A Verifiable Distributed Voting System

Without a Trusted Party

by

Spencer J Bouck

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved January 2021 by the
Graduate Supervisory Committee:

Rida Bazzi, Chair
Dragan Boscovic
Yan Shoshitaishvili

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

Cryptographic voting systems such as Helios rely heavily on a trusted party to maintain privacy or verifiability. This tradeoff can be done away with by using distributed substitutes for the components that need a trusted party. By replacing the encryption, shuffle, and decryption steps described by Helios with the Pedersen threshold encryption and Neff shuffle, it is possible to obtain a distributed voting system which achieves both privacy and verifiability without trusting any of the contributors. This thesis seeks to examine existing approaches to this problem, and their shortcomings. It provides empirical metrics for comparing different working solutions in detail.

ACKNOWLEDGMENTS

This work was supported in part by a grant from Dash, grant number AWD00034285.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

To probe the public opinion on certain topics, individuals or parties may utilize elections. Elections are a vital way of determining the thoughts and opinions of individuals without compromising their privacy. Compromising the privacy of the individual, or connecting a certain vote to that individual, represents a significant breach in the confidentiality implied in a voting system. It is of great importance, then, that the link between votes and voters is kept hidden from all parties.

There exist both theoretical and practical solutions to the problem of maintaining user privacy during an election. On the theoretical side, there is a broad range of approaches used [12][19][20]. Conversely, on the practical side, the approaches are much more confined. Implementations of voting systems exist, but have limitations, such as the need to trust a third party for voter privacy [1][6][10]. Although decentralized voting systems have been attempted, no existing implementation can handle attacks from within [3]. In this work, we show that a practical solution can operate independently of any trusted party by distributing the vote administration to number of voting authorities. The system can also handle non-compliant operators, and provide privacy for its users. We achieve this solution by combining existing techniques for shuffling encrypted votes[14][15] and threshold decryption[8][9]. While these techniques have been proposed in the literature and systems built from them exist, we are aware of no system that combines all of them to achieve secure and private voting in the absence of a trusted authority and non-compliant actors. We propose a system

that represents an improvement on existing systems in two main ways. First, unlike other systems, the system we propose supports free-form voting, in which the length of the votes is not restricted by the requirements of the encryption scheme. Second, and more novelly, our system can operate independently of any particular party: it is a completely distributed system. Further, our system can handle some users going against the scheme we describe, trying to disrupt it.

After implementing our scheme in the Go language, using the Kyber library [22] as a foundation, we ran several runtime tests. To demonstrate the feasibility of our system in real-world environments, we show performance measurements of different aspects of the scheme, noting how the runtimes tend to scale.

1.2 Results

In this work, we present a decentralized system that effectively provides voter privacy without a trusted party. To accomplish this, we combine existing theoretical solutions and practical implementations of cryptographic protocols. We evaluate the system's performance and find that it is feasible for an environment in which up to dozens of voting authorities run elections for thousands of voters. For example, for a system with 20 authorities and over a thousand voters, full execution time of the scheme takes less than 30 seconds on modest hardware.

The rest of this thesis is organized as follows. Chapter 2 introduces the system model and provides a formal overview of the voting problem. Chapter 3 gives an overview of related work and existing schemes. Chapter 4 delves into the workings of our voting scheme, giving an analysis of each portion. Chapter 5 sets forth performance data and test results for the individual portions of our scheme. Finally, chapter 6 concludes the work and examines potential avenues for future work.

Chapter 2

PROBLEM STATEMENT AND SYSTEM MODEL

2.1 Problem Overview

To describe the problem, we follow the example of Cranor and Cytron in the analysis of a voting scheme [6]. There are four main tasks to a voting scheme: registration, validation, collection, and tallying. Registration is the process of determining the list of voters that are allowed to vote in an election. For our purposes, we assume that this step is taken care of by a public consensus algorithm, which may use technologies such as a blockchain. Validation is the process of checking that a given voter is in the set of valid voters. Using the same or similar consensus algorithm, with the list of voter identities supplied, this step becomes a trivial search through the valid voter set. Collection is the process of accumulating the votes from the valid voters. The tallying phase is the process wherein the collected votes are transformed into a published result. We focus on describing the collection and tallying schemes, leaving the registration and validation phases out of our consideration.

The relevant properties we seek for in our scheme are:

- **Accuracy** It is impossible for a valid vote to be altered or omitted from the final tally, and it is impossible for an invalid vote to be counted.
- **Privacy** A vote cannot be connected to the voter who submitted it.
  - **Zero-Knowledge** No information about how any validated voter voted can be determined, other than can be determined by the results of the election.

- **Verifiability** Anyone can independently verify that all votes have been counted correctly.
- **Flexibility** The ballot can take different formats, such as allowing write-in responses.
- **No Trusted Party** The properties of the voting scheme are guaranteed to hold even if one party or a group of parties, up to a threshold, are compromised.

Note that we use a weaker definition of privacy than in other publications; specifically, we allow for voters to prove how they voted. Although existing implementations cover subsets of these properties, we know of none which contain all of them. Specifically, we seek to combine, modify and add theoretical components to existing schemes to achieve all these properties simultaneously.

As with all cryptographic works, our scheme relies on some assumptions in order to provide the security it does. The bulk of scheme relies on the decisional Diffie–Hellman assumption (DDH). This is to ensure semantic security for the ElGamal encryption used to encrypt the votes [14][15]. We assume the existence of a public bulletin board to post relevant data. This can be implemented in practice with a consensus algorithm or similar scheme [13][16]. An additional assumption we make is that some number of operators will not collude in deviating from the scheme. In practice, this is effectively equivalent to requiring a majority of operators to comply with the scheme, which has been shown to be optimal for any scheme with deviant operators [13].

We analyze the runtime of portions of these schemes with respect to various meta-parameters. In doing so, we discover the viability of running a fully decentralized election, and gain insight into the limitations that such a system presents. Among the variables we tweak when evaluating runtime are the number of voters participating in the election, the number of votes cast, and threshold of voters needed to decrypt

the votes.

2.2 Solution Outline

To illustrate what we seek to accomplish, we use an analogy to a classroom setting. Imagine that a class of students wishes to elect a class president. There are several ways to determine what the consensus is; for example, each student wishing to express their opinion could write the name of the person they wish to vote for on a whiteboard at the front of the class. This betrays the privacy of the students, as everyone can see what each student has written on the board. Another option would be to have the teacher go around the room and collect votes from the students. This way, only the teacher would know what each student voted for. However, this would require us to trust the teacher for both accuracy and privacy: if the teacher were to represent the results differently than reported, no student would be able to tell.

To eliminate the requirement of a trusted party, the scheme relies on a number of "authorities" to facilitate the vote. The authorities together create a public key that the voters can use to encrypt their votes, but each authority gets only a share of the corresponding private key so that no authority can decrypt on its own. The voters who wish to vote in the scheme may submit an encrypted vote along with a personal identifier to confirm that they are allowed to participate in the election. At a set time, all of the most recent votes from valid voters are collected. From this point, each of the voting authorities shuffles and re-encrypts the list of votes, removing the association between each vote and the voter who submitted it. The final list of shuffled votes is then decrypted one vote at a time by the authorities working together and contributing using their shares of the private key.

Because each vote is made public by the end, any onlooker may trivially compute the amount of votes for each candidate. This opens the way for use of more complex

counting methods, such as counting a voter's second choice as half a vote.

Chapter 3

RELATED WORK

3.1 Overview

Extensive work has been done in describing and achieving the goals of cryptographic voting systems [2][5][6][17][20]. Such systems aim at providing solutions to electronic voting problems that differ in their privacy, security, and performance requirements. Solutions to these problems use a variety of cryptographic tools, including homomorphic encryption [4], mixnets [1], and digital signatures [12]. Each of these methods for achieving voter privacy comes with benefits and detriments.

Existing voting systems have different aims depending on the problems they solve. They differ in cryptographic assumptions, security guarantees, use cases, usability, and performance. The table below is intended to illustrate the differences in existing schemes, but is not meant to be exhaustive.

Work	Accuracy	Privacy	Verifiability	Flexibility	No Trusted Party	ZKP Techniques
Fujioka *et al.* (1992) [10]	✓	✗	✗	✓	✗	✗
Sako and Kilian (1995) [19]	✓	✗	✓	✗	✗	✓
Cranor and Cytron (1997) [6]	✓	✗	✗	✓	✗	✗
Anane *et al.* (2007) [2]	✓	✓	✗	✓	✗	✗
Adida (2008) [1]	✓	✗	✓	✓	✗	✓
Bocek *et al.* (2009) [3]	✗	✗	✗	✗	✓	✗
Chaidos *et al.* (2016) [4]	✓	✓	✓	✓	✗	✓
This Work	✓	✓	✓	✓	✓	✓

Fujioka *et al.* were among the first authors to describe a practical cryptographic voting scheme [10]. The scheme they describe utilizes digital signatures to achieve a limited form of privacy. Although the scheme allows all voters to check if their vote has been properly counted, there is no way for a third party to verify that the outcome

of an election accurately reflects the intent of the voters. The main drawback of the scheme is its reliance on trusted parties.

Sako and Kilian make great strides in the use of zero-knowledge proofs to allow universal verifiability and a limited form of privacy [19]. However, to achieve accuracy, the scheme relies on a strong physical assumption of an untappable communication channel. As stated in the paper, this channel cannot simply be simulated with cryptographic means. The scheme is also not flexible; due to the zero-knowledge scheme used, the vote options are restricted to a binary 0 and 1.

The main contribution we use from the work of Cranor and Cytron is the list of important properties for a voting system [6]. As in the work of Fujioka *et al.*, the scheme described by Cranor and Cytron achieves a limited form of verifiability [10]. The scheme described also relies heavily on trusted parties.

In order to provide privacy, there is a need to obscure the relationship between the vote and the voter. Many voting schemes use a shuffle, where encrypted votes are rearranged and re-encrypted. One way of implementing this type of shuffle is cut and choose [19]. A novel zero-knowledge shuffling scheme is described by Neff [14]. This method of shuffling boasts greater speed than other comparable methods.

Neff further improves upon their earlier work and optimizes the scheme for messages encrypted in the style of ElGamal [15].

The scheme described by Anane *et al.* ensures vote accuracy, privacy, and flexibility with blind signatures [2]. However, since a voter cannot prove whether their vote was counted wrongly, the scheme provided does not provide the strong form of verifiability we seek. More importantly, the scheme, like its predecessors, relies on a trusted party to manage the votes.

Described by Adida is an implementation of a voting platform system called Helios [1]. The system described achieves unconditional accuracy and verifiability through

the use of zero knowledge proofs. Vote content is also flexible. Although it has limited privacy, it is adequate for small scale elections. This is achieved by gathering input from various voters through a JavaScript interface, and publishes the results of an election. One advantage of the techniques used in this procedure is the ability to publicly verify any election results. However, all of the data collected is given as plaintext to the Helios servers, which must be trusted for privacy.

Within the work of Bocek *et al.*, a specific use case of a voting system is described where peers can vote on modifications to documents [3]. The main contribution the work makes is the removal of centralization from the voting scheme, which it accomplishes by distributing various roles to peers. However, it is limited by a lack of privacy- all votes are public. Additionally, the scheme lacks flexibility, as votes may only be "positive" or "negative."

Chaidos *et al.* make significant progress in removing the need for interaction [4]. The main contribution made in this paper is the addition of a property called strong receipt-freeness. In essence, this is a form of privacy so extensive that a valid voter cannot prove to anyone how they voted. Although we value the contribution to privacy, the extent to which they provide coercion-resistance is beyond the requirements described in this paper. More importantly, the scheme still relies on a trusted party to carry out the scheme. An additional takeaway from the work of Chaidos *et al.* is the observation of a nuanced attack (known as a replay attack) on the privacy in systems such as those described by Adida [4][1]. This attack illustrates that in some cases, it is possible for an attacker to duplicate an encrypted valid ballot. We discuss this attack later in this paper, describing both how it can be eliminated, and cases where such an attack could be *beneficial* when leveraged by legitimate users.

From this analysis, we conclude that the decentralization of privacy represents a significant challenge to voting schemes.

Mixnets provide flexibility in potential vote content and can easily be used independently from any centralized authority. For this reason, we choose a mixnet as our source of privacy. Specifically, the mixnet described by Neff was selected for our procedure because it achieves speeds up to 80 times that of similar algorithms, such as those used by Sako and Kilian and Adida [15][19][1].

3.2 Fault Tolerance

Seeking to make our scheme robust to non-compliant or adversarial users, we sought ways to One solution to this problem to add a threshold portion to our scheme. Without this, our scheme would require all users involved in the scheme to contribute their portion of the pubic key in order to come to decrypt the votes. We chose the threshold system described by Pedersen because unlike other schemes, it relies on no trusted party to provide the public key [18]. In order to maintain this independence, we further chose to use a decryption scheme modeled after the methods described by Desmedt and Frankel [8]. This scheme excelled above more conventional approaches because it allowed for decryption in a decentralized fashion.

Chapter 4

THE VOTING SCHEME

4.1 Construction

In this chapter, we outline the design of our voting scheme. Our voting scheme can be described as having five main steps:

- **Initialization** Each user in the scheme supplies a portion of a public key for the voting system.
- **Communication** The public key is computed publicly from these portions.
- **Vote Collection** Any voter, whether involved in the scheme or not, may submit a ballot encrypted with the public key. This encrypted ballot will be linked with the voter, in order to verify that the ballot was submitted by a verified user.
- **Shuffle** After all ballots are submitted, they are shuffled in a publicly verifiable but untraceable manner. This obscures the relationship between each voter and ballot.
- **Decryption** Each shuffled ballot is decrypted publicly in a joint computation by the users. A threshold number of these users must all contribute for the ballot to be decrypted. Using a threshold here maintains accuracy and verifiability while accounting for defiant users, such that no individual party needs to be trusted.

