NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

MSc THESIS

A Distributed Bulletin Board Implementation for Practical Use in e-Voting Systems

Giorgos K. Metaxopoulos

Supervisors: **Alex Delis,** Professor NKUA
Dimitris Mitropoulos, Assistant Professor NKUA

ATHENS

March 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση ενός κατανεμημένου Πίνακα Ανακοινώσεων για πρακτική χρήση σε συστήματα ηλεκτρονικών ψηφοφοριών

Γιώργος Κ. Μεταξόπουλος

Επιβλέποντες: **Αλέξης Δελής,** Καθηγητής ΕΚΠΑ
Δημήτρης Μητρόπουλος, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

Μάρτιος 2022

MSc THESIS

A Distributed Bulletin Board Implementation for Practical Use in e-Voting Systems

Giorgos K. Metaxopoulos
S.N.: CS3190002

SUPERVISORS: **Alex Delis,** Professor NKUA
Dimitris Mitropoulos, Assistant Professor NKUA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση ενός κατανεμημένου Πίνακα Ανακοινώσεων για πρακτική χρήση σε συστήματα ηλεκτρονικών ψηφοφοριών

Γιώργος Κ. Μεταξόπουλος

Α.Μ.: CS3190002

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Αλέξης Δελής,** Καθηγητής ΕΚΠΑ
Δημήτρης Μητρόπουλος, Αναπληρωτής Καθηγητής ΕΚΠΑ

ABSTRACT

Contemporary e-voting systems are built around a central, publicly accessible, digital Bulletin Board (BB). All items posted to a BB are authenticated and no entity should be able to either erase or modify them. However, the BB can be a potential single point of failure. To address this issue, a number of distributed approaches have been proposed for the make up of BBs. By and large, such proposals lack either interoperability as they are tied to specific e-voting systems or a review under a formal security model. In this thesis, we discuss a set of proposed distributed BB protocols that do not suffer from the above shortcomings and analyze their security properties based on a specific security framework.

Our solution consists of a platform-independent set of modules that not only realize the above BB protocols but also can be applied on top of existing e-voting systems. As a proof of concept, we integrate our solution into Zeus, a well-established verifiable internet ballot casting and counting system. Moreover, we articulate key implementation aspects of our approach and underline the assumptions of our solution. Finally, we evaluate the scalability properties of the integrated protocols and provide an experimental security analysis. In this context, we simulate different adversarial scenarios and assess the guarantees that our realization of the BB protocols yields.

SUBJECT AREA: E-voting/Security/Distributed Systems

KEYWORDS: E-voting, Bulletin Board, Fault-tolerance

ΠΕΡΙΛΗΨΗ

Τα σύγχρονα συστήματα ηλεκτρονικών ψηφοφοριών χτίζονται γύρω από έναν κεντρικοποιημένο, δημοσίως διαθέσιμο, ψηφιακό Πίνακα Ανακοινώσεων (ΠΑ). Όλα τα αντικείμενα που αναρτώνται στον ΠΑ είναι αυθεντικοποιημένα και δεν πρέπει κανένας να έχει τη δυνατότητα είτε να τα διαγράψει, είτε να τα τροποποιήσει. Ένας ΠΑ μπορεί να αποτελέσει ένα μοναδικό σημείο αποτυχίας ενός συστήματος. Για να αντιμετωπιστεί αυτό το κρίσιμο ζήτημα, μια σειρά από κατανεμημένες προσεγγίσεις έχουν προταθεί για την κατασκευή του ΠΑ. Γενικά, αυτές οι προτάσεις στερούνται είτε διαλειτουργικότητας, διότι είναι στενά συνδεδεμένες με συγκεκριμένα συστήματα ηλεκτρονικών ψηφοφοριών, είτε επιθεώρησης βάσει ενός επίσημου μοντέλου ασφαλείας. Στην παρούσα Διπλωματική Εργασία, ερευνούμε ένα σύνολο προτεινόμενων κατανεμημένων πρωτοκόλλων για ΠΑ, τα οποία δεν πάσχουν από τα προαναφερθέντα ελαττώματα, και αναλύουμε τις ιδιότητες ασφαλείας τους βάσει ενός συγκεκριμένου πλαισίου ασφαλείας.

Η προτεινόμενη λύση μας αποτελείται από ένα ανεξάρτητο πλατφόρμας σύνολο δομοστοιχείων λογισμικού, τα οποία όχι μόνο υλοποιούν τα παραπάνω πρωτόκολλα, αλλά μπορούν και να εφαρμοστούν πάνω σε υπάρχοντα συστήματα ηλεκτρονικών ψηφοφοριών. Για την επικύρωση των ανωτέρω, ενσωματώνουμε τη λύση μας στο Zeus, ένα εδραιωμένο, επαληθεύσιμο, διαδικτυακό σύστημα κατάθεσης και καταμέτρησης ψηφοδελτίων. Επιπλέον, διατυπώνουμε βασικές πτυχές της υλοποίησης της προσέγγισής μας και επισημαίνουμε τις παραδοχές της προτεινόμενης λύσης. Στο τέλος, αξιολογούμε της ιδιότητες κλιμακωσιμότητας των ενσωματωμένων πρωτοκόλλων και παρέχουμε μία πειραματική ανάλυση ασφαλείας. Σε αυτήν, προσομοιώνουμε διαφορετικά σενάρια αντιπάλων και αξιολογούμε τις εγγυήσεις που παρέχει αυτή μας η υλοποίηση των πρωτοκόλλων για ΠΑ.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ηλεκτρονικές ψηφοφορίες/Ασφάλεια/Κατανεμημένα συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ηλεκτρονικές ψηφοφορίες, Πίνακας Ανακοινώσεων, Ανοχή βλαβών

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Alex Delis, for his pertinent comments and his overall cooperation in bringing this work to fruition.

I would also like to express my gratitude to my advisor, Dimitris Mitropoulos, for his support and guidance throughout the whole duration of my studies. His abilities in clarifying complex concepts and seeing the whole picture of a research endeavor are unmatched. His insightful advices played a big role in the successful realization of this work.

Finally, I owe a big thank you to my dear friend and classmate, Stefanos Chaliasos. His undeniable talent and passion for Computer Science served as a huge inspiration in every step of my studies. I am certain that his future career will show how lucky I was for working with him in so many projects.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

E-voting systems were introduced as a means of improving the accessibility of voters to elections, while reducing the associated costs. A critical property of e-voting systems [3, 23, 35, 19, 34] is end-to-end verifiability. This property allows voters and third-party auditors to verify that votes were properly cast, recorded and tallied into the election result. This in turn, requires a publicly available and reliable method of posting and reading all required election information.

To satisfy this need, e-voting systems employ a digital Bulletin Board (BB), which abstractly encompasses two key components: (1) an item collection (IC) subsystem that receives and stores the different items submitted by users, and (2) a publicly accessible audit board (AB) for publishing the final agreed-on record of items.

In most internet e-voting systems, the BB is realized as a centralized component, adding a single point of failure to the setup, with regards to service availability, voter secrecy and the integrity of the final result. Motivated by this shortcoming, many studies [11, 28, 17, 21, 5, 9, 8, 18] have attempted to introduce a distributed BB component, aiming to achieve tolerance against arbitrary failures. The two most concrete examples of distributed BB designs include the Secure Web Bulletin Board (WBB) [9] and the BB of the D-DEMOS Internet Voting system [8]. In the latter case, the proposed BB is an integral part of a specific e-voting system, making the BB unsuitable for use in other e-voting settings. In the former case, while the proposed BB design is independent of the underlying e-voting system, it lacks review under a formal security model.

In reference [18], the authors introduce a security framework for the formal analysis of BB systems. The framework consists of three main properties: *Confirmable Liveness*, *Persistence*, and *Confirmable Persistence*. *Confirmable Liveness* dictates that all honest users that post an item x will obtain a valid receipt for its proper submission, and x will be published by the AB. *Persistence* asserts that published items cannot be removed from the AB, and only items that have been posted may appear on it. *Confirmable Persistence* accounts for an AB subsystem that is fully controlled by an adversary and requires any malicious AB behavior to be detectable by any auditor. The authors of reference [18] also examine the guarantees of the BB system introduced in reference [9] under their framework. Inspired by an attack on Confirmable Liveness, they modify one of the BB protocols presented in [9] and proceed to evaluate the properties of their new design based on their formal security model.

Motivated by the BB protocols described in reference [9] and the refinements reported by Kiayias et al. [18], we propose an implementation of the following:

- A Posting Protocol for a BB system as defined in [9]. We refer to this system as the “CS BB system”.
- The Publishing Protocol of the CS BB system, as defined in [9]. We refer to it as the “CS BB Publishing Protocol”.
- The Publishing Protocol of the BB system as defined in [18]. We refer to this system as the “KKLSZ BB system” and to the respective Publishing Protocol as the “KKLSZ BB Publishing Protocol”. The KKLSZ BB system uses the same Posting Protocol as the CS BB system.

Our implementation can be applied on top of any e-voting system, as long as the existing

BB component is not restrictively tied to other components of the system.

As a proof of concept, we integrate the CS BB and the KKLSZ BB protocols on top of Zeus [34], a well-established e-voting system, that is able to handle any kind of voting schemes in a completely open source implementation.

We evaluate the scalability properties of our solution, by measuring latency and throughput for a varying number of IC nodes and concurrent voters. Our initial results indicate that the current implementation is suitable for small to medium sized elections, as the user-perceived latency when casting a vote may grow out of acceptable bounds (in a realistic election scenario), if we employ too many IC nodes and/or expect too many concurrent client requests. We also simulate a series of adversarial scenarios and provide an experimental security evaluation of Confirmable Liveness, Persistence, and Confirmable Persistence for our implementation of the CS BB and the KKLSZ BB systems. Our experimental results suggest that our implementation of each system preserves the properties that were defined in [18]. A formal study of our implementation and its respective assumptions would be required to properly verify that all expected properties hold, yet, our experimental findings provide a promising first assessment.

The rest of this thesis is structured as follows: In Section 2, we provide some fundamental terminology for e-voting, and proceed with an overview of the main e-voting and distributed e-voting systems that influenced our proposal. Section 3, presents a comprehensive description of the BB systems we have implemented, as originally defined in [9] and [18]. Accordingly, we discuss the security properties of each system, under the formal framework proposed in [18]. Then in Section 4, we demonstrate our implementation of the BB protocols. In particular, we discuss any assumptions made during the development of the respective modules and provide guidelines for executing our proof-of-concept application of our modules into Zeus. Section 5, evaluates our implementation, in terms of both scalability and security, by simulating a number of different election setting and setups and leveraging our integration of the BB protocols on top of Zeus. Finally, in Section 6, we summarize the main findings of our study and provide guidelines for future work.

2. BACKGROUND AND RELATED WORK

2.1 E-voting Primitives

In this section, we will provide definitions as well as a brief overview of the key components that comprise modern e-voting systems. Given that the objective of this work is focused on providing a practical software solution, rather than an extensive research on e-voting systems from a cryptographic perspective, we will only focus on aspects of e-voting that are relevant to our work and the respective analyses are purposely designed to provide a general understanding of the underlying concepts, rather than an extensive investigation of cryptographic properties and proofs.

2.1.1 E-voting Primitives

2.1.1.1 E2E Verifiability

In an end-to-end (E2E) verifiable election system, voters have the ability to verify that their vote was properly cast, recorded and tallied into the election result. Intuitively, the security property that an E2E verifiable election intends to capture is the ability of the voters to detect a malicious election authority that tries to misrepresent the election outcome. E2E verifiability mandates that the voter can obtain a receipt at the end of the ballot casting procedure that can allow her to verify that her vote was (i) cast as intended, (ii) recorded as cast, and (iii) tallied as recorded. Furthermore, any external third party should be able to verify that the election procedure is executed properly [20].

Cast-as-intended indicates that voters can verify that the voting system correctly marked her candidate choice on the ballot. Recorded-as-cast means that the voter can verify that her vote was correctly recorded by the voting system. Tallied-as-recorded means that the voter can verify that her vote was counted as recorded. This translates as follows: the voter first confirms the system has correctly encrypted her vote. She then tracks her vote on the bulletin board using her receipt and confirms that it is correctly recorded. Integrity of the result is ensured by rigorously auditing the tallying process and requiring the system to publish cryptographic proofs of correct operation. This three-step verification therefore covers the entire life cycle of the vote, and the voter can be confident that if there is any tampering or breakdown in the system it will be discovered in one of the checks [4].

Cryptography makes end-to-end voting verification possible. At a high level, cryptographic voting systems effectively bring back the voting systems of yore, when all eligible citizens voted publicly, with tallying also carried out in public for all to see and audit. Cryptographic schemes augment this approach with [2]:

1. encryption to provide ballot secrecy, and
2. zero-knowledge proofs to provide public auditing of the tallying process.

2.1.1.2 Bulletin Board

Cryptographic voting protocols revolve around a central, publicly accessible, digital Bulletin Board (BB). All messages posted to the BB are authenticated, and it is assumed that

any data written to the BB cannot be erased or tampered with. The names or identification numbers of voters are posted in plaintext, along with the voter's ballot in encrypted form. Two processes surround the BB. The ballot casting process lets Alice prepare her encrypted vote and cast it to the BB. The tallying process involves election officials performing various operations to aggregate the encrypted votes and produce a decrypted tally, with proofs of correctness of this process also posted to the BB for all observers to see. Effectively, the BB is the verifiable transfer point from identified to de-identified ballots. Votes first appear on the BB encrypted and attached to the voter's identity. After multiple transformations by election officials, the votes end up on the BB, decrypted and now unlinked from the original voter identity [2].

2.1.1.3 Distributed Bulletin Board

Many of the proposed e-voting solutions in the literature relied on a centralized BB component. This centralized approach introduces a single point of failure to the respective setup. In addition, the centralized approach raises issues with regards to availability (in terms of both service and data), integrity of the result and trust. That is why, a series of distributed BB designs have been introduced [11, 28, 17, 21, 5, 9, 8, 18], striving to provide tolerance against Byzantine failures, increased availability and voter secrecy.

2.1.1.4 Overview of a voting protocol

Most verifiable voting protocols in the literature present the following sequence of events:

1. Setup: Election setup parameters are generated and published.
2. Ballot Preparation: Alice, the voter, prepares her ballot with the help of a special ballot or machine. The result is an encrypted vote.
3. Ballot Recording: Alice's encrypted ballot is posted on a world-readable bulletin board, paired with Alice's identity in plaintext.
4. Anonymization & Aggregation: A publicly-verifiable shuffling (and potentially aggregation) algorithm is run, with intermediate results posted on the bulletin board.
5. Results: Election officials cooperate to produce a plaintext tally for each race, again with publicly-verifiable proofs posted to the bulletin board.

In general, two broad categories of schemes exist, i.e., aggregate voting schemes and ballot-preserving voting schemes. In the former, the output of the protocol indicates only the aggregate number of votes for each candidate in each race. In ballot-preserving voting schemes, all plaintext ballots are preserved in their entirety all the way through the tallying process [2].

2.1.1.5 Mixnet

A mix network or mixnet is a cryptographic construction that invokes a set of servers to create private communication channels. In general, a mix network accepts as input a collection of ciphertexts, and outputs associated plaintexts in a randomly permuted order.

The special security property of a mix network is the secrecy it creates in the correspondence between inputs and outputs. In particular, a well constructed mix network makes it unfeasible for an adversary to determine which input ciphertext corresponds to which output plaintext any more effectively than by guessing at random [16].

Since mix-nodes also perform a transformation process that modifies the values of the set of input encrypted votes, it is important to be able to verify the mixing and decryption procedures in such a way that privacy and integrity are preserved. In 1995, Sako and Kilian [31] introduced the concept of "universal verifiability" for their proposal of a vote decryption process based on a mixnet approach. This verifiability is focused on providing means for any auditor or observer to verify the correct decryption of the votes, using cryptographic proofs that are generated by the decryption process [29].

