D-DEMOS: A Distributed, End-to-End Verifiable, Internet Voting System

D-DEMOS: A Distributed, End-to-end Verifiable, Internet Voting system

Nikos Chondros*, Bingsheng Zhang†, Thomas Zacharias*, Panos Diamantopoulos*, Stathis Maneas‡,
Christos Patsonakis*, Alex Delis*, Aggelos Kiayias* and Mema Roussopoulos*
* Department of Informatics and Telecommunications, University of Athens, Greece
† Schools of Computing and Communications, University of Lancaster, UK
‡ Department of Computer Science, University of Toronto, Canada

***Abstract*—E-voting systems have emerged as a powerful technology for improving democracy by reducing election cost, increasing voter participation, and even allowing voters to directly verify the entire election procedure. Prior internet voting systems have single points of failure, which may result in the compromise of availability, voter secrecy, or integrity of the election results.**

In this paper, we present the design, implementation, security analysis, and evaluation of D-DEMOS, a complete e-voting system that is distributed, privacy-preserving and end-to-end verifiable. Our system includes a fully asynchronous vote collection subsystem that provides immediate assurance to the voter her vote was recorded as cast, without requiring cryptographic operations on behalf of the voter. We also include a distributed, replicated and fault-tolerant Bulletin Board component, that stores all necessary election-related information, and allows any party to read and verify the complete election process. Finally, we also incorporate trustees, i.e., individuals who control election result production while guaranteeing privacy and end-to-end-verifiability as long as their strong majority is honest.

Our system is the first e-voting system whose voting operation is human verifiable, i.e., a voter can vote over the web, even when her web client stack is potentially unsafe, without sacrificing her privacy, and still be assured her vote was recorded as cast. Additionally, a voter can outsource election auditing to third parties, still without sacrificing privacy. Finally, as the number of auditors increases, the probability of election fraud going undetected is diminished exponentially.

We provide a model and security analysis of the system. We implement a prototype of the complete system, we measure its performance experimentally, and we demonstrate its ability to handle large-scale elections.

I. Introduction

E-voting systems have emerged as a powerful technology to improve the election process. Kiosk-based e-voting systems, e.g., [12], [15], [26], [13], [10], [23], allow the tally to be produced faster, but require the voter's physical presence at the booth. Internet e-voting systems, e.g., [21], [8], [17], [31], [27], [35], [12], [13], [35], [29], however, allow voters to cast their votes remotely. Internet voting systems have the potential to enhance the democratic process by reducing election costs and by increasing voter participation for social groups that face considerable physical barriers and overseas voters. In addition, several internet voting systems [8], [31], [35], [29] allow voters and auditors to directly verify the integrity of the entire election process, providing *end-to-end verifiability*. This is a highly desired property that has emerged in the last decade, where voters can be assured that no entities, even the election authorities, have manipulated the election result. Despite their potential, existing internet voting systems suffer from single points of failure, which may result in the compromise of voter secrecy, service availability, or integrity of the result [12], [15], [26], [13], [10], [21], [8], [17], [31], [27], [35], [29].

In this paper, we present the design and prototype implementation of D-DEMOS, a distributed, end-to-end verifiable internet voting system, with no single point of failure during the election process (that is, besides setup). We set out to overcome two major limitations in existing internet voting systems. The first, is their dependency on centralized components. The second is their requirement for the voter to run special software on their devices, which processes cryptographic operations. Overcoming the latter allows votes to be cast with a greater variety of client devices, such as feature phones using SMS, or untrusted public web terminals. Our design is inspired by the novel approach proposed in [29], where the voters are used as a source of randomness to challenge the zero-knowledge proof protocols; the latter are used to enable end-to-end verifiability.

We design a distributed vote collection subsystem that is able to collect votes from voters and assure them their vote was recorded as cast, without requiring any cryptographic operation from the client device. This allows voters to vote via SMS, a simple console client over a telnet session, or a public web terminal, while preserving their privacy. At election end time, vote collectors agree on a single set of votes asynchronously, and upload it to a second distributed component, the Bulletin Board (BB). This is a replicated service that publishes its data immediately and makes it available to the public forever. Our third distributed subsystem, *trustees* are a set of persons entrusted with secret keys that can unlock information from the BB. We share these secret keys among the trustees, making sure only an honest majority can uncover information from the BB. Trustees interact with the BB once the votes are uploaded to it, to produce and publish the final election tally.

The resulting voting system is end-to-end verifiable, by the voters themselves, as well as third-party auditors; all this while preserving voter privacy. A voter can provide an auditor information from her ballot; the auditor can read from the distributed BB and verify the complete process, including the correctness of the election setup by election authorities. Additionally, as the number of auditors increases, the probability of election fraud going undetected diminishes

exponentially.

Finally, we implement a prototype of the complete D-DEMOS voting system. We measure its performance experimentally, under a variety of election settings, demonstrating its ability to handle thousands of concurrent connections, and thus manage large-scale elections.

To summarize, we make the following contributions:

- We present the world's first complete, state-of-the-art, end-to-end verifiable, distributed voting system with no single point of failure besides setup.
- The system allows voters to verify their vote was tallied-as-intended without the assistance of special software or trusted devices, and external auditors to verify the correctness of the election process. Additionally, the system allows voters to delegate auditing to a third party auditor, without sacrificing their privacy.
- We provide a model and a security analysis of our voting system.
- We implement a prototype of the integrated system, measure its performance and demonstrate its ability to handle large-scale elections.

II. Related work

Voting systems: Several end-to-end verifiable e-voting systems have been introduced, e.g. the kiosk-based systems [15], [26], [13], [10] and the internet voting systems [8], [31], [35], [29]. In all these works, the Bulletin Board (BB) is a single point of failure and has to be trusted.

Dini presents a distributed e-voting system, which however is not end-to-end verifiable [24]. In [23], there is a distributed BB implementation, also handling vote collection, according to the design of the vVote end-to-end verifiable e-voting system [22], which in turn is an adaptation of the Prêt à Voter e-voting system [15]. In [23], the proper operation of the BB during ballot casting requires a trusted device for signature verification. In contrast, our vote collection subsystem is done so that correct execution of ballot casting can be "human verifiable", i.e., by simply checking the validity of the obtained receipt. Additionally, our vote collection subsystem is fully asynchronous, always deciding with exactly $n-f$ inputs, while in [23], the system uses a synchronous approach based on the FloodSet algorithm from [32] to agree on a single version of the state.

DEMOS [29] is an end-to-end verifiable e-voting system, which introduces the novel idea of extracting the challenge of the zero-knowledge proof protocols from the voters' random choices; we leverage this idea in our system too. However, DEMOS uses a centralized Election Authority (EA), which maintains all secrets throughout the entire election procedure, collects votes, produces the result and commits to verification data in the BB. Hence, the EA is a single point of failure, and because it knows the voters' votes, it is also a critical privacy vulnerability. In this work, we address these issues by introducing distributed components for vote collection and result tabulation, and we do not assume any trusted component during election. Additionally, DEMOS does not provide any recorded-as-cast feedback to the voter, whereas our system includes such a mechanism. Moreover, in our design, the committed verification data in the BB support auditing with asymptotically lower computational cost w.r.t. the number of options, compared to DEMOS. Finally, the zero-knowledge proofs in DEMOS have a large soundness error which decreases the effectiveness of zero-knowledge application, while in this work we obtain nearly optimal overall zero-knowledge soundness.

