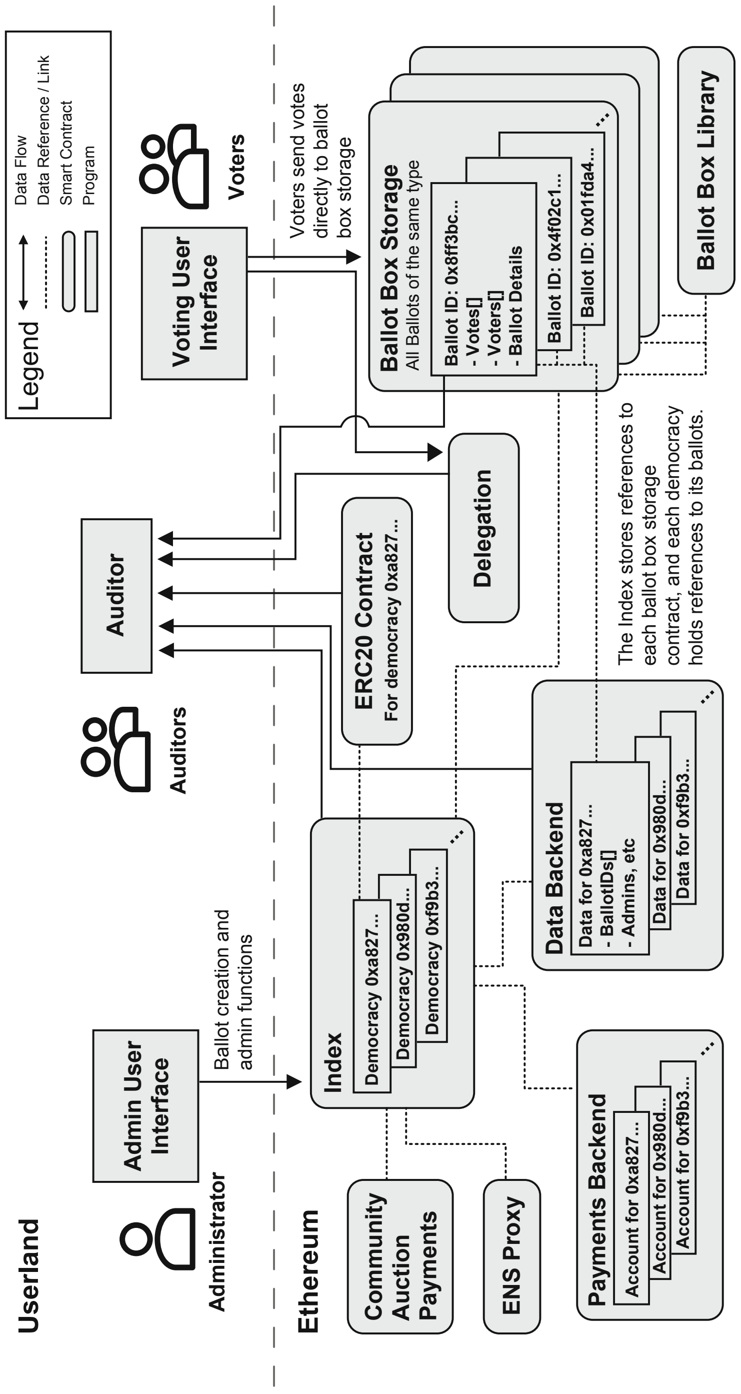

Fig. 13.3 The architecture for Tokenvote at deployment. Many alterations were needed before SecureVote considered the dapp production quality

In Chapter 5, we discussed ways in which blockchain can be used architecturally. In the Tokenvote architecture, we note the following:

- Tokenvote uses the blockchain as a storage element, as a communication mechanism for publishing ballots and votes, and as an asset management and control mechanism for payments and checking stakes.
- The use of blockchain as a computational element in this architecture is limited. Most computation is done off-chain, for the reasons outlined earlier. On-chain computation serves primarily to enforce checks such as authorization, what to store, and hash integrity. Other smart contract codes implement schemas for data, particularly ballot data stored in the ballot box storage. Not all data schemas are implemented in smart contract code, to allow for more flexibility in schemas that are immaterial to the core concerns of the solution.
- Regarding the integration of blockchain into a system as a component, this architecture is rather interesting, in that there are only in-browser components in addition to blockchain. However, running the auditor in-browser relies on SecureVote's full blockchain nodes, which introduces some level of trust on their integrity and truthfulness. Alternatively, anyone could host their own full node to function as an auditor. However, an auditor requires the full history including all states, which prevents syncing the blockchain through fast-sync, and requires over 1TB of SSD space at the time of writing.