2.2 E-Voting

In this section, we will briefly describe certain e-voting systems that currently employ a centralized BB component. While there several such E2E verifiable e-voting systems [3, 23, 35, 19, 34], we focus on Helios [3] and Zeus [34], as they are the two systems that are mostly related to our implementation. In particular, even though the set of modules we have implemented (cf. Section 4) can be built on top of various e-voting systems, our proof of concept consists of an intergration of the BB protocols into Zeus. That is why, an overview of the key procedures of Zeus may be useful for a better understanding of our prototype integration. Given the historical significance of Helios, as well as the fact that Zeus is derived from it, we also provide a concise summary of Helios' main components and operations.

2.2.1 Helios

In 2008, Ben Adida introduce Helios [3], the first web-based, open-audit voting system. Using a modern web browser, anyone can set up an election, invite voters to cast a secret ballot, compute a tally, and generate a validity proof for the entire process. One key limitation of the original Helios system was the fact that it only supported simple elections, where a voter selects 1 or more out of the proposed candidates.

Helios promotes unconditional integrity over unconditional privacy. Thus, there is only 1 trustee, the Helios server itself. Privacy is guaranteed only if you trust Helios. With regards to integrity, even if the Helios server is corrupt, the election results can still be fully audited. The Helios Protocol consists of vote preparation and casting, the bulletin board of votes and the Sako-Kilian mixnet [31].

With regards to vote preparation and casting, a ballot for an election can be completed by anyone, meaning that authentication is only required at ballot casting time. Once a voter has filled in her ballot and confirmed her choices, she can choose to either audit or seal the ballot. When auditing, the Ballot Preparation System (BPS) encrypts the voter's choices and displays a hash of the ciphertext. In addition, the BPS displays the ciphertext and the randomness used to create it, so that the voter can verify that the BPS had correctly encrypted her choices. If this process is selected, a new encryption of her choices is generated to proceed with sealing the ballot. When sealing, the BPS discards all randomness and plaintext information, leaving only the ciphertext, ready for casting. The voter is then

prompted to authenticate. If successful, the encrypted vote, which the BPS committed to earlier, is recorded as the final vote of the respective voter.

The BB of votes is publicly available and cast votes are displayed next to either a voter name or a voter ID. All subsequent processing is also posted on the BB for the public to download and verify. The BB in Helios is run by a single server and its integrity can be verified by both auditors and voters. Auditors, are expected to check the BB's integrity over time and individual voters are expected to check that their encrypted vote appears on the BB.

For mixing, Helios employs the Sako-Kilian protocol in order to preserve individual ballots and potentially support write-in votes, while achieving anonymization of the ballots. A mix server is responsible for shuffling and re-randomizing the cast ciphertexts before jointly decrypting them. When an election closes, Helios shuffles all encrypted ballots and produces a non-interactive proof of correct shuffling, correct with overwhelming probability.

After a reasonable period to let auditors check the shuffling, Helios decrypts all shuffled ballots, provides a decryption proof for each, and performs a tally. An auditor can download the entire election data and verify the shuffle, decryptions, and tally.

2.2.2 Zeus

In 2012, the Greek Research and Education Network (GRNET) was asked to provide a system for electronic voting to be used in elections in universities in Greece. As expected, GRNET's approach began with determining whether Helios could facilitate the requested election type. The version of Helios available at the time (version 3), used homomorphic tallying in place of mixnets, rendering the use of Helios for the specific election impossible. In particular, the election that GRNET was asked to accommodate used the Single Transferable Vote (STV) system [33], in which voters do not simply indicate the candidates of their preference, but also rank them in order of preference. In STV, homomorphic tallying as then implemented in Helios could only pass to the STV algorithm the information that a certain candidate has been selected in rank r by n voters. That was not sufficient though, as STV's counting rounds require the whole ballot and not just each rank separately.

The aforementioned limitation ultimately led to the creation of Zeus, an extension of Helios that is able to handle any kind of voting systems in a completely open source implementation [34].

Each poll in Zeus comprises of the following phases:

- Poll preparation: A Zeus administrator account is created for the organizers of the poll, allowing the poll administrator to create a new poll, choose who the trustees are going to be, select the type of the election and define the content of the poll accordingly. The administrator also uploads the list of voters that may participate in the poll. Before voting starts each of the trustees must visit Zeus's website and generate and register their encryption keys for the poll. Upon their designation as trustees, Zeus sends them a confidential invitation which they can use to log in to the system and perform their duties. Trustees must follow their invitation and log in to generate their key pair. After key creation, the trustee is prompted to save her private key. At this point, the session is ended and Zeus send a new invitation to the trustee. Following the link, a new browser session is initiated and the trustee is shown the hash value of her public key, prompted to locate their saved private key.

The browser verifies the key locally by comparing its hash value to the one registered by the server. This process is important to ensure that the trustees have indeed saved their keys, otherwise the decryption will not be possible. Once administrators decide to finalize the poll, all registered voters are sent their confidential invitations to vote. As with trustees, voters do not need an account, but use the secret information in their invitation.

- Voting: The voting booth operates entirely locally without further interactions with the Zeus server or any other site. Once the booth is open, the voters select, review and confirm their choices. After vote submission, a vote submission receipt is generated. This receipt can immediately be downloaded, but it is also sent to the respective voter via e-mail. The receipt is also registered in the poll document. Vote submission receipts are text files which list cryptographic information about the vote and poll. The receipt text is signed with Zeus's own trustee key for the poll (Zeus is a trustee for every poll) and contains a unique identification of the cast vote, the vote (if any) which it replaces, the cryptosystem used, the poll's public key along with the public keys of all trustees, the list of candidates, the cryptographic proof that it was a valid encryption and the signature itself. The validity of the encryption can be verified by checking the discrete log proof submitted, along with the encrypted ballot.
- Processing: Processing consists of the *mixing*, *decrypting* and *finished* stages. Once the poll administrator has explicitly select to close the poll, the poll is put into the *mixing* stage, where the encrypted ballots are anonymized. Only the ballots eligible for counting are selected, by excluding all audit votes, all replaced votes and all votes by excluded voters. The eligible encrypted votes are first mixed by Zeus itself by a Sako-Kilian mixnet. After the first mix is complete, additional mixes by external agents may be verified and added in a mix list using Zeus's command line toolkit. Once mixing is complete, the final mixed ballots are exported to the trustees to be partially decrypted (*decrypting* stage). The resulting decryption factors are then verified and imported into the poll document, along with Zeus's own decryption factors which are computed last. In the transition to the final stage of the processing phase, denoted as *finished*, all decryption factors are distributed in parallel workers to be verified and combined together to yield the final decrypted ballots. The decrypted ballots are recorded into the poll document and the document is cast into a canonical representation in text. This textual representation is hashed to obtain a cryptographically secure identifier for it, which can be published and recorded in the proceedings.
- Tallying: The ballots are either sent into another system for tallying and reporting the results, or the ballots are tallied and reported by Zeus itself, depending on the poll type.

2.3 Distributed E-Voting

In the following sections, we will be briefly discussing certain important solutions on the distributed e-voting realm. Most of these solutions focus on providing a distributed BB component [28, 17, 21, 5, 9, 18], while others have proposed fully distributed e-voting architectures [11, 8].

Before proceeding with the aforementioned analysis, it is important to highlight the key motivations behind designing e-voting solutions based on distributed components. The main problem deriving from centralized solutions is the introduction of a single point of

failure to the respective deployment. If any of the centralized components of the system suffers a failure, the whole service becomes unavailable until the failure has been restored. Apart from service availability, single point of failures due to the presence of centralized components may result in the compromise of voter secrecy or the integrity of the result [8]. Therefore, some key considerations when designing such systems include tolerance over Byzantine failures (a corrupt party may behave arbitrarily or even in cooperation with other corrupt parties to maliciously manipulate the behavior of the service), proper replication of the components of a distributed (sub)system, consistency of the data copies of each component, integrity and privacy.

2.3.1 Dini's E-voting Distributed Service

Dini introduces an e-voting service suitable for large-scale distributed systems such as the Internet, focusing on the availability and the security of the system, as well as tolerance against benign failures of voters [11]. To achieve the first two properties, the proposed solution follows a replication-based approach. Trust is distributed among a set of servers that are collectively responsible to carry out the voting operations. If enough servers are correct, service availability and security can be ensured despite the presence of faulty servers acting against the voting process, both individually and in collusion with one another or with malicious voters [11]. With regards to voter failures, the author proves that the system may tolerate voter crashes by allowing voter validation and vote casting to be repeated across failures and recoveries while preserving the security requirements of the service. To achieve the aforementioned properties, the author employs dissemination quorums [24] on every given voting operation. Since quorums are small ($O(\sqrt{n})$ out of n servers) and do not require server-to-server or voter-to-voter interaction, the e-voting service can be deployed efficiently in large-scale systems. Only eligible voters are allowed to vote, an eligible voter cannot vote more than once and no one can determine how any individual voted (voter eligibility and privacy). In addition, The proposed e-voting service supports common security requirements: only eligible voters should be able to vote; an eligible voter should not vote more than once; no one should be able to determine how any individual voted. In addition, any party, even external observers, can verify that the voting outcome was computed fairly from the correctly cast ballots (tally accuracy and verifiability).

2.3.2 Peter's Bulletin Board

Peters studied the design of a secure bulletin board, implemented by a distributed protocol that can be executed among several parties. The goal was to propose an e-voting system that can operate efficiently and correctly even under the presence of malicious voters and/or talliers. In order to achieve that, Peters examined a series of existing protocols, namely Rampart [30], a protocol described by Kursawe and Shoup [22], Phalanx [26], another version of Phalanx [25], as well as an adapted version of Rampant and the original Phalanx protocol, so that they employ threshold signatures to boost their performance. After thorough examination of the performance characteristics of each protocol, Peters opted for the adapted version of Rampant and implemented the proposed bulletin board, discussing its resilience to various attacks, its efficiency and other implementation considerations. Users can post messages to this board, and once they have received a signed acknowledgment, they have the assurance that their message will never be deleted, will never be changed, and will be available to every other user. Also, no authorized user can

be denied access to posting and reading messages. These properties hold even when up to one-third of the parties comprising the bulletin board are corrupted [28].

2.3.3 Heather & Lundin's Bulletin Board

Heather and Lundin [17] identify three key agents participating in an e-voting system: the web bulletin board, readers and writers. Based on their assumptions, the web bulletin board allows various authorized parties to publish information on it, which can then be read by any of the readers. Once something new is published to the board, it should be placed at the end of the sequence messages of the board and it should never be removed or altered (append-only web bulletin board). The authors begin by providing a scheme for implementing such a board, including desired security properties of the scheme, such as, certified publishing, timely publication, unalterable history, and verifiability, along with the respective proofs of each property. This scheme, however, does not yet provide any liveness guarantees. Although the history of the board cannot be manipulated, it is totally possible that the board may -for some reason- refuse to communicate with one or more agents. That is why, the authors decided to investigate how their board can be distributed among a number of peers, making it robust against arbitrary failures. The distributed web bulletin board that the authors outline suggests the presence of a collective of bulletin board peers. The certificates issued by the web bulletin board to the writer are now issued by the web bulletin board peers as a collective. This can be enabled by a threshold cryptography scheme. As long as some threshold set k out of n bulletin board peers survive and function correctly, the integrity of the election should be guaranteed by the collective. With regards to writing messages on this distributed board, the authors suggest two possible solutions, a synchronous and an asynchronous one. In the synchronous case, a peer receives a message from a writer and attempts to form a threshold set of peers to sign the respective receipt. All k peers involved in the signing add the message to their history and the message is also sent to the remaining $n - k$ peers for publication. A threshold set must contain more than half of the peers ($2k > n$) according to the authors. In the asynchronous case, the remaining $n - k$ peers do not need to necessarily replicate the message that has been accepted by a threshold number of peers. Each peer now maintains a local append-only structure and two or more messages can be written concurrently. Yet, apart from outlining the aforementioned design, the authors do not provide an analysis of the security and liveness properties of the distributed web bulletin board, nor do they contribute an implementation of their proposal.

2.3.4 Krummenacher's Bulletin Board

Krummenacher [21] builds on the Web Bulletin Board (WBB) described by Heather and Lundin [17], by restating and complementing the proposed designs and properties of both a single WBB and a distributed WBB, and -more importantly- by providing a scheme for the practical implementation of such a distributed WBB. Focusing on the latter, the author defines n as the total number of WBBs that constitute the distributed WBB, k as the maximum number of WBBs that could fail or be controlled by a single adversary, l (with $l \geq k + 1$) as the size of a threshold set (a group of WBBs that redundantly publish a message m), and p as the number of parties (such as political parties, election observation organizations) that operate a WBB. After some investigation on certain variants of the synchronous and the asynchronous WBBs outlined by Heather and Lundin, Krummenacher proceeds to define a model where each party p operates a single WBB component

($p = n$) and every WBB component can have up to a predefined number of histories. The final proposed distributed WBB implementation ensures that a message is either not published at all, or it is published on $k + 1$ boards, that the writer receives a valid receipt only if the message is published in at least $k + 1$ boards, that the system is tolerant to arbitrary failures of up to k WBB components (otherwise the correctness of the distributed WBB cannot be guaranteed), and that the system scales in proportion to the number of existing histories. While the author provides certain useful considerations with regards to the implementation of each variant and the proper selection of the respective parameters (p, l, number of histories per WBB etc.), it is stated that an effective scheme relies on experienced data or estimations so that the safety of the system remains guaranteed.

2.3.5 Culnane & Schneider's Bulletin Board

Arising from the need to implement a bulletin board as part of the vVote system for the Victorian State election of 2014 [7], Culnane and Schneider [9] introduced a novel distributed protocol for running a bulletin board consisting of a network of peers, that operates correctly even in the presence of individual peers going down, external attacks and a minority of dishonest peers. In particular, for n peers, a threshold of $t > \frac{2n}{3}$ peers behaving correctly is sufficient to ensure the proper behavior of the proposed BB design. It is worth noting that different threshold sets of peers may be operational at different times (thus some peers may be missing some posts), without threatening the availability and the correctness of the system. The authors use the Event-B modelling and refinement approach [1] to formally verify their protocol in the context of a Dolev-Yao adversary [12] having control over the network and a minority of peers. The authors utilize a threshold signature scheme to allow a subset of the peers (any subset of $t > \frac{2n}{3}$ honest peers) to jointly provide signatures on data. The bulletin board accepts items to be posted (if they do not clash with previous posts), issues receipts and periodically publishes the posts it has received thus far. The bulletin board published for any particular period must include all items for which receipts were issued during that period. The authors provide detailed protocols for the posting of an item to the BB, as well as the publishing of the BB (an optimistic one and a fallback). Each peer maintains a local copy of its view of the BB and agreement on the BB is only required during the publishing phase.

2.3.6 D-DEMOS

D-DEMOS [8] is a complete distributed, end-to-end (E2E) verifiable e-voting system, that eliminates the presence of potential single points of failure (besides setup). The authors define three main goals for their system. First, it has to be E2E verifiable and voters should be able to outsource auditing to third parties without revealing their voting choice. Second, it has to be fault-tolerant, meaning that attacks on both availability and correctness of the system are hard. Third, the voters do not need to trust the terminals they use to vote. Voters should be assured that their vote was recorded as cast without any information disclosure of how they voted to the entity that controls this potentially malicious device. The proposed system consist of four key components:

- the Election Authority (EA). This is the centralized election setup component of D-DEMOS, which is responsible for initializing all the other components of the system, before getting destroyed to preserve privacy. The EA encodes each election option, commits to it using a commitment scheme, and creates a votecode and a receipt

for each option. Finally, it creates one ballot per voter, consisting of two functionally equivalent parts. Each part contains contains a list of options, along with their corresponding votecodes and receipts.

- the Vote Collection (VC) subsystem. This is a distributed subsystem, responsible for collecting the votes from voters and accepting up to one vote code per voter. The EA initializes every VC node with the vote codes and the receipts of the voters' ballots, hiding the vote codes using a commitment scheme. In addition, each receipt is secret-shared across all VC peers using a VSS scheme [27], ensuring that a receipt can be retrieved and returned to the voter only when a strong majority of the VC nodes successfully participates in the voting protocol. Voters randomly select one part of their ballot at random and post their selected vote code to one of the VC nodes. If the receipt that is returned by the system matches the one on their ballot (corresponding to the selected vote code), the voters can be certain that their vote was recorded as cast and will be included in the election tally. The other part of the ballot is used for auditing, betraying a malicious EA with $\frac{1}{2}$ probability per audited ballot.