Furthermore, none of the above works provide any performance evaluation results.

State Machine Replication: Castro et al. [11] introduce a practical Byzantine Fault Tolerant replicated state machine protocol. In the last several years, several protocols for Byzantine Fault Tolerant state machine replication have been introduced to improve performance ([20], [30]), robustness ([9], [19]), or both ([18]). Our system does not use the state machine replication approach to handle vote collection, as it would be inevitably more costly. Each of our vote collection nodes can validate a voter's requests on its own. In addition, we are able to process multiple different voters' requests concurrently, without enforcing the total ordering inherent in replicated state machines. Finally, we do not wish voters to use special client-side software to access our system.

III. System description

A. Problem Definition and Goals

We consider an *election* with a single *question* and m *options*, for a voter population of size n, where voting takes place between a certain *begin* and *end* time (the *voting hours*), and each voter may select a single *option*.

Our major goals in designing our voting system are three. 1) It has to be end-to-end verifiable, so that anyone can verify the complete election process. Additionally, voters should be able to outsource auditing to third parties, without revealing their voting choice. 2) It has to be fault-tolerant, so that an attack on system availability and correctness is hard. 3) Voters should not have to trust the terminals they use to vote, as such devices may be malicious. Instead, voters should be assured their vote was recorded, without disclosing any information on how they voted to the malicious entity controlling their device.

B. System overview

We employ an election setup component in our system, which we call the Election Authority (EA), to alleviate the voter from employing any cryptographic operations. The EA initializes all other system components, and then gets immediately destroyed to preserve privacy. The *Vote Collection* (VC) subsystem collects the votes from the voters during election hours, and assures them their vote was *recorded-as-cast*. Our *Bulletin Board* (BB) subsystem, which is a public repository of all election-related information, is used to hold all ballots, votes, and the result, either in encrypted or plain form, allowing any party to read from the BB and verify the complete election process. The VC subsystem uploads all votes to the BB at election end time. Finally, our design includes *trustees*, who are

persons entrusted with managing all actions needed until result tabulation and publication, including all actions supporting end-to-end verifiability. Trustees hold the keys to uncover any information hidden in the *BB*, and we use threshold cryptography to make sure a malicious minority cannot uncover any secrets or corrupt the process.

Our system starts with the *EA* generating initialization data for every component. The *EA* encodes each election option, and *commits* to it using a commitment scheme, as described below. It encodes the i-th option as $\vec{e}_i$, a unit vector where the i-th element is 1 and the remaining elements are 0. The commitment of an option encoding is a vector of (lifted) ElGamal ciphertexts [25] over elliptic curve, that element-wise encrypts a unit vector. Note that this commitment scheme is also additively homomorphic, i.e., the commitment of $e_a + e_b$ can be computed by component-wise multiplying the corresponding commitments of e_a and e_b. The *EA* then creates a votecode and a receipt for each option. Subsequently, the *EA* prepares one ballot for each voter, with two functionally equivalent parts. Each part contains a list of options, along with their corresponding vote codes and receipts. We consider ballot distribution to be outside the scope of this paper, but we do assume ballots, after being produced by the *EA*, are distributed in a secure manner to each voter; thus only each voter knows the vote codes listed in her ballot. We make sure vote codes are not stored in clear form anywhere besides the voter's ballot.

Our *VC* subsystem collects the votes from the voters during election hours, by accepting up to one vote code from each voter. The *EA* initializes each *VC* node with the vote codes and the receipts of the voters' ballots. However, it hides the vote codes, using a simple commitment scheme based on symmetric encryption of the plaintext along with a random salt value. This way, each *VC* node can verify if a vote code is indeed part of a specific ballot, but cannot recover any vote code until the voter actually chooses to disclose it. Additionally, we secret-share each receipt across all *VC*-nodes using an $(N - f, N)$-VSS (verifiable secret-sharing) scheme with trusted dealer [34], making sure that a receipt can be recovered and posted back to the voter only when a strong majority of *VC* nodes participates successfully in our voting protocol. With this design, our system adheres to the following contract with the voters: *Any honest voter who receives a valid receipt from a Vote Collector node, is assured her vote will be published on the BB, and thus it will be included in the election tally*.

The voter selects one part of her ballot at random, and posts her selected vote code to one of the *VC* nodes. When she receives a receipt, she compares it with the one on her ballot corresponding to the selected vote code. If it matches, she is assured her vote was correctly recorded and will be included in the election tally. The other part of her ballot, the one not used for voting, will be used for auditing purposes. This design is essential for verifiability, in the sense that the *EA* cannot predict which part a voter may use, and the unused part will betray a malicious *EA* with $\frac{1}{2}$ probability per audited ballot.

Our second distributed subsystem is the *BB*, which is a replicated service of isolated nodes. Each *BB* node is initialized from the *EA* with vote codes and associated option encodings in committed form (again, for vote code secrecy), and each *BB* node provides public access to its stored information. At election end time, *VC* nodes run our Vote Set Consensus protocol, which guarantees all *VC* nodes agree on a single set of voted vote codes. After agreement, each *VC* node uploads this set to every *BB* node. In turn, each *BB* node publishes this set once it receives the same copy from enough *VC* nodes.

Our third distributed subsystem is a set of trustees, who are persons entrusted with managing all actions needed after vote collection, until result tabulation and publication; this includes all actions supporting end-to-end verifiability. Secrets that may uncover information in the *BB* are shared across trustees, making sure malicious trustees under a certain threshold cannot uncover and disclose sensitive information. We use Pedersen's Verifiable linear Secret Sharing (VSS) [33] to split the election data among the trustees. In a (k, n)-VSS, at least k shares are required to reconstruct the original data, and any collection of less than k shares leaks no information about the original data. Moreover, Pedersen's VSS is additively homomorphic, i.e., one can compute the share of $a + b$ by adding the share of a and the share of b respectively. This approach allows trustees to perform homomorphic "addition" on the option-encodings of cast vote codes, and contribute back a share of the opening of the homomorphic "total". Once enough trustees upload their shares of the "total", the election tally is uncovered and published at each *BB* node.

To ensure voter privacy, the system cannot reveal the content inside an option encoding commitment at any point. However, a malicious *EA* might put an arbitrary value (say 9000 votes for option 1) inside such a commitment, causing an incorrect tally result. To prevent this, we utilize the Chaum-Pedersen zero-knowledge proof [14], allowing the *EA* to show that the content inside each commitment is a valid option encoding, without revealing its actual content. Namely, the prover uses Sigma OR proof to show that each ElGamal ciphertext encrypts either 0 or 1, and the sum of all elements in a vector is 1. Our zero knowledge proof is organized as follows. First, the *EA* posts the initial part of the proofs on the *BB*. Second, during the election, each voter's A/B part choice is viewed as a source of randomness, $0/1$, and all the voters' choices are collected and used as the challenge of our zero knowledge proof. Finally, the trustees will jointly produce the final part of the proofs and post it on the *BB* before the opening of the tally. Hence, everyone can verify those proofs on the *BB*. Due to space, we omit the zero-knowledge proof components in this paper and refer the interested reader to [14].

