GenVote: Blockchain-based Customizable and Secure Voting Platform

Praneeth Babu Marella[1], Matea Milojkovic[2], Jordan Mohler[3], and Gaby G. Dagher[1]

[1] Boise State University, Boise, Idaho, USA
[2] Winthrop University, Rock Hill, South Carolina, USA
[3] University of Denver, Denver, Colorado, USA

Abstract. Electronic voting has been popularized in recent years as an alternative to traditional voting. Even though electronic voting addresses the problems that traditional voting brings, it is not a perfect solution. Electronic voting brings its own set of concerns which include: election fraud, voter privacy, data integrity, and confidentiality. To ensure fairness in electronic voting, a centralized system is required and the complete process has to be overseen by an authority. Due to these requirements it can be very expensive to roll-out on a large scale during every voting period. Blockchain, the distributed data structure popularized by Bitcoin, can be integrated into electronic voting systems to alleviate some the problems involved with them while being cost-effective. With the use of blockchain, we propose a voting system that is easily accessible, customizable, transparent, and in-expensive. GenVote is a distributed electronic voting system that utilizes Ethereum Blockchain, smart contracts, and homomorphic encryption to achieve a transparent voting process with non-authority based tallying and voter privacy. GenVote also allows the ballot creation and voting process to be customizable with different types of ballots and logic based voting. GenVote is currently a viable solution for university-scaled elections and has been deployed on Ethereum Ropsten testing network to evaluate its viability and scalability as an electronic voting system.

Keywords: Blockchain · Ethereum · Smart Contracts · Voting · Privacy · Encryption.

1 INTRODUCTION

Voting is a fundamental part of every democratic process. It allows for us to have a voice in government process and be represented for issues that matter most to us. With the technology advancements we made, it could be assumed voting has become easily accessible for everyone and their votes are securely tallied. However, even at the university level, voter fraud has been a problem. In 2016, a fraud at Kennesaw State University brought forth the issues of voter registration fraud. Students at the university believed they had signed up to vote in the 2016 Presidential Election without knowing that their registration forms were simply trashed. Due to this, the students were unable

to cast their votes on the day of the election [4]. The same year city officials in Green Bay, Wisconsin refused to allow early voting on the University of Wisconsin's satellite campus[5]. So the only other option for students to vote was driving fifteen-minutes to a near-by voting location. But the location was only open during business hours so it was even more difficult for students to access the voting site. This led to a lot of students being excluded from voicing their opinions on the election. To make it even more difficult, student IDs are not considered a suitable identification for voter registration at many locations. Voter registration fraud and lack of access to voting sites for university students are important issues that must be addressed.

Secure and privacy preserving voting systems are necessary for university-scale elections. For instance, at many universities, one of the major objectives of the student government organization Associated Students (AS) is to "advocate for the interests of students at the University". To achieve this goal, they must provide a easily accessible platform for students to voice their choice on different matters. This is where electronic voting systems come in to fill the need. One such system, TIVI, uses digital authentication of voters through facial biometrics: specifically, selfies [6]. Although TIVI solves the accessibility issue previously mentioned, it does not completely stop fraudulent activity. Using public photos and 3-D rendering, malicious users can break into accounts [7]. Helios is the first online, open-audit voting protocol. The primary goal of Helios is data integrity but it also provides voter privacy to an extent. To ensure data integrity, any observer may audit the election process during its active voting period. At the start of the voting process, voters name and encrypted vote are posted. But after the election is completed, the votes are shuffled and then tallied to compute the end result. Helios claims to be the optimal voting system for small groups where coercion is unlikely but private voting is necessary [2]. Although Helios maintains data integrity, voter privacy is not preserved to the utmost. Another major limitation associated with current electronic voting systems is voting fraud in the form of database/platform manipulation [14]. With the use of centralized data storage, current implementations of electronic voting platforms are susceptible to vote altering. Our systems aims to address the security concerns of current electronic voting systems by incorporating blockchain elements to it. Due to the distributed nature of blockchains, voting systems that use a blockchain to record and tally their votes do not have a central point of failure [5]. Voters can also verify their vote has been recorded and not been tampered with by inspecting the blockchain. This can be achieved because every vote is recorded on the distributed ledger through transactions to the blockchain [1].

The blockchain is a distributed append-only data structure that grows through adding blocks. The blocks contain transactions submitted by participants, or nodes, of a peer-

[4] http://bettergeorgia.org/2016/09/11/a-different-kind-of-voter-fraud-one-to-actually-be-worried-about/

[5] https://www.thenation.com/article/city-clerk-opposed-early-voting-site-at-uw-green-bay-because-students-lean-more-toward-the-democrats/

[6] https://eandt.theiet.org/content/articles/2016/10/voting-online-made-possible-with-selfie-recognition-technology/

[7] https://www.wired.com/2016/08/hackers-trick-facial-recognition-logins-photos-facebook-thanks-zuck/

to-peer network. When a transaction is submitted, it goes into a pool that a validating node, also referred to as a miner, can extract. Miners can gather a set of transactions from the pool into a block and append it to the blockchain. In order for a miner to append his block to the blockchain, he must complete a consensus proof such as Proof-of-Work or Proof-of-Stake. Participating in the consensus proof process requires either computation power or a stake which is cost of participation. Once a transaction has been validated as part of a block and added to the chain, it cannot be altered. Because of these properties, the blockchain is considered an immutable, secure data structure. The Ethereum Blockchain expands this functionality by implementing smart contracts [7].

Smart Contracts are blocks of code written in specific languages, usually dictated by the blockchain being used, and contain methods/events. Methods contained within a Smart Contract allow for interaction with the blockchain through either external or internal calls. Smart Contracts are stored within the blockchain so once the code is deployed, it cannot be altered and is publicly available for anyone to interact with it. In Ethereum, to preserve the network, every interaction with a smart contract that changes its state needs to pay a computational fee, referred to as "gas". Gas is the unit of measure used to calculate amount of work a validating node will need to perform for an operation and the gas price, the amount user needs to pay, is measured in terms of ether in Ethereum [7]. Smart contracts are also extended to private implementation of blockchains; as opposed to public blockchains, private blockchains are implemented to be utilized within a single organization. While this sacrifices the decentralized nature of the blockchain, it enhances the privacy of the blockchain [6]. For the purpose of our system, GenVote implements a private blockchain. We believe that a private blockchain is best suited for maintaining the integrity and privacy of the ballots within an organization scale.