- the distributed Bulletin Board (BB) subsystem. Each BB node is initialized from the EA with vote codes and the associated option encoding in commited form. All BB nodes provide public access to their published information. After an election has ended, VC nodes run a consensus protocol and agree on a single set of (serial-no, vote-code) tuples. VC nodes then upload this set to every BB node and the BB node publishes the respective set once enough VC nodes have agreed on its content.

- a distributed subsystem consisting of a set of trustees, who are responsible for all further actions until result tabulation and publication. Secrets that may uncover information in the BB are shared among trustees, ensuring that malicious trustees under a certain threshold cannot disclose sensitive information. The authors propose a scheme through which the election tally is uncovered and published at each BB node only when a threshold number of trustees provide a share of the homomorphic total of the option-encoding of cast vote codes.

With regards to tolerance against Byzantine failures, the VC subsystem operates correctly as long as the number of faulty nodes is strictly less than $\frac{1}{7}3$ of total VC nodes. Accordingly, the BB susbystem is able to tolerate a number of faulty nodes that is strictly less than $\frac{1}{7}2$ of total BB nodes. As for the subsystem of trustees, the respective threshold of malicious trustees the system can tolerate is $f_t = N_t - h_t$, where N_t is the total number of trustees and h_t is the number of honest trustees (since h_t out-of N_t threshold secret sharing is applied).

The authors provide a model and a security analysis of the proposed e-voting system, implement a prototype, and measure its performance, in order to highlight its ability to facilitate large scale elections.

2.3.7 On the Security Properties of e-Voting Bulletin Boards

In this work [18], the authors introduce a complete framework for the formal security analysis of the functionality of e-voting BBs. The authors propose and define the properties of Confirmable Liveness, Persistence and Confirmable Persistence. Based on these properties and their formal framework, they analyze the Culnane and Schneider Bulletin Board

system (CS BB) and reveal two vulnerabilities of the CS BB system, one of which challenges the reasoning of liveness as it was stipulated in [10].

They proceed to enhance the CS BB system with a novel Publishing protocol, that overcomes the detected violation of Confirmable Liveness. The proposed BB Publishing Protocol, when combined with the CS BB Posting Protocol, comprises a BB system that achieves Confirmable Liveness and Persistence against a computationally bounded general Byzantine adversary, when the number of corrupted peers is strictly less than $\frac{1}{3}$ of the total peers. Confirmable Liveness holds in a partially synchronous model, while Persistence holds in an asynchronous model. Persistence of the system may also be Confirmable, if the audit board (AB) is distributed as a replicated service and reading of items is done via honest majority.

3. SYSTEM DESCRIPTION

In this section, we will be providing an in-depth description of the system we have implemented. We will begin by introducing the key components and constructs involved in our implementation, and we will proceed by describing their main properties and interactions within the protocols that we have integrated into Zeus. We should note that our work constitutes a practical implementation of the Culnane-Schneider BB protocols [9], as well as the novel Publishing Protocol introduced by Kiayias et al [18]. Therefore, we will not be repeating all aspects of the BB protocols proposed in the aforementioned works, but we will attempt to provide a comprehensive analysis of those parts that are essential for the illustration of our system. The BB protocols proposed by Culnane and Schneider as part of their distributed BB system are the Posting Protocol and the Publishing Protocol. Kiayias et al. adopt the exact same Posting Protocol as part of their proposed BB framework, but introduce their own variant of the Publishing Protocol. Since the protocol for posting items is identical in both BB systems, we will be simply referring to it as the "Posting Protocol". With regards to the Publishing Protocol, we will be referring to the one proposed by Culnane and Schneider as the "CS BB Publishing Protocol", and to the one proposed by Kiayias et al. as the "KKLSZ BB Publishing Protocol" (from the initials of the authors of [18]). In the following sections, we will be adapting the terminology to conform with an e-voting setting, however, the same procedures can apply for the posting and publishing of any item x, which may not necessarily be a vote. Finally, one deviation from the proposed solutions that we should mention before discussing the details of our systems, is the omission of the notion of periods from our practical implementation. Our work serves as a proof of concept on the applicability of the proposed BB protocols on top of e-voting systems, so we decided to focus on evaluating their main attributes, especially given that in each period, we would basically run almost the same iteration of requests and procedures as we already do in our implementation (that considers the whole procedure as one period).

3.1 Architecture

3.1.1 Preliminaries

Before we proceed with the description of our system, it is important to provide definitions regarding digital signatures schemes and threshold signature schemes. We write PPT for probabilistic polynomial-time, and $f(\kappa) = negl(\kappa)$ if function f is negligible in κ, where κ is the security parameter. We also denote $[N] := \{1, 2, ..., N\}$, for any $N \in \mathbb{N}$.

3.1.1.1 Digital Signature Schemes

A digital signature scheme $DS = (KGen, Sig, Vf)$ consists of the following three PPT algorithms:

1. $KGen$ is the key generation algorithm. It outputs a keypair $(vk, sk) \leftarrow KGen(1^\kappa)$, where vk is a public verification key and sk is a secret signing key.
2. Sig is the signing algorithm. On input m (a message to be signed) and sk (a signing key), it outputs a signature $\sigma \leftarrow Sig_{sk}(m)$.

3. Vf is the verification algorithm. On input vk, a message m, and a signature σ, Vf outputs a bit $b \leftarrow Vf_{vk}(m, \sigma)$.

The correctness of DS requires that for each $(vk, sk) \in KGen(1^\kappa)$ and a valid message m, it must hold that $Vf_{vk}(m, Sig_{sk}(m)) = 1$. The security of DS is formalized via the notion of existential unforgeability against chosen message attacks(EUFCMA, [15]).

3.1.1.2 Threshold Signature Schemes

Let $t_s < N$ be two positive integers and $P_1, ..., P_N$ a set of peers. A non-interactive Threshold Signature Scheme $TSS = (DistKeygen, ShareSig, ShareVerify, Combine, TVf)$ is a tuple of the following five efficient algorithms:

1. $DistKeygen(1^\kappa, t_s, N)$ is the distributed key generation algorithm. It generates a keypair (tsk_i, pk_i) for each Peer P_i and a public key pk, such that exactly $t_s + 1$ secret keys tsk_i are required to recover the secret key tsk corresponding to pk. The public output of $DistKeygen$ is pk together with a tuple of public verification keys $(pk_1, ..., pk_N)$.

2. $ShareSig_{tsk_i}(m)$ is the signing algorithm. It returns a signature share σ_i of the message m.

3. $ShareVerify(pk, pk_1, ..., pk_N, m, (i, \sigma_i))$ is the share verification algorithm. It outputs 1 iff σ_i is the valid ith signature share of message m.

4. $Combine(pk, pk_1, ..., pk_N, m, (i, \sigma_i)_{i \in S})$ is the share combining algorithm. Given a subset of $t_s + 1$ valid signature shares on message m, it outputs a full signature $\sigma \leftarrow TSign(tsk, m)$ on m.

5. $TVf_{pk}(m, \sigma)$ is the verification algorithm. It outputs 1 or 0, depending on whether σ is a valid (threshold) signature of m or not, respectively.

The correctness of TSS requires that for a vector $(tsk, pk, tsk_1, ..., tsk_N, pk_1, ..., pk_N)$ that has been generated by $DistKeygen(1^\kappa, t_s, N)$, if $S \subseteq [N]$ s.t $|S| = t_s + 1$, it holds that:

i. $\sigma_i = ShareSig_{tsk_i}(m)$, and

ii. If $\sigma = Combine(pk, pk_1, ..., pk_N, m, (i, \sigma_i)_{i \in S})$, then:
$ShareVerify(pk, pk_1, ..., pk_N, m, (i, \sigma_i)) = 1$ for $i \in S$ and $TVf_{pk}(m, \sigma) = 1$.

TSS is existentially (t_s, N)-unforgeable against chosen-message attacks ((t_s, N)-EUFCMA-secure) if every PPT adversary A has $negl(\kappa)$ advantage in performing a successful EUFCMA forgery for a message m^* even when the number of parties that A corrupts and the number of parties for which A made a signing query for m^* is no more than t_s.

TSS is (t_s, N)-robust, if A controlling t_s peers, cannot prevent honest peers from creating a valid signature. Robustness can only be achieved for $t_s < \frac{N}{2}$.

3.1.2 Entities

The following entities comprise our system:

- Users: We define users as clients of our system, that initiate cast vote requests in order to submit a vote of their choice, and that produce a receipt (TSS signature) signifying the proper posting of their vote, once they have acquired a sufficient amount of signature shares from IC peers.
- Votes: When casting a vote as part of an election in Zeus, users submit their encrypted ballots and a discrete log knowledge proof that they possess the randomness the ballot was encrypted with. For each vote that was properly cast, the vote is posted in a record of posted votes ($B_i \forall i \in [N_c]$, where $[N_c] := \{1, 2, ..., N_c\}$ and N_c is the total number of IC peers of the system), and users receive TSS signature shares from the IC peers that were able to properly post the vote in their local record during the Posting Protocol. By combining a threshold number of these shares, the users are able to generate a TSS signature of their vote, and identify it in the final record of posted votes, which is published by the AB after the voting period has finished.
- Subsystem of item collection (IC): The subsystem of item collection consists of N_c IC peers($P_1, ..., P_{N_c}$) that carry out all the required interactions among themselves and users, in order to properly post the users' submitted votes to their local record of posted votes, and that interact with the audit board (AB) component, to publish the recorded votes.
- Audit Board: The audit board (AB) component is responsible for publishing all the properly posted votes, after the Publishing Protocol has finished. The AB component can also consist of multiple AB peers, that together constitute the AB subsystem.
- Setup Authority: The Setup Authority (SA) generates the setup information and initializes all other entities of the system, along with their private inputs.

3.1.3 Setup

At the preparation period, the SA specifies a posting policy $P = (Accept, Select(\cdot))$ where $Accept$ is a check that the IC peers execute to verify that a user is allowed to submit a vote and $Select$ is responsible for resolving any conflict among clashing items for a user.

Upon specifying the posting policy P, the SA provides all entities with the description of an EUFCMA-secure digital signature scheme $DS = (KGen, Sig, Vf)$ and a (t_s, N_c)-EUFCMA-secure Threshold Signature Scheme $TSS = (DistKeygen, ShareSig, ShareVerify, Combine, TVf)$ (cf. 3.1.1). Then, each IC Peer P_i runs $KGen(1^\kappa)$ to obtain a signing key sk_i and a verification key vk_i, while all IC peers jointly execute $DistKeygen(1^\kappa, t_s, N_c)$ to produce secret keys $(tsk_1, ..., tsk_{N_c})$, implicitly defining tsk, and the corresponding public output $pk, (pk_1, ..., pk_{N_c})$.

Upon key generation, the IC peers broadcast ***pk*** $:= \{pk, (pk_1, ..., pk_{N_c}), (vk_1, ..., vk_{N_c})\}$ to all other entities. The public parameters $params$ include the description of DS, TSS, P, as well as ***pk***. For each IC peer P_i, we denote by B_i the local record of P_i including all votes v recorded as posted, and by D_i the database of received votes v, together with other peers' signature on them. In the beginning of the election, each P_i sets $B_i, D_i \leftarrow \emptyset$.

3.2 BB Protocols

3.2.1 Posting Protocol

Now that we have described the key components and properties of our system, we will be focusing on the interactions between them when a cast vote request is initiated by a user. The protocol for posting a vote v and issuing the corresponding acknowledgment will be called the Posting Protocol.

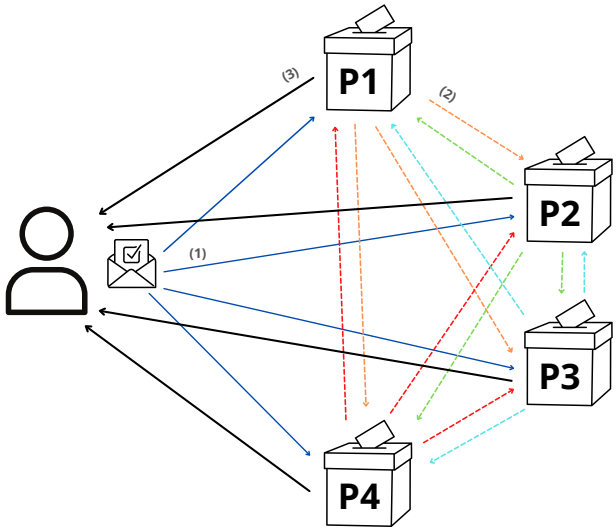


Figure 1: Distributed vote casting overview

Figure 1 provides a brief overview of the vote casting procedure in the distributed setting. In step (1), a user initiates a cast vote request by broadcasting (blue solid lines) its vote (as it was previously defined) to all IC peers. Upon receiving the vote from the user, each IC peer P_i signs the received vote with its own signing key and broadcasts the vote along with its signature to the rest of its peers(step (2), dotted lines). Finally, once an IC peer has collected a threshold number of signatures from its peers(including its own), it threshold signs the vote and returns its share of the threshold signature back to the user (step 3, black solid lines). Once a user has received a threshold number of such signature shares, she is able to combine them to obtain a threshold signature on v, which serves as her receipt of successful submission of her vote. The fault-tolerance threshold on the number of corrupted IC peers, t_c, that the Posting Protocol imposes is:

$$t_c < \frac{N_c}{3} \quad \text{and} \quad t_s + 1 = N_c - t_c$$

We will be discussing more on the fault-tolerance and security properties of the system in section 3.3.

Figure 2 presents the full set of messages and operations of vote casting, as dictated by the Posting Protocol:

- During the “Vote Submission” phase, the user broadcasts her vote v to every IC Peer $P_i, \forall i \in [N_c]$.
- Upon receiving the vote v, each IC peer begins the “Signature Exchange” phase. Specifically, each IC peer P_i signs vote v with its individual signing key sk_i and

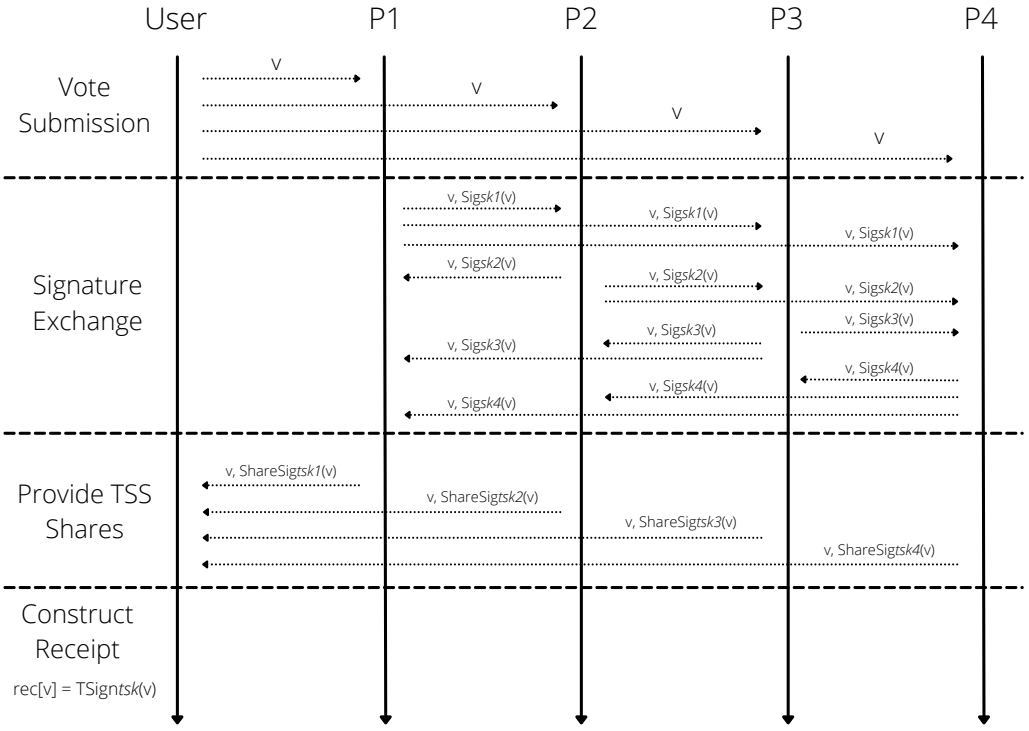


Figure 2: The Posting Protocol

broadcasts $(v, Sig_{sk_i}(v))$ to all of its peers $P_j, \forall j \in [N_c] \setminus \{i\}$. Each time an IC Peer P_i receives a valid message $(v, Sig_{sk_j}(v))$ from one of its peers, it appends it to its dataset of received signed votes D_i. Upon receiving $N_c - t_c$ valid signatures on v (including its own), P_i adds v to its local record B_i.