Our design allows any voter to read information from the *BB*, combine it with her private ballot, and verify her ballot was included in the tally. Additionally, any third-party auditor can read the *BB* and verify the complete election process. As the number of auditors increases, the probability of election fraud going undetected diminishes exponentially. For example, even if only 10 people audit, with each one having $\frac{1}{2}$ probability of detecting ballot fraud, the probability of ballot fraud going undetected is only $\frac{1}{2}^{10} = 0.00097$. Thus, even if the *EA* is

malicious and, e.g., tries to point all vote codes to a specific option, this faulty setup will be detected because of the end-to-end verifiability of the complete system.

C. System and Threat Model

We assume a fully connected network, where each node can reach any other node with which it needs to communicate. The network can drop, delay, duplicate, or deliver messages out of order. However, we assume messages are eventually delivered, provided the sender keeps retransmitting them. For all nodes, we make no assumptions regarding processor speeds. We assume the clocks of *VC* nodes are synchronized with real world time; this is needed simply to prohibit voters from casting votes outside election hours. Besides this, we make no other timing assumptions in our system. We assume the *EA* sets up the election and is destroyed upon completion of the setup, as it does not directly interact with the remaining components of the system, thus reducing the attack surface of the privacy of the voting system as a whole. We also assume initialization data for every system component is relayed to it via untappable channels. We assume the adversary does not have the computational power to violate any underlying cryptographic assumptions. To ensure liveness, we additionally assume the adversary cannot delay communication between honest nodes above a certain threshold. We place no bound on the number of faulty nodes the adversary can coordinate, as long as the number of malicious nodes of each subsystem is below its corresponding fault threshold. We consider arbitrary (Byzantine) failures, because we expect our system to be deployed across separate administrative domains. Let N_v, N_b, and N_t be the number of *VC* nodes, *BB* nodes, and trustees respectively. For each of the subsystems, we have the following fault tolerance thresholds:

- The number of faulty *VC* nodes, f_v, is strictly less than $1/3$ of N_v.
- The number of faulty *BB* nodes, f_b, is strictly less than $1/2$ of N_b.
- For the trustees' subsystem, we apply h_t out-of N_t verifiable secret sharing, where h_t is the number of honest trustees, thus we tolerate $f_t = N_t - h_t$ malicious trustees.

D. Election Authority

EA produces the initialization data for each election entity in the setup phase. To enhance the system robustness, we let the *EA* generate all the public/private key pairs for all the system components (except voters) without relying on external PKI support. We use zero knowledge proofs to ensure the correctness of all the initialization data produced by the *EA*.

Voter ballots: The *EA* generates one ballot ballot_ℓ for each voter ℓ, and assigns a unique 64-bit $\mathsf{serial\text{-}no}_\ell$ to it. As shown below, each ballot consists of two parts: Part A and Part B. Each part contains a list of m $\langle \mathsf{vote\text{-}code}, \mathsf{option}, \mathsf{receipt} \rangle$ tuples, one tuple for each election option. The *EA* generates the vote-code as a 128-bit random number, unique within the ballot, and the receipt as 64-bit random number.

$\mathsf{serial\text{-}no}_\ell$		
	Part A	
$\mathsf{vote\text{-}code}_{\ell,1}$	$\mathsf{option}_{\ell,1}$	$\mathsf{receipt}_{\ell,1}$
...	...	...
$\mathsf{vote\text{-}code}_{\ell,m}$	$\mathsf{option}_{\ell,m}$	$\mathsf{receipt}_{\ell,m}$
	Part B	
$\mathsf{vote\text{-}code}_{\ell,1}$	$\mathsf{option}_{\ell,1}$	$\mathsf{receipt}_{\ell,1}$
...	...	...
$\mathsf{vote\text{-}code}_{\ell,m}$	$\mathsf{option}_{\ell,m}$	$\mathsf{receipt}_{\ell,m}$

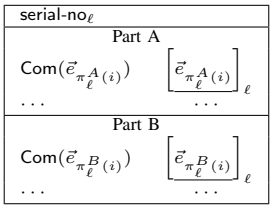
***BB* initialization data:** The initialization data for all *BB* nodes is identical, and each *BB* node publishes its initialization data immediately. The *BB*'s data is used to show the correspondence between the vote codes and their associated cryptographic payload. This payload comprises the committed option encodings, and their respective zero knowledge proofs of valid encoding (first move of the prover), as described in section III-B. However, the vote codes must be kept secret during the election, to prevent the adversary from "stealing" the voters' ballots and using the stolen vote codes to vote. To achieve this, the *EA* first randomly picks a 128-bit key, msk, and encrypts each $\mathsf{vote\text{-}code}$ using AES-128-CBC with random initialization vector (AES-128-CBC$) encryption, denoted as $[\mathsf{vote\text{-}code}]_{\mathsf{msk}}$. Each *BB* is given $H_{\mathsf{msk}} \leftarrow SHA256(\mathsf{msk}, \mathsf{salt}_{\mathsf{msk}})$ and $\mathsf{salt}_{\mathsf{msk}}$, where $\mathsf{salt}_{\mathsf{msk}}$ is a fresh 64-bit random salt. Hence, each *BB* node can be assured the key it reconstructs from *VC* key-shares (see below) is indeed the key that was used to encrypt these vote-codes.

The rest of the *BB* initialization data is as follows: for each $\mathsf{serial\text{-}no}_\ell$, and for each ballot part, there is a *shuffled* list of $\left\langle [\mathsf{vote\text{-}code}_{\ell,\pi^X_\ell(j)}]_{\mathsf{msk}}, \mathsf{payload}_{\ell,\pi^X_\ell(j)} \right\rangle$ tuples, where $\pi^X_\ell \in S_m$ is a random permutation (X is A or B). We shuffle the list of tuples of each part to ensure voter's privacy. This way, nobody can guess the voter's choice from the position of the cast vote-code in this list.

***VC* initialization data:** The *EA* uses an $(N_v - f_v, N_v)$-VSS (Verifiable Secret-Sharing) scheme to split msk and every $\mathsf{receipt}_{\ell,j}$, denoted as $(\|\mathsf{msk}\|_1, \ldots, \|\mathsf{msk}\|_{N_v})$ and $(\|\mathsf{receipt}_{\ell,j}\|_1, \ldots, \|\mathsf{receipt}_{\ell,j}\|_{N_v})$. For each $\mathsf{vote\text{-}code}_{\ell,j}$ in each ballot, the *EA* also computes $H_{\ell,j} \leftarrow SHA256(\mathsf{vote\text{-}code}_{\ell,j}, \mathsf{salt}_{\ell,j})$, where $\mathsf{salt}_{\ell,j}$ is a 64-bit random number. $H_{\ell,j}$ allows each *VC* node to validate a $\mathsf{vote\text{-}code}_{\ell,j}$ individually (without network communication), while still keeping the $\mathsf{vote\text{-}code}_{\ell,j}$ secret. To preserve voter privacy, these tuples are also shuffled using π^X_ℓ. The initialization data for VC_i is structured as below:

$\|\mathsf{msk}\|_i$	
$\mathsf{serial\text{-}no}_\ell$	
Part A	
$(H_{\ell,\pi^A_\ell(1)}, \mathsf{salt}_{\ell,\pi^A_\ell(1)})$	$\|\mathsf{receipt}_{\ell,\pi^A_\ell(1)}\|_i$
...	...
$(H_{\ell,\pi^A_\ell(m)}, \mathsf{salt}_{\ell,\pi^A_\ell(m)})$	$\|\mathsf{receipt}_{\ell,\pi^A_\ell(m)}\|_i$
Part B	
$(H_{\ell,\pi^B_\ell(1)}, \mathsf{salt}_{\ell,\pi^B_\ell(1)})$	$\|\mathsf{receipt}_{\ell,\pi^B_\ell(1)}\|_i$
...	...
$(H_{\ell,\pi^B_\ell(m)}, \mathsf{salt}_{\ell,\pi^B_\ell(m)})$	$\|\mathsf{receipt}_{\ell,\pi^B_\ell(m)}\|_i$

Trustee initialization data: The *EA* uses (h_t, N_t)-VSS to split the opening of encoded option commitments $\mathsf{Com}(\vec{e}_i)$, denoted as $(\left[\vec{e}_i\right]_1, \ldots, \left[\vec{e}_i\right]_{N_t})$. The initialization data for $\mathsf{Trustee}_i$ is structured as below:



Similarly, the state of zero knowledge proofs for ballot correctness is shared among the trustees using (h_t, N_t)-VSS. For further details, we refer the interested reader to [14].