SecureVote decided to use Ethereum as a technology platform for two main reasons. First, Ethereum's ecosystem was the most attractive, especially the support for ERC20 tokens. Second, despite its limitations, Ethereum was the best available option in terms of security, the execution environment, and the network. To benefit from lower fees, Tokenvote contracts can also operate on Ethereum Classic: transaction fees in fiat currency were a factor of 10–30 lower during the development of Tokenvote.

Qualities and Trade-offs

The common blockchain trade-off is between *transparency* and *confidentiality*, and this is present in Tokenvote. How voters voted needs to remain confidential, but each voter needs the transparency and certainty that their vote has been counted. Out of the options for how data can be stored (discussed in Section 6.3.3), SecureVote decided to use smart contract variables. This was to (i) avoid the need for offline interpretation as much as possible and (ii) ensure integrity, since, e.g., logs are computed in each full node and are not directly part of consensus.[7]

Cost, as discussed in Chapter 9, played an important role. Gas cost, complexity bounds, and limitations of the platform and their impact led to the revision of certain

[7]https://ethereum.stackexchange.com/a/1309.

design decisions, such as collapsing all ballots into one smart contract and storing most ballot details (e.g. title, description, and options) off-chain.

In terms of *performance*, discussed in Chapter 10, throughput plays the most important role. The latency requirement is that feedback to users confirming the recording of their vote should be given within reasonable time, on the order of 1–2 min. For both throughput performance and cost reasons, SecureVote minimized the complexity of voting transactions.

Dependability and *security* concerns (Chapter 11) are of course very important for a voting platform. In terms of *availability*, the most impactful issue would be transactions that are not included. This concerns primarily new or upgraded contracts and the transactions deploying them, since these transactions can incur high gas costs. This risk has partly been mitigated by collapsing all ballot contracts into a few reused contracts. *Reliability* is prominent when running full nodes with full history over a long time, due to high network load and high requirements on fast and sizeable disk space. *Maintainability* and *upgradability* are addressed using the patterns discussed throughout this chapter. *Safety* in the Lamport-Alpern-Schneider sense (see Section 11.3) is addressed through good coding practices and thorough testing with close to full code coverage, including negative tests that test failure cases. In terms of *integrity*, the solution relies on the strong, inherent integrity features of blockchain, and on implementing tight authorization checks for all functions. To ensure integrity for the stake weighting, stake holdings are taken from before and after each ballot. Also, the `Auditor` components ensure that all votes are counted. Auditing starts only 15 min after end of a ballot, which corresponds to approximately 60 confirmation blocks.

13.4 Details and Code Samples

In the following, we discuss some of the issues and lessons learned in detail and provide code samples where they are helpful.

13.4.1 Indexing and Externally Accessing Data

SecureVote's earliest component for Tokenvote was the multi-democracy framework (the `Index`), which would allow ballots to be created within a *namespace* that only the democracy's administrators had access to. Each time a ballot was to be created, a new `BallotBox` smart contract would be deployed. To begin, the voting smart contract MVP was reused, but a different approach was ultimately required.

```
// contract: Index
mapping (bytes32 => Democracy) public democs;
bytes32[] public democList;

struct Democracy {
  address erc20;
  address admin;
  Ballot[] ballots;
}

// return the number of democracies
function getDemocN() external view returns (uint) {
  return democList.length;
}
```

Listing 13.1 Referencing rich data types via unique IDs. SecureVote still uses this pattern today

Initially, each democracy had a unique identifier, via a hash generated from a number of parameters.[8] Unique identifiers are important because they facilitate cheap lookups via arrays or mappings.