We utilize a mixnet described by Neff to achieve voter privacy, chosen for its speed in execution above other methods [14][15]. Furthermore, we use the threshold scheme

described by Pedersen in [18] to provide a public key for voters to submit their votes with, as it can generate such a key in a decentralized setting. Achieving decryption of shuffled votes without allowing for the decryption of the unshuffled votes, the techniques described in [8] maintain user privacy. We implement this combined system and analyze its performance.

We chose to use a system where the results of the voting system are only revealed once and cannot be modified after the results have been revealed. Note that until this result is revealed, votes may still be modified. Behind this decision is the fact that revealing votes before the final votes are submitted may leak information about how a voter voted. We prove a theorem below. Informally, this theorem states that revealing the tally while an election is running can reveal information about how individual voters voted. We use the following terminology to prove the theorem:

- Timestep: The number of times the tally has been published.
- T_i: The tally published at timestep i.
- V_i: The set of voters who contributed at timestep i. V_0 is the empty set.

Theorem 1. *If it is possible to determine who contributed at each timestep, any voting system that does not have all contributors vote between updating published results is not zero-knowledge.*

Proof. Assume to the contrary that such a voting system exists and is zero-knowledge. Then by definition of Zero-Knowledge, at all timesteps other than the final, no information about how each voter voted can be determined other than what would be given in the last timestep. Without loss of generality, we show that this can happen on timestep 1. The voters in V_1 must be responsible for the change from T_0 to T_1, and by our assumption, V_1 is a strict subset of the set of all voters. Because it is possible

to determine who voted at each timestep, V_1 is public. This means that the difference encountered can be traced back to a strict subset of the voters, and some information about how those voters voted is given away. This contradicts our assumption, and so, any voting system with these properties cannot be zero-knowledge. □

The assumption that it is possible to determine who voted is easily justifiable, as each vote submitted has to be verified, so that only votes from authorized voters are counted. In a system where vote re-submissions are allowed, this is almost unavoidable, as the overwritten vote must be linked to the identity of the voter so that it may be discarded.

This extends to a system where some voters can re-submit votes. However, an important implication of this is that all voters who intend to vote in the election must finish voting before the first tally is shown. Note that up to this point, voters may re-submit as many votes as they would like. Because forcing all voters to re-submit their votes is impractical in practice, this amounts to a system where only one tally is performed. We choose to use such a system to achieve the zero-knowledge property.

4.2 Description of the Steps

4.2.1 Initialization

To begin the scheme, each voting authority must create a secret private key (1a) and publish a corresponding public key (1b). The public keys, collectively (1c) are then stored by each user along with their private key (1d) for later computation.

4.2.2 Creating Shares for Threshold Decryption

To create the shares and global public key, each user takes in combination their own private key and the collection of public keys (2a) to create an intermediary value

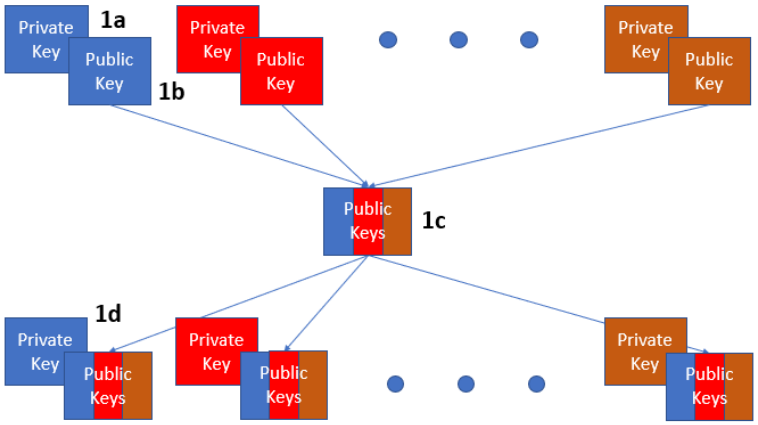

Figure 4.1: Analysis of the initialization step

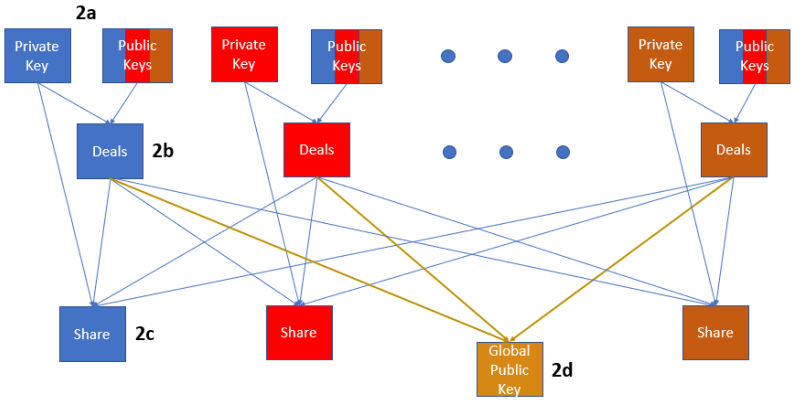

Figure 4.2: Analysis of the share creation step

referred to as a "deal" (2b), which is shared with each other voting authority. With a rather involved calculation that involves the deals and the authority's private key, each authority computes their own private share (2c) of the global private key. Each authority can also compute and publicize the same value for the public key (2d): $h = \prod_{i=1}^{n} s_i$. For more details on the calculations performed, readers are encouraged to consult [9][18].

4.2.3 Vote Collection

Each voter then would take the global public key (3a), asymmetrically encrypting their vote (3b). Each voter then submits their encrypted vote (3c).

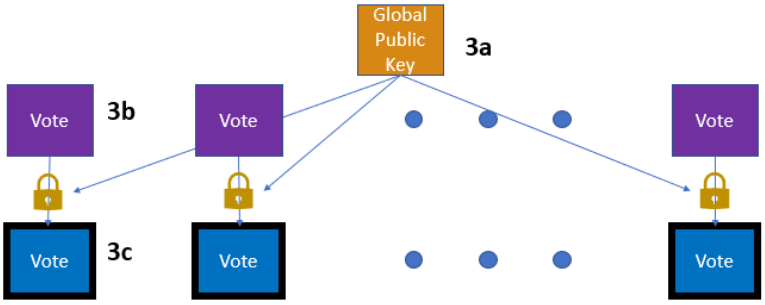

Figure 4.3: Analysis of the vote collection step

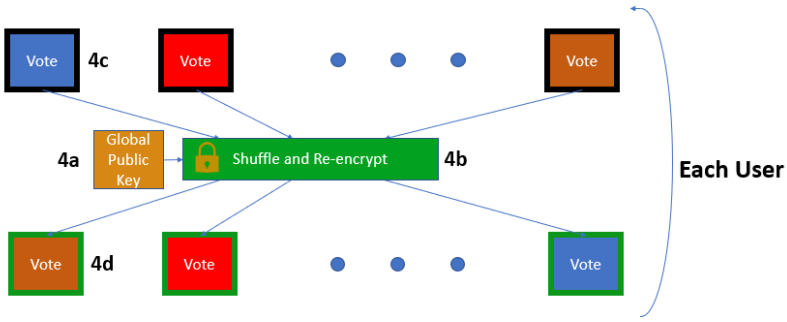

Figure 4.4: Analysis of the shuffle step

4.2.4 Shuffle and Re-encryption

Using the global public key (4a), the authorities would follow the procedure outlined by Neff to shuffle and re-encrypt (4b) the votes (4c). This is performed by each of the authorities to be thorough, though as long as at least one authority shuffles the list randomly and does not collude with the other authorities, the final permutation is random and untraceable. Having each authority shuffle guarantees this. Following this procedure results in a new list of encrypted votes (4d) guaranteed to have the same contents as the originals. Only the shuffler knows the permutation from the original encrypted votes to the new list. The formulas used in the shuffle itself are rather involved. Interested readers can find the formulas and descriptions in [15].

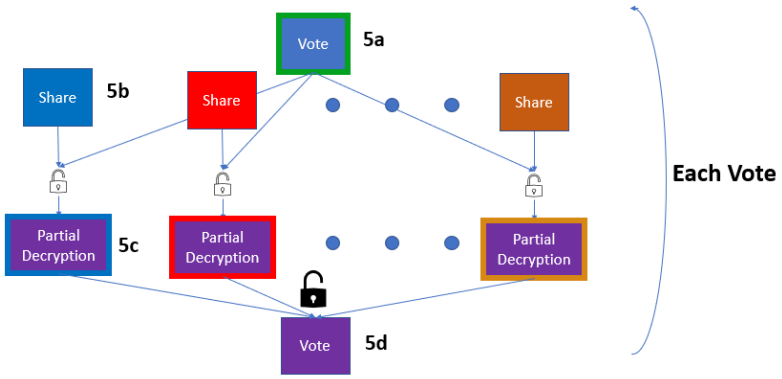

Figure 4.5: Analysis of the decryption step

4.2.5 Decryption

After each authority has shuffled the votes, the final list of votes is ready to be decrypted. Each vote (5a) is taken in combination with each authority's share of the private key (5b) to create a partial decryption for the vote (5c). These parial decryptions combine into a single decrypted vote (5d). In technical terms, the partial decryptions represent points on a polynomial function, with the decrypted vote as the y-intercept. This value is found via Lagrange interpolation [8].

4.3 Security Analysis

Security is an important issue to consider. Behind our system are several security proofs based upon the assumptions discussed in chapter 2, primarily the DDH assumption. The proofs we refer to can be found in the cited works [14][8]. The shuffle used provides unconditional accuracy– that is, no party, even one with unlimited computing power, could falsify the results of the shuffle. The privacy of the shuffle is guaranteed computationally– to determine the results of a shuffle, an observer would need to solve the decisional Diffie Helman problem. The threshold encryption and decryption provides computational privacy and computational accuracy. This is because the underlying ElGamal system used has these properties [21]. In combination, this

means our scheme is guaranteed to have computational privacy and computational accuracy. Any attack that could be done on this system must exploit a vulnerability in the implementation of the system, or it cannot manipulate submitted votes or determine the source of the votes.

An important note to make about this scheme is that it is subject to an attack whereby an attacker (Eve) may submit a vote with identical content to a vote submitted by another voter (Alice). All encrypted votes take the form of El-Gamal pairs: for a message M, this takes the form (g^y, Mg^{xy}), where g is the generator for the finite field, x is the private key and y is a random nonce. To decrypt this message, a user who knows x would raise g^y the the power of x, find its multiplicative inverse, and multiply by Mg^{xy}, yielding $Mg^{xy}g^{-xy} = M$. Note that another valid encryption for M is $(2g^y, 2Mg^{xy})$, constructing which only requires knowledge of the original (g^y, Mg^{xy}). Many more valid re-encryptions of M are possible to derive from the cyphertext. This can reveal information about how an honest voter voted. In an extreme example, say that every voter other than Eve submits a unique vote. Eve can copy Alice's vote, and, when votes are revealed, Alice's vote will be the only one with a duplicate. Eve can thus glean information about how Alice voted.

To counter this, we can simply have each voter commit to a vote before revealing it, by revealing a hash of the encrypted vote. After all commitments are submitted, each voter may reveal the original encrypted vote. This would prevent any vote duplication from taking place.

On the other hand, this type of attack represents a powerful form of delegation. If Bob wishes to delegate his vote to Alice, Alice could reveal her encrypted vote publicly before Bob submits his ballot. Bob could then re-encrypt Alice's vote, effectively delegating his vote. The only parties that would gain information about how Alice voted would be the parties that voted the same way as Alice, which could be seen as

an affordable tradeoff.

Chapter 5

SYSTEM IMPLEMENTATION AND PERFORMANCE RESULTS

5.1 Overview

We implemented our voting scheme in the Go language using the dedis kyber library [22]. Kyber is a library that provides cryptographic building blocks, such as points and operations in finite fields. It also includes the more advanced functionalities of the Neff cryptographic shuffle and Lagrange interpolation (to provide threshold decryption).

Our implementation adds several aspects to the building blocks provided by Kyber. To start, given Kyber's classes for manipulating finite fields, we created a function to encrypt a message, using the ElGamal cryptosystem. The most significant contribution in this category is the addition of arbitrarily lengthed messages (discussed later). Since the shuffle portion of the scheme is already implemented in Kyber, incorporating the shuffle into the voting protocol required an understanding of the scheme, but not much coding. The setup for the threshold cryptosystem was implemented by Kyber as well. However, in order to use the system they provided, our own modifications were needed. Specifically, we personally implemented a method of decrypting messages without revealing the global secret key. Originally, we implemented this system ourselves, but after discovering Kyber's functions for Lagrange interpolation, we re-wrote our code to simplify the codebase and take advantage of Kyber's builtin methods.

To test the runtime of different portions of our scheme, we simulated its execution on a single computer. We used the inbuilt "time" package of the Go language,

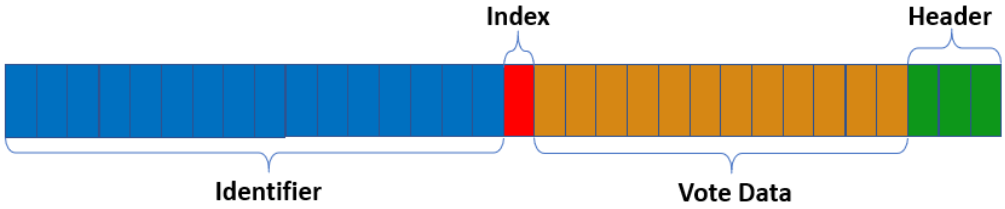

Figure 5.1: Organization of bytes in a single chunk of a longer message

recording the time before and after execution of the block of code associated with the procedure in question. The tests were preformed on a machine with an Intel® core™ i5-7200U CPU @ 2.50 GHz. All operations were performed sequentially, without multi-threading.