E. Vote Collectors

VC is a distributed system of N_v nodes, running our *voting* and *vote-set consensus* protocols. *VC* nodes have private and authenticated channels to each other, and a public (unsecured) channel for voters. The *voting* protocol starts when a voter submits a VOTE⟨serial-no, vote-code⟩ message to a *VC* node. We call this node the *responder*, as it is responsible for delivering the receipt to the voter. The *VC* node confirms the current system time is within the defined election hours, and locates the ballot with the specified serial-no. It also verifies this ballot has not been used for this election, with either the same or a different vote code. Then, it compares the vote-code against every hashed vote code in each ballot line, until it locates the correct entry. At this point, it multicasts an ENDORSE⟨serial-no, vote-code⟩ message to all *VC* nodes. Each *VC* node, after making sure it has not endorsed another vote code for this ballot, responds with an ENDORSEMENT⟨serial-no, vote-code,sig$_{\mathsf{VC_i}}$⟩ message, where sig$_{\mathsf{VC_i}}$ is a digital signature of the specific serial-no and vote-code, with VC_i's private key. The responder collects $N_v - f_v$ valid signatures and forms a uniqueness certificate UCERT for this ballot. Subsequently, it obtains the receipt-share corresponding to the specific vote-code from its local database, and marks the ballot as pending for the specific vote-code. Finally, it multicasts a VOTE_P⟨serial-no, vote-code, receipt-share, UCERT⟩ message to all *VC* nodes, disclosing its share of the receipt. In case the located ballot is marked as voted for the specific vote-code, the *VC* node sends the stored receipt to the voter without any further interaction with other *VC* nodes.

Each *VC* node that receives a VOTE_P message, first verifies the validity of UCERT, and validates the received receipt-share according to the verifiable secret sharing scheme used. Then, it performs the same validations as the responder, and multicasts another VOTE_P message (only once), disclosing its share of the receipt. When a node collects $h_v = N_v - f_v$ valid shares, it uses the verifiable secret sharing reconstruction algorithm to reconstruct the receipt (the secret) and marks the ballot as voted for the specific vote-code. Additionally, the *responder* node sends this receipt back to the voter. The formation of a valid UCERT gives our algorithms the following guarantees:

a) No matter how many responders and vote codes are active at the same time for the same ballot, if a UCERT is formed for vote code vc_a, no other uniqueness certificate for any vote code different than vc_a can be formed.

b) By verifying the UCERT before disclosing a *VC* node's receipt share, we guarantee the voter's receipt cannot be reconstructed unless a valid UCERT is present.

At election end time, each *VC* node stops processing ENDORSE, ENDORSEMENT, VOTE and VOTE_P messages, and initiates the *vote-set consensus* protocol, by performing the following steps for each registered ballot:

1) Send ANNOUNCE⟨serial-no, vote-code, UCERT⟩ to all nodes. The vote-code will be *null* if the node knows of no vote code for this ballot.
2) Wait for $N_v - f_v$ such messages. If any of these messages contains a valid vote code vc_a, accompanied by a valid UCERT, change the local state immediately, by setting vc_a as the vote code used for this ballot.
3) Participate in a Binary Consensus protocol, with the subject "Is there a valid vote code for this ballot?". Enter with an opinion of 1, if a valid vote code is locally known, or a 0 otherwise.
4) If the result of Binary Consensus is 0, consider the ballot not voted.
5) Else, if the result of Binary Consensus is 1, consider the ballot voted. There are two sub-cases here:
   a) If vote code vc_a, accompanied by a valid UCERT is locally known, consider the ballot voted for vc_a.
   b) If, however, vc_a is not known, send a RECOVER-REQUEST⟨serial-no⟩ message to all *VC* nodes, wait for the first valid RECOVER-RESPONSE⟨serial-no, vc_a, UCERT⟩ response, and update the local state accordingly.

Steps 1-2 ensure used vote codes are dispersed across nodes. Recall our receipt generation requires $N_v - f_v$ shares to be revealed by distinct *VC* nodes, of which at least $N_v - 2f_v$ are honest. Note that any two $N_v - f_v$ subsets of N_v have at least one honest node in common. Because of this, if a receipt was generated, at least one honest node's ANNOUNCE will be processed by every honest node, and all honest *VC* nodes will obtain the corresponding vote code in these two steps. Thus, all honest nodes enter step 3 with an opinion of 1, and binary consensus is guaranteed to deliver 1 as the resulting value, thus safeguarding our contract against the voters. In any case, step 3 guarantees all *VC* nodes arrive at the same conclusion, on whether this ballot is voted or not.

In the algorithm outlined above, the result from binary consensus is translated from 0/1 to a status of "not-voted" or a unique valid vote code, in steps 4-5. The 5b case of this translation, in particular, requires additional justification. Assume, for example, that a voter submitted a valid vote code vc_a, but a receipt was not generated before election end time. In this case, an honest vote collector node VC_i may not be aware of vc_a at step 3, as steps 1-2 do not make any guarantees in this case. Thus, VC_i may rightfully enter consensus with a value of 0. However, when honest nodes' opinions are mixed, the consensus algorithm may produce any result. In case the result is 1, VC_i will not possess the correct vote code vc_a, and

thus will not be able to properly translate the result to a vote code. This is what our recovery sub-protocol is designed for. VC_i will issue a RECOVER-REQUEST multicast, and we claim that another honest node, VC_h exists that *possesses* vc_a and *replies* with it. The reason for the existence of an honest VC_h is straightforward and stems from the properties of the binary consensus problem definition. If all honest nodes enter binary consensus with the same opinion a, the result of any consensus algorithm is guaranteed to be a. Since we have an honest node VC_i, that entered consensus with a value of 0, but a result of 1 was produced, there has to exist another honest node VC_h that entered consensus with an opinion of 1. Since VC_h is honest, it must *possess* vc_a, along with the corresponding UCERT, as no other vote code vc_b can be active at the same time for this ballot. Again, because VC_h is honest, it will follow the protocol and *reply* with a well formed RECOVER-REPLY. Additionally, the existence of UCERT guarantees that any malicious replies can be safely identified and discarded.

At the end of this algorithm, each node submits the resulting set of *voted* $\langle \mathsf{serial\text{-}no}, \mathsf{vote\text{-}code} \rangle$ tuples to each *BB* node, which concludes its operation for the specific election.