In this case, SecureVote stored democracies in the `Index` as in Listing 13.1. This code sample uses patterns that are important when upgrading smart contracts. Since blockchain data cannot be moved easily during an upgrade, it should either be stored in a separate data contract, following the data contract pattern (Section 7.4.2), or remain in the older version of the smart contract and locked from further mutations. This is a direct consequence of Requirement R2.3 and the nature of smart contracts. For Tokenvote, we adopted the latter solution.

Note that in Listing 13.1, the array `democList` keeps an index of known keys, and the function `getDemocN` returns the number of democracies. These provide means for external users to discover information about the contract. Solidity does not have primitives for this nor otherwise directly supports discovery of this kind of information, so developers have to implement it specifically. The combination of this mapping, array, and length getter means that it is possible for everything but the ballots (stored in `Democracy.ballots`) to be easily read externally.[9] Reading the `Ballots` in each `Democracy` requires another set of getter functions (shown in Listing 13.2).

Accessing all data in a smart contract is an important prerequisite to ensure upgrade paths are available. For this reason, important state variables (for the most

[8]Choosing multiple parameters, particularly parameters outside of the user's control, is important to avoid collisions when using this technique.

[9]Technically it is always possible to read any arbitrary data stored on a blockchain at some level; in the case of Ethereum and Solidity, the curious reader can find out about accessing arbitrary variables in contract storage here: https://medium.com/aigang-network/how-to-read-ethereum-contract-storage-44252c8af925.

```
// contract: Index
struct Ballot {
  bytes32 ballotSpec;    // hash of the ballot specification
  BallotBox ballotBox;   // external smart contract reference
}

function getBallotsN(bytes32 democID) public returns (uint) {
  return democs[democID].ballots.length;
}

function getBallot(bytes32 democID, uint ballotID)
  public
  returns (bytes32, BallotBox)
{
  Ballot memory b = democs[democID].ballots[ballotID];
  return (b.ballotSpec, b.ballotBox);
}
```

Listing 13.2 Accessing nested dynamic elements in arrays and mappings

part) require external getters. This allows other smart contracts to read the complete state and helps maximize upgrade potential.

13.4.2 Splitting Up Contracts

Design patterns can increase code size. In general, adding an external function has a very low runtime overhead. However, the additional space required can easily be hundreds of bytes, depending on the number of arguments and data returned. Although this will usually be inconsequential, it can cause issues due to many blockchain platforms setting limits for the size and deployment cost of smart contracts. In Ethereum (due to EIP-170[10]) smart contracts are limited to `0x6000` bytes (approximately 24 KB). Refer also to the deployment risk of large contracts mentioned in Section 11.6.

In SecureVote's case, `Index` grew rapidly during development and hit Ethereum's deployment limit. Managing the size of smart contracts is necessary for any complex dapp deployed to Ethereum and similar networks. In this section we show three approaches: using auxiliary contracts for non-core but useful operations, using tightly coupled 'backend' contracts to offload storage and data processing, and using libraries to allow common code to be used by multiple contracts and hosted separately on the blockchain.

[10] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md.

Augmenting Smart Contraction Functionality via Auxiliary Contracts

Often, smart contracts serve two purposes: storing data and processing that data. For example, a ballot box smart contract might include logic for storing, tracking, and retrieving individual votes. However, to avoid separate `HTTP` calls to retrieve each vote, it makes sense to try and batch these requests to return all votes at once. A straightforward approach would be to add this function to the ballot box smart contract itself. Rather than just the function for individual votes, `getVote(uint voteID)`, we could add another function `getAllVotes()` to return all votes, and functions such as `getAllVotesFrom(address voter)` to return all votes from a particular voter. However, adding such functions increases contract size and introduces complexity to the integrity-critical primary contract.

As an alternative approach, we can use auxiliary contracts. In this example, the primary contract would retain the individual `getVote(..)` function. However, `getAllVotes()` would not be added to the primary ballot box contract. Rather, we create a second contract with a function `getAllVotes(BallotBox bb)` which calls `bb.getVote(..)` for every vote and returns a corresponding array. A single auxiliary contract can work with every instance of the primary contract (in this case `BallotBox`).