- Consequently, during the "Provide TSS Share" phase, each IC Peer P_i signs vote v with its individual threshold secret key tsk_i via the $ShareSig$ algorithm of the TSS and sends its TSS signature share $(v, ShareSig_{tsk_i}(v))$ to the user that submitted v.
- The user waits for $N_c - t_c$ such valid TSS signature shares from IC peers. Let S be the set of those peers from which the user can combine the signature share. Then the user passes to the "Construct Receipt" phase, where the receipt for vote v is:

$$rec[v] := TSign_{tsk}(v) \leftarrow Combine(pk, pk_1, ..., pk_{N_c}, v, (k, \sigma_k)_{k \in S})$$

3.2.2 The CS BB Publishing Protocol

Once the voting phase has ended, the IC peers engage in the CS BB Publishing Protocol, and the final record of posted items is published after its termination. The aim of the CS BB Publishing Protocol is for the peers to agree on the contents of the final record and to issue their signature share on it to a single trusted AB component, that can combine the shares and make the resulting TSS signature of the final record (along with its contents) publicly available.

The CS BB Publishing Protocol consists of two protocols, the Optimistic Protocol and the Fallback Protocol.

Figure 3 presents the Optimistic Protocol, which entails the following steps:

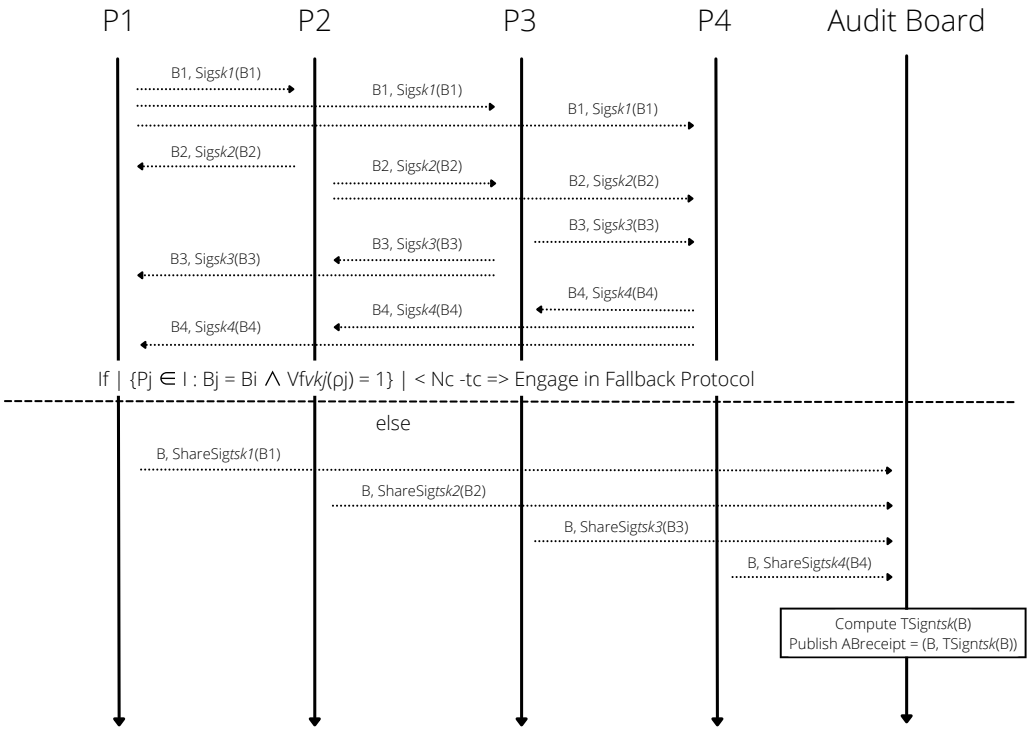


Figure 3: The Optimistic Protocol

1. Each IC Peer P_i signs its local record B_i and broadcasts $(B_i, Sig_{sk_i}(B_i))$ to each of its peers $P_j, j \in [N_c] \setminus \{i\}$.
    - Each P_i waits until it receives $N_c - t_c$ valid signatures $(B_j, \rho_j := Sig_{sk_j}(B_j))$ from different IC peers(including its own). Let I be the set of those peers.
    - If $|\{P_j \in I : B_j = B_i \wedge Vf_{vk_j}(\rho_j) = 1) < N_c - t_c\}|$, then IC peer P_i engages in the Fallback Protocol, by broadcasting a message $(D_i, Sig_{sk_i}(D_i))$ to all of its peers.
2. Each P_i generates its TSS signature share of its local record B_i and sends $(B_i, ShareSig_{tsk_i}(B_i))$ to the AB.
3. The AB waits for messages $(B_j, \sigma_j = ShareSig_{tsk_j}(B_j))$ from at least $N_c - t_c \geq t_s + 1$ IC peers P_j.
    - Let S be a set of those $\geq N_c - t_c$ peers that agree on the contents of the final record of posted votes, and let B be the board of agreed-on contents. Then, compute $TSign_{tsk}(B) \leftarrow Combine(pk, pk_1, ..., pk_{N_c}, B, (j, \sigma_j)_{j \in S})$.
    - Publish $ABreceipt[B] := (B, TSign_{tsk}(B))$.

The Fallback Protocol, which may be initiated in step (1) of the Optimistic Protocol above, is presented in figure 4 and entails the following steps:

1. As previously stated, the Fallback Protocol starts with every peer P_i that engages in it broadcasting $(D_i, Sig_{sk_i}(D_i))$ to each of its peers $P_j, j \in [N_c] \setminus \{i\}$.
2. Subsequently, each P_i updates its local database of received votes D_i with the signatures it is missing, and then updates its local record B_i. In particular, if $Vf_{vk_j}(D_j, \sigma_j) =$

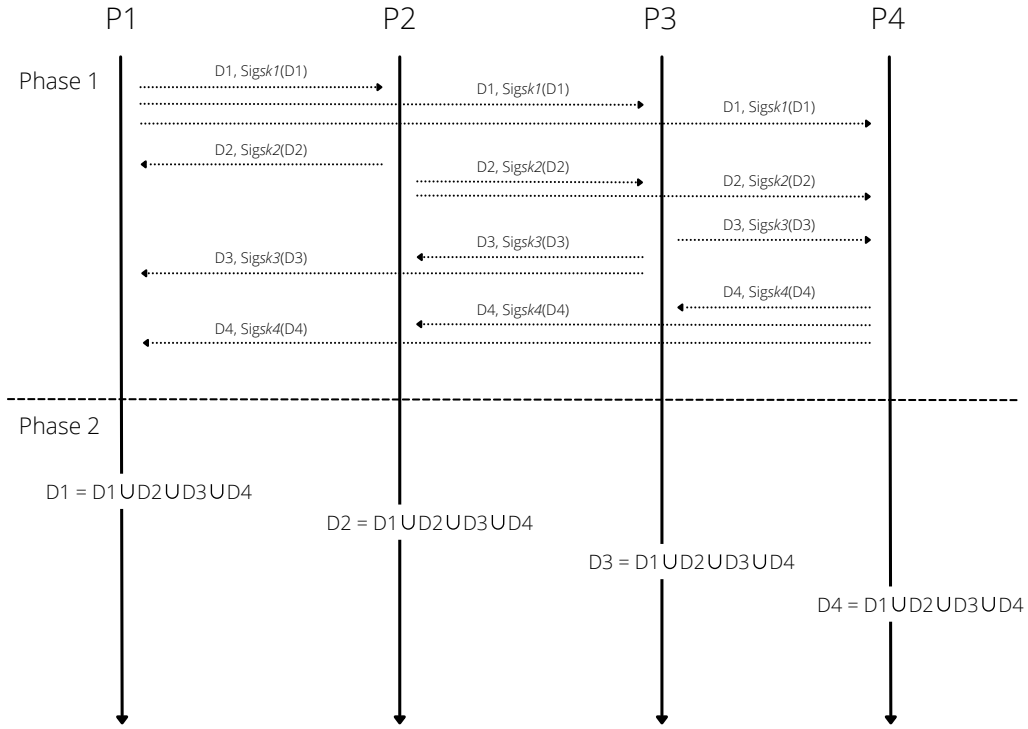


Figure 4: The Fallback Protocol

1, then P_i will insert all new signatures in D_j to its local database D_i, thus P_i sets $D_i \leftarrow D_i \cup D_j$. For any vote v for which P_i now has $N_c - t_c$ valid signatures on, P_i adds v to its local record B_i.

3. P_i broadcasts a message $(B_i, Sig_{sk_i}(B_i))$ with its updated local record B_i, indicating that it re-engages in step (1) of the Optimistic Protocol.

The Optimistic Protocol always terminates succesfully in the first run if all peers are honest. If it does not, but all users posted their votes honestly, then the Fallback Protocol needs to run only once. Otherwise, it needs to be executed up to $N_c - t_c + 1$ times before reaching a final consensus on the contents of the final record of posted items. When consensus is reached and the AB has gathered all the corresponding TSS shares from IC Peers, the AB calculates the TSS signature of the agreed record from the received shares, and publishes the final record along with the TSS signature.

3.2.3 The KKLSZ BB Publishing Protocol

In this section, we will be providing a concise description of the KKLSZ BB Publishing Protocol. An in-depth analysis of all the steps and procedures associated with the protocol can be found in [18] (cf. Section 6). We will be discussing the motivation behind this upgraded version of the CS BB Publishing Protocol in section 3.3.

The KKLSZ BB Publishing protocol consists of the following phases:

- The **Initialization** phase: Each IC Peer P_i initializes the following vectors:

  i. Its direct view of local records, $View_i := \langle \tilde{B}_{i,1}, ..., \tilde{B}_{i,N_c} \rangle$, by setting $\tilde{B}_{i,j} \leftarrow \perp$ for $j \neq i$ and $\tilde{B}_{i,i} \leftarrow B_i$.

ii. For every $j \in [N_c] \setminus \{i\}$, its indirect view of local records as provided by P_j, denoted by $View_{i,j} := \langle \tilde{B}^i_{j,1}, ..., \tilde{B}^i_{j,N_c} \rangle$, by setting $View_{i,j} \leftarrow \langle \perp, ..., \perp \rangle$.

iii. A variabe vector $\langle b_{i,1}, ..., b_{i,N_c} \rangle$, where $b_{i,j}$ is a value in $\{?, 0, 1\}$ and expresses the opinion of P_i on the validity of P_j's behavior. Initially all $b_{i,j}$ for $j \neq i$ are set to the pending value '?', and only $b_{i,i}$ is fixed to 1. Whenever P_i fixes $b_{i,j}$ to some value $1/0$ for all $j \in N_c$, it engages in the Consensus phase of the protocol.

- The **Collection** phase: During this phase, all IC peers exchange their local records as they were created during the Posting Protocol (direct views), as well as their opinion on the records they receive for other peers by rebroadcasting them (indirect views). By the end of the Collection phase, all (honest) peers will have updated their local views, so that they include (when combined) all the honestly posted votes.

- The **Consensus** phase: When P_i fixes its opinion $b_{i,j}$ to some value $1/0$ for P_j for every $j \in [N_c]$, it engages in the partially synchronous Binary Consensus (BC) protocol BC [13], for the question "Is my view of P_j's record set to a non-$\perp$ value?". Each peer P_i engages in BC with input $b_{i,j} = 1$ if $\tilde{B}_{i,j}$ is non-$\perp$ and $b_{i,j} = 0$ if $\tilde{B}_{i,j}$ is $\perp$. During the Collection phase, all $N_c - t_c \geq t_c$ honest peers have received each other's records, so they engage in BC for an honest peer P_j on input 1. Thus, by the validity property of BC, the honest peers will agree on 1 for P_j. When all N_c BC executions are completed, the (honest) peers will have discarded their views about any peer for the record of which they decided on 0.

- The **Finalization** phase: During this phase, each IC peer P_i decides on the final votes that will be included in its local record. In particular, for every vote $v \in \bigcup_{j:B_{i,j} \neq \perp} \tilde{B}_{i,j}$, P_i defines the set $N_i(x)$ that denotes the number of IC peers that, according to P_i's view, have included v in their records. Then, P_i updates its original record B_i as follows:

  i. If $v \notin B_i$, but $N_i(v) \geq t_c + 1$, then it adds v to its final local record B_i.

  ii. If $v \in B_i$, but $N_i(v) < t_c + 1$, then it removes v from its final local record B_i.

  We should note that every honestly posted vote for which a receipt was generated during the Posting Protocol is stored in the records of at least $N_c - 2t_c \geq t_c + 1$ honest peers. Thus, $N_i(v) < t_c + 1$ implies that either v was maliciously injected or that a receipt for v was never generated. That implies that in all honest peer's final records all honestly posted votes will be included and only maliciously injected votes will be removed. Yet, there may be maliciously posted votes (initiated by a malicious user and signed by t_c corrupted peers and $t_c + 1$ honest peers) for which a receipt was generated during the Posting Protocol and that will be included in an honest peer's final record. However, this does not violate the security requirements of the system as they will be defined in section 3.3. We will also encounter such an example in our experimental security evaluation, in section 5.2.

- The **Publication** phase: During this phase, every IC peer threshold signs its final local record B_i, as it has been updated in the Finalization phase, by threshold signing each vote in the final record individually. Formally, this is denoted as $ShareSig(tsk_i, B_i) := \bigcup_{v \in B_i} ShareSig(tsk_i, v)$. Then, it sends the message $(B_i, ShareSig(tsk_i, B_i))$ to the AB. The AB receives and records the threshold signature share for posted votes. For every posted item v for which the AB receives $N_c - t_c$ distinct successfully verified signature shares $(j, \sigma_j)_{j \in S}$, where S is a subset of $\geq N_c - t_c$ peers, it adds v to the record B (initialized as empty) and computes

the TSS signature on v as $TSign(tsk, v) \leftarrow Combine(pk, pk_1, ..., pk_{N_c}, v, (j, \sigma_j)_{j \in S})$. Finally, the AB fixes the signed record $TSign(tsk, B) := \bigcup_{v \in B} TSign(tsk, v)$ and publishes the record $ABreceipt[B] := (B, TSign(tsk, B))$, which includes the final posted votes of the election, along with the threshold signature of each vote.

3.3 Security Properties

In this section, we will seek to elaborate more on the fault-tolerance and security guarantees of our system. In particular, we will first introduce the key security properties that define a secure BB system, as stipulated in [9] and formalized in the security framework proposed by [18]. We will be referring to the BB system proposed by Culnane and Schneider as the "CS BB system", and to the BB system proposed by Kiayias et al. as the "KKLSZ BB system". Upon presenting the properties of the formal security framework, we will be discussing the key conditions that each system must satisfy in order to guarantee the proposed security properties.

3.3.1 Definitions of Security Properties

In [9], Culnane and Schneider propose four key properties that a secure BB system must satisfy:

(bb.1). Only items that have been posted may appear on the AB. This property expresses safety against illegitimate data injection.

(bb.2). Any item that has a valid receipt issued must appear on the AB.

(bb.3). No clashing items must both appear on the AB.

(bb.4). Once published, no items can be removed from the AB.