F. Voter

We expect the voter, who has received a ballot from *EA*, to know the URLs of at least $f_v + 1$ *VC* nodes. To vote, she picks one part of the ballot at random, selects the vote code representing her chosen option, and loops, selecting a *VC* node at random and posting the vote code, until she receives a valid receipt. After the election, the voter can verify two things from the updated *BB*. First, she can verify her cast vote code is included in the tally set. Second, she can verify that the unused part of her ballot, as "opened" at the *BB*, matches the copy she received before the election started. This step verifies that the vote codes are associated with the expected options as printed in the ballot. Finally, the voter can delegate both of these checks to an *auditor*, without sacrificing her privacy. This is because the cast vote code does not reveal her choice, and because the unused part of the ballot is completely unrelated to the used one.

G. Bulletin Board

A *BB* node functions as public repository of election-specific information. By definition, it can be read via a public and anonymous channel. Writes, on the other hand, happen over an authenticated channel, implemented with PKI originating from the voting system. *BB* nodes are independent from each other, as a *BB* node never directly contacts another *BB* node. Readers are expected to issue a read request to all *BB* nodes, and trust the reply that comes from the majority. Writers are also expected to write to all *BB* nodes; their submissions are always verified, and explained in more detail below.

After the setup phase, each *BB* node publishes its initialization data. During election hours, *BB* nodes remain inert. After the voting phase, each *BB* node receives from each *VC* node, the final vote-code set and the shares of msk. Once it receives $f_v + 1$ identical final vote code sets, it accepts and publishes the final vote code set. Once it receives $N_v - f_v$ valid key shares (again from *VC* nodes), it reconstructs the msk, decrypts all the encrypted vote codes in its initialization data, and publishes them.

At this point, the cryptographic payloads corresponding to the cast vote codes are made available to the trustees. Trustees, in turn, read from the *BB* subsystem, perform their individual calculations and then write to the *BB*s; these writes are verified by the trustees' keys, generated by the *EA*. Once enough trustees have posted valid data, the *BB* node combines them and publishes the final election result.

We intentionally designed our *BB* nodes to be as simple as possible for the reader, refraining from using a *Replicated State Machine*, which would require readers to run algorithm-specific software. The robustness of *BB* nodes comes from controlling all write accesses to them. Writes from *VC* nodes are verified against their honest majority threshold. Further writes are allowed only from trustees, verified by their keys.

Finally, a reader of our *BB* nodes should post her read request to all nodes, and accept what the majority responds with ($f_b + 1$ is enough). We acknowledge there might be temporary state divergence (among *BB* nodes), from the time a writer updates the first *BB* node, until the same writer updates the last *BB* node. However, given our thresholds, this should be only momentary, alleviated with simple retries. Thus, if there is no reply backed by a clear majority, the reader should retry until there is one.

H. Trustees

After the end of election hours, each trustee fetches all the election data from the *BB* subsystem and verifies its validity. For each ballot, there are two possible valid outcomes: i) one of the A/B parts are voted, ii) none of the A/B parts are voted. If both A/B parts of a ballot are marked as voted, then the ballot is considered as invalid and is discarded. Similarly, trustees also discard those ballots where more than one commitments in a single part (A or B) are marked as voted. In case (i), for each encoded option commitment in the unused part, $\mathsf{Trustee}_\ell$ submits its corresponding share of the opening of the commitment to the *BB*. For each encoded option commitment in the voted part, $\mathsf{Trustee}_\ell$ computes and posts the share of the final message of the corresponding zero knowledge proof, showing the validity of those commitments. Meanwhile, those commitments marked as voted are collected to a tally set $\mathbf{E}_{\text{tally}}$. In case (ii), for each encoded option commitment in both parts, $\mathsf{Trustee}_\ell$ submits its corresponding share of the opening of the commitment to the *BB*. Finally, denote $\mathbf{D}^{(\ell)}_{\text{tally}}$ as $\mathsf{Trustee}_\ell$'s set of shares of option encoding commitment openings, corresponding to the commitments in $\mathbf{E}_{\text{tally}}$. $\mathsf{Trustee}_\ell$ computes the opening share for E_{sum} as $T_\ell = \sum_{D \in \mathbf{D}^{(\ell)}_{\text{tally}}}$ and then submits T_ℓ to each *BB* node.

I. Auditors

Auditors are participants of our system who can independently verify the election process. The role of the auditor can be assumed by voters or any other party. After election end

time, auditors read information from the *BB* and verify the correct execution of the election, by verifying the following: a) within each opened ballot, no two vote codes are the same, b) there are no two submitted vote codes associated with any single ballot part, c) within each ballot, no more than one part has been used, d) all the openings of the commitments are valid, e) all the zero-knowledge proofs that are associated with the used ballot parts are completed and valid. In case they received audit information (an unused ballot part and a cast vote code) from voters who wish to delegate verification, they can also verify: i) the submitted vote codes are consistent with the ones received from the voters, ii) the openings of the unused ballot parts are consistent with the ones received from the voters.

IV. Security of D-DEMOS

In this section, we show that our e-voting system achieves liveness and safety, as well as end-to-end verifiability and voter privacy at the same level of [29][1]. Due to space constraints, we provide high-level overviews of our proof strategies, and refer the reader to the extended version [16] for the full proofs.

We use m, n to denote the number of options and voters respectively. We denote by λ the cryptographic security parameter and we write $\mathsf{negl}(\lambda)$ to denote that a function is negligible in λ. We assume the following security guarantees for the underlying cryptographic tools:

1) The probability that an adversary running in λ steps forges digital signatures is $\mathsf{negl}(\lambda)$.

2) There exists a constant $c < 1$ s.t. the probability an adversary running in $O(2^{\lambda^c})$ steps breaks the hiding property of the option-encoding commitments is $\mathsf{negl}(\lambda)$.

Liveness. We prove the liveness that our system can guarantee in the following theorem.

Theorem 1 (Liveness)**.** *Let δ be an upper bound on the communication delay and Δ be an upper bound on the synchronization loss in all node's clocks with respect to a global clock. Let T_{comp} be the worst-case running time of any procedure run by the VC nodes and the voters during the voting protocol. Then, every honest voter that is engaged in the voting protocol at least $(f_v+1)\cdot\big((2N_v+5)T_{\mathsf{comp}}+12\Delta+6\delta\big)$ clock steps before election end, will obtain a valid receipt.*

Proof strategy overview. If an honest voter submits her vote to an honest responder, then by the description of the *VC* nodes in Section III-E and the bounds $\delta, \Delta, T_{\mathsf{comp}}$, we can show that the upper bound on the time required for the honest voter to obtain and verify the validity of her receipt is $T_{\mathsf{wait}} := (2N_v + 5)T_{\mathsf{comp}} + 12\Delta + 6\delta$. Thus, after T_{wait} steps, she will blacklist this *VC* node and submit the same vote to another randomly selected *VC* node. By the *VC* fault tolerance threshold, she will run into a honest responder after at most $f_v + 1$ attempts.

Safety. Our safety theorem is stated in the form of a contract adhered by the *VC* subsystem.

[1]In [29], the authors use the term *voter privacy/receipt-freeness*, but they actually refer to the same property.