This technique has significant benefits: the auxiliary and primary smart contracts are only loosely coupled, so the auxiliary contract can be more easily upgraded or deprecated; code for complex data processing is moved out of the primary contract, reducing testing and attack surfaces; and there is greater flexibility around the type of data returned.

SecureVote uses auxiliary contracts for several purposes:

- Delegation between voters and self-delegation across network boundaries
- Retrieval and preprocessing of votes
- As a lookup table for human-readable names

Adding a Backend Smart Contract

Sometimes a large, interconnected contract must be broken up. We discuss several methods for this purpose here.

The simplest approach is to separate the data storage and longer-term data processing into different contracts. Exactly what is split between ‘frontend’ and ‘backend’ contracts should be based on an assessment of what functionality is likely to be more stable in the long term and what is likely to be more frequently upgraded or replaced. There is a small performance cost to this approach. As a smart contract is split up, it will need to know about the backend contract (and load its address from storage), and the backend contract will need to grant permissions to the frontend contract and verify these permissions. Each call between contracts will also incur some additional fee for invocation and parameter passing. So, this approach is valuable for infrequently called functions (such as democracy creation) rather than for frequently called functions (such as voting).

In Tokenvote, the `Index` is split in this manner. For example, when democracies are created (and administrated), most of the operations (like calculating the unique ID, storing data, and setting initial permissions) take place in the backend. Only minimal functions are left in the frontend contract. As mentioned, a consequence of this approach is that two sets of permissions must be verified. The first is for the user when calling the frontend contract, and the second is for the frontend contract when calling the backend.

This approach has some nice properties. More than one `editor` can be added, provided the backend authentication has been architected appropriately. This means an alternate frontend can be introduced, or the original frontend swapped out, augmenting or introducing functionality.

Tokenvote required two dedicated backends and several other supporting smart contracts all using this approach. Examples are Tokenvote's ENS (Ethereum Name Service)[11] integration and an auction system for publishing a particular kind of ballot. The latter contract is currently simply a placeholder for future functionality.

Using Libraries

Another way to split up functionality and code is to use a library. These are deployed like Solidity contracts but cannot be called directly and have no state of their own. Rather, they are 'linked'[12] (similarly to the way libraries are linked in `C`) and allow the library to modify the state of the contract calling it. This is particularly useful for logic repeated in multiple contracts. Examples of these sorts of libraries are in the OpenZeppelin[13] framework, which includes many code examples, base contracts, and useful libraries, such as the ubiquitously used `SafeMath` library that has safety checks on inputs.

SecureVote uses libraries in a few specific cases. First, a library is used to handle code that needs to be identical across contracts, such as extracting data from packed variables. Second, a library is used to version and manage the handling of votes and ballots, leaving the *container* contract (which holds many votes and ballots) with a cleaner and simpler codebase. Third, libraries are used like macros across multiple contracts to wrap common multistep operations. Only the first two uses reduce the calling contract's size.

[11] A simple name system has been implemented over Ethereum allowing names like `data61.eth` to be registered and resolved to an address (in the same way domain names resolve to an IP address). In this case the Tokenvote `index` is resolved via `index.tokenvote.eth`. More information can be found at https://docs.ens.domains/en/latest/.

[12] Libraries are used via the `delegatecall` operation, which means 'run this code as if it were inline here and give it direct access to my storage'.

[13] https://openzeppelin.org/.

Libraries are not as simple to upgrade as contracts, and while this is technically possible,[14] it requires preparation and a deep understanding of the underlying blockchain. It can be easier and safer to upgrade an individual contract linked to a new library, rather than the library itself.

13.4.3 Upgrades and Trade-offs

Recently, there has been increased interest in upgradable smart contracts. There are many reasons to upgrade, including to add functionality, to mitigate potential attacks, or to fix bugs. In this section we will describe two techniques used by SecureVote which, when used together, allow for atomic upgrades without downtime for other smart contracts. We also describe how SecureVote plans to improve their upgrade procedure to expand atomic interactions to all users and eliminate any sorts of race conditions entirely.[15]

We start by looking at a simple case of replacing an existing contract. Then, we examine the upgrade of a prototype delegation contract and how SecureVote currently manages upgrading the `Index`. Finally, we describe an oversight and how SecureVote plans to address this.