Our proposed system is an extension of our previous work [8]. We expand the functionality of our previous system by allowing voters to cast votes logically and giving ballot creators the freedom to create different types of ballots. Our system still uses similar concepts as [10], [4], and [9], specifically in the areas of privacy and smart contracts. Votes in all those systems are encrypted and stored on the blockchain to achieve voter privacy and ballot transparency. These systems also utilize hashing to ensure strong data integrity within the voting process. In [10], the voting system may have an optional round in which voters hash and post their encrypted vote to the blockchain. Transactions consisting of votes are hashed before being stored on the blockchain in the system described in [4]. In addition, [10] uses the smart contracts as part of their voting system to allow for easy election process and perform cryptography functions.

1.1 Contributions

Our implemented system, GenVote, provides a secure and private electronic voting system that is easily accessible and customizable. GenVote is intended to be used in a university scale voting system. GenVote utilizes smart contracts in Ethereum and Paillier Homomorphic Encryption to achieve voter privacy and ballot integrity. Our system also allows elections to be customized with different types of ballots. Creators, of the ballots, have the freedom to create polls, standard elections, or first to X number of votes. Creators also have the option to either open voting for everyone or whitelist a certain set

of voters to participate in the ballot. Voters have the option to vote in a traditional way or opt to vote logically using one of these options: vote for the current leading, vote for the runner-up, or vote for the losing candidate/choice. GenVote provides voter privacy on all our ballots by homomorphically encrypting every vote, tallying, and revealing the vote count using the Paillier cryptosystem. To maintain transparency, all ballot and voting data is publicly available as part of the smart contracts within the blockchain used in our system.

2 PRELIMINARIES

2.1 Blockchain Mining

'Mining' is a process that is used in a trust-less blockchain network to reach a consensus about the state of the blockchain [11]. The role of a 'mining' node is to verify a group of transactions into a block by solving a computationally intensive puzzle. The puzzle involves the 'mining' node to find the hash of the block that begins with a certain number of zeros. To achieve this, a number called a 'nonce' is included in each block; each time miners hash the block without solving the computational problem, they increment the nonce and rehash the block [11]. The difficulty of solving the hashing problem is described as 'Proof of Work,' signifying the computational power and difficulty needed to append a new block to the blockchain [11]. Once the puzzle has been solved the block can be appended to the blockchain and the 'mining' node is rewarded with the appropriate cryptocurrency.

2.2 *Eth.calls*

Every valid transaction executed is stored on the blockchain [7]. Due to this, blockchains can suffer from scalability issues. Valid transactions sent to smart contracts in the Ethereum blockchain are considered state changeable calls and consume gas. To reduce gas consumption and the number of transactions on the blockchain, the Ethereum blockchain allows *eth.calls* to be utilized in addition to transactions. *Eth.calls* allow nodes to send messages to other nodes or smart contracts to retrieve its current state without storing the message on the blockchain [8]. Therefore, *eth.calls* are similar to simulations of transactions. By executing *eth.calls* to send notifications/messages or to retrieve current states, the size of the blockchain can be greatly reduced [8].

2.3 Paillier Encryption

Full homomorphic encryption allows us to perform computations on encrypted data. The encrypted data can then be decrypted to reveal the same value as it would be if the computations were done on plain data [16]. However, fully homomorphic encryption requires fully modular multiplication which can be computationally intensive and very slow [15]. However, because of the advantages provided by homomorphic encryption, it is still a prominent encryption scheme and partial homomorphic encryption scheme

[8] https://github.com/ethereum/wiki/wiki/JSON-RPC

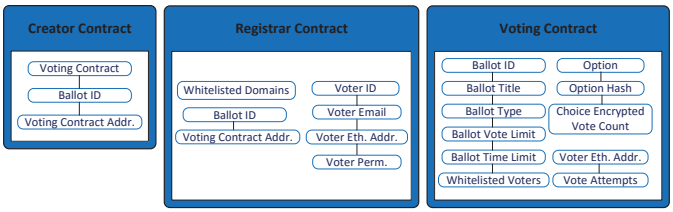

Fig. 1. Memory field structure of smart contracts in GenVote, where lines between fields represent relational data [8].

has been introduced for faster encryption. One such scheme is the Paillier Homomorphic Encryption. This probabilistic public-key encryption method supports addition and multiplication [16]. Paillier system can homomorphically add two ciphertexts but it can only multiply a ciphertext with a plaintext integer. Since the Paillier system cannot homomorphically multiply two ciphertexts, it is considered partially homomorphic. The process of encryption is not completely intuitive: multiplying ciphertexts is equivalent to adding the plaintexts and raising a ciphertext to the power of another ciphertext is equivalent to multiplying the plaintexts [12]. To achieve the advantages of homomorphic encryption without the substantial reduction in processing speed, Paillier Encryption is one of the ideal encryption schemes.

2.4 MetaMask

MetaMask is a web broswer plugin that was created to make it easy for average users to interact with Ethereum blockchain based Dapps. MetaMask acts as an Ethereum browser, which allows the users to easily manage their Ethereum wallet and interactions with decentralized applications, or Dapps, and smart contracts. Using MetaMask removes the need for users to download a local copy of the blockchain. Users are also able to easily manage multiple accounts and switch between test or main network [8] . MetaMask facilitates the user transaction broadcasting by using a set of trusted nodes that relay the transactions to the pool. Since transactions are signed using the sender's private key, which is stored locally on the user's machine, MetaMask cannot impersonate the user and send transactions on the user's behalf or modify outgoing transactions. MetaMask makes it convenient and secure for average users to interact with Dapps on the blockchain using a simple web browser.