5.2 Encrypting Longer Messages

In our initial implementation, we supported only messages whose length was less than the block size required for field operations. The limit on the actual data that we can include in a block (embeddable data) was 29 bytes, which is equal to the length of the block size (32 bytes) minus the length of of header data (3 bytes). To create messages longer than 29 bytes, we split each message into several pieces, 12 bytes each, before shuffling: the first 12 bytes of the message would be held by the first message piece, the next few bytes by the next piece, and so on. To ensure that no information about individual messages is leaked from the message length, each user would have to contribute the same number of chunks as each other user. So, before voting, the maximum message length must be decided upon, and all messages need to be padded to be exactly that maximum length. The since the number of pieces the message could be split into could be quite large, the aggregated messages can be much larger than the original block size. To identify the message pieces, each message portion contained two additional sections along with the message contents.

The first was a fixed-length (16 byte) random number to uniquely identify which

long message the chunk belonged to (see Figure 5.1). The next section was a single byte reserved for the positional index of the message chunk. For instance, the first message chunk would have an index of 0, the second message chunk would have an index of 1, and so on. In the case of short messages, any non-message bytes under may be filled by using a standard padding scheme, or, if using ASCII encoding, by using the null character to indicate the early end of a string. After decrypting all of these message pieces, we would use these segments to recombine messages into coherent wholes. If we sort the message chunks in ascending order, it becomes trivial to recombine them. Say, for example, that we split each message into 8 pieces. Then, in the sorted list of message chunks, the first 8 chunks would all belong to the same message. In fact, every group of 8 chunks would correspond to a unique message. This is because each chunk begins with its identifier, immediately followed by the index byte. The most significant bytes, therefore, belong to the identifier. This clumps all similarly identified chunks together. The next significant byte is the positional byte, which will be used to order all like-identified chunks in ascending order.

This, of course, assumes that the numbers chosen to identify the message portions are truly random, and therefore unique. Such an assumption worked well for our proof-of-concept implementation, but is not comprehensive. From this point on, we describe a theoretical system whereby we can work around such assumptions. If identifiers were not unique, it would be obvious by the fact that two consecutive chunks in the sorted list have the same index bit. The only legitimate way this could occur is in the case of a re-encrypted ballot. In section 4.3, we discussed the possibility of duplicate messages. Such messages can be eliminated by having the voters commit their votes by publishing a hash of the vote. If duplicated votes are allowed, the contents of the entire chunks must be identical, not just the identifier and index. We simply need to count up the number of duplicate chunks while reconstructing

the vote, and count that vote that many times. Note that duplicating only some subset of these vote chunks instead of the whole vote is possible, but is easy to detect and work around. To do this, we only need to count the duplicates for each chunk, and take the minimum as the number of times to count the vote. For example, if the chunk with index 0 showed up 3 times, and all other chunks appeared twice, we would count the vote twice. The vote can be unambiguously reconstructed despite these duplicates, because there is only one message content available at each index. If two chunks with identical identifier and index number contain different contents, all chunks signified that id may safely be discarded, as this would indicate that all voters using that id have deviated from the scheme, either by not choosing a truly random number, or communicating that number with another voter outside of the bounds of the scheme. By ignoring these kinds of duplicate identifiers, we remove all incentive for any legitimate voter to use any number other than a random one, while effectively forcing any attacker wishing to invalidate another voter's vote to guess a 16 byte long number exactly. This puts the probability of invalidating another voter's vote on the order of $\frac{1}{2^{128}}$.

Other approaches for re-combining the votes were considered, such as message authentication codes and use of multiple independent shuffles. One more space-conservative approach would be to use a single hash to reconstruct the message instead of having an identifier and index for each portion. This would allow us to embed much more data into each message portion– all 29 bytes per message segment could hold message data. The cost of this would be a single additional encrypted chunk, which would hold a hash of the entire message. These hashes would be stored in a list separate from the message portions, and could be shuffled independently from them. To reconstruct the message, one could try piecing together all possible combinations of message segments. As an optimization to this scheme, multiple lists of message

segments could be kept, e.g. a list of message chunks with index 0, an list of message chunks with index 1, and so on. Even still, this method would be very slow for higher numbers of message partitions: the number of possible messages that could be constructed would be on the order of the number of messages to the power of the number of chunks each message is split into. As such, we deem it impractical for all but the smallest of scenarios. Another available technique that we considered for reconstructing messages would be to "chain" the messages together. To do this, we would set aside some bytes of a given data chunk to contain a signifier of the next message portion, such as a small hash. However, to be effective, this hash would have to take about as much space as a unique random identifier would, as well causing additional time to organize the messages. Ultimately, this technique proves less useful than id-based reconstruction. The optimization of separating differently-indexed messages into different lists before shuffling has very little impact on the scheme that uses id-based message reconstruction. The runtime complexity is unaffected (linear in the number of message chunks in both cases). However, this method could allow the saving of an additional byte per message chunk, as well as allowing more a message to be split into more than 256 pieces.

An advanced technique we leave as an open problem is the use of zero-knowledge proofs to remove any need for using encrypted data to reconstruct the messages. By constructing a zero-knowledge proof that two lists have been permuted in the same way, it would be possible to prove that all lists have been shuffled in the same way. To begin, each vote would be split into a set number of pieces. Each piece with the same positional index would be gathered into a list, resulting in a list for each number up to the number of pieces each message was split into. Crucially, the order of the message portions must be the same in each list: the first message portion in each list, when appended together, would create the first complete message, the second

message chunks in each list would combine to form the second message, and so on through the remainder of the lists. After all votes have been submitted, each of these lists would be permuted the same way, publishing a zero-knowledge proof that the first list was shuffled in the same way as the second, the second the same as the third, and so on. This would mean, for example, that if the first element of list 1 was moved to the fourth position on the shuffled list, every list would have the first element moved to the fourth position. After all shuffles are completed, reconstructing the original messages would be as simple as removing the first element from every list and appending them, repeating this process until all lists are empty. An equivalent formulation of this process would be to see all message portions as entries in a matrix, with each row representing a complete message, and each column representing one of our lists of like-indexed message portions. We can see the shuffling process as a re-ordering of these rows, provable by the fact that every column was shuffled in the same way. Such a technique would remove the relatively costly overhead of embedding identifiers into the data chunks, but constructing the required proofs that two shuffles are identical is a challenge exceeding the scope of our project.

The effect augmenting our scheme with longer messages has on the runtime is to multiply the runtime of all operations dependent on the number of messages by a constant factor, particularly the number of pieces the messages is split into. This is because all operations affected by this change are at worst linear to the size of the input they are given. Any new operations required can be done linearly in the number of message pieces as well. For example, by using a radix (or other linear-time) sort [7] to organize the messages, we can recombine the messages with a runtime linear in the number of message chunks, i.e. linear in the product of the number of messages and the number of chunks each message is split into.

5.3 Implementation Structure

Our code is organized into several helper functions, designed to be usable in a distributed setting. One of these functions is responsible for creating a public-private keypair, used to represent a voting authority's identity. Another function deals the shares of the private key to each of the authorities, as well as publishing the public key. Another decrypts a message using these shares, without compromising the integrity of the secret key. One of the most important functions is one which shuffles the list of encrypted votes, and checks that all calculations have been done correctly by verifying a published zero-knowledge proof of the shuffle.

5.4 Performance Measurements

5.4.1 Environment Creation

We implemented the Pedersen threshold system described in [18]. The threshold system we use requires significant setup time, but the cryptographic environment that results from this setup may be reused for multiple elections. For this reason, the runtimes of this section should be considered independently of the runtimes of subsequent sections. The environment creation step is said to begin immediately after the environment parameters (contributor count, threshold) are chosen, and ends immediately after the public key for the scheme is agreed upon and each contributor calculates his share of the private key. We show the runtime of the environment creation step as a function of the number of contributors. Results were averaged over 10 trials for each number of contributors. Figure 5.2 shows the runtime when the threshold is held constant. Figure 5.3 shows the runtime when the threshold varies with the number of contributors.

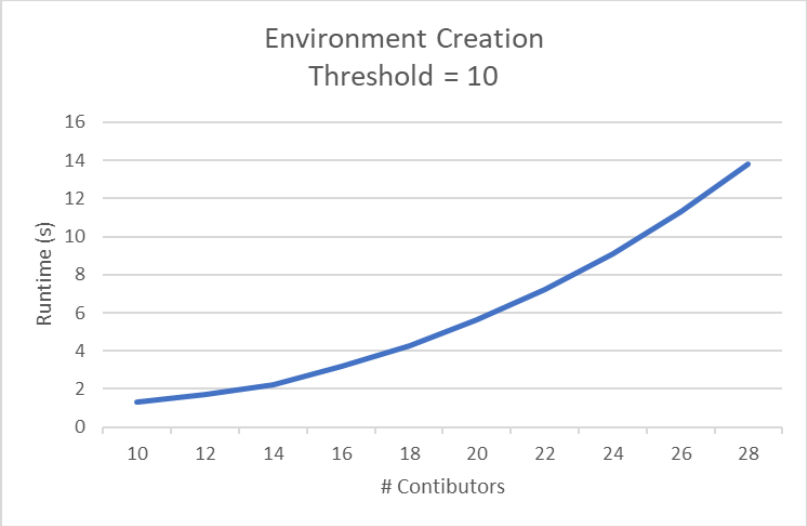

Figure 5.2: Environment creation runtime with constant threshold

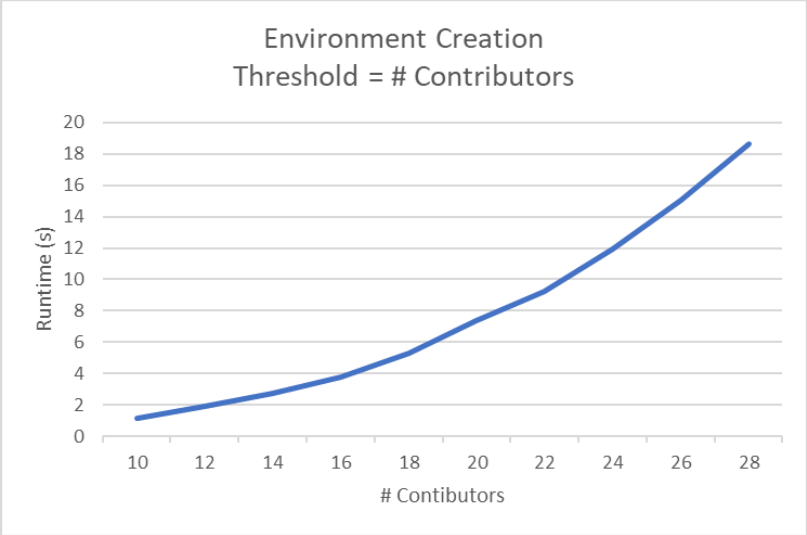

Figure 5.3: Environment creation runtime with variable threshold

In both 5.2 and 5.3, the runtime seems to be quadratic in the number of contributors. This matches the theory well, as each contributor must communicate with each other contributor to form the public key and private shares. This is the most taxing part of the scheme, because of how poorly it scales. However, for a setup scheme, these delays are manageably small, even with 28 authorities. We leave the problem open as to whether there exists a scheme with a lower computational complexity than

this.

5.4.2 Encryption

The encryption portion of the scheme is said to begin after the public key is given to the voters, when the voters being encrypting their votes, and ends after the list of pre-shuffle votes is published. Below we show the runtime of the encryption step as a function of the number of ballots. Results were averaged over 50 trials for each number of ballots. Figure 5.4 shows the total runtime of the section. Figure 5.5 shows the averaged runtime of processing a single ballot.