Replacing Smart Contracts

When the interface to a smart contract is well known, it is trivial to replace it at a future date. A simple way to upgrade a contract factory is shown in Listing 13.3. There are three smart contracts here: an instance of `Frontend` and two instances of contracts which implement the `Backend` interface. The `Index` instance stores a reference to a `Backend` implementation, and upgrading is as simple as replacing this reference. This method is very general and forms the foundation for other methods discussed below.

At compile time, Solidity knows about the interface of remote smart contracts, to generate logic for communicating with them and to check type safety. At runtime, these checks are not performed. In Listing 13.3, the `doUpgrade` function accepts an address `newBackend`, and after the contract is deployed, the owner is free to call `doUpgrade` with any address, including that of a smart contract which does not adhere to the `AuxContract` interface. We can use this lack of runtime checks to implement upgrade schemes.

[14]One way to implement upgradable libraries: https://blog.zeppelin.solutions/proxy-libraries-in-solidity-79fbe4b970fd.

[15]Although well-designed smart contracts can interact with Tokenvote atomically, a race condition exists where an upgrade could take place between the creation of a transaction to the index and the inclusion of said transaction. In this case the transaction would only affect the old index, and would revert in most cases as the index would have lost permissions to modify data on the backend.

```
interface Backend {
  function replaceWith(AuxContract newExternal) public view;
}

contract Frontend {
  Backend _backend;
  address owner;

  constructor(Backend initBackend) public {
    _backend = initBackend;
    owner = msg.sender;
  }

  function doUpgrade(Backend newBackend) public {
    require(msg.sender == owner);
    // let the current _backend know we are upgrading, if
        needed
    _backend.replaceWith(newBackend);
    _backend = newBackend;
  }
}
```
Listing 13.3 A simple, general way to replace smart contracts

SecureVote's First Upgrade

Delegation in voting systems is usually straight forward. A voter can choose someone else (the delegate) to act on their behalf, and if the voter abstains the delegate's vote is used instead. So each delegate votes with the combined power of all their delegators (the voters doing the delegation). Many systems of delegation also include delegation by categories or similar ways for voters to choose one of multiple delegates depending on context.

Less than a month after deploying their first delegation smart contract, SecureVote found they had made an oversight. Although users could make and check delegations, there was no way to iterate through them, and there was no way to find delegators given some delegate. Although, functionally, delegation only works in one direction (where a voter chooses a delegate), *resolving* delegations is more complex. There were two complications:

- In standard ERC20 implementations, there is no complete list of account holders. This means it is impractical to iterate through all potential voters to check their delegations.
- If the voter abstains, only the delegate's vote and address are known. Without delegation backlinks it is not possible to efficiently find the voters who have selected a particular delegate.

```
contract Version1 {
  mapping (uint => bytes32) data;
  function getData(uint i) external returns (bytes32) {
    data[i];
  }
}

contract Version2 {
  mapping (uint => bytes32) data;
  Version1 prevContract;

  constructor(Version1 _prev) public {
    prevContract = _prev;
  }

  function getData(uint i) external returns (bytes32) {
    bytes32 r = data[i]
    if (r == bytes32(0))
      r = prevContract.getData(i)
    return r
  }
}
```

Listing 13.4 The general pattern of a layered upgrade

Since looping through (and caching) all historic delegations was not particularly elegant, SecureVote opted to upgrade their delegation functionality by implementing a second delegation contract which operated ‘over the top’ of the first. Only when this new contract could not find a delegation would it check the original contract. Since SecureVote references most contracts via ENS names, the upgrade was a simple matter of deploying the new contract and updating the ENS resolution. This general pattern is shown in Listing 13.4.

This simple pattern is very useful in the right contexts. Ideally the first contract can be locked down when upgrading, such that no data can be added, but often this is not necessary. That is because the first contract is simply a fall-back, and only in the case that no *new* data exists (which would be stored in the second contract) is the first ever called.