3 PROPOSED SOLUTION: GenVote

3.1 OVERVIEW

Prior to discussing our proposed voting system, we would like to mention that the Ethereum blockchain used in our system has not been modified in any way and the

[8] https://github.com/MetaMask/metamask-extension

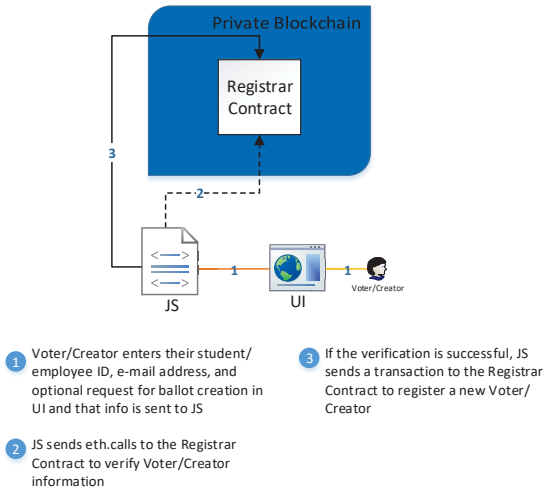

Fig. 2. The process for registering a voter in GenVote, where black dotted line represent eth.calls and solid line represent transactions to the blockchain [8].

standard proof-of-work was used for validating transactions. Our system, GenVote, uses existing functionality and features provided by Ethereum and Solidity to provide the ability for creating and voting on ballots. Our implementation consists of three smart contracts coded in Ethereum's Solidity language, two scripts written in JavaScript, and one HTML page. GenVote is an open source project and the entirety of the code is available for public use [9].

In order to participate in the system, the users have to utilize MetaMask plugin or become a node by downloading the Ethereum blockchain. We assume the administrator, creators, and voters have one of options setup and can create and manage Ethereum accounts to interact with our system. We utilize Ethereum's Web3 framework internally, this allows our users to easily manage signed transactions and interactions with the Ethereum blockchain. The only action required of users when registering, voting, or creating ballots using MetaMask is to use their passwords/private keys to unlock their Ethereum accounts and securely interact with the blockchain. If the user decides not to utilize the Metamask plugin then they are responsible for running a node on their local machine and managing appropriate accounts to interact with our system using Web3 [8].

A brief description of all the user parts of GenVote follows:

- **Administrator** is responsible for deploying the initial Registrar and Creator smart contracts. The administrator also has the ability to grant or revoke ballot creation permission for registered voters/creators.
- **Voter** registers in our system with a valid student/employee ID and e-mail address to vote on given ballot ID numbers.

[9] https://bit.ly/2GEVtwk

– **Creator** is a voter with ballot creation permission.

A brief description of the front/back-end pages implemented in GenVote follows:

– **VoteUI.html** page is the user interface for our users. This page allows users to enter necessary information for each of the different use cases. Once the user enters the necessary information, the corresponding click buttons will invoke functions in *App.js*.
– **VotingApp.js** gathers information from *VoteUI.html* and interacts with *Crypto.js* and the Ethereum Blockchain. For each corresponding request from *VoteUI.html*, it utilizes *eth.calls*, *Crypto.js* server calls, and Ethereum transactions to verify, encrypt/decrypt votes, and store ballot/vote information.
– **Crypto.js** acts as a cryptographic server. All votes are encrypted, homomorphically added, and decrypted using the Paillier homomorphic encryption system key pair in this server.

A brief overview of the smart contracts implemented in GenVote follows:

– **Registrar.sol** takes the role of a record and gate keeper. It keeps track of all registered voters and creators, ballot IDs, voting contract addresses, and whitelisted e-mail domains. Information regarding the voter and different ballots are linked together in the contract, as seen in Figure 1. This allows the contract to perform voter verification, permission modification, and *Voting.sol* address retrieval easily. The owner of this contract is the administrator.
– **Creator.sol** functions as a spawner for new *Voting.sol* contracts. The Creator defines the voting contract details from the required information entered in *VoteUI.html*. The owner of this contract is the administrator.
– **Voting.sol** functions as a virtual ballot and handles the voting on the ballot. Another set of voter verification, including vote attempts and ballot time limit, is also conducted in this contract. As we can see in Figure 1, ballot title and the choice encrypted votes are also stored here so that we can retrieve at later stages. The owner of this contract is the contract creator.

A brief overview of the types of ballots and voting styles implemented in GenVote follows:

– **Polling Ballot** lets you create a ballot that displays results live as users participate in this ballot and ends when the ballot reaches its end time. The choice that has most votes wins.
– **Election Ballot** is a traditional election style ballot that displays its results after the ballot end time has reached and the choice with the most votes wins.
– **First to X Ballot** is a hybrid ballot that displays its results live and when one of the choices reaches the required number of votes to win then the ballots ends. The winner is the one that reaches the X votes defined by the creator.
– **Vote Limiting on Ballot** can be imposed by the creator when creating any type of ballot. This will create a limit on the number of votes each voter/Ethereum address can send to a ballot.

- **Whitelisting on Ballot** can be utilized by the creator when creating any type of ballot. This will set a restriction on who can participate on a particular ballot. Currently the whitelisting is based on e-mail but can be customized to many different forms of identification whiteliting.
- **Traditional Voting** can be used on any type of ballot. The user manually decides which choice to vote for and submits the vote. The vote is then immediately processed and updated on the UI once its been verified.
- **Logical Voting** is a special type of voting. The user chooses from one of the three options: Winning Vote, Runner-Up Vote, and Losing Vote. Depending on which option is chosen, the system determines who to vote for and notifies the user once the vote has been verified. The user can also choose to allow the system to vote on their behalf at a later time with one of the options.

3.2 Initial Setup

The initial setup process needs to be kick-started by the administrator of the system. The administrator needs to deploy *Registrar* and *Creator* contracts to activate the system. The administrator is also responsible for including a set of e-mail domains that are permitted to be used for registering in our system. Once the system has been activated, the users can start registering, creating, and voting on ballots in our system.