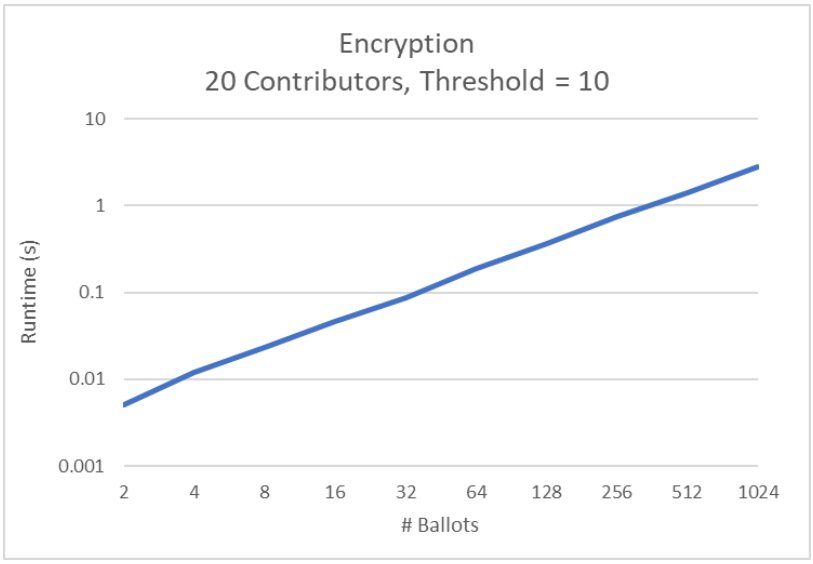

Figure 5.4: Encryption runtime

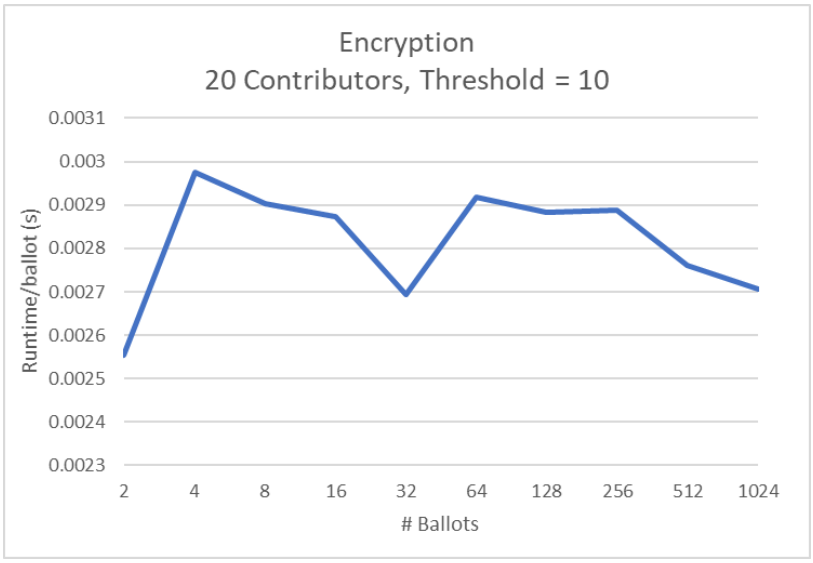

Figure 5.5: Encryption runtime, averaged

In Figure 5.4, the total runtime seems to be linear in the number of voters. This meshes well with theoretical analysis, as each over needs only a constant-time operation to encrypt their vote, illustrated by Figure 5.5. In practice, with voters encrypting their votes in parallel, likely on different machines, the runtime contribution of this step is negligible.

5.4.3 Shuffle

We implemented the Neff shuffle [15]. The shuffle portion of the scheme is said to begin when all the encrypted votes are given to the shuffler, and ends after a permutation of the votes has been provided, and the associated zero-knowledge proof of the validity of that shuffle has been checked and verified. Below we show the runtime of the shuffle step as a function of the number of ballots. Results were averaged over 50 trials for each number of ballots. Figure 5.6 shows the total runtime of the section.

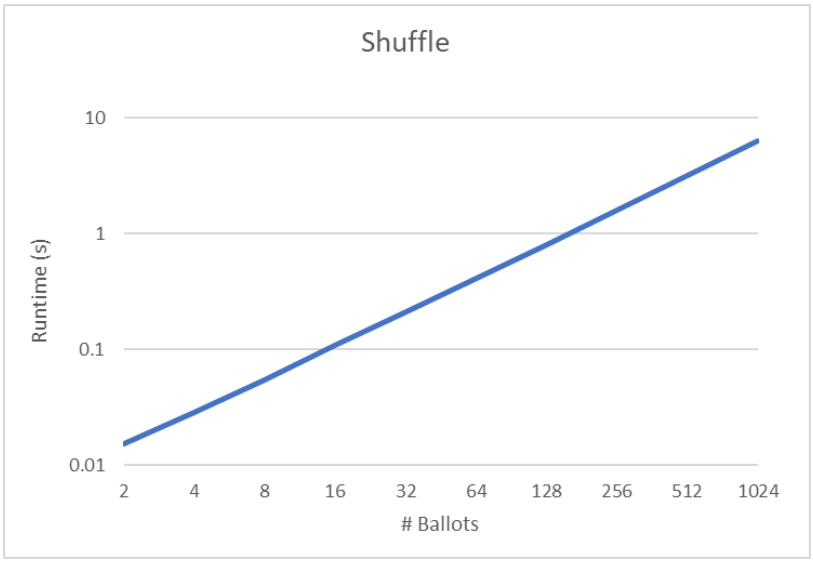

Figure 5.6: Shuffle runtime

In Figure 5.6, the total runtime seems to be linear in the number of votes. This meshes well with theoretical analysis. Note that here we only analyze the runtime of a single shuffle. In practice, since each authority needs to perform a shuffle that depends on the results of the previous shuffle (meaning no parallelization is possible), the total runtime would be roughly proportional to the product of the number of voters and the number of authorities.

5.4.4 Decryption

We implemented the decryption method described by Desmedt and Frankel [8]. The decryption portion of the scheme is said to begin after all the encrypted votes are eligible for decryption, and ends as soon as the plaintext of the last message is revealed. Below we show the runtime of the decryption step as a function of the number of ballots. Results were averaged over 50 trials for each number of ballots. Figure 5.7 shows the total runtime of the section. Figure 5.8 shows the averaged runtime of processing a single ballot.

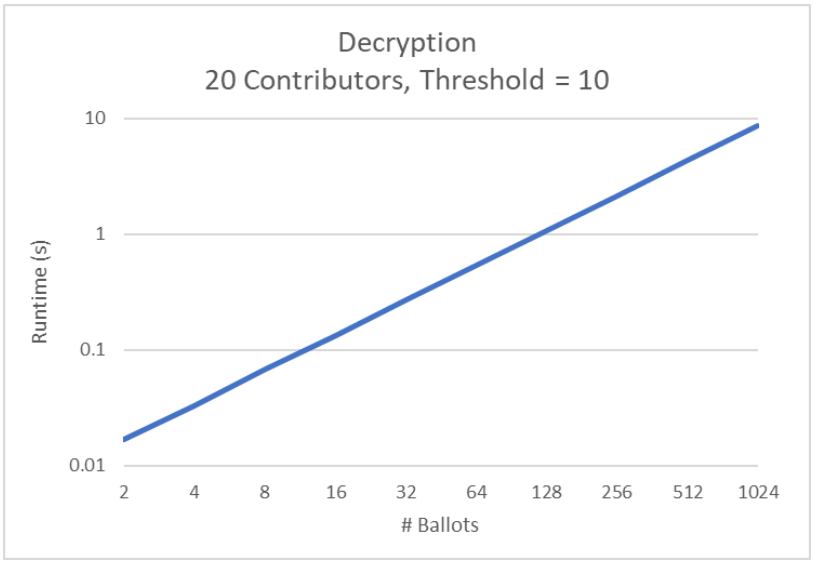

Figure 5.7: Decryption runtime

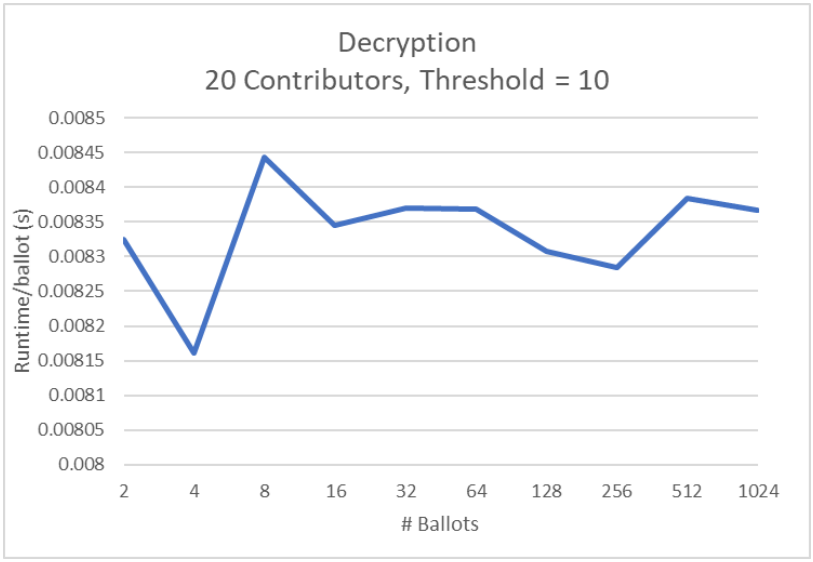

Figure 5.8: Decryption runtime, averaged

In Figure 5.7, the total runtime seems to be linear in the number of votes. This meshes well with theoretical analysis, as each vote needs to be decrypted individually. Note that this task may be done in parallel, with each vote only taking some constant amount of time proportional to the number of voting authorities, illustrated by Figure 5.8. Decryption seems to take longer than encryption in our tests. We

hypothesize that this is due to the asymmetric nature of encryption; only a single key value is needed to encrypt a message, but at least a threshold of the shares of the private key must be used to decrypt a message.

Chapter 6

CONCLUSION

6.1 Results

By our runtime analysis we conclude that our scheme is workable for sets of thousands of voters and dozens of authorities with reasonable delay. The greatest runtime contributor is the setup phase, which can be re-used for subsequent elections with the same set of authorities. The next most significant contibutor to runtime is the shuffle phase, which runs linearly to the product of the number of voters and the number of authorities. Although this potentially contributes quite significantly to the total runtime, each authority needs only consider the time it takes to shuffle a single list of votes, which takes time linear to the number of votes. Both encryption and decryption of the votes can be done quickly, with a runtime linear to the number of votes to process. By using zero-knowledge proofs, we have demonstrated the feasibility of a trust-free voting architecture.

Further work we accomplished was extending our scheme to include variable-length messages. Using this extension only expands the runtime by a constant factor.

6.2 Future Work

The algorithms we analyze provide a basis for a verifiable ballot collection scheme without a trusted party. Beyond what we accomplish, more work could be done to optimize various aspects of the algorithm. For instance, we put forth that the runtime of the procedure could be improved substantially by using an extension provided in [14] of shuffling DSA keys before the voting process begins. In this way, the identity of

each of the voters could be obscured, and only the fact that the voter was verified could be determined by the key submitted with each ballot. This has different challenges than the method we used, and would form an interesting modification of our scheme. For instance, one challenge that could arise is the difficulty of assigning each identity to a unique identifier. Another difficulty could regard the security concerns arising from the process of submitting a vote: if the origin of the vote could physically be traced to the voter's computer in another way, this could represent a significant security vulnerability. We conjecture that a system using DSA key shuffling would require further measures and careful planning to avoid exposing voter identities inadvertently.

We leave open the problem of efficiently assigning shuffles to certain voting authorities; having each authority be required to shuffle the votes in the scheme is a squandering of computation. By selecting some subset of authorites to shuffle, instead of the entire set, this computation time could be reduced. Some techniques that could be used to select the subset of shuffling authorities include random selection, like those used by cryptographic currencies for selecting peers or trusted nodes, such as Algorand [11]. Additional optimizations may be possible with pipelining the shuffles using clever applications of commitments, by having some users begin shuffling while another publishes the proof of a prior shuffle.

An especially promising open problem we propose is the creation of a zero-knowledge proof that two lists have been shuffled with the same permutation. Discussed in greater detail in section 5.2, this proof could be used to construct a more efficient rendition of a long-form message voting system.

REFERENCES

1. Adida, B., "Helios: Web-based open-audit voting.", in "USENIX security symposium", vol. 17, pp. 335–348 (2008).

2. Anane, R., R. Freeland and G. Theodoropoulos, "E-voting requirements and implementation", in "The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)", pp. 382–392 (IEEE, 2007).

3. Bocek, T., D. Peric, F. Hecht, D. Hausheer and B. Stiller, "Peervote: a decentralized voting mechanism for p2p collaboration systems", in "IFIP International Conference on Autonomous Infrastructure, Management and Security", pp. 56–69 (Springer, 2009).

4. Chaidos, P., V. Cortier, G. Fuchsbauer and D. Galindo, "Beleniosrf: A non-interactive receipt-free electronic voting scheme", in "Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security", pp. 1614–1625 (2016).

5. Chevallier-Mames, B., P.-A. Fouque, D. Pointcheval, J. Stern and J. Traoré, "On some incompatible properties of voting schemes", in "Towards Trustworthy Elections", pp. 191–199 (Springer, 2010).

6. Cranor, L. F. and R. K. Cytron, "Sensus: A security-conscious electronic polling system for the internet", in "Proceedings of the Thirtieth Hawaii International Conference on system sciences", vol. 3, pp. 561–570 (IEEE, 1997).

7. Davis, I. J., "A fast radix sort", The computer journal **35**, 6, 636–642 (1992).

8. Desmedt, Y. and Y. Frankel, "Threshold cryptosystems", in "Conference on the Theory and Application of Cryptology", pp. 307–315 (Springer, 1989).

9. Desmedt, Y. G., "Threshold cryptography", European Transactions on Telecommunications **5**, 4, 449–458 (1994).

10. Fujioka, A., T. Okamoto and K. Ohta, "A practical secret voting scheme for large scale elections", in "International Workshop on the Theory and Application of Cryptographic Techniques", pp. 244–251 (Springer, 1992).

11. Gilad, Y., R. Hemo, S. Micali, G. Vlachos and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies", in "Proceedings of the 26th Symposium on Operating Systems Principles", pp. 51–68 (2017).

12. Ibrahim, S., M. Kamat, M. Salleh and S. R. A. Aziz, "Secure e-voting with blind signature", in "4th National Conference of Telecommunication Technology, 2003. NCTT 2003 Proceedings.", pp. 193–197 (IEEE, 2003).

13. Lamport, L. *et al.*, "Paxos made simple", ACM Sigact News **32**, 4, 18–25 (2001).

14. Neff, C. A., "A verifiable secret shuffle and its application to e-voting", in "Proceedings of the 8th ACM conference on Computer and Communications Security", pp. 116–125 (2001).

15. Neff, C. A., "Verifiable mixing (shuffling) of elgamal pairs", VHTi Technical Document, VoteHere, Inc p. 112 (2003).

16. Ongaro, D. and J. Ousterhout, "In search of an understandable consensus algorithm", in "2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)", pp. 305–319 (2014).