Theorem 2 (Safety)**.** *Any honest voter who receives a valid receipt from a VC node, is assured her vote will be published on the honest BB nodes and included in the election tally, with probability at least $1 - \mathsf{negl}(\lambda) - \frac{f_v}{2^{64}-f_v}$.*

Proof strategy overview. Assume an adversary that attempts to produce a valid receipt without interacting with the honest *VC* nodes by either (i) forging digital signatures, hence producing fake UCERT certificates during vote collection, or (ii) guessing the randomly generated valid 64-bit receipt for some honest voter. By the security of digital signatures, (i) happens only with $\mathsf{negl}(\lambda)$ probability. Further, since there are at most f_v malicious *VC* nodes, the adversary has at most f_v attempts (there are $2^{64}-i$ choices left after i attempts) to guess the receipt for each voter, thus (ii) happens with probability no more than $\sum_{i=0}^{f_v-1} \frac{1}{2^{64}-i} \leq \frac{f_v}{2^{64}-f_v}$.

Now, let V be an honest voter that has obtained a receipt reconstructed from a complete *VC* interaction. Then, by the security arguments stated in Section III-E (steps 1-5), every honest *VC* node will submit V's vote to each *BB* node by including it in the set of voted tuples. By the fault tolerance thresholds, the honest *BB* nodes will publish V's vote, while the h_t out-of N_t honest trustees will read V's vote from the majority of *BB* nodes and include it in the election tally.

End-to-end verifiability. We define end-to-end verifiability by modifying the framework introduced in [29] accordingly to our setting. We require fault tolerance only for the *BB* nodes and prove the end-to-end verifiability of D-DEMOS in the following theorem.

Theorem 3 (End-to-end verifiability)**.** *Let θ be the number of honest voters. Let $\mathcal{A}$ be an adversary that controls the EA, all the VC nodes, all the trustee nodes and can statically corrupt up to f_b BB nodes. Then, if the honest voters and at least one auditor perform verification, the probability that $\mathcal{A}$ causes tally deviation d from the intended election result without being detected, is no more than $2^{-\theta} + 2^{-d}$.*

Proof strategy overview. The proof follows the lines of [29, Theorem 4]. Specifically, by the number of honest voters, the entropy of the collected voters' coins is at least θ. As in [29, Section 3.4], we can show that the verification of the Chaum-Pedersen zero-knowledge proofs guarantees the correctness of all the committed ballots in the *BB*, except some probability error $2^{-\theta}$. In case of all valid zero-knowledge proofs, $\mathcal{A}$ may attack by pointing the honest voter to audit in a *BB* location where the audit data is inconsistent with the respective information in at least one part of the voter's ballot. As in [29, Theorem 4], we can show that every such single attack has 1/2 success probability (the voter had chosen to vote with the inconsistent ballot part) and in case of success, adds 1 to the tally deviation. Thus, in this case, the probability that $\mathcal{A}$ causes tally deviation d is no more than 2^{-d}.

Voter Privacy. Our definition of voter privacy is similar to [29]. That is, an adversary instructs the honest voters to

vote according to either one of two alternative ways under the restriction that election tally is the same for both ways. The systems achieves voter privacy if the adversary cannot distinguish which alternative was followed by the honest voters. We require that the *EA* is destroyed after setup.

Theorem 4 (Voter Privacy)**.** *Let c, c' be constants s.t. $c' \in (0, c)$ and $n^2(n+1)^m \cdot 2^\phi = O(2^{\lambda^{c'}})$. Let $\mathcal{A}$ be an adversary that controls all the VC nodes, up to f_b BB nodes, up to f_t trustees, and up to ϕ voters, observes the network during election and obtains all the voters' audit information. Then, $\mathcal{A}$ cannot break voter privacy if the underlying commitment scheme is hiding against all 2^{λ^c} adversaries.*

Proof strategy overview. The proof follows the lines of [29, Theorem 5]. Due to full *VC* corruption, $\mathcal{A}$ learns all the vote-codes. Even so, the audit information of every voter leaks nothing about her vote, as each ballot part is independently and randomly generated, and the voter could "lie" about her used ballot part (i.e. switch the vote-code and option correspondence in the used ballot part, so that the submitted vote-code appears associated with the option in the alternative the voter did not follow). Moreover, we can show that if $\mathcal{A}$ distinguishes the alternative followed by honest voters, then we can construct an algorithm $\mathcal{B}$ that invokes $\mathcal{A}$ and simulates an election execution where it guesses (i) the corrupted voters' coins (in 2^ϕ expected attempts) and (ii) the election tally (in $(n+1)^m$ expected attempts). Thus, $\mathcal{B}$ finishes a compete simulation with high probability running in $n^2(n+1)^m \cdot 2^\phi = O(2^{\lambda^{c'}})$ steps. By exploiting the distinguishing advantage of $\mathcal{A}$, $\mathcal{B}$ can break the hiding property of the option-encoding commitment scheme in $O(2^{\lambda^{c'}}) = o(2^{\lambda^c})$ steps, thus leading to contradiction.

V. Implementation and evaluation

Implementation: We implement the Election Authority component of our system as a standalone C++ application, and all other components in Java. Whenever we store data structures on disk, or transmit them over the network, we use Google Protocol Buffers [2] to encode and decode them efficiently. We use the MIRACL library [4] for elliptic-curve cryptographic operations. In all applications requiring a database, we use PostgreSQL [6].

We build an *asynchronous communications stack* (ACS) on top of Java, using Netty [5] and the asynchronous PostgreSQL driver from [1], using TLS based authenticated channels for inter-node communication, and a public HTTP channel for public access. This infrastructure uses connection-oriented sockets, but allows the applications running on the upper layers to operate in a message-oriented fashion. We use this infrastructure to implement *VC* and *BB* nodes. We implement Bracha's Binary Consensus directly on top of the ACS, and we use that to implement our Vote Set Consensus algorithm of Section III-E. We introduce a version of Binary Consensus that operates in batches of arbitrary size; this way, we achieve greater network efficiency. We implement "verifiable secret sharing with honest dealer", by utilizing Shamir's Secret Share library implementation [7], and having the *EA* sign each share.

We implement a Mozilla Firefox extension which automates the task of reading from the *BB*, by intercepting the initial read request, replicating it to all *BB* nodes, capturing all replies, and showing a single correct reply only when it comes from the majority. For more details, see [16].

Evaluation: We experimentally evaluate the performance of our voting system, focusing mostly on our vote collection algorithm, which is the most performance critical part. We conduct our experiments using a cluster of 12 machines, connected over a Gigabit Ethernet switch. The first 4 are equipped with Hexa-core Intel Xeon E5-2420 @ 1.90GHz, 16GB RAM, and one 1TB SATA disk, running CentOS 7 Linux, and we use them to run our *VC* nodes. The remaining 8 comprise dual Intel(R) Xeon(TM) CPUs @ 2.80GHz, with 4GB of main memory, and two 50GB disks, running CentOS 6 Linux, and we use them as clients.

We implement a multi-threaded voting client to simulate concurrency. This client starts the requested number of threads, each of which loads its corresponding ballots from disk and waits for a signal to start. From then on, the thread enters a loop where it picks one *VC* node and vote code at random, requests the voting page from the selected *VC* (HTTP GET), submits its vote (HTTP POST), and waits for the reply (receipt). This simulates multiple concurrent voters casting their votes in parallel, and gives an understanding of the behavior of the system under the corresponding load.