In [18], Kiayias et al. integrate the above four properties into a formal security framework, which consist of the following three properties:

1. **Confirmable Liveness**: In the e-voting setting, we require that the users will eventually get a receipt when engaging in the Posting Protocol. The Confirmable Liveness property, also denoted as Confirmable θ-Liveness, dictates that any honest user that submits an item x, will obtain a valid receipt for x within time θ, and x will be published on the AB. Thus, Confirmable Liveness encompasses (bb.2) of the aforementioned properties. We should note that Confirmable Liveness and property (bb.3) cannot be satisfied concurrently, if we assume that honest users may submit post request for clashing items (e.g., in voting systems where voters may vote multiple times and only their last vote should count). To resolve this conflict, we make the following relaxation: we do not require that (bb.3) holds, thus allowing every successfully posted item to be published. Then, we specify the subset of valid items for each user via the $Select(\cdot)$ function over the set of published items. The final description of $Select(\cdot)$ depends on the respective election setting. Thus, in the special case where honest users should be restricted from submitting clashing items, we can match (bb.3) by fixing $Select(\cdot)$ to be the identity function.

2. **Persistence:** In order to encompass (bb.1) and (bb.4), we introduce the notion of Persistence, where conflict resolution is achieved by applying $Select(\cdot)$ on the AB view. In [24], the definition of persistence informally states that once an honest peer reports an item x as posted, then all honest peers will either agree on the AB position that x should be published, or not report x, which is enough to encompass the unremovability property (bb.4). In order to also capture (bb.1), we require that in a setting where t_c out-of N_c IC peers are corrupted, an item x posted by a user will not be reported as valid, if at least $t_c + 1$ honest IC peers did not record x at some execution of the Posting Protocol.

3. **Confirmable Persistence:** Confirmabe Persistence extends the notion of Persistence and is introduced to account for a AB susbsystem that is fully controlled by an adversary. The Confirmable Persistence property dictates that any malicious AB behavior will be detected via the $VerifyPub$ algorithm (cf. [18], section 6).

We have now provided a comprehensible description of the key properties that define a secure BB system based on Kiayias et al's formal framework. For a more thorough analysis of these properties, along with the corresponding proofs, we refer the reader to [18]. In the following sections, we will be specifying under which conditions the CS BB system and the KKLSZ system are able to guarantee Confirmable Liveness, Persistence and Confirmable Persistence.

3.3.2 Security Properties of the CS BB system

The CS BB system comprises of the setup authority SA, the users, the IC peers $P_1, ..., P_{N_c}$ and a single trusted AB component(we denote as N_w the number of AB components that constitute the AB susbsystem, thus for CS BB $N_w = 1$). The fault-tolerance threshold on the number of corrupted IC peers t_c that the CS BB system requires is:

$$t_c < \frac{N_c}{3} \quad \text{and} \quad t_s + 1 = N_c - t_c \tag{1}$$

Given the above, we will be assessing under which conditions the CS BB system achieves Confirmable Liveness, Persistence and Confirmable Persistence.

3.3.2.1 Confirmable Liveness

In [10], the authors argue that the liveness of the CS BB system can be achieved if one of the following conditions hold:

1. All the peers follow the protocol honestly and are online.
2. A threshold of $t_c < \frac{N_c}{3}$ is malicious, but all users are honest.
3. Not all users are honest and malicious peers may choose any database in their capability, but do not change their database once it has been fixed, and do not send different databases to different peers.

The more general condition (3) above relies on a presumed "fear of detection" when peers send different databases to different peers, but the authors do not address this condition

rigorously. In [18], Kiayias et al. present an attack against the Confirmable Liveness of the CS BB system, where the liveness adversary (both malicious users and IC peers) is able to intelligently split honest peers into two groups and only behave consistently towards the one group, while providing no information to the other group. This malicious behavior cannot be proved by the peers in the first group, because they cannot determine whether the peers on the other group are honest and did not receive anything, or malicious and falsely claim a denial of service. This attack will be described in more detail, and demonstrated experimentally in section 5.2.

Therefore, Confirmable Liveness is not satisfied in the CS BB system. In order for the CS BB system to guarantee Confirmable Liveness, it would need to be enhanced with an explicit detection mechanism against any deviations from the IC consensus protocol specifications.

3.3.2.2 Persistence

In the CS BB system, if the centralized AB component is honest, it AB will neither accept inconsistently signed data, nor remove any existing published items, as ensured by the Posting and Publishing Protocols. Therefore, the CS BB system satisfies properties (bb.1) and (bb.4), which are captured by the aforementioned definition of Persistence.

As a result, Persistence is satisfied in the CS BB system.

3.3.2.3 Confirmable Persistence

In the case of the CS BB system, where the AB is a single component, Confirmable Persistence extends the notion of Persistence to account for a malicious AB. All data published by the AB must be signed by a TSS signature, thus, any attack that includes the injection of illegitimate items or the removal of published items will be detected as long as the t_s out-of N_c TSS fault tolerance threshold is preserved, since the malicious AB cannot forge signatures on inconsistent records without the consent of at least some honest peers.

This leads to a conflict. TSS fault-tolerance must satisfy the conditions of equation (1), thus $t_s + 1 > \frac{2}{3}N_c$, yet the robustness requirement $t_s < \frac{N_c}{2}$ as defined in [14] must also hold in order to preserve liveness. These two requirements are contradictory.

Kiayias et al.(cf. [18]) prove that in the general case, the CS BB system satisfies both robustness ($t_s < \frac{N_c}{2}$) and Confirmable Persistence for $t_c < \frac{N_c}{4}$. If we discard the robustness bound, the CS BB system achieves Confirmable Persistence for $t_c < \frac{(t_s+1)}{2}$.

If we wish to achieve both robustness (which is necessary for liveness), and also preserve the original CS BB bound of $t_s = \frac{2}{3}N_c$, care must be taking in the selection of the TSS scheme to be used. In [32], Shoup proposes a definition of a $(k, t_s, N_c) - TSS$ scheme, where t_s is the number of malicious peers and $k \geq t_s + 1$ the number of signature shares required to provide a valid TSS signature. Therefore, we would still get robustness by having $t_s = \lceil \frac{N_c}{3} \rceil - 1 < \frac{N_c}{2}$, and satisfy the conditions of (1) by setting $k = N_c - t_c = \lfloor \frac{2}{3}N_c \rfloor - 1$. Shoup's construction uses a trusted dealer, but this can be avoided by employing the trivial TSS scheme proposed in section A.3 of [18] (or any other secure $(k, t_s, N_c) - TSS$).

Thus, the CS BB system can concurrently satisfy the bounds of (1) and achieve Confirmable Persistence with the proper TSS scheme.

3.3.3 Security Properties of the KKLSZ system

3.3.3.1 Introduction

In [18], the authors propose the KKLSZ BB system that satisfies the properties of Confirmable Liveness, Persistence and Confirmable Persistence under specific models. We will be providing a short description of these models and then we will be succinctly mentioning which of these properties are guaranteed under which model in the KKLSZ BB system. For a more extensive analysis of the models, along with the corresponding proofs of each security property, we refer the reader to the original paper [18].

The security framework proposed by Kiayias et al. assumes the existence of a global clock variable $Clock \in \mathbb{N}$ and every entity X is equipped with an internal variable $Clock[X] \in \mathbb{N}$. The event $Init(X)$ initializes X by synchronizing its local clock with the global clock, and the event $Inc(Clock[X])$ causes some $Clock[X]$ to advance by one time unit. The threat model of the system is parameterized by an upper bound δ on the delay of message delivery and an upper bound Δ on the synchronization loss of the internal clocks of the entities in relation to the global clock. Based on these definitions, the authors specify the following two models:

- The asynchronous model: In the asynchronous model, we set $\Delta = \infty$. Also, $\delta = \infty$ may be used to denote that an adversary may drop messages.
- The partially-synchronous model: In the partially-synchronous model, $\delta, \Delta \in [0, \infty)$ and δ, Δ are unknown to the system's entities.

Based on the aforementioned models, we will proceed with the assessment of the three security properties of the formal framework for the KKLSZ BB system.

3.3.3.2 Confirmable Liveness

The KKLSZ BB system achieves Confirmable Liveness in a partially synchronous model, against a computationally bounded Byzantine adversary and assuming a threshold of $t_c < \frac{N_c}{3}$ corrupted IC peers. For a formal proof, we refer the reader to [18] (Appendix C.2).

3.3.3.3 Persistence

The Persistence of the KKLSZ BB system holds in the asynchronous model against a computationally bounded Byzantine adversary for a threshold number of up to t_c corrupted peers where $t_c < \frac{N_c}{3}$. For a formal proof, we refer the reader to [18] (Appendix C.1).

3.3.3.4 Confirmable Persistence

In order for the KKLSZ system to also achieve Confirmable Persistence in the asynchronous model, we would need to replace a single AB with a subsystem of N_w AB peers. In particular, the IC peers would then broadcast their local records to all AB peers, and each AB peer would behave exactly as the centralized AB when publishing its final record (cf. section 3.2.3). Reading of the AB data would now need to be performed via honest majority of matching data among the AB peers, and the respective threshold of corrupted AB

peers would be $t_w < \frac{N_w}{2}$. As a result, any alteration of previously published data by some AB peer would be detected by an auditor, thus achieving Confirmable Persistence.

4. IMPLEMENTATION

4.1 Introduction

In this section, we will be discussing the details of our implementation of the protocols we have illustrated in section 3. Specifically, we will be describing:

- The exact parts of the protocols we have implemented and any assumptions that were made during the development of the respective software modules.
- Any third-party libraries that were used and are essential to the evaluation of our implementation.
- Our integration of the implementation of the aforementioned protocols on top of Zeus [1].
- Instructions on how to execute our integrated solution and how to interpret the respective outputs.

The codebase of our implementation can be found in our Github repository [2].

4.2 Preliminaries and Assumptions

In this part we will be discussing certain assumptions that were made during the development of our implementation, focusing on any deviations from the description of the BB systems in Section 3.

With regards to the model of our distributed system, we do not strictly follow neither the asynchronous model, nor the partially synchronous model as defined in Section 3.3.3.1. Specifically, we do not formally implement the notion of global and internal locks, though we set some practical bounds on message delivery and the completion of each protocol phase. The model under which our implementation functions assumes the following:

- With regards to the execution of the phases of each protocol from Sections 3.2.1, 3.2.2, and 3.2.3, our system works in a synchronous fashion, meaning that if a specific phase needs to be completed before proceeding to the next, our system respects this order and is able to re-engage in specific phases if a timeout without an acceptable response has occurred.
- With regards to the messages that are exchanged among all entities in a specific phase of the aforementioned protocols, our system operates in an asynchronous fashion, with some practical bounds on message delivery. As long as the delivery of the messages respects the specified bounds (and the bound on the completion of the specific phase), we do not require messages to be processed or sent in a specific order, allowing the concurrent handling of multiple messages by each entity.

[1] `https://github.com/grnet/zeus`
[2] `https://github.com/gmetaxo/dzeus`

Regarding the KKLSZ BB Publishing Protocol, we have omitted the implementation of the Consensus phase, and proceed to the Finalization phase even with the presence of malicious peers' records in the direct views of the peers. In our experimental evaluation, we did not encounter a scenario where a maliciously injected item for which a receipt was never generated was included in the final published record. We leave it as future work to investigate the consequences of skipping that part. Next iterations of our implementation could incorporate the Consensus phase, along with the corresponding partially synchronous Binary Consensus (BC) protocol BC [13], following a more thorough study of its necessity in a practical setting.

4.3 Implementation Details

Here, we will be presenting key details of our implementation, with regards to the modules we have developed, any third-party libraries essential to our implementation, and the integration of our implementation on top of Zeus.

The *src* directory of our Github repository [3] contains the source code of our implementation, which is written in *Python 3.8.10*. The two most significant modules are:

- *server.py*: This module encompasses our implementation of the IC peers, along with all the corresponding procedures that are dictated as part of the BB protocols in Section 3.
- *client.py*: This module includes our implementation of the users of the BB system, providing all the required functions for properly submitting an item and receiving the respective receipt. We have also included all the functions required for a Setup Authority to initiate interactions with the IC peers, for operations such as the generation and sharing of private and public inputs and the coordination of the phases of the BB protocols.

The two aforementioned modules encapsulate the main functionalities of the BB protocols described in Section 3. In order to evaluate our implementation, we decided to integrate our modules into the core script provided by Zeus [4]. The respective source code of our integration can also be found inside the *src* directory of our Github repository, specifically in our *zeus_core.py* [5] module. Thus, *zeus_core.py* contains all the functions required by Zeus to initiate and successfully run an election, along with the proper functions from *server.py* and *client.py* so that the election incorporates the distributed components and BB protocols described in Section 3. In order to test our implementation, we utilize the `--generate` option of *zeus_core.py*, which allows us to automatically initiate random votes, voters and candidates, and run a full election scenario preserving Zeus' e-voting properties and procedures(cf. Section 2.2.2), enhanced by our distributed BB implementation.

As part of our integration, we have assigned the role of the Setup Authority to Zeus. Therefore, the *ZeusCoreElection* server (initialized when running an election via *zeus_core.py*) also operates as a trusted dealer for the generation and distribution of private and public inputs of all entitities of our distributed setup (e.g. public TSS verification keys pk_i and secret TSS signing keys tsk_i, as defined in Section 3.1.1). The *ZeusCoreElection* server

[3] `https://github.com/gmetaxo/dzeus`
[4] `https://github.com/grnet/zeus/blob/master/zeus/core.py`
[5] `https://github.com/gmetaxo/dzeus/blob/main/src/zeus_core.py`

also coordinates the phases of the election, such as the end of the voting period and the beginning of the publishing phase, as well as the execution of the steps of the Publishing Protocols within acceptable time boundaries. After all the stages of the BB protocols and Zeus' election workflow have concluded, the final poll document, along with the final record of published votes are also published by the *ZeusCoreElection* server. Thus, the *ZeusCoreElection* server also serves as the centralized AB component of the integrated BB systems. Zeus' final poll document contains the decrypted tally of the election in a portable binary format that can be fed to any other system that actually calculates the election results. With regards to votes, the poll document contains the exact same votes as the ones in the final published BB record of the AB. In future iterations of our integrations we could incorporate the final published record as part of the poll document, in order to release only one document that incorporates all the required information for both the production of results and the verification of the validity of the data on the AB.

Finally, it is worth mentioning two of the third-party libraries our implementation relies on in order to run properly:

- *asyncio:* The *asyncio* [6] library provides APIs that allow the creation and management of event loops, which provide asynchronous APIs for networking and allow the concurrent handling of network IO events. Our implementation leverages these APIs for the concurrent handling of all messages that are exchanged as part of our proposed solutions.
- *bls-lib:* The *bls-lib* [7] library provides a simple Python implementation of threshold BLS signatures [6]. We utilize *bls-lib* to generate TSS public and secret keys, to sign items and provide TSS signature shares, and to combine TSS signature shares into final TSS signatures on items. The library also allows us to verify TSS signature shares and TSS signatures, which is also a critical part of our implementation. We should note that we also use the threshold public and secret keys(pk_i, tsk_i, cf. Section 3.1.1) of each peer as the respective verification and signing keys of the DS scheme(vk_i, sk_i, cf. Section 3.1.1). While these choices may be acceptable as part of a proof of concept, it is important to either evaluate them formally before including them in production code, or to implement one of the TSS schemes discussed in Section 3.3.2.3 instead of using *bls-lib*. Finally, we should mention that we do not directly install *bls-lib*. Instead, we clone a forked repository [8] of the original implementation that includes a minor modification and build *bls-lib* manually.

4.4 Execution

In this part, we will be describing how to setup the environment for the execution of our implementation, how to run our scripts with various inputs and how to interpret the respective outputs. Any relevant screenshots will be included in Appendix A.

The following commands are used to setup our virtual environment and install all the required packages:

```
# Clone the repository
$ git clone git@github.com:gmetaxo/dzeus.git
```

[6] https://docs.python.org/3/library/asyncio.html
[7] https://pypi.org/project/bls-lib
[8] https://github.com/StefanosChaliasos/bls

```
# Go to the source directory
$ cd dzeus/src
# Create and activate the virtual environment
$ python3 -m venv env
$ source env/bin/activate
# Install required dependencies
$ pip install pycryptodome petlib
$ git clone https://github.com/StefanosChaliasos/bls
$ cd bls && pip install -e . && cd ..
```

In order to start the IC peers, we need to run *server.py* and provide it with a configuration file. The server configuration file is a JSON file that contains the following information:

- The BB *protocol* to be used. The value *bbcs* instructs the servers to run according to the protocols of the CS BB system, while the *bbcs_advanced* value represents the KKLSZ BB system.
- A list of *hosts* that constitute the IC peers of the respective setup. Each host is identified by its *ip*, its *port* (where it listens for requests to establish new connections via sockets), and a unique *id*.