3.3 Register Voter

Since GenVote was implemented with a university setting in mind, anyone with a student/employee ID number and a specific e-mail can register and use the system. The user has to have an e-mail that contains one of the whitelisted domains setup by the administrator to register as a voter and request ballot creation permission. Once the user has entered the ID number, e-mail address, and ballot creation permission fields in the registering section of *VoteUI.html*, the information is parsed by *VotingApp.js*. The parsed information is then used to make *eth.calls* to the *registrar* contract, as seen in step two of Figure 2, to verify the user's e-mail and registration status. If the user passes the checks, then *VotingApp.js* sends a transaction to the *registrar* contract to store the new voter information, including the voter's ID, e-mail, and Ethereum address. The user's e-mail address and Ethereum address are linked so we can use it in the future to prevent the user from re-registering. Users that also requested access to create ballots are placed in a queue so that they can be manually reviewed by the administrator.

3.4 Create Ballot

Users with ballot creation ability can use the ballot creation section in *VoteUI.html* to spawn new voting contracts using the *Creator* contract. To create a new ballot, the creator must provide their registered e-mail address then choose whether this ballot will be a election, poll, or first to X votes type. Then determine the title of the ballot to let voters are aware of what they are voting on, voting options for the ballot, and the number of votes allowed per voter on this ballot. During this process, the creator can

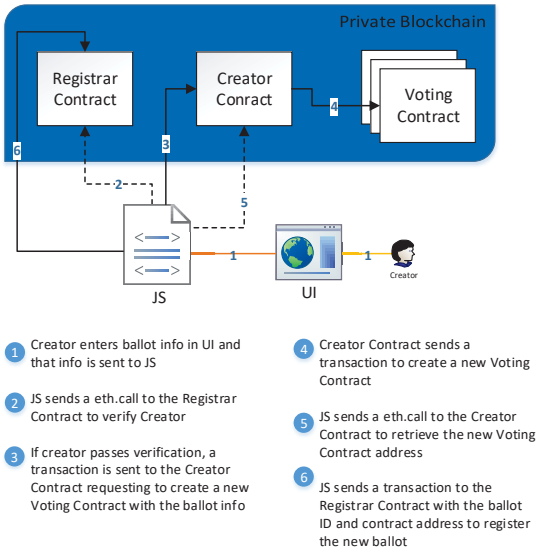

Fig. 3. The process for creating a ballot as a creator in GenVote, where black dotted lines represent eth.calls and solid lines represent transactions to the blockchain [8].

also opt to make it a whitelisted ballot. If the whitelisted ballot option is chosen, the creator is required to enter the list of e-mail addresses allowed to vote on the ballot. But if the creator chooses to not make it a whitelisted ballot, everyone with a registered e-mail address will be allowed to vote on the ballot. Lastly, the creator sets the ending date and time for the voting period for the ballot.

Once the creator has provided the necessary information to create a ballot, *VotingApp.js* parses the information and continues to step two in Figure 3 to verify the creator status. To verify the status, *VotingApp.js* sends the two *eth.calls* to the *Registrar* Contract to verify the creator's provided e-mail address is registered in our system and whether the request originates from the registered Ethereum address linked to the e-mail address or not. If those two checks are passed, then *VotingApp.js* sends the third *eth.call* to determine if the user has the permission to create ballots. If it was determined that the user is allowed to create ballots, *VotingApp.js* gathers the parsed data and a randomly generated ballot ID number to include in a transaction to the *Creator* Contract. The *Creator* Contract uses the information provided in the transaction to create a new *Voting* Contract and deploy it onto the blockchain. Once the new *Voting* Contract has been deployed successfully, the contract address is returned to the *Creator* Contract.

VotingApp.js then sends a final *eth.call* to the *Creator* Contract to retrieve the new *Voting* contract address to register it in the system. Step six in Figure 3 shows the transaction to the *Registrar* Contract for storing the newly linked ballot ID and contract address. The ballot ID is then displayed afterwards and the *creator* is reminded to write down this ballot ID since it will be the unique identifier for this ballot. The ballot identifier is then used by voters to load and vote on the ballot.

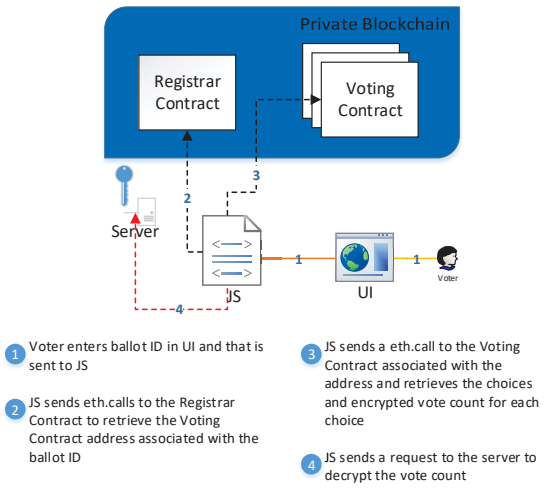

Fig. 4. The process for loading a ballot as a voter in GenVote, where black dotted lines represent eth.calls to the blockchain and red dotted line represents decryption calls to the server [8].

3.5 Load Ballot

Users can load the ballot information by using the unique ballot identifier provided to the voters by the Creator of the ballot. After loading the ballot, the user can check the results of the ballot or vote on the ballot if the voting period has not passed. Once the voter enters the ballot ID in load ballot section of *VoteUI.html*, *VotingApp.js* sends an *eth.call* to the *Registrar* Contract to determine if the provided ballot ID is valid. If the ballot ID is valid, then the title, voting options, and the encrypted vote count for each choice are gathered from the *Voting* contract. If the voting period has ended or if the ballot is a poll or first to X type, the vote count is decrypted and displayed in *VoteUI.html*. In order for the decrypted vote count to be displayed, there is a step involved, which can be seen in Figure 4 as step four, that send the encrypted vote count to the *Crypto.js* server. *Crypto.js* server will then decrypt the votes and send them back to *VoteUI.html*.