There are also some drawbacks to be aware of. If users continue using the first contract, they might be able to change the data returned from the second, newer contract. The approach may also require software updates depending on the dapp in question. Finally, this technique does not work when contracts need to return dynamic arrays or strings, as these cannot be passed between contracts. So whether this technique is appropriate depends on the nature of the contract being upgraded.

SecureVote’s delegation contract was designed to handle all delegation requirements across all democracies, so voters could simultaneously delegate on a per-token basis and globally. If a delegation for a particular token was not found, the global delegation would be used. This is useful for self-delegating from a cold wallet to a hot wallet. SecureVote’s first improvement was logging all known tokens for which delegations had been made, allowing them to iterate through all known tokens easily.

SecureVote's other improvement was to look up all delegations to a particular address, exposed in a function called `findPossibleDelegatorsOf`. This was done by looping through all known delegations and constructing an in-memory array of delegations matching the delegate in question. One consequence of this approach is that only *potential* delegators are returned; delegations need to be checked individually before being treated as valid when calculating the results of a ballot. This demonstrates a trade-off between work done on-chain and work done off-chain. If backlinks were stored with the delegations themselves, the contract would also require additional data structures and logic to store and maintain the accuracy of these backlinks, increasing the cost for the voter. However, the chosen approach implies that the `findPossibleDelegatorsOf` function cannot be called from other smart contracts. When this function is called, the computation is only ever done on the Ethereum node responding to the call, not across all full nodes on the Ethereum network itself.

Complex Upgrades

The method above may be applicable for individual contracts but does not support upgrades of a complex system of contracts. The pattern used by SecureVote in Listing 13.5 (called an 'upgrade pointer') is suitable for singly linked contracts, where the contract being called might be upgraded. In this example `AContract` would call `checkIndexForUpgrade()` before sending any data to `Index`.

This example code shows the core idea, but the `doUpgrade` function could be easily extended to allow for upgrade hooks or notifications to be sent to other contracts. SecureVote extensively use such an extension to manage the multiple interactions and permissions between Tokenvote's smart contracts. A sample from their `Index` contract is shown in Listing 13.6.

Multiple other contracts are notified of an upgrade via their `upgradeMe (address)` method. SecureVote use this mostly for permission management, but it supports other complex upgrades. When `Index` calls `upgradeMe` on another contract, the permissions of `Index` are transferred to the new contract. Note also the modifiers `only_owner`, which allows only the owner of the contract to execute this function, and `not_upgraded`, which checks that the function can only be called on the latest version of the `Index`.

Atomic Upgrades and Tokenvote

Two lines in Listing 13.6 differ from the others: `ensOwnerPx.setAddr (nextSC);` and `ensOwnerPx.upgradeMeAdmin(nextSC);`. The contract `ensOwnerPx` is an 'owner proxy' for `index.tokenvote.eth`, defined using the Ethereum Name Service. Since this contract has administrative control over this

```
contract Upgradable {
  address public _upgradePtr;
}

contract Index is Upgradable {
  function doUpgrade(address next) external {
    require(msg.sender == owner);
    _upgradePtr = next;
  }
}

contract AContract {
  Index index;

  function checkIndexForUpgrade() internal {
    if (index._upgradePtr() != address(0))
      index = Index(index._upgradePtr);
  }
}
```

Listing 13.5 This pattern allows other smart contracts to know about an upgrade and act accordingly

```
function doUpgrade(address nextSC) only_owner() not_upgraded
    () external {
  doUpgradeInternal(nextSC);
  backend.upgradeMe(nextSC);
  payments.upgradeMe(nextSC);
  ensOwnerPx.setAddr(nextSC);
  ensOwnerPx.upgradeMeAdmin(nextSC);
  commAuction.upgradeMe(nextSC);

  for (uint i = 0; i < bbFarms.length; i++) {
    bbFarms[i].upgradeMe(nextSC);
  }
}
```

Listing 13.6 A sample from SecureVote's Index contract showing how upgrades notify other linked contracts. © 2018 SecureVote, reprinted with permission

name, it can expose functionality (like setting the address associated with the ENS name) to multiple other accounts. In this case, the other accounts are a SecureVote cold wallet and the `Index` contract.