17. Oo, H. N. and A. M. Aung, "Implementation and analysis of secure electronic voting system", Int. J. Sci. Technol. Res **2**, 3, 158–161 (2013).

18. Pedersen, T. P., "A threshold cryptosystem without a trusted party", in "Workshop on the Theory and Application of of Cryptographic Techniques", pp. 522–526 (Springer, 1991).

19. Sako, K. and J. Kilian, "Receipt-free mix-type voting scheme", in "International Conference on the Theory and Applications of Cryptographic Techniques", pp. 393–403 (Springer, 1995).

20. Sastry, N. K. and D. Wagner, *Verifying security properties in electronic voting machines* (University of California, Berkeley, 2007).

21. Tsiounis, Y. and M. Yung, "On the security of elgamal based encryption", in "International Workshop on Public Key Cryptography", pp. 117–134 (Springer, 1998).

22. Wolinsky, D. I., B. Ford, N. Gailly, J. R. Allen, P. Jovanovic, L. Gasser, F. Huang, M. Raynal, G. Bosson, D. Visher, I. Khoffi, kc1212, I. Tamas, Christophe, C. Cook, rkopiga, N. Kocher, G. Narula, V. Rousset, T. Bowers, K. Nikitin, A. Lu, J. A. G. de Sa Sousa, N. Artem, liminalsheeps, franklinsxx, ferhat elmas and S. Celi, "kyber", `https://github.com/dedis/kyber` (2020).

APPENDIX A

RAW DATA

Environment Creation runtimes
Threshold = 10
Independent variable: Contributor Count

10	12	14	16	18	20	22	24	26	28
1.2257	2.1089	2.2180	3.1477	4.2310	5.5955	7.1922	9.0628	12.1974	13.6958
1.6171	1.8353	2.2336	3.1178	4.2312	5.5886	7.3434	9.0363	11.3024	14.1441
1.3111	1.8341	2.2195	3.1076	4.2487	5.5788	7.1896	9.0516	11.2321	13.6326
1.1518	1.8222	2.2471	3.1059	4.2604	5.7211	7.2344	9.0450	11.1472	13.6526
1.4114	1.5769	2.2287	3.2660	4.2479	5.7364	7.2403	9.0734	11.2431	13.8114
1.1564	1.6112	2.2277	3.1241	4.2533	5.5928	7.2700	9.3264	11.1620	13.7001
1.1467	1.6605	2.2125	3.5263	4.2364	5.6233	7.2162	9.0634	11.1844	13.7844
1.2671	1.5051	2.2205	3.1430	4.2491	5.5929	7.2190	9.0588	11.2122	13.6136
1.4220	1.5030	2.2550	3.1937	4.3585	5.6121	7.1803	9.1381	11.1657	13.9880
1.3387	1.5201	2.2742	3.1196	4.2953	5.5733	7.2345	9.0706	11.1411	14.2908

Environment Creation runtimes
Threshold = Contributor Count
Independent variable: Contributor Count

10	12	14	16	18	20	22	24	26	28
1.0412	1.7722	3.6695	3.8434	5.2663	8.2271	9.2264	11.7783	14.8998	18.6699
1.0239	1.8678	2.6508	3.7367	5.2233	7.5577	9.1897	12.1658	16.0694	18.5131
1.1032	2.0266	2.5711	3.7522	5.2050	7.0488	9.1691	12.1890	14.8754	18.7311
1.0607	1.7321	2.5907	3.7501	5.2551	7.0147	9.2036	11.9595	14.8142	20.0535
1.1688	1.7469	2.5921	3.7417	5.2819	7.3482	9.2929	11.8045	15.5334	18.3889
1.2694	1.7442	2.5648	3.7296	5.1978	7.7507	9.4460	11.7538	14.8695	18.4902
1.0178	1.7672	2.5921	3.7485	5.4290	7.2688	9.3188	12.0708	14.8451	18.3943
1.0114	1.8757	2.5767	3.8044	5.2214	7.1258	9.1984	11.8465	14.8399	18.4882
1.1230	1.8891	2.5930	3.7107	5.4303	7.0951	9.2052	11.7738	14.9870	18.3941
1.5680	2.3199	2.6261	3.7554	5.6941	7.0003	9.1562	11.7581	15.0362	18.4099

Encryption runtimes
Threshold = 10
Contributor Count = 20
Independent variable: Ballot Count

2	4	8	16	32	64	128	256	512	1024
0.0070	0.0130	0.0279	0.0379	0.0738	0.1711	0.3501	0.6308	1.3973	2.6901
0.0070	0.0090	0.0220	0.0419	0.0638	0.1667	0.3650	0.7537	1.4064	2.7109
0.0049	0.0140	0.0156	0.0453	0.0718	0.1795	0.3212	0.7137	1.3744	2.7971
0.0080	0.0110	0.0289	0.0459	0.1097	0.1785	0.3691	0.7057	1.4478	2.7058
0.0050	0.0050	0.0220	0.0579	0.0883	0.1981	0.4089	0.7178	1.4292	2.8582
0.0030	0.0160	0.0240	0.0529	0.1347	0.2059	0.3491	0.7176	1.3526	2.7641
0.0050	0.0120	0.0239	0.0589	0.0973	0.1605	0.3996	0.6588	1.4102	2.8163
0.0050	0.0150	0.0219	0.0519	0.0764	0.1960	0.4318	0.7357	1.5112	2.7067
0.0030	0.0110	0.0209	0.0594	0.0827	0.2047	0.3491	0.7392	1.4553	2.7308
0.0020	0.0060	0.0229	0.0409	0.0638	0.1915	0.3276	0.7643	1.4886	2.7054
0.0010	0.0159	0.0180	0.0330	0.0853	0.1801	0.3496	0.7336	1.4321	2.8486
0.0120	0.0100	0.0229	0.0350	0.0868	0.1651	0.3681	0.7899	1.3590	2.9627
0.0010	0.0139	0.0240	0.0409	0.0958	0.1935	0.3426	0.7480	1.3777	2.7704
0.0050	0.0100	0.0200	0.0448	0.0584	0.1900	0.3447	0.7095	1.4398	2.7261
0.0080	0.0150	0.0249	0.0359	0.0788	0.2677	0.4369	0.7357	1.3273	2.7899
0.0040	0.0180	0.0229	0.0439	0.0967	0.1890	0.3675	0.7710	1.4014	2.7461
0.0050	0.0110	0.0229	0.0359	0.0918	0.1746	0.3910	0.7525	1.4597	2.8063
0.0109	0.0070	0.0130	0.0489	0.0719	0.1685	0.3900	0.7241	1.4268	2.8150
0.0040	0.0209	0.0239	0.0299	0.0928	0.1805	0.3780	0.7715	1.4563	2.7749
0.0040	0.0230	0.0259	0.0419	0.0878	0.1855	0.3610	0.7326	1.3373	2.7795
0.0120	0.0140	0.0170	0.0564	0.1007	0.2170	0.4069	0.6982	1.4497	2.7208
0.0030	0.0060	0.0195	0.0469	0.0938	0.1685	0.3650	0.7546	1.3554	2.7356
0.0020	0.0080	0.0310	0.0320	0.0987	0.1965	0.3780	0.7416	1.4527	2.9586
0.0030	0.0130	0.0240	0.0359	0.1027	0.1710	0.3541	0.6817	1.4749	2.7821
0.0060	0.0130	0.0289	0.0459	0.0628	0.2035	0.3700	0.7670	1.4806	2.8574

0.0060	0.0080	0.0170	0.0554	0.0898	0.1935	0.3616	0.7266	1.3632	2.7512
0.0030	0.0130	0.0160	0.0434	0.0917	0.1626	0.3709	0.8165	1.2933	2.8025
0.0020	0.0275	0.0280	0.0519	0.0793	0.1770	0.3790	0.7768	1.4111	2.6353
0.0020	0.0210	0.0270	0.0559	0.1077	0.1815	0.3658	0.7969	1.4815	2.8809
0.0050	0.0090	0.0200	0.0534	0.0798	0.1626	0.3446	0.6663	1.4255	2.7898
0.0040	0.0150	0.0200	0.0508	0.0828	0.2044	0.3920	0.6995	1.4454	2.7636
0.0050	0.0110	0.0140	0.0459	0.1067	0.1835	0.3675	0.7335	1.3640	2.8897
0.0060	0.0080	0.0270	0.0369	0.0808	0.1885	0.3580	0.7261	1.3419	2.7615
0.0060	0.0080	0.0239	0.0429	0.1052	0.1825	0.3917	0.7435	1.4482	2.6281
0.0030	0.0100	0.0220	0.0389	0.0728	0.1476	0.3836	0.7665	1.3476	2.7107
0.0060	0.0140	0.0309	0.0618	0.0738	0.2064	0.3850	0.7447	1.3254	2.7455
0.0040	0.0069	0.0230	0.0509	0.0748	0.1945	0.3566	0.7286	1.3860	2.7101
0.0070	0.0140	0.0160	0.0409	0.0788	0.1831	0.3751	0.7216	1.6327	2.7769
0.0069	0.0060	0.0280	0.0379	0.1057	0.1622	0.3501	0.7580	1.3292	2.7055
0.0070	0.0080	0.0213	0.0479	0.0977	0.2074	0.3032	0.7913	1.4412	2.8075
0.0030	0.0100	0.0180	0.0648	0.0997	0.2060	0.3615	0.7944	1.5106	2.7468
0.0030	0.0060	0.0190	0.0439	0.0794	0.1864	0.3632	0.7500	1.3387	2.6256
0.0090	0.0160	0.0279	0.0504	0.0928	0.1835	0.4290	0.8238	1.4020	2.7612
0.0100	0.0040	0.0249	0.0329	0.0957	0.1645	0.3362	0.8130	1.4148	2.7320
0.0020	0.0160	0.0164	0.0379	0.0683	0.1656	0.3865	0.6533	1.3896	2.8421
0.0050	0.0079	0.0259	0.0559	0.0628	0.1815	0.3516	0.7243	1.4178	2.8257
0.0020	0.0120	0.0259	0.0459	0.0733	0.2319	0.3916	0.7904	1.4315	2.8362
0.0080	0.0060	0.0350	0.0598	0.0673	0.2244	0.3849	0.7605	1.3035	2.7318
0.0020	0.0129	0.0220	0.0498	0.0828	0.1713	0.3661	0.7252	1.4416	2.8589
0.0080	0.0150	0.0438	0.0419	0.0938	0.1826	0.3261	0.6807	1.4933	2.6835

Shuffle runtimes
Independent variable: Ballot Count

2	4	8	16	32	64	128	256	512	1024
0.0140	0.0289	0.0578	0.1067	0.2024	0.4069	0.8114	1.5944	3.1871	6.2665
0.0150	0.0270	0.0588	0.1047	0.2075	0.4048	0.8130	1.6189	3.1684	6.3210
0.0150	0.0254	0.0569	0.1053	0.2294	0.4035	0.7958	1.5922	3.1385	6.4904
0.0180	0.0279	0.0529	0.1037	0.2573	0.4079	0.7941	1.5989	3.1621	6.4752
0.0160	0.0299	0.0538	0.1297	0.2294	0.3980	0.8099	1.5997	3.2087	6.3769
0.0170	0.0289	0.0538	0.1277	0.2025	0.4021	0.7935	1.5948	3.1487	6.3164
0.0180	0.0259	0.0524	0.1087	0.2144	0.4468	0.8004	1.5938	3.1424	6.3714
0.0150	0.0269	0.0599	0.1053	0.2029	0.4040	0.8307	1.5804	3.1556	6.3180
0.0135	0.0269	0.0549	0.1042	0.2005	0.4078	0.8011	1.5811	3.1788	6.2947
0.0140	0.0289	0.0564	0.1038	0.2045	0.4030	0.7990	1.5860	3.1780	6.2663
0.0160	0.0290	0.0539	0.1037	0.2035	0.4099	0.8094	1.5794	3.1948	6.2798
0.0135	0.0279	0.0594	0.1077	0.2104	0.4070	0.8020	1.5944	3.1513	6.6402
0.0150	0.0269	0.0538	0.1057	0.2065	0.3999	0.8029	1.5909	3.1625	6.2794
0.0219	0.0269	0.0788	0.1067	0.2100	0.5116	0.7994	1.5754	3.1494	6.3203
0.0140	0.0269	0.0598	0.1082	0.2034	0.4688	0.8234	1.5945	3.1692	6.2986
0.0140	0.0279	0.0549	0.1077	0.2115	0.3970	0.8150	1.5890	3.1678	6.2773
0.0150	0.0259	0.0539	0.1037	0.2054	0.3969	0.8213	1.5729	3.1723	6.2900
0.0140	0.0269	0.0539	0.1028	0.2030	0.3985	0.8101	1.5984	3.1723	6.2718
0.0150	0.0350	0.0529	0.1057	0.2151	0.4010	0.8263	1.5935	3.4522	6.2775
0.0140	0.0299	0.0559	0.1037	0.2025	0.4478	0.8094	1.6096	3.1510	6.2814
0.0140	0.0269	0.0504	0.1042	0.2099	0.4069	0.8339	1.6369	3.1710	6.2878
0.0150	0.0269	0.0548	0.1017	0.2105	0.4074	0.8004	1.5980	3.4751	6.3586
0.0150	0.0269	0.0519	0.1077	0.2135	0.4479	0.8171	1.5906	3.1654	6.3765
0.0150	0.0289	0.0508	0.1037	0.2095	0.4030	0.8058	1.5727	3.2015	6.3060
0.0139	0.0254	0.0499	0.1207	0.2095	0.4045	0.7948	1.5945	3.1753	6.3553