A screenshot of such a configuration file can be found in figure 8. All server configuration files are located under the *src/data* directory. Thus, if we wish to start the IC peers of a setup of 4 peers, we need to type:

```
# server.py takes two positional arguments,
ip and port.
# We provide the configuration file of the
setup with the --config flag.
# The --debug flag is used to generate debug logs.
# If we provide the --interactive flag, debug
logs are printed in the
# respective shell window, otherwise they
are saved in a file(1 for each peer).
# s4.json uses the CS BB system.
To use the KKLSZ system, switch it to s4-advanced.json
# Each command needs to run in a different shell.

$ python server.py 127.0.0.1 65432 --config data/s4.json --debug --
    ↪ interactive
$ python server.py 127.0.0.1 65433 --config data/s4.json --debug --
    ↪ interactive
$ python server.py 127.0.0.1 65434 --config data/s4.json --debug --
    ↪ interactive
$ python server.py 127.0.0.1 65435 --config data/s4.json --debug --
    ↪ interactive
```

Each IC peer P_i is associated with a JSON file that contains useful information about the setup (e.g. peers' public keys), the local record B_i of the peer, and its local database of signatures D_i. A screenshot taken after an IC peer has received a number of requests is presented in figure 9.

As previously mentioned, *zeus_core.py* allows us to use the `--generate` flag in order to automatically run a Zeus election that leverages our implementation of the BB protocols. The following command can be used to simulate an election with the 4 IC peers we have initiated above (which implement the CS BB system), 5 voters, and 3 candidates, with each voter casting a single vote:

```
$ python zeus_core.py data/s4.json --generate out.json --voters 5 --votes
   ↪ 5 --candidates 3
```

The respective logs generated by *zeus_core.py* when simulating the election are presented in figure 10. The output of the previous command also includes Zeus' final poll document, titled *out.json*, and the final published record of the AB, titled *final_ab_bbcs.json*. Figure 11a and figure 11b list part of the contents of each file after the successful execution of the previous commands, signifying the proper publishing and inclusion in the tally of the vote with fingerprint *"0e8c6d70e873c3651b5220e61eac0f98f8f44f95a7b6bc3170d6c7c4f9e4e152"*. This fingerprint is a Zeus-generated cryptographically signed receipt for votes that were submitted successfully and will be included in the final tally, which is different than the TSS signature of the vote. Such receipts are issued to the voter only for votes for which the she has gathered a threshold number of partial signature shares and is able to generate a TSS signature of the vote. In figure 11b we can also see that the AB's record includes the receipt (TSS signature) of the final published AB record.

5. EVALUATION

5.1 Perfomance and Scalability Evaluation

In this section, we provide an initial evaluation of the performance and scalability properties of our implementation, aiming to highlight and analyze both desired attributes and possible points of improvement for future revisions of the current system. As part of this evaluation, we will be focusing on the performance and scalability features of only the Posting Protocol(cf. Section 3.2.1). There are two main reasons behind this choice. Firstly, the messages exchanged as part of the Posting Protocol are more time-sensitive, since honest voters wish to receive receipts for their posted items within an acceptable timeframe. Both Publishing Protocols, that is the "CS BB Publishing Protocol" (cf. Section 3.2.2) and the "KKLSZ BB Publishing Protocol" (cf. Section 3.2.3), are initiated after the voting phase has terminated, meaning that their performance aspects are less critical to the applicability of the proposed solutions. Secondly, the Posting protocol is the phase of the voting procedure that encompasses the majority of all messages exchanged. We leave a review of the performance and scalability properties of both Publishing Protocols as future work. Given that both protocols we have implemented share the exact same algorithm for posting items to the Bulletin Board, the following results are pertinent to both of the implemented distributed BB solutions.

Experimental Setup: We conducted our experiments in a local setting, meaning that the client and each server(IC peer) instance run in the same machine as separate processes. In particular, we used a Lenovo X1 Thinkpad with 12 Intel(R) Core(TM) i7-10710U CPUs 1.10GHz, and 16GB RAM. Each server is launched in a separate process and listens to a specified port for incoming connections from clients and other peers, and is able to create sockets for the exchange of messages between them. We also implemented a multi-threaded voting client to simulate concurrent connections. The client starts the requested number of threads, each of which performs a vote casting request, as defined in the Posting Protocol. For the coordination of our experiments, and the automated initiation of random votes, we used the `--generate` option of the core script provided by Zeus [1]. In all our experiments, a random vote includes the selection of one candidate among 100 randomly created candidates, though our implementation can easily facilitate a wide variety of voting schemes.

5.1.1 Latency

We define latency as the average time(in seconds) between the moment a client(voter) broadcasts a cast vote request to all IC peers, and the moment it receives the respective responses corresponding to that request. These responses are actually a set of partial threshold signatures on the vote, allowing for the creation of a threshold signature on the vote which serves as the receipt of the posted item. In our experiments, we consider at least a threshold number of peers are honest, thus all cast vote requests received the required amount of partial threshold signatures on the respective vote.

Before we proceed with the analysis of our findings, it is important to touch upon certain aspects of the vote casting procedure, with regards to the number of messages exchanged on each request. Whenever a client wishes to post her vote, the Posting Protocol (cf.

[1] `https://github.com/grnet/zeus/blob/master/zeus/core.py`

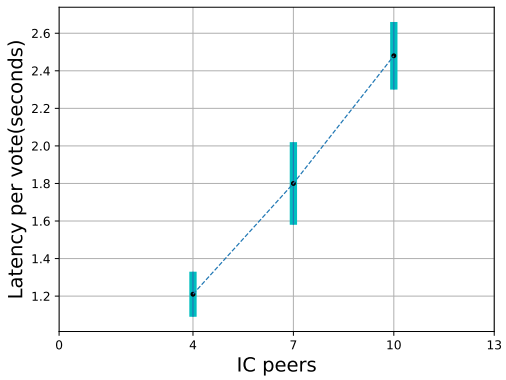
Section 3.2.1) dictates the following number of messages to be exchanged, where N_c is the number of total IC peers and $t_c = \lceil \frac{N_c}{3} \rceil - 1$ is the maximum number of allowed corrupted IC peers of the respective setup:

- The client broadcasts its vote to all N_c IC peers, and waits for a threshold number of $N_c - t_c$ partial signatures on the vote as responses to its request.
- Upon receiving a cast vote request, all N_c IC peers sign the vote with their own signing key and broadcast it to their $N_c - 1$ peers (without awaiting for a response).
- Once an IC peer has received a threshold number of signatures via the aforementioned broadcasting process, it is able to generate the aforementioned response to the client, which includes its share of the threshold signature for the respective vote.

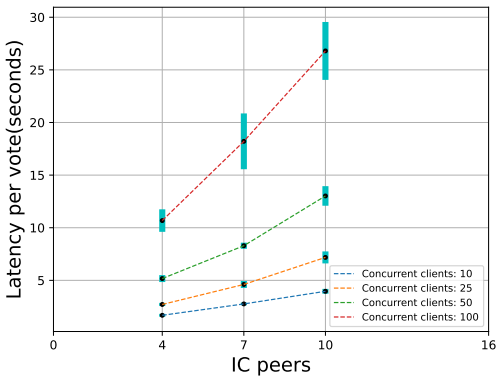
Given the above, the total number of messages(requests and responses) that are required as part of the cast vote request is:

$$\begin{aligned} TotalMessages(N_c) &= [N_c + (N_c - t_c)] + N_c * (N_c - 1) \\ &= N_c + N_c - \lceil \frac{N_c}{3} \rceil + 1 + N_c^2 - N_c \\ &= N_c^2 + N_c - \lceil \frac{N_c}{3} \rceil + 1 \end{aligned}$$

Assuming that the average latency per message sent as part of the vote casting procedure is independent of the number of IC peers (which is true in our case, since servers can concurrently handle a large number of requests by taking advantage of asyncio's concurrency model), the average latency of casting a vote should follow the quadratic growth of the equation above as new IC peers are added in the setup.



(a) Vote casting latency without concurrent clients follows a sublinear growth



(b) Vote casting latency with concurrent clients does not grow linearly with the addition of more clients

Figure 5: Vote Casting Latency

For our first experiment, a client initiates a single cast vote request. We run this process 10 times for different setups of IC peers and report the respective average latency per vote

and standard deviations for each setup. Figure 5a displays the corresponding results. In particular, for 4, 7, and 10 IC peers, the average latency per vote is 1.21, 1.8 and 2.48 seconds respectively, with standard deviations of 0.12, 0.22, and 0.18 seconds. Based on our previous observations, we would expect a quadratic growth in the latency per vote as we raise the number of IC peers, yet, it is evident that our implementation follows a sub-linear growth. That is achieved through the concurrent handling of individual messages that are sent as part of the cast vote request, a feature enabled by asyncio's concurrency model.

Apart from the concurrent handling of individual messages, a practical implementation of the Posting Protocol also needs to consider the case of concurrent client cast vote requests. In order to achieve that, we utilize our implementation of a multi-threaded voting client, which starts a specified number of threads, one for each cast vote request we wish to concurrently handle.

Specifically, we measured the average latency per votes(in seconds) for 10, 25, 50, and 100 concurrent cast vote request, for setups consisting of 4, 7 and 10 IC peers. Figure 5b presents our findings. For 10 concurrent clients, the average latency for 4, 7, and 10 IC peers is 1.68, 2.76, 3.96 seconds respectively. The corresponding values for 25, 50, and 100 concurrent clients are (2.71, 4.61, 7.18), (5.17, 8.29, 13.02), and (10.68, 18.21, 26.8).

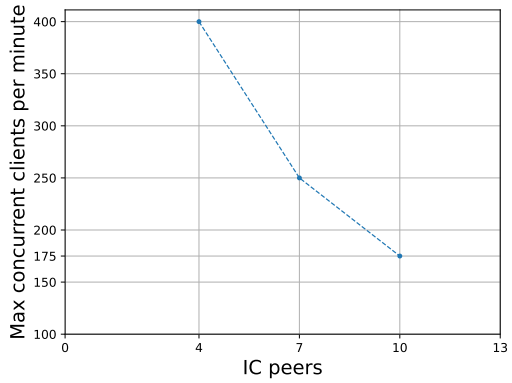
There are two main observations deriving from figure 5b. Firstly, since we are able to handle votes concurrently, the average latency per cast vote request does not grow linearly with the addition of new concurrent clients, as it would be the case if the votes were handled sequentially. For example, with 4 IC peers and 100 concurrent clients, we are able to provide an average latency of less than 9 times the average latency of the previous experiment, instead of the expected 100 times increase in a non-concurrent setting. Secondly, as we increase the number of servers, we are still able to maintain a non-quadratic growth in the latency per vote, indicating that the implementation performs adequately even in the presence of a very large number of concurrent messages.

5.1.2 Throughput

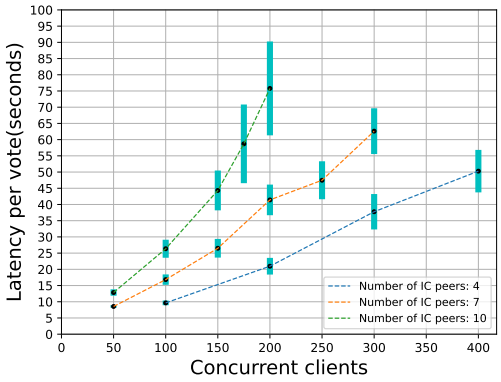
Having established that our implementation is able to concurrently handle a number of clients, while preserving reasonable latency properties(especially when compare to a sequential approach), we now need to identify the throughput attributes of our system.

In particular, we define throughput as the maximum number of cast vote requests that our system is able to facilitate within a minute, for specified numbers of IC peers and concurrent clients. In other words, since votes are handled concurrently, our goal was to measure how many concurrent votes our implementation of the Posting Protocol is able to facilitate while maintaining an average latency per vote of up to 1 minute.

In order to identify the maximum number of cast vote requests that each setup was able to accommodate, we simply ran our multi-threaded voting client while raising the number of concurrent votes to be initiated, for setups of 4, 7, and 10 IC peers. Figure 6a displays the corresponding results. As we can see from the graph, a setup consisting of 4 IC peers was able to concurrently handle up to 400 cast vote requests. Accordingly, a setup of 7 IC peers manages to handle up to 250 concurrent cast vote requests, and a setup of 10 IC peers is able to manipulate up to 175 concurrent cast vote requests. We should note that this inversely proportional relation between throughput(per minute) and the number of IC peers -though expected- indicates a limitation of our current implementation. While



(a) Maximum Throughput diminishes steeply as new IC peers are added to the setup



(b) Average latency per concurrent clients grows more rapidly for setups with more IC peers

Figure 6: Vote Casting Throughput

adding more peers to the setup allows for better fault tolerance and improved reliability, our experiment indicates that if we wish to maintain a latency per vote of under a minute while also facilitating a realistic number of concurrent requests, the number of IC peers should be kept relatively low.

Stemming from the previous observation, we also present in Figure 6b the corresponding latency per vote for different values of concurrent client requests, and for 4, 7, and 10 IC peers. Confirming our previous remark, it is evident that the lower the number of IC peers in the setup, the more concurrent clients we are able to facilitate and the smoother the rise in the respective latency per vote is when adding more concurrent client requests. Thus, the election authority should be very careful when determining the right trade-off between performance and fault-tolerance.

Remarks: Our findings indicate that while the current implementation of the Posting Protocol showcases certain desired properties, its scalability with regards to both concurrent clients and IC peers is limited. Therefore, our implementation may properly accommodate small to medium sized elections, with regards to the number of expected concurrent client cast vote requests and the fault-tolerance and security guarantees that IC peers provide to our system. Some key areas of improvement would include a more fine-grained multi-threaded approach in our IC peer implementation, instead of relying solely on asyncio for concurrent handling of messages and their respective tasks, as well as a more compact encoding of messages that would reduce the associated message processing procedures.

5.2 Security Evaluation

In this part, we will be providing an experimental evaluation of the security properties of our implementation, as defined in Section 3.3. It is worth noting that our work currently serves as a proof of concept on the applicability of the BB systems as part of practical e-voting systems. Thus, a more extensive formal analysis of the final implementation would be required before integrating our modules in a production e-voting environment.

Before proceeding with our experiments, it is important to provide some clarifications regarding the security properties that we will be targeting, based on the assumptions of our implementation. Specifically:

- Confirmable Liveness formally encompasses properties (bb.2) and (bb.3) of the security framework introduced by Kiayias et al. (cf. Section 3.3). The security framework of the BB systems does not require (bb.3) to hold, and conflict resolution depends on the description of function $Select(\cdot)$, which is subject to the election setting that leverages the BB protocols. Thus, assessing whether Confirmable Liveness holds in our implementation is equivalent to testing whether our modules satisfy condition (bb.2).
- With regards to Persistence, assuming the existence of an honest centralized AB component, the "unremovability" property (bb.4) (cf. Section 3.3) is obviously satisfied, since an honest AB component would never remove items for which the IC peers have agreed on. Given that we do not currently support multiple periods, and the AB is honest, our experiments on Persistence are equivalent to testing that property (bb.1) holds in our implementations of both the CS BB and the KKLSZ systems.
- We will not be evaluating the Confirmable Persistence property of Section 3.3. As already stated in Section 3.3.2, the CS BB system always employs a single AB component. If the AB is malicious, Confirmable Persistence may hold by employing a proper TSS scheme. We leave the evaluation of *bls-lib*(cf. Section 4.3) as a potentially secure $(k, t_s, N_c) - TSS$ scheme as part of our future assignments. Another alternative would be the implementation of one of the schemes proposed in Section 3.3.2.3. With regards to the KKLSZ BB system, Confirmable Persistence does not hold without the existence of a distributed AB subsystem. The development of such a subsystem is also part of our upcoming enhancements, and will follow the guidelines showcased in Section 3.3.3.

Any relevant screenshots that may facilitate our interpretations of the experimental results will be included in Appendix B.