3.6 Vote (Traditional)

To vote on a ballot, the user needs to first load the ballot. Once the ballot has been loaded, the user types in the choice they want to vote for along with their registered e-mail address in the voting section of *VoteUI.html*. The voting information is parsed by *VotingApp.js* and sends an *eth.call* to the *Registrar* Contract to verify the voter as well as the Ethereum address linked to the e-mail. If the voter is verified successfully, then another *eth.call* is sent to the *Voting* Contract to check if this ballot is whitelisted. If the ballot is whitelisted then it checks if the voter's e-mail is part of the list or not. Once that check is completed then more *eth.call*s are made to the *Voting* Contract to check if the user has exceeded his/her voting limit and if the ballot voting period has

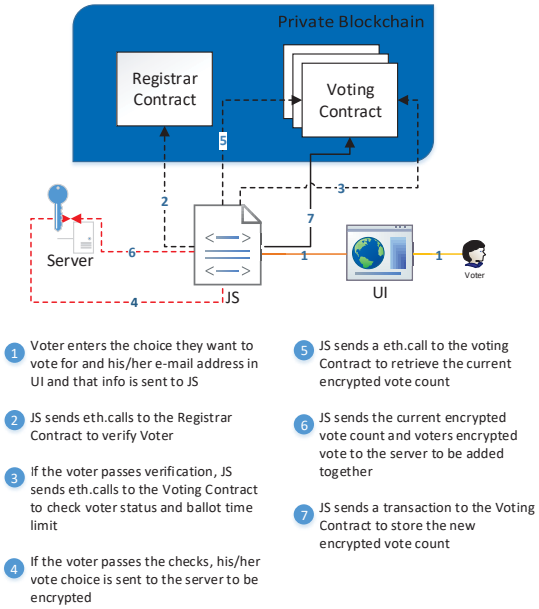

Fig. 5. The process for voting on a ballot as a voter in GenVote, where black dotted lines represent eth.calls, black solid lines represent transactions to the blockchain, and red dotted lines represent encryption calls to the server [8].

passed. The voting period is checked by comparing the end time set by the creator with the current block timestamp. If those checks were passed then the vote for the chosen choice is set to one and the rest of the votes are set to zero for the other choices on the client side. Then those votes are sent to *Crypto.js* server, as we can see in step four of Figure 5, to be encrypted using the previously generated public key in *Crypto.js* server. Once all the votes have been encrypted, the previously encrypted vote count for every choice is retrieved using an *eth.call*. Then the current encrypted votes and previously retrieved encrypted vote count are sent to the *Crypto.js* server to be homomorphically added together. Then the newly encrypted vote count for every choice is sent as an array in a transaction to the *Voting* Contract. Once the transaction has been verified, the *Voting* Contract has the updated encrypted vote count for each choice. Through this process we preserve voter privacy because we hide what their voting choice was by sending a encrypted vote to every single choice.

In the case of First to X wins ballot type, there are additional steps that need to be taken to verify and vote on the ballot. In between steps three and four of Figure 5, *VotingApp.js* checks the status of the ballot by sending an *eth.call* to the *Voting* Contract. If the status returned as not closed then it checks to see if any of the choices have met the X to win condition. The check is done by getting every individual encrypted vote with an *eth.call* and decrypting them using the *Crypto.js* server. If none of the votes met the X to win condition then we proceed to step four in Figure 5. But if the condition

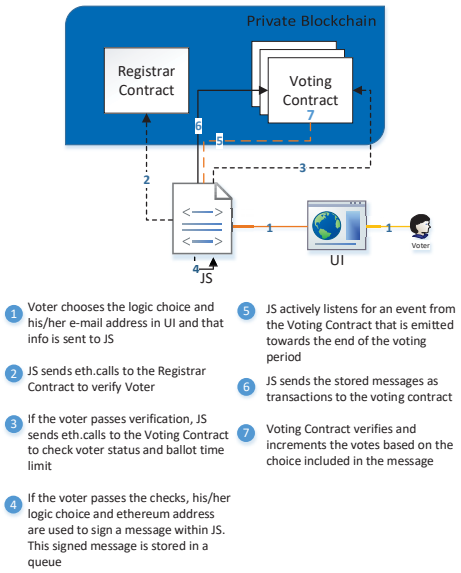

Fig. 6. The process for voting logically at a later time on a ballot as a voter in GenVote, where black dotted lines represent eth.calls, black solid lines represent transactions to the blockchain, and orange dotted lines represent listening events.

was met then we let the user know the ballot voting period has ended and we set the status of the ballot as closed in the *Voting* Contract.

3.7 Vote (Logical)

Logical voting allows the user to vote on a ballot using a calculated method. The user needs to first load the ballot. Once the ballot has been loaded, the user can choose between three options: Winning Vote, Runner-Up Vote, or Losing Vote and the option to case the vote now or at a later time. If the user chooses one of the choices and the vote now option, the process is simple. In this scenario there is only one extra step that is needed between step three and four in Figure 5. That step would involve the *VotingApp.js* calculating what the current standing for all the voters are internally to choose the appropriate choice to vote for. After the choice has been calculated then it continues with the traditional process of voting. At the end the user gets a vote verified notification but not the choice that was chosen on their behalf. This prevents voters from gaining knowledge about the current standings in an election prematurely.