0.0149	0.0309	0.0508	0.1088	0.2484	0.3979	0.8129	1.5955	3.3551	6.3204
0.0140	0.0259	0.0508	0.1038	0.2205	0.4198	0.8079	1.6035	3.1471	6.2699
0.0160	0.0429	0.0498	0.1067	0.2069	0.3989	0.8244	1.5837	3.1486	6.3139
0.0140	0.0289	0.0529	0.1107	0.2134	0.4374	0.8018	1.6006	3.1823	6.3350
0.0150	0.0260	0.0509	0.1062	0.2085	0.4094	0.7993	1.5909	3.2272	6.2609
0.0200	0.0329	0.0518	0.1097	0.2064	0.4029	0.8059	1.5906	3.1466	6.2864
0.0170	0.0309	0.0514	0.1072	0.2065	0.4106	0.7994	1.6168	3.1886	6.3161
0.0150	0.0269	0.0508	0.1004	0.2076	0.4040	0.8259	1.5964	3.1904	6.3215
0.0219	0.0329	0.0509	0.1037	0.2134	0.4020	0.8130	1.5828	3.1366	6.3238
0.0149	0.0269	0.0528	0.1057	0.2114	0.4088	0.8098	1.5974	3.1386	6.2808
0.0140	0.0289	0.0529	0.1067	0.2125	0.4060	0.7981	1.5749	3.1492	6.3386
0.0140	0.0279	0.0518	0.1088	0.2045	0.4129	0.8068	1.5925	3.1588	6.2780
0.0160	0.0269	0.0509	0.1047	0.2030	0.4089	0.8088	1.5829	3.1408	6.3078
0.0140	0.0260	0.0503	0.1038	0.2184	0.4174	0.7965	1.5928	3.1665	6.3324
0.0140	0.0259	0.0509	0.1028	0.2154	0.4539	0.8009	1.6033	3.1485	6.3011
0.0210	0.0314	0.0509	0.1108	0.2145	0.4109	0.8014	1.5753	3.1454	6.4663
0.0149	0.0259	0.0499	0.1087	0.2045	0.4035	0.8072	1.5963	3.1490	6.3083
0.0140	0.0299	0.0549	0.1087	0.2184	0.4070	0.8219	1.6098	3.1643	6.3126
0.0160	0.0259	0.0504	0.1067	0.2125	0.4538	0.8069	1.5900	3.5338	6.2919
0.0150	0.0269	0.0539	0.1117	0.2095	0.4164	0.8150	1.5910	3.1507	6.4092
0.0139	0.0309	0.0509	0.1263	0.2254	0.4034	0.8121	1.5759	3.1884	6.2934
0.0149	0.0269	0.0509	0.1097	0.2055	0.4199	0.8093	1.6094	3.1471	6.3059
0.0150	0.0340	0.0519	0.1053	0.2114	0.4638	0.8209	1.5875	3.1844	6.2671
0.0150	0.0259	0.0499	0.1137	0.2045	0.4090	0.8278	1.5854	3.1491	6.3169
0.0139	0.0289	0.0758	0.1068	0.2124	0.4184	0.8189	1.5843	3.1544	6.2848

Decryption runtimes
Threshold = 10
Contributor Count = 20
Independent variable: Ballot Count

2	4	8	16	32	64	128	256	512	1024
0.0159	0.0309	0.0638	0.1407	0.2656	0.5177	1.1022	2.1031	4.2517	8.5927
0.0160	0.0369	0.0618	0.1382	0.2560	0.5226	1.0673	2.1221	4.2334	8.6115
0.0180	0.0299	0.0648	0.1301	0.3490	0.5178	1.0813	2.1033	4.3058	8.5184
0.0190	0.0344	0.0628	0.1336	0.3212	0.5148	1.0443	2.1020	4.2770	8.6280
0.0200	0.0309	0.0663	0.1496	0.2857	0.5247	1.0683	2.1620	4.3142	8.4876
0.0169	0.0309	0.0629	0.1322	0.2633	0.5258	1.0462	2.1476	4.2754	8.5453
0.0180	0.0319	0.0628	0.1277	0.2667	0.5402	1.0458	2.1011	4.2822	8.5079
0.0160	0.0309	0.0638	0.1312	0.2564	0.5206	1.0813	2.1406	4.2975	8.6821
0.0160	0.0349	0.0628	0.1257	0.2598	0.5196	1.0409	2.1102	4.2506	8.4852
0.0170	0.0309	0.0643	0.1356	0.2573	0.5288	1.1036	2.1265	4.2480	8.6459
0.0160	0.0319	0.0624	0.1317	0.2583	0.5201	1.0579	2.1485	4.3126	8.5217
0.0160	0.0339	0.0648	0.1322	0.2624	0.5415	1.0921	2.0927	4.2644	8.5933
0.0159	0.0309	0.0618	0.1327	0.3092	0.5237	1.0443	2.0999	4.2359	8.5414
0.0155	0.0359	0.0808	0.1337	0.2663	0.6662	1.1016	2.1399	4.2489	8.4970
0.0149	0.0309	0.0658	0.1293	0.2614	0.6243	1.0518	2.1238	4.3096	8.5564
0.0160	0.0299	0.0618	0.1287	0.2588	0.5236	1.0584	2.1006	4.2322	8.6190
0.0195	0.0349	0.0678	0.1317	0.2603	0.5183	1.0563	2.1221	4.2348	8.5258
0.0160	0.0309	0.0618	0.1301	0.2584	0.5206	1.0597	2.0983	4.2659	8.5394
0.0160	0.0319	0.0648	0.1317	0.2613	0.5251	1.0592	2.1154	4.2489	8.5076
0.0190	0.0359	0.0718	0.1322	0.2763	0.5335	1.0576	2.0985	4.2788	8.5874
0.0160	0.0310	0.0658	0.1352	0.2633	0.5183	1.0542	2.1055	4.3124	8.6217
0.0150	0.0329	0.0678	0.1307	0.2624	0.5236	1.0553	2.0952	4.3038	8.6371
0.0150	0.0321	0.0698	0.1326	0.2603	0.5312	1.0553	2.0992	4.2764	8.6397
0.0190	0.0329	0.0728	0.1486	0.2633	0.5266	1.0605	2.1199	4.2779	8.5160
0.0160	0.0329	0.0698	0.1366	0.2623	0.5186	1.0605	2.1111	4.3088	8.5782

0.0150	0.0314	0.0698	0.1331	0.2693	0.5221	1.0642	2.1320	4.2870	8.6294
0.0199	0.0329	0.0698	0.1326	0.2593	0.5177	1.0918	2.1028	4.2983	8.5822
0.0160	0.0439	0.0700	0.1346	0.2613	0.5412	1.0543	2.0971	4.2260	8.5306
0.0160	0.0309	0.0678	0.1297	0.2698	0.5230	1.0578	2.1767	4.2441	8.5585
0.0159	0.0339	0.0738	0.1317	0.2611	0.5671	1.0593	2.1019	4.2704	8.5633
0.0170	0.0329	0.0688	0.1327	0.2613	0.5273	1.0627	2.1295	4.5272	8.6243
0.0159	0.0339	0.0708	0.1347	0.2638	0.5211	1.0668	2.1106	4.3354	8.5660
0.0150	0.0309	0.0848	0.1326	0.2614	0.5222	1.0577	2.1175	4.3196	8.6378
0.0180	0.0309	0.0698	0.1386	0.2633	0.5241	1.0783	2.1124	4.2635	8.6740
0.0160	0.0309	0.0698	0.1317	0.2619	0.5746	1.0699	2.1242	4.2676	8.6248
0.0160	0.0310	0.0658	0.1357	0.2608	0.5256	1.0697	2.1684	4.2732	8.5239
0.0179	0.0339	0.0678	0.1331	0.2593	0.5212	1.0442	2.1000	4.3346	8.5457
0.0160	0.0329	0.0708	0.1337	0.2633	0.5366	1.0575	2.2529	4.3301	8.5693
0.0160	0.0334	0.0668	0.1346	0.2605	0.5186	1.0593	2.1154	4.2953	8.4945
0.0150	0.0309	0.0738	0.1316	0.2673	0.5362	1.1087	2.0979	4.2961	8.4966
0.0219	0.0339	0.0668	0.1306	0.2593	0.5291	1.0726	2.0957	4.2989	8.6507
0.0150	0.0350	0.0644	0.1287	0.2639	0.5226	1.0499	2.1136	4.2750	8.4764
0.0160	0.0309	0.0688	0.1287	0.2672	0.5287	1.0553	2.1299	4.3070	8.5698
0.0167	0.0359	0.0659	0.1297	0.2628	0.5480	1.0444	2.1211	4.6176	8.5008
0.0160	0.0310	0.0663	0.1307	0.3072	0.5282	1.0558	2.1131	4.2988	8.5809
0.0160	0.0359	0.0698	0.1611	0.2574	0.6070	1.0517	2.1077	4.3098	8.4712
0.0190	0.0309	0.0648	0.1346	0.2633	0.5196	1.0546	2.1422	4.2444	8.5502
0.0159	0.0309	0.0678	0.1298	0.2633	0.6092	1.0624	2.1055	4.2971	8.5495
0.0159	0.0309	0.0668	0.1346	0.2693	0.5305	1.0575	2.1076	4.2783	8.6472
0.0170	0.0309	0.0698	0.1256	0.2688	0.5296	1.0544	2.1664	4.2909	8.5900

APPENDIX B

SOURCE CODE

B.1 Operation Instructions

To run a test for the default system, run main in testComplete.go. To run a test for longer messages, remove testComplete.go from the configuration and rename the main2 in testLongMessage.go to main before running it. Only one of testComplete.go or testLongMessage.go needs to be present for either to run. The other files hold the functionality needed to facilitate the tests.

B.2 testComplete.go

```
package main

import (
"fmt"
"log"
"os"
"strconv"
"time"
)

func main() {
// the filepath
environmentFilepath := "EnvironmentData0.txt"

shuffleFilepath := "ShuffleData0.txt"
encryptionFilepath := "EncryptionData0.txt"
decryptionFilepath := "DecryptionData0.txt"

// open a file for recording the test data
environmentFile, err := os.Create(environmentFilepath)
check(err)                    // make sure nothing's wrong
defer environmentFile.Close() // close the file eventually

// open a file for recording the test data
shuffleFile, err := os.Create(shuffleFilepath)
check(err)                // make sure nothing's wrong
defer shuffleFile.Close() // close the file eventually

// open a file for recording the test data
encryptionFile, err := os.Create(encryptionFilepath)
check(err)                   // make sure nothing's wrong
defer encryptionFile.Close() // close the file eventually
```

```
// open a file for recording the test data
decryptionFile, err := os.Create(decryptionFilepath)
check(err)                    // make sure nothing's wrong
defer decryptionFile.Close() // close the file eventually

n := 20 // the number of contributors in the scheme
t := 10 // the threshold

// record the parameters of the test
_, err = shuffleFile.WriteString("Environment Creation:\n")
check(err)
_, err = shuffleFile.WriteString("Using " + strconv.Itoa(n) +
" contributors with threshold " + strconv.Itoa(t) + "\n\n")
check(err)

// record the parameters of the test
_, err = encryptionFile.WriteString("Environment Creation:\n")
check(err)
_, err = encryptionFile.WriteString("Using " + strconv.Itoa(n) +
" contributors with threshold " + strconv.Itoa(t) + "\n\n")
check(err)

// record the parameters of the test
_, err = decryptionFile.WriteString("Environment Creation:\n")
check(err)
_, err = decryptionFile.WriteString("Using " + strconv.Itoa(n) +
" contributors with threshold " + strconv.Itoa(t) + "\n\n")
check(err)

for i := 0; i < 10; i++ {

newContributorCount := n - 10 + i*2
t = newContributorCount
// record the parameters of the test
_, err = environmentFile.WriteString("Environment Creation:\n")
check(err)
_, err = environmentFile.WriteString("Using " +
strconv.Itoa(newContributorCount) + " contributors with threshold "
+ strconv.Itoa(t) + "\n\n")
check(err)

for j := 0; j < 10; j++ {

start := time.Now() // start timer

// create the environment for the tests
createThresholdShares(newContributorCount, t)
```

```
// this is where the bulk of the time is spent:
// overhead for creating the system

// end timer
elapsed := time.Since(start)
// log the time
log.Printf("Environment Creation took %s", elapsed)
_, err = environmentFile.WriteString(
fmt.Sprintf("%.5f", elapsed.Seconds()) + "\n")
// record the time in file
check(err)
}
}

start := time.Now() // start timer

// create the environment for the tests
// this is where the bulk of the time is spent: overhead for creating the system
shares := createThresholdShares(n, t)
// each user's share will contain the public key
// Since these are all the same, we choose to use the copy at index 0 arbitrarlily
publicKey := shares[0].Public()

elapsed := time.Since(start)                          // end timer
log.Printf("Environment Creation took %s", elapsed) // log the time
// _, err = file.WriteString(elapsed.String() + "\n")  // record the time in file
// check(err)

// check each with an exponentially increasing number of ballots
for ballotCount := 2; ballotCount < 1050; ballotCount *= 2 {

// state the number of ballots encrypted
_, err = shuffleFile.WriteString("\nShuffling " + strconv.Itoa(ballotCount)
+ " ballots\n")
check(err)
// state the number of ballots encrypted
_, err = encryptionFile.WriteString("\nEncrypting " + strconv.Itoa(ballotCount)
+ " ballots\n")
check(err)
// state the number of ballots encrypted
_, err = decryptionFile.WriteString("\nDecrypting " + strconv.Itoa(ballotCount)
+ " ballots\n")
check(err)

for i := 0; i < 50; i++ { // do 50 tests

start := time.Now() // start timer
```

```
// doThresholdTest(ballotCount, n, t) // do test

// generate messages
messages, elGamal1, elGamal2 :=
generateMessageEncryptions(ballotCount, publicKey)

// end timer
elapsed := time.Since(start)
// log the time
log.Printf("Encryption took %s", elapsed)
_, err = encryptionFile.WriteString(fmt.Sprintf(
"%.5f", elapsed.Seconds()) + "\n") // record the time in file
check(err)
start = time.Now() // restart timer

// shuffle the messages
elGamal1, elGamal2 = shuffleAndCheck(publicKey, elGamal1, elGamal2)

// end timer
elapsed = time.Since(start)
// log the time
log.Printf("Shuffle took %s", elapsed)
_, err = shuffleFile.WriteString(fmt.Sprintf(
"%.5f", elapsed.Seconds()) + "\n") // record the time in file
check(err)
start = time.Now() // restart timer

// decrypt the messages, using the distributed shares
decryptedMessages := decryptMessages(elGamal1, elGamal2, shares, t, n)

fmt.Println("Byte Length:")
fmt.Println(suite.Point().EmbedLen())
fmt.Println("Original Values:")
for _, item := range messages {
value, err := item.Data()
check(err)
fmt.Println(string(value))
}
fmt.Println("Decrypted Values:")
for _, item := range decryptedMessages {
value, err := item.Data()
check(err)
fmt.Println(string(value))
}

// assures all decryptions are correct
//checkDecryption(messages, decryptedMessages)
```

```
// end timer
elapsed = time.Since(start)
// log the time
log.Printf("Decryption took %s", elapsed)
_, err = decryptionFile.WriteString(fmt.Sprintf(
"%.5f", elapsed.Seconds()) + "\n") // record the time in file
check(err)
}
}

}
```