5.2.1 Confirmable Liveness experiment without malicious entities

In this section, we will be providing an initial assessment of the Confirmable Liveness of our implementation of the CS BB and the KKLSZ systems, assuming the absence of malicious parties. In particular, we aim to investigate whether:

- Condition (bb.2) holds when IC peers become unavailable during the execution of the Posting Protocol.
- The Posting Protocol operates properly with only a threshold number of IC peers available during its execution. While this is not a property defined by the security framework of Kiayias et al., it is an attribute of the systems that should be preserved under our implementation.

With regards to the Publishing Protocols of both implemented BB systems, our current implementation presumes that all IC peers are live and participate in the respective Publishing Protocol (and are also honest in this experiment).

The CS BB system dictates that if at every execution of the Posting Protocol a threshold set of $N_c - t_c = N_c - \lceil \frac{N_c}{3} \rceil + 1$ IC peers were live and able to communicate, then only one round of the Fallback Protocol is required for (bb.2) to be satisfied. In order to verify that in our implementation, we utilize the *run_scenarios.py* script [2]. In order to run *run_scenarios.py*, we need to provide it with a configuration file that includes the BB system to be utilized, the IC peers to instantiate, which of the IC peers should shut down and at which votes, and whether there is an adversary in the setup. Based on this information, the script initializes the IC peers by running subprocesses of *server.py* and simulates an election consisting of 10 voters (each casting a single vote) and 3 candidates by running a subproccess of *zeus_core.py*. All subprocesses use the configuration file supplied to *run_scenarios.py*, so that the simulation includes the shutting down and restarting of the respective IC peers (in this experiment, there is no adversary in the setup).

To test the Confirmable Liveness guarantees for our implementation of the CS BB system in the absence of malicious entities, we initiate a scenario with 4 IC peers, where at most 1 of which is unavailable during an execution of the Posting Protocol for each vote. The corresponding command is:

```
$ python run_scenarios.py data/security-configs/scenario1.json
```

Since some of the IC peers missed some of the votes, during the first execution of the Optimistic Protocol, there will not be $N_c - t_c$ identical records of posted items exchanged between the IC peers, forcing them to engage in the Fallback Protocol.

After running our experiment, the final published record includes all votes for which a valid TSS signature was issued and only one execution of the Fallback Protocol was required (see figure 12). Also, all votes were properly handled and included in the final AB record, even though some of the peers were unavailable during executions of the Posting Protocol. This can be verified by examining the output of *zeus_core.py*, as we showed in Section 4.4.

With regards to the KKLSZ BB system, Confirmable Liveness holds in a partially synchronous setting with $< \frac{N_c}{3}$ corrupted IC peers, against a general computationally bounded Byzantine adversary. Given that all IC peers of this experiment are honest, we expect (bb.2) to hold, meaning that all votes for which the users received enough TSS signature shares to generate a TSS signature, must appear on the final record of the AB. Considering the assumptions of our implementation (cf. Section 4.2), our experiment serves as an indication of whether we have achieved the desired properties. Yet, a formal analysis would need to be performed before integrating our solution in production environments that require guarantees of the security properties of our implementation.

To simulate the previous experiment for the KKLSZ BB system, we need to run:

```
$ python run_scenarios.py data/security-configs/scenario1-advanced.json
```

Again, similar to the previous experiment, the final AB record includes all votes for which a valid TSS signature was issued and all votes were properly handled during the Posting Protocol. To verify that, we can follow the process explained in Section 4.4.

Overall, our results indicate that both systems preserve Confirmable Liveness in the absence of malicious entitites, and are able to handle the unavailability of $< \frac{N_c}{3}$ IC peers during individual executions of the Posting Protocol.

[2] `https://github.com/gmetaxo/dzeus/blob/main/src/run_scenarios.py`

5.2.2 Confirmable Liveness experiment with malicious entities

As previously mentioned in Section 3.3.2, Kiayias et al. (cf. [18], Section 5.2) demonstrate an attack against the Confirmable Liveness of the CS BB system. This attack requires the existence and the collaboration of malicious entities (users and IC peers), and seeks to violate condition (bb.2). We will first briefly describe the attack and then perform experiments that simulate this adversarial scenario for both BB systems. Given the analysis in Section 3.3, we expect condition (bb.2) to not hold for the CS BB system, given that our implementation has not incorporated any detection mechanisms and strictly follows the system description of [9]. Regarding the KKLSZ BB system, Confirmable Liveness holds in a partially synchronous setting with $< \frac{N_c}{3}$ corrupted IC peers, against a general computationally bounded Byzantine adversary. Considering the assumptions of our implementation (cf. Section 4.2), our experiment for the KKLSZ BB system may only serve as an indication of whether we have achieved to preserve Confirmable Liveness. We will first describe the attack targeting the CS BB system, and then we will be following the same steps for the KKLSZ BB system up to the point when the KKLSZ BB Publishing Protocol begins.

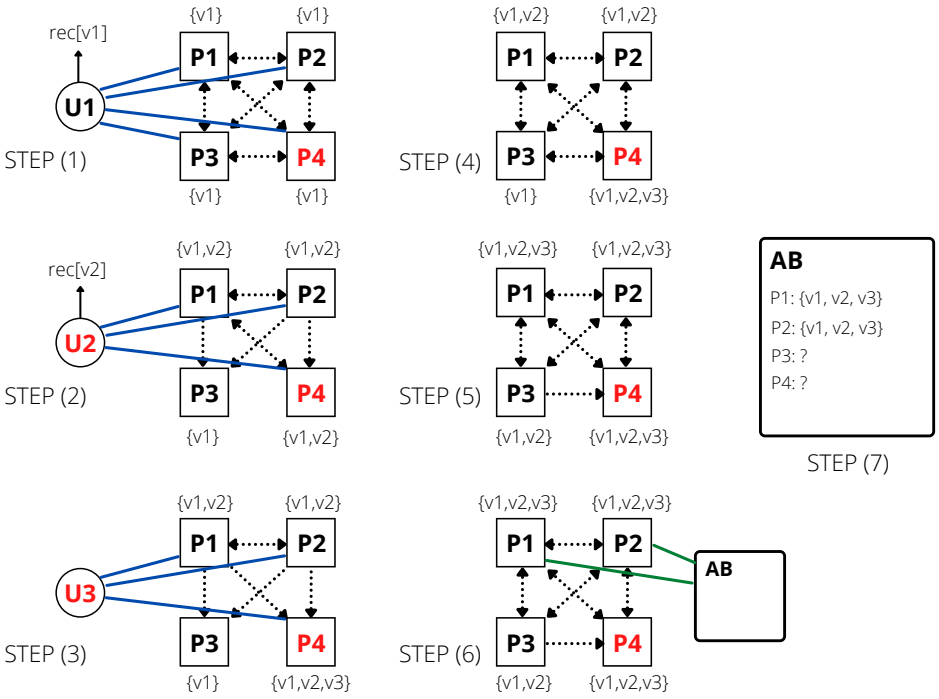


Figure 7: Attacking Confirmable Liveness for the CS BB system for $N_c = 4$ and $t_c = 1$

Following figure 7, the steps of the attack for $N_c = 4$ IC peers and $t_c = 1$ corrupted IC peers (peer P_4 is the malicious peer) are:

1. An honest user U_1 broadcasts item v_1 to all IC peers and obtains a valid receipt $rec[v_1]$.
2. A malicious user U_2 broadcasts item v_2 to the malicious peer P_4 and to $N_c - 2t_c = 2$ honest peers, P_1 and P_2. We denote this set of honest peers as $H_{in} = \{P_1, p_2\}$. The malicious peer only interacts with peers in H_{in} during the Posting Protocol. Even though there are $t_c = 1$ honest peers that did not receive the item and do not participate in the post request, the collaboration of $t_c + (N_c - 2t_c) = N_c - tc$ peers is

enough for U_2 to obtain a valid receipt $rec[v_2]$. Item $v_2 \in B_i$ only for honest peers $P_i \in H_{in}$. We denote as H_{out} the $t_c = 1$ peers for which $v_2 \notin B_i$, so $H_{out} = \{P_3\}$.

3. Malicious user U_3 sends item v_3 to all peers in H_{in} and the malicious peer P_4. Now, the malicious peer does not engage in the Posting Protocol, so the peers in H_{in} do not suffice for U_3 to obtain a receipt.

4. When the Publishing Protocol starts, all peers engage in the Optimistic Protocol and send their respective signed local records to each other. None of the three records $(\{v_1\}, \{v_1, v_2\}, \{v_1, v_2, v_3\})$ that are exchanged is signed by at least $N_c - t_c = 3$ peers, so the peers engage in the Fallback protocol.

5. During the Fallback Protocol, all honest peers exchange their local databases of signatures D_i. In this way, peer P_3 receives $N_c - t_c$ signatures for item v_2, but only $N_c - 2t_c$ signatures for item v_3, so it updates its local record to $\{v_1, v_2\}$. The malicious peer P_4 send its database of signatures only to peers in H_{in}. Thus, P_1 and P_2 have now collected $N_c - t_c$ signatures of item v_3 and update their local records to $\{v_1, v_2, v_3\}$.

6. After finishing the first Fallback Protocol run, all peers re-engage in the Optimistic Protocol with their updated records. The malicious peer P_4 signs and sends its local record $\{v_1, v_2, v_3\}$ only to peers in H_{in}, while honest peers sign and broadcast their updated local records. Now, the peers in H_{in} have $N_c - t_c$ signatures for record $\{v_1, v_2, v_3\}$ and finalize their engagement in the Publishing Protocol by sending their TSS signature shares on $\{v_1, v_2, v_3\}$ to the AB.

7. After the peers in H_{in} have finished their participation in the Publishing Protocol, P_4 becomes inert, forcing P_3 to remain pending for a new run of the Fallback Protocol, in which no other peer will participate. Therefore, the AB cannot obtain $N_c - t_c$ TSS signature shares for an agreed-on record, and cannot publish a final record. This is a violation of Confirmable Liveness, and specifically of condition (bb.2), as items that have a valid receipt do not appear on the AB.

In order to simulate the above, we first need to run the following commands to setup our IC peers:

```
# Each command needs to run in a different shell.

$ python server.py 127.0.0.1 65432 --config data/security-configs/liveness
    ↪ -attack.json --debug --interactive
$ python server.py 127.0.0.1 65433 --config data/security-configs/liveness
    ↪ -attack.json --debug --interactive
$ python server.py 127.0.0.1 65434 --config data/security-configs/liveness
    ↪ -attack.json --debug --interactive
$ python server.py 127.0.0.1 65435 --config data/security-configs/liveness
    ↪ -attack.json --debug --interactive
```

Then, we need to run the following:

```
$ python zeus_core.py data/security-configs/liveness-attack.json --
    ↪ generate out.json --attack
```

The command above simulates an election were 3 users cast their votes, with only 2 of them receiving a valid receipt following the steps in figure 7, and IC peers behave exactly as described above. One small deviation from the example, is that we have have included a timeout in our implementation of the Fallback Protocol, so instead of remaining pending for a new Fallback Protocol run, the IC peer in H_{out} re-engages in the Optimistic Protocol. Since no other peer follows the Optimistic Protocol, the peer in H_{out} re-engages in the Fallback Protocol. Yet again, no other peer participates in the Fallback Protocol and the peer in H_{out} re-engages in the Optimistic Protocol and so forth. After 10 such unsuccessful runs, we exit our program and announce the inability to publish a final agreed-on record. A corresponding screenshot can be found in figure 13. Confirmable Liveness, as expected, does not hold for our implementation of the CS BB system. In order to satisfy Confirmable Liveness, we would need to enhance our implementation with a detection mechanism of such malicious patterns.

We have also simulated the previous attack scenario for the KKLSZ BB system. In particular, we follow steps (1-3) as part of the attack, but once the Publishing Protocol begins (step 4), we let it run according to its original process(cf. Section 3.2.3) and the assumptions of our implementation(cf. Section 4.2). In order to simulate this scenario, we first need to run the following commands to setup our IC peers:

```
# Each command needs to run in a different shell.

$ python server.py 127.0.0.1 65432 --config data/security-configs/liveness
   ↪ -attack-advanced.json --debug --interactive
$ python server.py 127.0.0.1 65433 --config data/security-configs/liveness
   ↪ -attack-advanced.json --debug --interactive
$ python server.py 127.0.0.1 65434 --config data/security-configs/liveness
   ↪ -attack-advanced.json --debug --interactive
$ python server.py 127.0.0.1 65435 --config data/security-configs/liveness
   ↪ -attack-advanced.json --debug --interactive
```

Then, we need to run the following:

```
$ python zeus_core.py data/security-configs/liveness-attack-advanced.json
   ↪ --generate out.json --attack
```

The command above simulates an election were 3 users cast their votes, with only 2 of them receiving a valid receipt, following the steps (1-3) in figure 7. The script ends successfully, and the AB is able to publish a final agreed-on record, which includes items v_1 and v_2. Therefore, condition (bb.2) is satisfied, and all items that received a valid receipt appear in the final published AB record. While this is an indication that our implementation achieves the Confirmable Liveness property, a more formal analysis of the implications of our implementation would need to be performed before we claim that as a guarantee. A final interesting remark is that though the second vote was initiated by a malicious user and posted with the collaboration of honest and malicious peers, it appears in the final published record. However, this does not violate any liveness or persistence requirements as defined in Section 3.3.

5.2.3 Persistence experiment

Finally, we will be examining whether our implementation of both the CS BB system and the KKLSZ BB system satisfies Persistence. As already mentioned in Section 5.2, given the assumptions and the characteristics of our implementation, evaluating Persistence is equivalent to evaluating whether condition (bb.1) holds.

Since condition (bb.1) expresses safety against publishing items that have not been posted, our experiment consists of the following:

- For the CS BB system, after the Posting Protocol has finished, a malicious peer adds an illegitimate item to its local record, and also signs it and adds the signature to its local database of signatures. We also shut down and restart certain IC peers during the Posting Protocol (as in Section 5.2.1), in order to force a Fallback Protocol execution and verify that the exchange of signature databases that may contain the illegitimate item does not violate Persistence.
- For the KKLSZ BB system, after the Posting Protocol has finished, a malicious peer adds an illegitimate item to its local record and proceed normally with the execution of the Publishing Protocol.

We run this experiment for $N_c = 4$ IC peers, where $t_c = 1$ of them is malicious and performs the illegitimate item (the letter "X" in our experiment) injection on its local record. As in our experiment in Section 5.2.1, we simulate an election consisting of 10 voters (each casting a single vote) and 3 candidates, but in this experiment we have also included the item injection described above before proceeding to the Publishing Protocol. To simulate the experiment for the CS BB system, we would need to run:

```
$ python run_scenarios.py data/security-configs/scenario2.json
```

Accordingly, to simulate the experiment for the KKLSZ BB system, we would need to run:

```
$ python run_scenarios.py data/security-configs/scenario2-advanced.json
```

For both systems, if we examine the final published record we can verify that the illegitimate item is not part of it. Therefore, for the CS BB system, we can claim that our implementation guarantees Persistence. While we cannot guarantee the same for the KKLSZ system (given our deviations from the formal description of the system), we can claim that our experiment serves as an indication that our implementation of the KKLSZ system achieves Persistence.

6. CONCLUSIONS AND FUTURE WORK

Our work revolves around the implementation of a set of distributed protocols, that can be applied in e-voting applications seeking to incorporate a distributed BB component to their respective system.

First, we provide a concise description of the BB systems we have implemented, namely the CS BB system [9] and the KKLSZ BB system [18], focusing on the protocols for posting items to the local records of item collection (IC) peers and for publishing the agreed-on record on the audit board (AB). Following the security framework and the respective analysis presented by Kiayias et al. [18], we also present the fault-tolerance thresholds of each BB system in terms of Confirmable Liveness, Persistence and Confirmable Persistence.

Furthermore, we implement a set of platform-independent modules that realize the BB protocols of both systems, and that can be used on top of e-voting systems that seek a distributed BB component. We stated the assumptions of our implementation, provided an analysis of the developed modules, and argued on any deviations from the descriptions of the original protocols.