If the user chooses one of the choices and the vote later option, the process gets complex and its partially detailed in Figure 6. In this process once the user has chosen the choice they want to use for voting, that choice is signed using *ethsignedTypedData*.

ethsignedTypedData is a function that currently only available through *MetaMask* and it allows us to use the user's Ethereum address to sign a message. Once the message containing the users choice has been signed, it is then stored in a queue where it will reside until an event is emitted from the *Voting* Contract. In Solidity events are a mechanism to log that something has happened. When a function with an emit call gets invoked within the smart contract, then an event is triggered and that is logged. We can take advantage of this logging functionality by setting a function that listens for it on our front end, specifically using JavaScript. Since we need to wait until the close to end of the voting period, we setup a function that listens for the event from the *Voting* Contract. When that event is triggered, transactions are sent to the ballot using the Administrator Ethereum address with the user signed messages from the queue. The *Voting* Contract will verify the signed messages and increment the appropriate choice votes using separate functions. After the validation is complete, the ballot closes and when someone loads ballot the votes are retrieved to be decrypted to be displayed. Due to the experimental nature of the external libraries required to complete the Logical Vote Later process, we only provide the complete theoretical implementation. There are future plans to bring this functionality to Web3 so when that rolls out we will be able to complete the implementation.

3.8 Get Votes

getVotes acts as a data retrieval function. Whenever a user loads the ballot or successfully votes on a ballot, *getVotes* is invoked in *VotingApp.js*. *getVotes* sends an *eth.call* with the hashed choices to get the current total encrypted votes. Depending on the time-limit and election type, it would either decrypt the votes and display them or display the time when users can check back for the results. To decrypt the votes, *getVotes* sends the encrypted vote count to the *Crypto.js* server to be decrypted by the private key.

4 TESTNET EXPERIMENT ANALYSIS

In order to collect data and test the viability of our system, we deployed it onto the Ropsten Ethereum testnet and collected the gas cost for every use case. We chose to deploy it onto the testnet to simulate a mature blockchain and test the functionality on a blockchain that has enough validating nodes participating. Our primary focus for data collected was gathering gas costs for each process since it is closely related what the performance cost would be. By gathering the performance costs we can provide better estimates for resources that would be needed when the system is deployed onto a private blockchain. We conducted experiments on varying styles of ballots and specified the gas and time costs for every user, including Administrator (A), Creator (C), and Voter (V), involved in our system.

Contract (User):	Gas Cost:
Registrar (A)	755634
Creator (A)	2235815
Voting (A/C)	1944081*

Note: * = Base cost of creating a Voting contract

Fig. 7. Initial deployment gas costs for the Administrator to activate the system.

The Administrator deploys the Registrar, Creator, and a base Voting contract to activate the system on the blockchain. The deployment costs for this initialization step are shown in Table 7. The gas cost for deploying the Registrar contract can vary depending on the set of whitelisted domains the Administrator chose to include during initialization. We chose to whitelist three domains which contain an average of nine ASCII characters in length.

	Register (V)	Create Ballot (C)	Load Ballot (C/V)	Vote (V)	Logic Vote (V)
Ballot Types (Poll):			Gas Cost:		
1a. 4 Voting Options and 0 Whitelisted Voters	106517	2093911	0	249056	248992
1b. 6 Voting Options and 0 Whitelisted Voters	106453	2310099	0	387966	390206
1c. 8 Voting Options and 0 Whitelisted Voters	106389	2496288	0	556564	556436
1d. 10 Voting Options and 0 Whitelisted Voters	106641	2697477	0	748131	748195
2a. 6 Voting Options and 4 Whitelisted Voters	106387	2477293	0	396180	396308
2b. 6 Voting Options and 6 Whitelisted Voters	106541	2542039	0	396451	396244
2c. 6 Voting Options and 8 Whitelisted Voters	106874	2621785	0	396244	396308
2d. 6 Voting Options and 10 Whitelisted Voters	106215	2671467	0	396308	396308
3a. 4 Voting Options and 10 Whitelisted Voters	106389	2470343	0	255094	255030
3b. 6 Voting Options and 10 Whitelisted Voters	106287	2671531	0	396244	396180
3c. 8 Voting Options and 10 Whitelisted Voters	106453	2872720	0	562730	562666
3d. 10 Voting Options and 10 Whitelisted Voters	106515	3073909	0	754233	754425

Note: (A) = Administrator, (C) = Creator, (V) = Voter

Fig. 8. Gas Costs for different types of ballots and use cases in our system.

After the system has been deployed onto the testnet, our next step in experimentation was creating, loading, and voting on different types of ballots with varying sizes of voting options and whitelisted voters. We chose to use a poll style of ballot for all the tests but we did check other types and learned that there was no significant cost difference. The gas costs for those tests can be viewed in Table 8. The number of ASCII characters in the data being passed into the contracts has a noticeable effect on the gas cost so we used an average ASCII character length of 10 for ID numbers, 10 for voting choices, 25 for ballot titles and 18 for e-mail addresses. If we analyze the results in

Table 8, we notice that it costs nothing to load ballot information because it sends no transactions to the blockchain. We also notice that the gas costs for a user to register into the system also stay fairly constant because it always a set of amount of information going to the blockchain. In our tests we chose to register all user with ballot creation permission but if the user chose to not opt for that then the gas cost would be lowered by roughly 10,000. In the test data we can also notice that the number of whitelisted voters doesn't affect the gas cost as significantly as the number of voting choices. This is because we set the whitelisted voters once whereas we need to manipulate the choices everytime to update vote count. We calculated that the average change in gas cost per increment of voting choice when creating a ballot is 100,000 whereas for voting it grows exponentially as voting choices increase. Finally, the difference between traditional voting and the logical voting we implemented in our extension has no significant cost difference as we can see in Table 8.

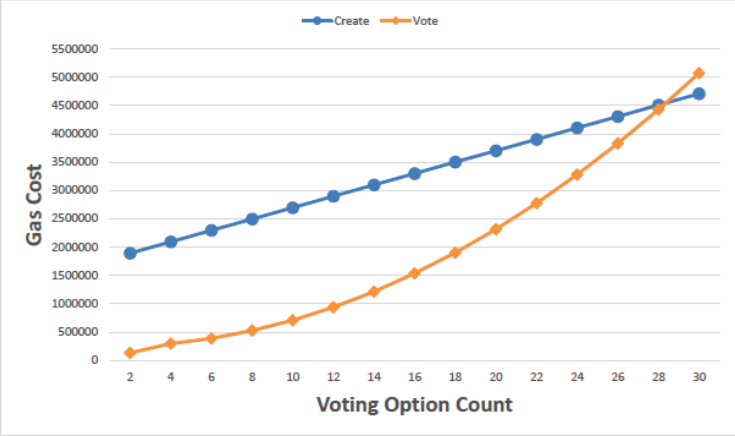

Fig. 9. Relationship between voting options and gas cost for creating and voting on ballots.