B.3 testLongMessage.go

```
package main

import (
"fmt"
"log"
"os"
"sort"
"strconv"
"time"

"go.dedis.ch/kyber"
)

// hyper-parameters
var messagePartitions = 8
var randomnessLength = 16

func main2() {
// the filepath
filepath := "longMessageTestData.txt"

// open a file for recording the test data
file, err := os.Create(filepath)
check(err)         // make sure nothing's wrong
defer file.Close() // close the file eventually

n := 5 // the number of contributors in the scheme
t := 5 // the threshold

// record the parameters of the test
_, err = file.WriteString("Environment Creation:\n")
check(err)
_, err = file.WriteString("Using " + strconv.Itoa(n) +
```

```
" cotruibutors with threshold " + strconv.Itoa(t) + "\n\n")
check(err)

start := time.Now() // start timer

// create the environment for the tests
// this is where the bulk of the time is spent: overhead for creating the system
shares := createThresholdShares(n, t)
// each user's share will contain the public key
// Since these are all the same, we choose to use the copy at index 0 arbitrarlily
publicKey := shares[0].Public()

elapsed := time.Since(start)                          // end timer
log.Printf("Environment Creation took %s", elapsed) // log the time
_, err = file.WriteString(elapsed.String() + "\n")  // record the time in file
check(err)

// check each with an exponentially increasing number of ballots
for ballotCount := 50; ballotCount < 100; ballotCount *= 2 {

// state the number of ballots encrypted
_, err = file.WriteString("\nEncrypting " +
strconv.Itoa(ballotCount) + " ballots\n")
check(err)

for i := 0; i < 1; i++ { // do 1 tests

start := time.Now() // start timer
// doThresholdTest(ballotCount, n, t) // do test

// generate messages
messages, elGamal1, elGamal2 :=
generateLongMessageEncryptions(ballotCount, publicKey)

elapsed := time.Since(start)                        // end timer
log.Printf("Encryption took %s", elapsed)          // log the time
_, err = file.WriteString(elapsed.String() + "\n") // record the time in file
check(err)
start = time.Now() // restart timer

// shuffle the messages
elGamal1, elGamal2 = shuffleAndCheck(publicKey, elGamal1, elGamal2)

elapsed = time.Since(start)                         // end timer
log.Printf("Shuffle took %s", elapsed)             // log the time
_, err = file.WriteString(elapsed.String() + "\n") // record the time in file
check(err)
```

```
start = time.Now() // restart timer

// decrypt the messages, using the distributed shares
decryptedMessages := decryptMessages(elGamal1, elGamal2, shares, t, n)

/*
fmt.Println("Byte Length:")
fmt.Println(suite.Point().EmbedLen())
fmt.Println("Original Values:")
for _, item := range messages {
fmt.Println(item)
}
fmt.Println("Decrypted Values:")
for _, item := range decryptedMessages {
value, err := item.Data()
check(err)
fmt.Println(string(value))
}
*/
fmt.Println("Compiled Values:")
for _, item := range compileMessages(decryptedMessages) {
fmt.Println(item)
}

// assures all decryptions are correct
checkDecryption(messages, decryptedMessages)

elapsed = time.Since(start)                          // end timer
log.Printf("Decryption took %s", elapsed)            // log the time
_, err = file.WriteString(elapsed.String() + "\n") // record the time in file
check(err)
}
}
}

func encryptLongMessage(data []byte, h kyber.Point)
(messagePortions, elGamal1, elGamal2 []kyber.Point) {

// create a blank byte array to XOR with randomness
blankData := make([]byte, randomnessLength)
for i := 0; i < randomnessLength; i++ {
blankData[i] = 0 // this doesn't matter, as it will be XORed with random bytes
}

messagePortions = make([]kyber.Point, messagePartitions)
elGamal1 = make([]kyber.Point, messagePartitions)
elGamal2 = make([]kyber.Point, messagePartitions)
```

```
// the length of meaningful data: length available - randomness length - 1 postional
meaningfulDataLength := suite.Point().EmbedLen() - randomnessLength - 1
// the data left in the message
remainingData := data
// the point to hold this message portion
var embeddedMessage kyber.Point
// the array to hold the randomness, to be reused in each message chunk
randomBytes := make([]byte, randomnessLength)
// initialize the randomness in the buffer
suite.RandomStream().XORKeyStream(randomBytes, blankData)

// split the message into parts, encrypt each
for i := range messagePortions {

// a buffer to hold the message portion
buffer := make([]byte, suite.Point().EmbedLen())

// for k := range buffer {
//  buffer[k] = 0 // initialize buffer to nil
// }

// transfer the randomness to the buffer
copy(buffer[0:randomnessLength], randomBytes)

// the Byte to determine message portion order
buffer[randomnessLength] = byte(i) // index of this message portion

if len(remainingData) >= meaningfulDataLength { // the buffer will be full
// copy as much data to the buffer as will fit
copy(buffer[randomnessLength+1:len(buffer)], remainingData[0:meaningfulDataLength])
// remove coppied data from the list of data to copy
remainingData = remainingData[meaningfulDataLength:len(remainingData)]
} else { // the buffer will not be full
// copy all remaining data to the buffer
copy(buffer[randomnessLength+1:len(buffer)], remainingData[0:len(remainingData)])
// remove all data, there is none left to copy
remainingData = remainingData[0:0]
}

// embed the message portion
embeddedMessage = suite.Point().Embed(buffer, suite.RandomStream())
// record the message portion embedding
messagePortions[i] = embeddedMessage
// encrypt the message portion
elGamal1[i], elGamal2[i] = encryptMessage(embeddedMessage, h)
//fmt.Printf(string(buffer))
```

```
}

return // messagePortions, elGamal1, elGamal2
}

// generates long messages
func generateLongMessageEncryptions(n int, h kyber.Point)
(messages, elGamal1, elGamal2 []kyber.Point) {

// these three slices are given at size 0, as we will append to them later on
messages = make([]kyber.Point, 0) // the el gamal messages
elGamal1 = make([]kyber.Point, 0) // the el gamal pairs
elGamal2 = make([]kyber.Point, 0) // the el gamal pairs

// declare the messages to be used
defaultLongMessages := make([]string, 0) // the default, sample messages
defaultLongMessages = append(defaultLongMessages,
"Hello. My name is BOB. I vote for Smith.")
defaultLongMessages = append(defaultLongMessages,
"Bonjour. My name is ALBERT. I vote for QUEEN.")
defaultLongMessages = append(defaultLongMessages,
"Hi. My name is ALSO BOB. I vote for NIL.")
defaultLongMessages = append(defaultLongMessages,
"HMMM... I pass.")
defaultLongMessages = append(defaultLongMessages,
"This is an example message. As you can see, these messages can be quite long indeed!

// initialize the elGamal pairs
// // pick random messages
// pick meaningful messages
// and encrypt them with the threshold public key
for i := 0; i < n; i++ {
// messages[i] = suite.Point().Pick(suite.RandomStream())
var data []byte
if i < len(defaultLongMessages) { // use a manually created message
data = []byte(defaultLongMessages[i])
} else { // use a generated sample message
data = []byte("This is Sample Long Message #" + strconv.Itoa(i))
}

// encrypt the long message
messagesToAdd, elGamal1ToAdd, elGamal2ToAdd := encryptLongMessage(data, h)
messages = append(messages, messagesToAdd...)       // add messages
elGamal1 = append(elGamal1, elGamal1ToAdd...)       // add elGamal1
elGamal2 = append(elGamal2, elGamal2ToAdd...)       // add elGamal2

}
```

```
return // messages, elGamal1, elGamal2

}

// SortableBytesList is a wrapper for a double array of bytes, used to sort
type SortableBytesList [][]byte

// simple minimum function
func min(a, b int) int {
if a < b {
return a
}
return b
}

// functions required for sorting
func (a SortableBytesList) Len() int      { return len(a) }
func (a SortableBytesList) Swap(i, j int) { a[i], a[j] = a[j], a[i] }
func (a SortableBytesList) Less(i, j int) bool {
// loop through the byte array to see the first byte that differs
for k := 0; k < min(len(a[i]), len(a[j])); k++ {
if a[i][k] != a[j][k] { // if this byte differs...
return a[i][k] < a[j][k] // it is either greater or less than the other byte
}
}
return len(a[i]) < len(a[j]) // no bytes differed
}

// helper function to tell whether two arrays of bytes have the same data
func isDataEqual(a, b []byte) bool {
if len(a) != len(b) {
return false // the length of the slices to not match
}
for i := range a { // the arrays have the same length, so we can index over either
if a[i] != b[i] { // this byte is different
return false
}
}
// everything matched
return true
}

// combines all messages into individual long messages
func compileMessages(decryptedMessages []kyber.Point) (completeMessages []string) {
// make room for the decrypted messages
messageChunks := make([][]byte, len(decryptedMessages))
```

```
// find the decoding of all decrypted messages
for i := range decryptedMessages {
data, err := decryptedMessages[i].Data()   // retrieve the data from the point
messageChunks[i] = make([]byte, len(data)) // prepare room for the data to be stored
copy(messageChunks[i], data)    // copy the data to the message chunks array
check(err)                      // no error
}
// all data has been retrieved

// we don't know exactly how many messages there will be by the end,
// so we allocate space dynamically
completeMessages = make([]string, 0)

// fmt.Print("Message chunks! ")
// fmt.Println(messageChunks)
sort.Sort(SortableBytesList(messageChunks)) // sort the bytes
fmt.Println("We've sorted successfully")
// all similarly prefixed messages will be organized together,
// in increasing order of positional index
// eg, 6C31-0-I vote for
// and 6C31-1- Tom Smith
// will come one after the other

// Edge cases:
// messages have more than messagePartitions pieces:
// if the messages have the same content, merge them
// if there's a conflict, ignore the message
// messages have less than messagePartitions pieces:
// fit them together by index
//
currentPrefix := make([]byte, 0) // the current prefix: nothing
currentIndex := byte(0)

for i := range messageChunks {
// the random prefix/nonce of the message portion
prefix := messageChunks[i][0:randomnessLength]
// the position of the message chunk in the whole message
index := messageChunks[i][randomnessLength]
// the actual data of the message
data := messageChunks[i][randomnessLength+1 : len(messageChunks[i])]

// this is the same prefix as the privious message
if isDataEqual(currentPrefix, prefix) {

if currentIndex == index { // duplicate message portion, ignore it
continue // we skip recording this message
```

```
// NOTE: alternative implementations can record this duplicate message
// it is cryptographically assured that the contents of
// these message portions are identical
// UNLESS both parts are submitted by the same user
// eg. F3A8-1-I will vote for...
// and F3A8-1-I dont vote for...
} else { // the index is unique (and, because the list is sorted,
// greater than the previous)
// could check here to make sure index is one greater than the previous
// but this is largely unnecessary

currentIndex = index // update index
// add the new data onto the existing message
updatedMessage := completeMessages[len(completeMessages)-1] + string(data)
completeMessages[len(completeMessages)-1] = updatedMessage
}
} else { // different prefix
currentPrefix = prefix                    // update prefix
currentIndex = index                      // update index
newMessage := string(data)                // allocate space for a string
completeMessages = append(completeMessages, newMessage) // add the message
}

}

return // completeMessages
}
```