To prove the applicability of our solution, we integrate our distributed BB protocol suite into Zeus, which is a well-established e-voting system that currently employs a centralized BB component.

Subsequently, we evaluate our prototype integration, to assess the scalability and security features of our implementation, when applied in a practical e-voting setting. With regards to scalability, our findings indicate that our current implementation may properly accommodate small to medium sized elections, with regards to the number of expected concurrent client cast vote requests and the number of IC peers that comprise the respective setup.

Regarding the security evaluation, we simulate several adversarial scenarios and examined whether the properties defined in the formal framework of Kiayias et al. [18] hold in our implementation. Our experiments indicated that the implementation correctly preserves the expected security properties, yet, a formal analysis would be required in order to provide guarantees that the properties hold before integrating our solution in a production e-voting system.

Taking everything into consideration, there are certain key aspects of our proposed solution that may inspire future work endeavors.

In particular, one key enhancement would be to incorporate the notion of periods into our modules, so that publishing and verifying the validity and consistency of the AB happen at specified intervals. That would allow the timely detection of malicious parties, which could be excluded from the election process before its finalization.

Moreover, a more fine-grained performance and scalability analysis would be beneficial in identifying and eliminating any bottlenecks of our implementation. The inclusion of a multi-threaded approach in our IC peer implementation (instead of relying solely on *asyncio* for concurrency), as well as a more compact encoding of messages constitute two key areas of improvement that we have identified.

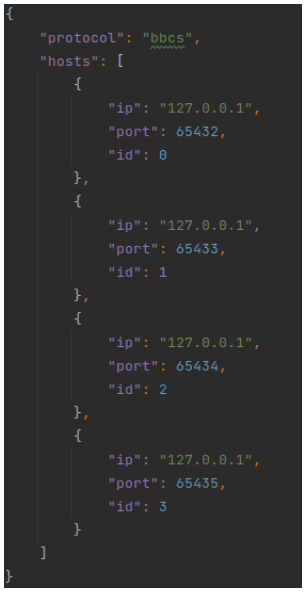
Finally, while we have argued that our deviations from the original protocol definitions (e.g.: omitting the Consensus phase from the KKLSZ BB Publishing Protocol) did not result in any violation of the expected security properties, we would need to perform a formal evaluation of our observations before deciding whether such alterations are safe,

or that we should proceed with the development of the corresponding procedures.

ABBREVIATIONS - ACRONYMS

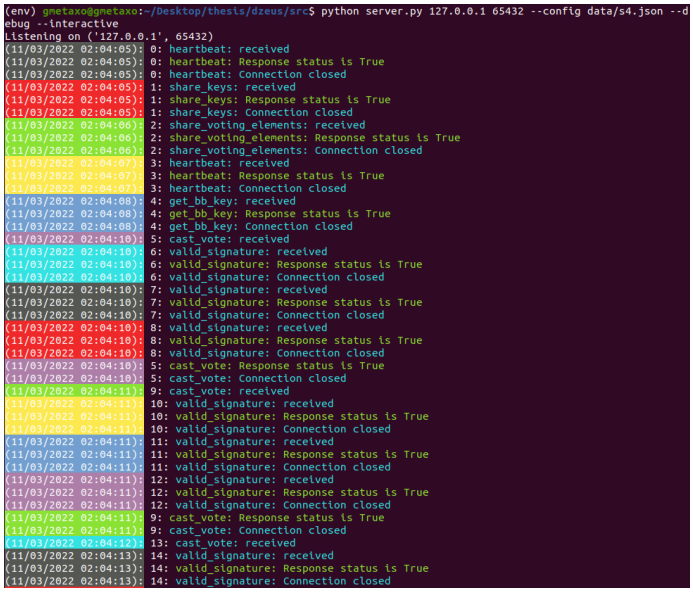
AB	Audit Board
BB	Bulletin Board
BLS	Boneh-Lynn-Shacham
BPS	Ballot Preparation System
CS	Culnane-Schneider
EA	Election Authority
EUFCMA	Existential Unforgeability under Chosen Message Attack
E2E	End-to-End
GRNET	Greek Research and Education Network
IC	Item Collection
KKLSZ	Kiayias-Kuldmaa-Lipmaa-Siim-Zacharias
SA	Setup Authority
STV	Single Transferable Vote
TSS	Threshold Signature Scheme
VC	Vote Collection
WBB	Web Bulletin Board

APPENDIX A. SCREENSHOTS OF EXECUTION



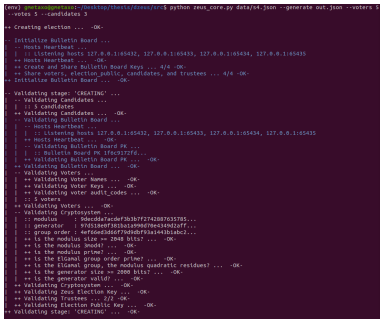
```
{
    "protocol": "bbcs",
    "hosts": [
        {
            "ip": "127.0.0.1",
            "port": 65432,
            "id": 0
        },
        {
            "ip": "127.0.0.1",
            "port": 65433,
            "id": 1
        },
        {
            "ip": "127.0.0.1",
            "port": 65434,
            "id": 2
        },
        {
            "ip": "127.0.0.1",
            "port": 65435,
            "id": 3
        }
    ]
}
```

Figure 8: Server configuration file with 4 IC peers for the CS BB system

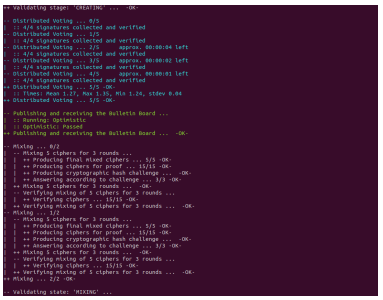


```
(env) gmetaxo@gmetaxo:~/Desktop/thesis/dzeus/src$ python server.py 127.0.0.1 65432 --config data/s4.json --d
ebug --interactive
Listening on ('127.0.0.1', 65432)
(11/03/2022 02:04:05): 0: heartbeat: received
(11/03/2022 02:04:05): 0: heartbeat: Response status is True
(11/03/2022 02:04:05): 0: heartbeat: Connection closed
(11/03/2022 02:04:05): 1: share_keys: received
(11/03/2022 02:04:05): 1: share_keys: Response status is True
(11/03/2022 02:04:05): 1: share_keys: Connection closed
(11/03/2022 02:04:06): 2: share_voting_elements: received
(11/03/2022 02:04:06): 2: share_voting_elements: Response status is True
(11/03/2022 02:04:06): 2: share_voting_elements: Connection closed
(11/03/2022 02:04:07): 3: heartbeat: received
(11/03/2022 02:04:07): 3: heartbeat: Response status is True
(11/03/2022 02:04:07): 3: heartbeat: Connection closed
(11/03/2022 02:04:08): 4: get_bb_key: received
(11/03/2022 02:04:08): 4: get_bb_key: Response status is True
(11/03/2022 02:04:08): 4: get_bb_key: Connection closed
(11/03/2022 02:04:10): 5: cast_vote: received
(11/03/2022 02:04:10): 6: valid_signature: received
(11/03/2022 02:04:10): 6: valid_signature: Response status is True
(11/03/2022 02:04:10): 6: valid_signature: Connection closed
(11/03/2022 02:04:10): 7: valid_signature: received
(11/03/2022 02:04:10): 7: valid_signature: Response status is True
(11/03/2022 02:04:10): 7: valid_signature: Connection closed
(11/03/2022 02:04:10): 8: valid_signature: received
(11/03/2022 02:04:10): 8: valid_signature: Response status is True
(11/03/2022 02:04:10): 8: valid_signature: Connection closed
(11/03/2022 02:04:10): 5: cast_vote: Response status is True
(11/03/2022 02:04:10): 5: cast_vote: Connection closed
(11/03/2022 02:04:11): 9: cast_vote: received
(11/03/2022 02:04:11): 10: valid_signature: received
(11/03/2022 02:04:11): 10: valid_signature: Response status is True
(11/03/2022 02:04:11): 10: valid_signature: Connection closed
(11/03/2022 02:04:11): 11: valid_signature: received
(11/03/2022 02:04:11): 11: valid_signature: Response status is True
(11/03/2022 02:04:11): 11: valid_signature: Connection closed
(11/03/2022 02:04:11): 12: valid_signature: received
(11/03/2022 02:04:11): 12: valid_signature: Response status is True
(11/03/2022 02:04:11): 12: valid_signature: Connection closed
(11/03/2022 02:04:11): 9: cast_vote: Response status is True
(11/03/2022 02:04:11): 9: cast_vote: Connection closed
(11/03/2022 02:04:12): 13: cast_vote: received
(11/03/2022 02:04:13): 14: valid_signature: received
(11/03/2022 02:04:13): 14: valid_signature: Response status is True
(11/03/2022 02:04:13): 14: valid_signature: Connection closed
```

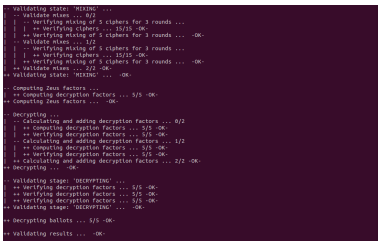
Figure 9: Screenshot of server logs while votes are being submitted



(a) Logs of zeus_core.py when running an election(pt.1)



(b) Logs of zeus_core.py when running an election(pt.2)



(c) Logs of zeus_core.py when running an election(pt.3)

Figure 10: Logs of zeus_core.py for an election that was properly finalized

(a) Part of the final Zeus' poll document and: Vote with the underlined fingerprint was properly submitted and included in the final tally

(b) Part of the final published BB: Vote with the underlined fingerprint and the final AB receipt are included in the published record

Figure 11: Screenshots of Zeus' final poll document and the final published BB record

APPENDIX B. SCREENSHOTS OF EXECUTION

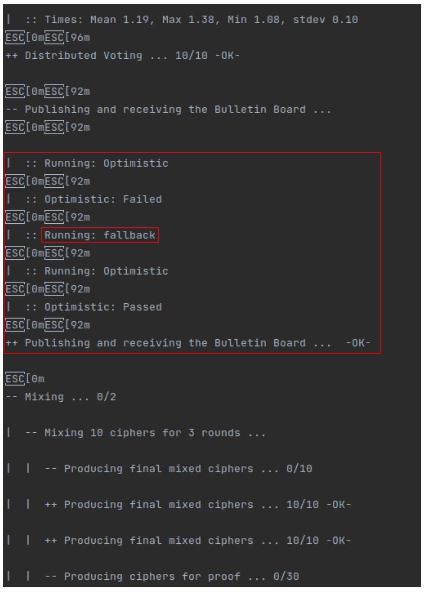


Figure 12: Logs of *zeus_core.py* verifying that the Fallback Protocol ran only once

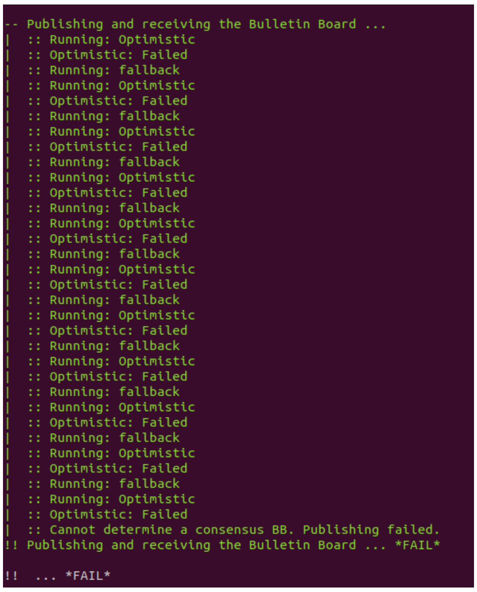


Figure 13: Logs of *zeus_core.py* verify that the CS BB system cannot reach a consensus on the final record. Confirmable Liveness is breached.

BIBLIOGRAPHY

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] B. Adida. Advances in cryptographic voting systems. 2006.

[3] B. Adida. Helios: Web-based open-audit voting. pages 335–348, Jan. 2008.

[4] S. T. Ali and J. Murray. An overview of end-to-end verifiable voting systems. *Real-World Electronic Voting*, pages 189–234, 2016.

[5] S. Bell, J. Benaloh, M. D. Byrne, D. DeBeauvoir, B. Eakin, P. Kortum, N. McBurnett, O. Pereira, P. B. Stark, D. S. Wallach, et al. Star-vote: A secure, transparent, auditable, and reliable voting system. In *2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 13)*, 2013.

[6] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In C. Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[7] C. Burton, C. Culnane, J. Heather, T. Peacock, P. Ryan, S. Schneider, S. Srinivasan, V. Teague, R. Wen, and Z. Xia. A supervised verifiable voting protocol for the victorian electoral commission. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, Jan. 2012.

[8] N. Chondros, B. Zhang, T. Zacharias, P. Diamantopoulos, S. Maneas, C. Patsonakis, A. Delis, A. Kiayias, and M. Roussopoulos. D-demos: A distributed, end-to-end verifiable, internet voting system. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 711–720. IEEE, 2016.

[9] C. Culnane and S. Schneider. A peered bulletin board for robust use in verifiable voting systems. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 169–183. IEEE, 2014.

[10] C. Culnane and S. A. Schneider. A peered bulletin board for robust use in verifiable voting systems. *CoRR*, abs/1401.4151, 2014.

[11] G. Dini. A secure and available electronic voting service for a large-scale distributed system. *Future Generation Computer Systems*, 19(1):69–85, 2003.

[12] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[13] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 04 1988.

[14] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and efficient sharing of rsa functions. volume 20, pages 157–172, 01 1996.

[15] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 04 1988.

[16] P. Golle, M. Jakobsson, A. Juels, and P. Syverson. Universal re-encryption for mixnets. volume 2964, pages 163–178, Feb. 2004.

[17] J. Heather and D. Lundin. The append-only web bulletin board. In *International Workshop on Formal Aspects in Security and Trust*, pages 242–256. Springer, 2008.

[18] A. Kiayias, A. Kuldmaa, H. Lipmaa, J. Siim, and T. Zacharias. On the security properties of e-voting bulletin boards. In *IACR Cryptol. ePrint Arch.*, 2018.

[19] A. Kiayias, T. Zacharias, and B. Zhang. End-to-end verifiable elections in the standard model. In *EUROCRYPT*, 2015.

[20] A. Kiayias, T. Zacharias, and B. Zhang. End-to-end verifiable elections in the standard model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 468–498. Springer, 2015.

[21] R. Krummenacher. Implementation of a web bulletin board for e-voting applications. *Project Report, Switzerland: Hochschule für Technik Rapperswil (HSR)*, 2010.

[22] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. In *Proceedings of the 32nd International Conference on Automata, Languages and Programming*, ICALP'05, page 204–215, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] M. Kutylowski and F. Zagorski. Scratch, click & vote: E2e voting over the internet. pages 343–356, 01 2010.

[24] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[25] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[26] D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. pages 51–58, 11 1998.

[27] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 129–140, Berlin, Heidelberg, 1991. Springer-Verlag.

[28] R. Peters. *A secure bulletin board*. PhD thesis, Master's thesis, Technische Universiteit Eindhoven, 2005.

[29] J. Puiggalí Allepuz and S. Guasch Castelló. Universally verifiable efficient re-encryption mixnet. In *4th International Conference on Electronic Voting 2010*. Gesellschaft für Informatik eV, 2010.

[30] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, CCS '94, page 68–80, New York, NY, USA, 1994. Association for Computing Machinery.

[31] K. Sako and J. Kilian. Receipt-free mix-type voting scheme. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 393–403. Springer, 1995.

[32] V. Shoup. Practical threshold signatures. In B. Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 207–220, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[33] N. Tideman. The single transferable vote. *Journal of Economic Perspectives*, 9(1):27–38, 1995.

[34] G. Tsoukalas, K. Papadimitriou, P. Louridas, and P. Tsanakas. From helios to zeus. In *2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 13)*, Washington, D.C., Aug. 2013. USENIX Association.

[35] F. Zagorski, R. Carback, D. Chaum, J. Clark, A. Essex, and P. Vora. Remotegrity: Design and use of an end-to-end verifiable remote voting system. pages 441–457, 06 2013.