To better demonstrate the relationship of increasing gas cost as voting options increase, we create a graph (Figure 9). The data used for the graph was derived from the use case data in Table 8. We calculated the average gas cost difference between creating and voting on non-whitelisted ballots. As we can see the increase in cost is linear when creating ballots with increasing options but voting on them starts to show signs of exponential growth in cost. Currently the Ropsten Ethereum testnet has a block gas limit of 4,700,000 gas so we were able to achieve a ballot with max ballot options of 28 without any whitelisted voters. If this system was deployed on a private blockchain with modified block gas limit then we could have larger ballots.

In our previous work we conducted a time cost analysis by calculating how long, in seconds, each use case would take. In this extension we chose to exclude that due to hugely varying times it can take to validate transactions for a single repeatable process. This happens due to the validating method, when the transactions are placed in a pool

the validating node doesn't necessarily always include your transaction in the validation process right away. But even without actual time data we can say that in general time to complete will vary significantly on each of the use cases. Use cases that require sending transactions compared to the ones that only use *eth.calls* will take a longer time since they need to be validated. So the more transactions a use case utilizes the longer it would take to complete that specific process. For example, the Load Ballot would take the least amount time due to it only using *eth.calls*, which bypasses the mining requirement, to retrieve ballot information. But in the case of Create Ballot or Vote, they would take the longest on average since they require sending a few transactions to the smart contracts in order to complete their process.

5 TECHNICAL DIFFICULTIES

While implementing GenVote, we encountered a few technical difficulties. One such difficulty is support for cryptography: the maximum data value in Solidity is unsigned int of 256 bit. Many of the cryptosystems require large int numbers that the ones currently supported in the Solidity language. Therefore, GenVote cryptography is facilitated through a server, which can introduce new vulnerabilities. However, for the purpose of this paper, we assume the server is secure and cannot be compromised. Currently signed messages via users functionality is only limited to MetaMask but there are plans to expand it into the standard protocol [9] . Until then we are hindered in the ability to complete the Logical Vote at a later time feature. It can also be difficult to debug while coding in Solidity because, currently, it lacks proper debugging tools. To overcome that difficulty we chose to debug smart contracts using Remix, an integrated development environment for Solidity [10]. To debug a transaction, Remix uses either the transaction's hash or the transaction's block number and index. From there, Remix provides details regarding the transaction's execution, including local and state variables, storage changes, and return values [10] . Remix allows users to step through the contract execution so we can check the state changes and the resulting global state in the system.

6 RELATED WORKS

Fair elections are heavily dependent on the privacy and correctness of the election process. Works by McCorry et. al [10], Zyskind et al. [17], Barnes et al. [4], Ernest [9], and Varshneya [3] explore different methods to utilize the blockchain for the purpose of data integrity. To protect the privacy of user data and to authenticate voters before the results are determined, [10] utilizes zero knowledge proofs. Whereas, [4] and [9] encrypt their voter data using symmetric encryption methods. In addition, [4] also stores segmented data on the blockchain. Follow My Vote and BitCongress are two seperate voting system analyzed by [3]. Follow My Vote is a voting protocol that is hosted online and encrypts voter data with symmetric encryption protcols. Voters in the Follow My Vote system are identified with unique addresses so their real identities are never revealed.

[9] https://github.com/ethereum/EIPs/pull/712

[10] https://media.readthedocs.org/pdf/remix/latest/remix.pdf

But the system allows for third parties, like government officials, with permission to access the real identity of the voters. The second voting protocol, BitCongress, maintains data integrity with the use of two consensus methods, proof of work and proof of tally. In BitCongress, a voter is authenticated using the digital signatures of the votes cast by them. When a voter casts a vote for a candidate in BitCongress, the action is public but other participants cannot trace the vote to any voter in particular. This is achieved by creating a new key pair for voters when participating in a new election. GenVote applies partial homomorphic encryption to secure the privacy of voters and their votes on the blockchain [8].

[17] introduces a peer-to-peer network called Enigma. Enigma based implementations consist of three components: a public distributed ledger, a hash-table that refers to encrypted data off-chain, and a secure multi-party computation that is distributed among random participating nodes. Enigma is mainly used to connect to the blockchain for performing computationally sensitive data and store these records off-chain. Data integrity and privacy is achieved in the Enigma network by using the secure multi-party computation component. Secure multi-party computation is used to perform data queries without having to reveal the contents to the participating nodes. When a multi-party computation is needed, data is distributed to a set of random nodes and the nodes process their part of the data without revealing to each other which part they have. Information leakage can only occur if the majority of the selected participating nodes collude [17]. The private blockchain employed by GenVote establishes a closed voting system to protect voters from outside privacy breaches. For internal privacy, homomorphic encryption mentioned above is used within the system [8].

With the use of smart contracts, voting processes can be automated. [10] and [17] utilize the smart contract components to enhance their voting systems. In [10] two smart contracts are implemented: a Voting Contract and a a Cryptography Contract. The Voting Contract is used to process the vote for different elections and the Cryptography Contract allows for the zero knowledge proof process used in the system. Since every participating node has a copy of the Smart Contract, they can reach an agreement on the contract output instead of having to rely on someone else. Similar to smart contracts, private contracts in [17] are applied to enhance the system's scalability. These contracts are designed to process the system's private information. Three smart contracts are utilized in GenVote: a Creation Contract, Voting Contract, and Registration Contract. The Creation Contract establishes the poll or election; once this contract is deployed, it can be used to create multiple, different ballots. The Registration Contract lists the eligible voters; and the Voting Contract allows eligible voters to vote for a candidate [8].