B.4 testShuffle.go

```
package main

import (
"go.dedis.ch/kyber"
"go.dedis.ch/kyber/proof"
"go.dedis.ch/kyber/shuffle"
)

// preform a shuffle test
func doShuffleTest(listLength int) {

a := suite.Scalar().Pick(suite.RandomStream()) // the private key
h := suite.Point().Mul(a, nil)                 // the public key

messages, elGamal1, elGamal2 :=
generateMessageEncryptions(listLength, h) // generate messages
newElGamal1, newElGamal2 :=
shuffleAndCheck(h, elGamal1, elGamal2)    // shuffle them
```

```
decryptedMessages :=
decryptAll(newElGamal1, newElGamal2, a) // decrypt the shuffled messages
checkDecryption(messages, decryptedMessages) // verify correct decryption
}

// shuffles and verifies the shuffle of elGamal encrypted points
// takes in the public key and two list which together
// represent a list of el Gamal pairs
// NOTE: this is the only function we need from this file for the complete scheme
func shuffleAndCheck(h kyber.Point, elGamal1, elGamal2 []kyber.Point)
(shuffledElGamal1, shuffledElGamal2 []kyber.Point) {

shuffledElGamal1, shuffledElGamal2, prover := shuffle.Shuffle
(suite, suite.Point().Base(), h, elGamal1[:], elGamal2[:], suite.RandomStream())

// Prove the shuffle
// This certifies that the shuffle was performed correctly,
// and prevents cheating
prf, err := proof.HashProve(suite, "PairShuffle", prover)
if err != nil {
panic("Shuffle proof failed: " + err.Error())
}

// Verify the proof
// each user could do this to the proof provided of the shuffle
// This will catch cheating done by the shuffler
verifier := shuffle.Verifier(suite, suite.Point().Base(), h,
elGamal1[:], elGamal2[:], shuffledElGamal1, shuffledElGamal2)
err = proof.HashVerify(suite, "PairShuffle", verifier, prf)
if err != nil {
panic("Shuffle verify failed: " + err.Error())
}

return // shuffledElGamal1, shuffledElGamal2
}

// decrypts an El Gamal message
// uses the secret key in the decryption process
// this would be infeasable in a distributed environment,
// but is useful for testing
func decryptMessage(elGamal1, elGamal2 kyber.Point, secret kyber.Scalar)
(message kyber.Point) {

toReverseElGamal := suite.Point().Mul(secret, elGamal1) // (g^y)^x == g^(xy)
message = suite.Point().Sub(elGamal2, toReverseElGamal) // M == Mg^(xy) / g^(xy)

return // message
```

```
}

// decrypts all El Gamal messages
func decryptAll(elGamal1, elGamal2 []kyber.Point, secret kyber.Scalar)
(decrpyedMessages []kyber.Point) {
// allocate space for the decrypted el gamal messages
decrpyedMessages = make([]kyber.Point, len(elGamal1))
for i := range elGamal1 {
// decrypt each messsage
decrpyedMessages[i] = decryptMessage(elGamal1[i], elGamal2[i], secret)
}
return // decryptedMessages
}
```

B.5 testThreshold.go

```
package main

import (
"sort"
"strconv"

"go.dedis.ch/kyber"
"go.dedis.ch/kyber/share"
vss "go.dedis.ch/kyber/share/dkg/pedersen"
"go.dedis.ch/kyber/suites"
)

var suite = suites.MustFind("ed25519") // Use the edwards25519-curve
//var suite = suites.MustFind("P256") // Use the NIST P-256 elliptic curve

// helper function to quickly make sure no error exists
func check(err error) {
if err != nil {
panic(err)
}
}

// from dedis github
// generates a public/private key pair randomly
func genPair() (kyber.Scalar, kyber.Point) {
sc := suite.Scalar().Pick(suite.RandomStream())
return sc, suite.Point().Mul(sc, nil)
}

// preform a shuffle test
// reading this function will provide a high-level understanding of the scheme used
func doThresholdTest(listLength, contributorCount, threshold int) {
```

```
// create the environment
// each user gets a share, which contains:
// 1) a portion of the secret key
// 2) the public key
// for the cryptosystem
shares := createThresholdShares(contributorCount, threshold)

// each user's share will contain the public key
// Since these are all the same, we choose to use the copy at index 0 arbitrarlily
publicKey := shares[0].Public()

// generate messages
messages, elGamal1, elGamal2 := generateMessageEncryptions(listLength, publicKey)

// ------------------------------------------------------- //
//                    Decryption Begins                    //
// ------------------------------------------------------  //

// decrypt the messages, using the distributed shares
decryptedMessages := decryptMessages(elGamal1, elGamal2,
shares, threshold, contributorCount)

// assures all decryptions are correct
checkDecryption(messages, decryptedMessages)
}

// helper function to decrypt a list of messages
// takes in the encrypted messages, the shares of the private threshold key,
// and the parameters of the threshold cryptosystem
// returns the list of decrypted messages
func decryptMessages(elGamal1, elGamal2 []kyber.Point, shares []*vss.DistKeyShare,
threshold, contributorCount int) (decryptedMessages []kyber.Point) {
// allocate space for the decrypted el gamal messages
decryptedMessages = make([]kyber.Point, len(elGamal1))

// decrypt each of the messages
for i := range elGamal1 { // for each message

// allocate space for the partial decryptions
shadows := make([]*share.PubShare, len(shares))
for j := range shares {
// each contributor could do this themselves
shadows[j] = extractShadow(elGamal1[i], elGamal2[i], shares[j])
// the shadow extracted is a partial decryption of the given message
// each user has their own shadow for the message
}
```

```
// to decrypt the message, we take the encrypted message,
// the parameters of the threshold system,
// and the list of shadows from each of the users
decryptedMessages[i] = decryptMessageSecretless(elGamal1[i],
elGamal2[i], shadows, threshold, contributorCount)
}

return // decryptedMessages
}

// does the preliminary step required for using a threshold cryptosystem
// returns the shares that belong to the users
// Each share has a private part, which is unique to that user,
// and a public part, which is the public key for the threshold cryptosystem
// (and is the same for all users)
func createThresholdShares(contributorCount, threshold int)
(shares []*vss.DistKeyShare) {

// the users create their own dkgs using the public keys of the other users
// each dkg is all that is needed for the threshold system
dkgs := generate(contributorCount, threshold)

fullShare(dkgs) // communicate between the dkgs

// collect shares
// each user should have their own share
// creating a share fulfills the purpose of the dkg
shares = make([]*vss.DistKeyShare, 0, len(dkgs)) // allocate space for the shares
for _, shareholder := range dkgs {               // for each generator
if shareholder.Certified() { // make sure it's certified
newShare, err := shareholder.DistKeyShare() // get the share
check(err)
shares = append(shares, newShare) // add it
}
}
return // shares
}

// adapted from dedis github
// makes public/private key pairs, and makes a dgk for each private key
// publicly executable, as each dkg created only uses one private key
// and the public keys
// follows "A Threshold Cryptosystem Without a Trusted Party"
func generate(n, t int) (dkgs []*vss.DistKeyGenerator) {
// Each public/private keypair represents the identity of a user
// in a distributed application, these keypairs will be created
```

```
// by each user independently
partPubs := make([]kyber.Point, n) // allocate space for public key parts
partSec := make([]kyber.Scalar, n) // allocate space for private key parts
for i := 0; i < n; i++ {           // for n users
sec, pub := genPair()
partPubs[i] = pub // the public key for the user
partSec[i] = sec  // the private key for the user
}

// allocate space for the key generators
dkgs = make([]*vss.DistKeyGenerator, n)

// each user...
for i := 0; i < n; i++ {

// creates a key generator
// uses one user's private key
// the dkg created is now linked to that user
dkg, err := vss.NewDistKeyGenerator(suite, partSec[i], partPubs, t)
check(err)
dkgs[i] = dkg
}
// after this point, the keypair is no longer used
// each dkg now effectively represents a user's identity

return dkgs
}

// encrypts an El Gamal message
func encryptMessage(message, pubKey kyber.Point) (elGamal1, elGamal2 kyber.Point) {
tempScalar := suite.Scalar().Pick(suite.RandomStream())
elGamal1 = suite.Point().Mul(tempScalar, nil)
elGamal2 = suite.Point().Mul(tempScalar, pubKey)
elGamal2.Add(elGamal2, message)

return
}

// decrypts an El Gamal message
// given the encrypted message, the list of shadows,
// and the parameters for the threshold system
// the decryption is "secretless" because the secret exponent is never revealed
// follows the scheme outlined in Sections 2.2 and 3.1 of
// "Threshold Cryptosystems" by Desmedt and Frankel
func decryptMessageSecretless(elGamal1, elGamal2 kyber.Point,
shadows []*share.PubShare, t, n int) (message kyber.Point) {
```

```
// With a message M and secret x, encrypted as the tuple (g^y, Mg^(xy))
// recovers the committment g^(xy)
// this acts as a key to decrypt the original message
// by dividing Mg^(xy) by g^(xy)

// recover g^(xy), essentially by multiplying its factors together
key, err := share.RecoverCommit(suite, shadows, t, n)
check(err)                                              // no error
message = suite.Point().Sub(elGamal2, key)              // M = Mg^(xy) / g^(xy)

return // message
}

// Follows the scheme outlined in Section 2.1 of
// "Threshold Cryptosystems" by Desmedt and Frankel
// extracts the tuple g^kV_i, i
// given a public el gamal encrpyed message,
// and a share (kept private),
// returns the corresponding shadow
// this shadow essentially is a factor of the committment used
// in the second half of an El Gamal encrypted message
func extractShadow(elGamal1, elGamal2 kyber.Point, distShare *vss.DistKeyShare)
(shadow *share.PubShare) {

priv := distShare.PriShare()                  // private share x_i
value := suite.Point().Mul(priv.V, elGamal1) // g^(y*x_i)
index := priv.I        // record the index of the user to keep the shadws ordered
shadow = &share.PubShare{I: index, V: value} // struct for recovering commit later
return                                        // shadow
}

// communicates the required information for the key generators to function
func fullShare(dkgs []*vss.DistKeyGenerator) {

// This function shares all of the information between all dkgs
// the outline for the communication steps were provided on the dedis github,
// following an error-resistant implementation of the scheme described in
// "A Threshold Cryptosystem without a Trusted Party"

// There are three phases for the communication:
// 1) Deals
// Each user conveys information about itself to all others
// 2) Responses
// Each user makes sure that all of the deals it recieves are consistent
// If a given deal is compliant, the response indicate that it was accepted
// otherwise, the response will indicate that a justification
// for the deal is needed
```

```
// 3) Justifications
// This phase gives users the chance to justify their deal
// This can occur when either party has made a mistake
// Usually, no justification is needed, and none is given

// allocate space for the responses
resps := make([]*vss.Response, 0, len(dkgs)*len(dkgs))

// here, the dkgs know only the public keys of the other users
// more information is needed in order to create a share
// of the public threshold key

// deal all shares
for _, generator := range dkgs {
deals, err := generator.Deals() // each dkg has a deal for each other user
check(err)

for j, deal := range deals { // for each deal
processor := dkgs[j]

//process the deal
response, err := processor.ProcessDeal(deal)
check(err) // no error
// record the response to the deal
// index of response is contributorCount * sender index + reciever index
resps = append(resps, response)
// each response is

}
}
// all deals dealt

// distribute responses
for _, response := range resps {
for i, dkg := range dkgs { // everone can process every response

// don't justify to yourself
if uint32(i) == response.Response.Index {
continue
}

// handle response to the deal, justify deal to responder
justification, err := dkg.ProcessResponse(response)
check(err) // no error

// process justification
// This can be done directly after the response is given,
```

```
// independently of other responses

// justification will be nil if there is nothing to justify
// this is normally the case
if justification != nil {
sender := dkgs[response.Response.Index]
err = sender.ProcessJustification(justification)
check(err) // make sure the justification is valid
}
}
}
// all responses distributed
}

// SortablePointList is a wrapper for []]kyber.Point
// this allows for these lists to be sorted
type SortablePointList []kyber.Point

// functions required for sorting
func (a SortablePointList) Len() int           { return len(a) }
func (a SortablePointList) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a SortablePointList) Less(i, j int) bool
{ return a[i].String() < a[j].String() }

// helper function, compares two lists of points
// checks if the decryptions of messages matches the original messages
func checkDecryption(messages, decryptedMessages []kyber.Point) {

// message lengths match up
if len(messages) != len(decryptedMessages) {
panic("Decrypted messages do not match up with messages! (different len)")
}

// sort the lists, as one may have been shuffled
sort.Sort(SortablePointList(messages))
sort.Sort(SortablePointList(decryptedMessages))

// check each pair of messages
for i := range messages {
// fmt.Println("Message:", messages[i])
// fmt.Println("Decrypted Message:", decryptedMessages[i])
// fmt.Println()
if messages[i].Equal(decryptedMessages[i]) == false {
panic("Message incorrectly decrypted!") // messages do not match
}
}
}
```

```
// helper function, generates a list of messages alongside their encryptions
// generates n messages and their encryptions with a given public key h
// encryptions are done in El Gamal,
// where each index represents an encrypted message
// in pseudocode: encrypt(message[i]) == (elGamal1[i], elGamal2[i])
func generateMessageEncryptions(n int, h kyber.Point)
(messages, elGamal1, elGamal2 []kyber.Point) {

messages = make([]kyber.Point, n) // the el gamal messages
elGamal1 = make([]kyber.Point, n) // the el gamal pairs
elGamal2 = make([]kyber.Point, n) // the el gamal pairs

// initialize the elGamal pairs
// // pick random messages
// pick meaningful messages
// and encrypt them with the threshold public key
for i := range messages {
// messages[i] = suite.Point().Pick(suite.RandomStream())
data := []byte("Sample Message " + strconv.Itoa(i))
messages[i] = suite.Point().Embed(data, suite.RandomStream())
// can use any share's public key
elGamal1[i], elGamal2[i] = encryptMessage(messages[i], h)
}

return // messages, elGamal1, elGamal2
}
```