We employ the PostgreSQL RDBMS [6] to store all *VC* initialization data from the *EA*. We start off by demonstrating our system's capability of handling large-scale elections. To this end, we generate election data for referendums, i.e., $m = 2$, and vary the total number of ballots n from 50 million to 250 million (note the 2012 US voting population size was 235 million). We fix the number of concurrent clients to 400 and cast a total of 200,000 ballots, which are enough for our system to reach its steady-state operation. Figure 1a shows the throughput of the system declines slowly, even with a five-fold increase in the number of eligible voters.

In our second experiment, we explore the effect of m, i.e., the number of election options, on system performance. We vary the number of options from $m = 2$ to $m = 10$. Each election has a total of $n = 200,000$ ballots which we spread evenly across 400 concurrent clients. As illustrated in Figure 1b, our vote collection protocol manages to deliver approximately the same throughput regardless of the value of m. Notice that the only extra overhead m induces during vote collection, is the increase in the number of hash verifications during vote code validation, as there are more vote codes per ballot.

Next, we evaluate the scalability of our vote collection protocol by varying the number of vote collectors and concurrent clients. We eliminate the database, by caching the election data in memory and servicing voters from the cache, to measure the net communication and processing costs of our voting protocol. We vary the number of *VC* nodes from 4 to 16, and distribute them across the 4 physical machines. Note that, co-located nodes are unable to produce vote receipts via local

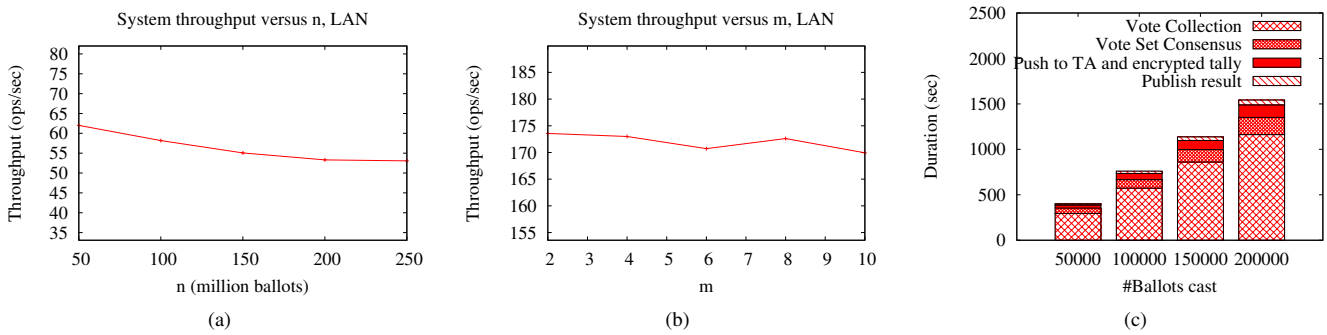


Figure 1. Throughput graphs of the vote collection phase versus the number of total election ballots n (1a) and the number of total election options m (1b). A total of 200,000 ballots were cast by 400 concurrent clients on 4 *VC* nodes. Figure 1c illustrates the duration of all system phases. Results depicted are for 4 *VC*s, n = 200,000 and m = 4. All these plots are for disk based experiments.

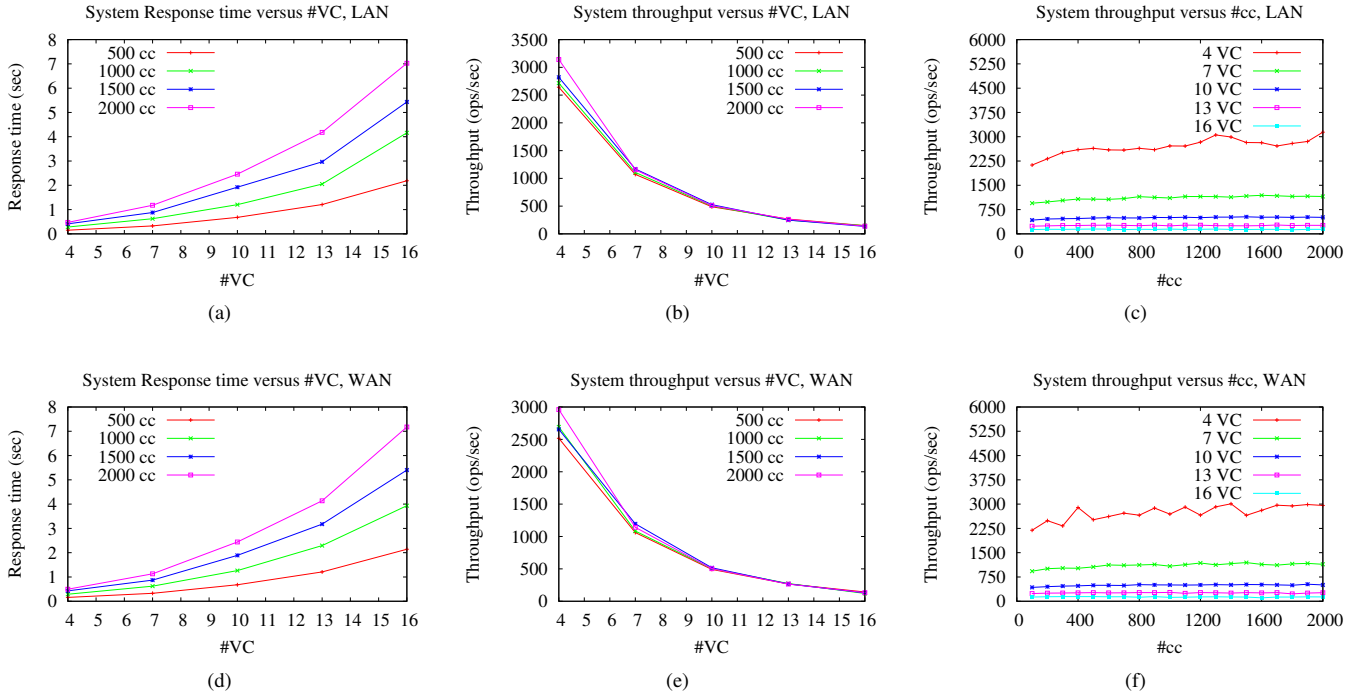


Figure 2. Latency (2a, 2d) and throughput graphs (2b, 2e) of the vote collection algorithm vs. the number of VC nodes. Figures (2c and 2f) illustrate throughput versus the number of concurrent clients. First row illustrates LAN setting plots. Second row illustrates WAN setting plots. Election parameters are n = 200,000 and m = 4.

messages only, since the $N_v - f_v$ threshold cannot be satisfied, i.e., cross-machine communication is still the dominant factor in receipt generation. For election data, we use the dataset with $n = 200,000$ ballots and $m = 4$ options.

In Figures 2a and 2b, we plot the average response time and throughput of our vote collection protocol, versus the number of vote collectors, under various concurrent client scenarios. Results illustrate an almost linear increase in the client-perceived latency, for all concurrency scenarios, up to 13 *VC* nodes. From this point on, when four logical *VC* nodes are placed on a single physical machine, we notice a non-linear increase in latency. We attribute this to the overloading of the memory bus, a resource shared among all processors of the system, which services all (in-memory) database operations.