Providing a user interface for the voting process increases the ease of access for voters. Ease of access can help with mass adoption of the voting systems as well. [10] created three potential HTML5/JavaScript pages that the voters use to access the voting system through a web browser. BitCongress [3] utilizes an application called Axiomity as the graphical user interface through which users create elections and vote. Axiomity also keeps a voting record history for users to review on demand externally. Similarly, voters in the GenVote system cast their ballots through an HTML website [8] and Javascript is used to process the votes.

The voting processes in [10], [13], [4], [9], and [3] are described below. The voting process is split into give stages in [10]. The first stage involves the election administrator creating a list of voters allowed and creates the election. The administrator also sets the election timers, deposit for registration, and toggle for optional commit stage. The second stage is when the voters register for the appropriate election. The third stage is the optional commit stage, the voters has to store a hash of their vote onto the blockchain before proceeding to the fourth stage. The fourth stage is where the voters publish their vote and a zero proof of knowledge onto the blockchain. Lastly, the final, fifth, stage computes the result of the election and reveals the outcome. It is important to note that in this system, voters can only vote for two options, typically "yes" or "no" [10].

BitCongress [13] follows a similar voting process in their system. In BitCongress, every "yes" or "no" is a token and candidate has an address. When the election is in progress, the voters cast their votes by sending their appropriate token. The tokens are then tallied and returned to voters at the end of the election process. In [4] the voting process is implemented in a hybrid way, it allows for online and offline voting. This is achieved by using two separate blockchains: one to store registered voters and one to store the actual votes. By using two separate blockchains, [4] ensures voter privacy and anonymity. Regardless of how a voter registers, the same information is required that uniquely identifies the voter. When the voter wants to vote online, they registration attempt is stored on the blockchain for government entities to mine for verification. Once verified, the voter is sent a ballot card and password to submit a vote, which is stored on the blockchain as a transaction. Some voting systems allow the voters to update their vote while it is active. This feature is implemented in both [9] and Follow My Vote discussed in [3]. Additionally, in Follow My Vote, voters can vote for multiple candidates. An election in GenVote is established when an administrator in the system deploys the Creation Contract in order to set up the ballot; this include defining the candidates of the election and the election timer. Next, the administrator defines within the Registrar Contract who is eligible to register. Lastly, the voters cast their ballots through the Voting Contract, which encrypts each ballot to provide security and privacy to the voters. Unlike the systems in [10] and [13], GenVote allows users to vote for multiple candidates with different styles of voting.

GenVote is currently a university-scaled voting system that is deployable on the Ethereum Blockchain. Voter privacy is handled through homomorphic encryption and the integrity of votes is ensured with the distributed ledger. To guarantee voter accessibility, voters cast their votes on an HTML website that can be accessed anywhere with Internet access. GenVote also has the ability to be used to conduct polls: similar to elections, polls allow individuals to voice opinions on matters. However, individuals are able to view poll statistics in real-time. Voters can also opt to let the system vote on their behalf for whoever is most (or least) favorable through logical voting. GenVote is a secure, economical voting system that is customizable and has the potential to be expanded from a university scale to a larger scale.

7 CONCLUSIONS AND FURTHER WORK

In this paper, we have presented a proof of concept system for GenVote. We also deployed it on the test network for Ethereum blockchain to gather data for the purpose of showcasing the ease of deployment and the viability of the voting system. GenVote is setup to be used in a private blockchain within a university setting and utilizes the smart contracts in Ethereum blockchain to achieve voter privacy and ballot integrity. The smart contracts implemented in GenVote are multi-functional, they act as the record keeper for all the voters and ballots, perform access control duties to prevent voter fraud, and self tally the votes for each ballot. With the use of Pallier Homomorphic cryptosystem, blockchain, and smart contracts we were able to propose a system that alleviated some of the problems that were inherited from the current electronic voting systems.

In future work, we will investigate the possibility of allowing for logical voting at a future time using event triggers and raw transaction signing. We will also further explore the possibility of implementing a partial Paillier cryptosystem as a library contract in Solidity. With the implementation of that library smart contract, it will help us generate individual key pairs for each ballot so that we can make the ballot verification process more modular. This will help us achieve individual ballot audit without the risk of compromising the other ballots in the system.

References

1. Blockchain technology in online voting. Web (2016), https://followmyvote.com/online-voting-technology/blockchain-technology/
2. Adida, B.: Helios: Web-based open-audit voting. In: USENIX security symposium. vol. 17, pp. 335–348 (2008)
3. A.J. Varshneya, Sugat Poudel, X.V.: Blockchain voting (2015)
4. Andrew Barnes, C.B., Perry, T.: Digital voting with the use of blockchain technology (2016)
5. Atzori, M.: Blockchain technology and decentralized governance: Is the state still necessary? (2015)
6. Buterin, V.: On public and private blockchains. Ethereum Blog **7** (2015)
7. Buterin, V., et al.: Ethereum white paper (2013)
8. Dagher, G.G., Marella, P.B., Milojkovic, M., Mohler, J.: Broncovote: Secure voting system using ethereums blockchain. In: Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP). pp. 96–107 (2018)
9. Ernest, A.K.: The key to unlocking the black box: Why the world needs a transparent voting dac (2014)
10. McCorry, P., Shahandashti, S.F., Hao, F.: A smart contract for boardroom voting with maximum voter privacy. IACR Cryptology ePrint Archive **2017**, 110 (2017)
11. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
12. OKeeffe, M.: The paillier cryptosystem: a look into the cryptosystem and its potential application. College of New Jersey (2008)
13. Rockwell, M.: Bitcongress whitepaper (2015)
14. Tarasov, P., Tewari, H.: Internet voting using zcash
15. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Exploring the feasibility of fully homomorphic encryption. IEEE Transactions on Computers **64**(3), 698–706 (2015)
16. Xun Yi, Russell Paulet, E.B.: Homomorphic Encryption and Applications. Springer International Publishing (2014)

17. Zyskind, G., Nathan, O., Pentland, A.: Enigma: Decentralized computation platform with guaranteed privacy. arXiv preprint arXiv:1506.03471 (2015)