When other smart contracts interact with Tokenvote, they first resolve the ENS name to an address to ensure they are invoking the current version of Tokenvote and not an old contract. Somewhat similarly, when voters use the Tokenvote UI, the software finds the `Index` address via an ENS lookup, but this only guarantees the address is correct at the time of the lookup. As mentioned above, there may be a race condition if an upgrade is made after the user loads the UI but before they create a new ballot.[16]

As an aside, SecureVote have made extensive use of events in their contracts, though unfortunately forgot to add an event for upgrades. If they had included one, it would be entirely feasible for the UI to listen for upgrade events and to reload contract instances when such an event is emitted. This would reduce the risk of race conditions causing failed transactions.

13.4.4 Reducing Complexity and Cost

In preparation for production deployment, SecureVote began benchmarking and optimizing many of the methods users would call regularly, like casting a vote or creating a ballot. Several optimizations greatly reduced transaction fees for the end user. One advantage of platforms like Ethereum is that the performance and gas cost of function invocation can be measured accurately in test environments. As mentioned in Chapter 9, rather than implementing dynamic gas costs, Ethereum's design opts for a dynamic price per gas operation. Thus measuring (and optimizing) gas costs is separate from the price of a transaction.

During SecureVote's benchmarking, the primary offender in terms of gas use was identified to be the creation of a new ballot. The original architecture used a contract factory to deploy individual contracts to manage each ballot. Although this allowed them to reuse much of the code from the MVP, as they added features the cost of deployment grew to 3,000,000 gas. During the worst periods of congestion,[17] this corresponded to a transaction fee of around US$30. SecureVote felt this was too high and designed a more sustainable architecture for ballot creation.

A standard solution to this kind of problem is to refrain from deploying new contracts. Instead of deploying one contract per ballot, SecureVote would deploy one contract per *type of ballot*: a ballot storage contract, implementing the data contract pattern described in Section 7.4.2. In order to maximize code reuse, ballot-specific functionality like recording votes was not included in the ballot storage. Rather, this was refactored out into a library of its own, allowing SecureVote to reuse the ballot storage contract and interface with different libraries for vastly different kinds of ballots. The results of this new architecture reduced ballot creation cost to between 200,000 and 300,000 gas, a reduction of 85–95%.

However, this kind of change has many flow-on impacts. For example, their previous voting and auditing architecture included the assumption that each ballot lived at its own address. Under this new pattern, many ballots lived at the same

[16] Direct user interaction with the index only occurs on write operations; read operations call the backend directly, so an update to the index does not affect this functionality.

[17] Such as the CryptoKitties Congestion Crisis of late 2017. https://media.consensys.net/the-inside-story-of-the-cryptokitties-congestion-crisis-499b35d119cc.

address, and each had a unique identifier. Not only did voting-specific code require updating, but the entire state model of the UI required refactoring to accommodate ballot storage contracts, each holding multiple ballots. Even the URL routing logic needed to be updated.

In general a ‘hub and spoke’ architecture (where new spokes are created via newly deployed contracts) should only be used in cases where the cost of such deployment is warranted. Any developers using this architecture should strongly consider whether refactoring to a single, heavily used contract will improve performance, user experience, and maintainability.

13.5 Summary

In this chapter, SecureVote described their experience of moving from an MVP to a production dapp. We contrast the initial, planned architecture and the final one, which resulted from many lessons learned and optimizations made during the development. We also describe how the architecture relates to the functions blockchain can play, which patterns are used and how, as well as the considerations of the qualities and the trade-offs in the architecture. Finally, the previous section covers many details, code samples, and (occasionally hard) lessons learned.

13.6 Further Reading

The Tokenvote source code is available on GitHub at https://github.com/secure-vote/sv-light-smart-contracts.

Solidity documentation, including a very good ‘by example’ section, is available at https://solidity.readthedocs.io/en/latest/.

The end-to-end verifiable voting design *Prêt à Voter* is described in Ryan et al. (2009).