In terms of overall system throughput, however, the penalty of tolerating extra failures, i.e., increasing the number of vote collectors, manifests early on. We notice an almost 50% decline in system throughput from 4 to 7 *VC* nodes. However, further increases in the number of vote collectors lead to a much smoother, linear decrease. We repeat the same experiment by emulating a WAN environment using *netem* [28], a network emulator for Linux. We inject a uniform latency of 25ms (typical for US coast-to-coast communication [3]) for each network packet exchanged between vote collector nodes, and

present our results in Figures 2d and 2e. A simple comparison between LAN and WAN plots illustrates our system manages to deliver the same level of throughput and average response time, regardless of the increased intra-VC communication latency. Finally, in Figures 2c and 2f, we plot system throughput versus the number of concurrent clients, in LAN and WAN settings respectively. Results show our system has the nice property of delivering nearly constant throughput, regardless of the incoming request load, for a given number of VC nodes.

Finally, in Figure 1c, we illustrate a breakdown of the duration of each phase of the complete voting system (D-DEMOS), versus the total number of ballots cast. We assume immediate phase succession, i.e., the vote collection phase ends when all votes have been cast, at which point the vote set consensus phase starts, and so on. The "Push to BB and encrypted tally" phase is the time it takes for the vote collectors to push the final vote code set to the BB nodes, including all actions necessary by the BB to calculate and publish the encrypted result. The "Publish result" phase is the time it takes for Trustees to calculate and push their share of the opening of the final tally to the BB, and for the BB to publish the final tally. Note that, in most voting procedures, the vote collection phase would in reality last several hours and even days as stipulated by national law (see Estonia voting system). Thus, looking only at the post-election phases of the system, we see the time it takes to publish the tally on the BB is quite fast.

Overall, although we introduced Byzantine Fault Tolerance across all phases of a voting system (besides setup), we demonstrate it achieves high performance, enough to run real-life elections of large electorate bodies.

VI. Conclusion

We have presented the world's first complete, state-of-the-art, end-to-end verifiable, distributed voting system with no single point of failure besides setup. The system allows voters to verify their vote was tallied-as-intended without the assistance of special software or trusted devices, and external auditors to verify the correctness of the election process. Additionally, the system allows voters to delegate auditing to a third party auditor, without sacrificing their privacy. We provided a model and security analysis of our voting system. Finally, we implemented a prototype of the integrated system, measured its performance and demonstrated its ability to handle large scale elections.

Acknowledgements: This work was partially supported by ERC Starting Grant # 279237 and by the FINER project funded by the Greek Secretariat of Research and Technology under action "ARISTEIA 1".

References

[1] "Asynchronous postgresql java driver." https://github.com/alaisi/postgres-async-driver/.
[2] "Google protocol buffers," https://code.google.com/p/protobuf/.
[3] "High performance browser networking: What every web developer should know about networking and web performance," http://chimera.labs.oreilly.com/books/1230000000545/ch01.html#PROPAGATION_LATENCY.
[4] "Miracl multi-precision integer and rational arithmetic c/c++ library." http://www.certivox.com/miracl/.
[5] "Netty, an asynchronous network application framework." http://netty.io/.
[6] "Postgresql rdbms." http://www.postgresql.org/.
[7] "Shamir's secret share in java." https://github.com/timtiemens/secretshare.
[8] B. Adida, "Helios: Web-based open-audit voting," in *USENIX Security Symposium*, 2008.
[9] P.-L. Aublin, S. Ben Mokhtar, and V. Quéma, "Rbft: Redundant byzantine fault tolerance," in *IEEE ICDCS*, 2013.
[10] J. Benaloh, M. D. Byrne, B. Eakin, P. T. Kortum, N. McBurnett, O. Pereira, P. B. Stark, D. S. Wallach, G. Fisher, J. Montoya, M. Parker, and M. Winn, "STAR-vote: A secure, transparent, auditable, and reliable voting system," in *EVT/WOTE '13*, Aug. 2013.
[11] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, February 1999.
[12] D. Chaum, "Surevote: Technical overview," in *Proceedings of the Workshop on Trustworthy Elections*, ser. WOTE, Aug. 2001.
[13] D. Chaum, A. Essex, R. Carback, J. Clark, S. Popoveniuc, A. Sherman, and P. Vora, "Scantegrity: End-to-end voter-verifiable optical-scan voting," *Security & Privacy, IEEE*, vol. 6, no. 3, pp. 40–46, 2008.
[14] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *CRYPTO '92*. Springer-Verlag, 1993, pp. 89–105.
[15] D. Chaum, P. Y. A. Ryan, and S. A. Schneider, "A practical voter-verifiable election scheme," in *ESORICS 2005*, Sept. 2005, pp. 118–139.
[16] N. Chondros, B. Zhang, T. Zacharias, P. Diamantopoulos, S. Maneas, C. Patsonakis, A. Delis, A. Kiayias, and M. Roussopoulos, "A distributed, end-to-end verifiable, internet voting system, extended version," *CoRR*, vol. abs/1507.06812, 2015. [Online]. Available: http://arxiv.org/abs/1507.06812
[17] M. R. Clarkson, S. Chong, and A. C. Myers, "Civitas: Toward a secure voting system," in *IEEE Symposium on Security and Privacy*, 2008.
[18] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proc. of ACM SOSP*, 2009.
[19] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults." in *NSDI*, vol. 9, 2009, pp. 153–168.
[20] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of USENIX OSDI*, 2006.
[21] R. Cramer, R. Gennaro, and B. Schoenmakers, "A secure and optimally efficient multi-authority election scheme," in *EUROCRYPT*, 1997.
[22] C. Culnane, P. Y. A. Ryan, S. Schneider, and V. Teague, "vvote: a verifiable voting system (DRAFT)," *CoRR*, vol. abs/1404.6822, 2014. [Online]. Available: http://arxiv.org/abs/1404.6822
[23] C. Culnane and S. Schneider, "A peered bulletin board for robust use in verifiable voting systems," in *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*. IEEE, 2014, pp. 169–183.
[24] G. Dini, "A secure and available electronic voting service for a large-scale distributed system," *Future Generation Computer Systems*, vol. 19, no. 1, pp. 69–85, 2003.
[25] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Springer-Verlag CRYPTO 1984*.
[26] K. Fisher, R. Carback, and A. Sherman, "Punchscan: introduction and system definition of a high-integrity election system," in *WOTE*, 2006.
[27] K. Gjøsteen, "The norwegian internet voting protocol," *IACR Cryptology ePrint Archive*, vol. 2013, p. 473, 2013. [Online]. Available: http://eprint.iacr.org/2013/473
[28] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*. Citeseer, 2005, pp. 18–23.
[29] A. Kiayias, T. Zacharias, and B. Zhang, "End-to-end verifiable elections in the standard model," in *EUROCRYPT 2015*, April 2015, pp. 468–498.
[30] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *SOSP*, Oct 2007.
[31] M. Kutylowski and F. Zagórski, "Scratch, click & vote: E2E voting over the internet," in *Towards Trustworthy Elections, New Directions in Electronic Voting*, 2010, pp. 343–356. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12980-3_21
[32] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.
[33] T. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology CRYPTO*, 1991.
[34] B. Schneier, *Applied cryptography*. John Wiley & Sons, 1996.
[35] F. Zagórski, R. T. Carback, D. Chaum, J. Clark, A. Essex, and P. L. Vora, "Remotegrity: Design and use of an end-to-end verifiable remote voting system," in *Applied Cryptography and Network Security*, 2013.