Cryptovote

Nicholas Boucher, Pasapol Saowakon, Kyle Swanson, Luka Govedič

***Abstract*—Ranked choice voting is a popular alternative to single choice voting as it represents a more equitable social choice function; voters can vote for particularly popular or unpopular candidates without fear of their vote going to waste. However, as in all election schemes, it is crucial that the election is implemented by a system that is secure, confidential, fair, and free of coercion. In this paper, we contribute an open-source Python implementation of the Shuffle-Sum protocol from [1], which securely tallies encrypted votes in a single transferable vote (STV) ranked choice election. For the underlying cryptography for Shuffle-Sum, we implemented the Damgård-Jurik cryptosystem [2], which is a threshold cryptosystem that enables the distribution of trust among multiple election authorities. Additionally, we created a website through which election authorities can easily and securely run an STV election using Shuffle-Sum. We have also profiled our implementation of Shuffle-Sum on real election data from the November 2016 San Francisco Board of Supervisors election, and we investigated the impact of using different levels of encryption and different numbers of ballots on the speed of our implementation.**

***Index Terms*—Ranked choice voting, homomorphic encryption.**

I. Introduction

Electronic voting (e-voting) is voting that uses electronic means to either aid or enable casting and counting votes [3]. It is much more efficient than traditional voting in many ways. For instance, electronic voting does not require physical polling stations, since it takes place online. Therefore, electronic voting eliminates the significant need not only for human resources but also for physical resources such as trees. Moreover, electronic voting likely boosts voter turnout, as votes can be cast anywhere, offering unparalleled flexibility to voters.

While the idea of electronic voting sounds appealing, there have been criticisms of its security. For example, there must be a mechanism to ensure that all votes are only cast by eligible voters, each casting a vote no more than once. At the same time, the voters should not be identifiable with the votes they cast. Ideally, the voters should also be able to verify that their votes have been counted, yet there should be no receipt so as to prevent coercion. And very importantly, an adversary must not be able to tamper with the outcome.

Electronic voting has garnered much attention from researchers, as evident from the recent progress in the field [4]–[6]. However, most prior work has focused on encryption for first-past-the-post voting [7], in which every voter votes for a single candidate and the candidate with the most votes wins. Although first-past-the-post is commonly used both in the United States and abroad, some communities have shifted to ranked choice voting systems since they represent a more equitable voting mechanism. One such system in particular, single transferable vote (STV) [8], is designed to overcome the limitations of first-past-the-post voting, where votes for particularly popular or unpopular candidates have little influence on the outcome of the election. In STV elections, votes for candidates who receive more than the required quota of votes are reweighted and redistributed to the voter's next preference based on the number of excess votes for their first-choice candidate. Additionally, candidates who receive very few votes are eliminated from the election, and the full weight of the vote is redistributed to the voter's next preference, meaning that no votes are wasted by voting for candidates who are unlikely to win. This also circumvents the "spoiler effect", where a 3rd party candidate draws enough votes away from one of the two major party candidates to flip the election [9][1]. For these reasons, the STV election system has been adopted by cities and nations such as Cambridge, Massachusetts; San Francisco, California; and Australia.

Due to the rising popularity of STV, Benaloh et al. designed an encrypted, coercion-resistant, verifiable protocol for tallying votes in an STV election [1]. Their protocol, called Shuffle-Sum, cycles between a variety of encrypted ballot forms to perform the reweighting, redistribution, and elimination operations of an STV election. Furthermore, it leverages an additively homomorphic encryption scheme to make tallying of encrypted votes possible. The encryption scheme is also designed to be a threshold cryptosystem, meaning that the private key used for decryption is shared among a number of election authorities. The threshold for decryption can be chosen so that as long as at least one of those authorities is honest, decryption of individual votes is not possible and the integrity of the election is preserved.

Despite the excellent design of the Shuffle-Sum protocol, we were unable to find a publicly available implementation of Shuffle-Sum or of any algorithm for tallying an encrypted STV election. Thus, our goal was to build a working, easy-to-use, open-source implementation of Shuffle-Sum. To this end, we implemented Shuffle-Sum in Python with an easily accessible web interface that allows users to create, administer, and vote in STV elections. We also implemented the Damgård-Jurik cryptosystem to provide the underlying cryptography for Shuffle-Sum. Our web interface is available at https://cryptovote.ml and our source code is available for audit on GitHub at https://github.com/cryptovoting.

II. Damgård-Jurik Cryptosystem

A. Overview

The Damgård-Jurik cryptosystem [2] is a generalization of Paillier's public key cryptosystem [11]. As with Paillier's

[1]In fact, some have claimed that 3rd party candidates Gary Johnson and Jill Stein may have drawn enough votes from Hillary Clinton in the 2016 US presidential election to help elect Donald Trump [10].

cryptosystem, the security of Damgård-Jurik is due to the difficulty of factoring an RSA modulus $n = pq$ where p and q are large primes. However, while Paillier operates in the group $Z^*_{n^2}$ with plaintexts living in Z_n, Damgård-Jurik operates in the group $Z^*_{n^{s+1}}$ with plaintexts in Z_{n^s} for any $s > 0$. This makes it possible for Damgård-Jurik to encrypt and decrypt large plaintexts without changing the size of the underlying keys.

B. Applicability to Voting

Damgård-Jurik is particularly applicable to voting systems for two reasons.

1) Additively Homomorphic: Like Paillier encryption, Damgård-Jurik is additively homomorphic, meaning it is possible to combine ciphertexts in such a way as to obtain an encryption of the sum of the corresponding plaintexts. This is ideal for tallying encrypted votes because it means that individual votes never need to be decrypted; instead, the encrypted votes can be added directly and only the sum needs to be decrypted. Besides enabling addition of encrypted numbers, Damgård-Jurik's homomorphic property also means that it is also possible to combine an encrypted number with an unencrypted scalar in such a way as to obtain the product of the corresponding plaintext and the scalar. This is necessary for tallying an STV election because votes need to be multiplied by an (unencrypted) weight when a candidate is elected and his or her excess votes need to be redistributed.

2) Threshold Decryption: The second feature of Damgård-Jurik that makes it useful for voting is that it can be structured as a threshold cryptosystem. In a threshold cryptosystem, there is a single public key as usual, but the private key is split up and shared among a number of authorities. The shares are constructed so that decryption is only possible with at least w shares, where w is a threshold decided during key generation. The distributed nature of the private key is useful in an election because it means that as long as the number of corrupt election authorities is less than the threshold, the confidentiality of individual votes is preserved. This makes the election robust to as many as $w - 1$ corrupt authorities. If w is chosen to be $w = l$, then the election is robust as long as at least one authority is honest.

In the next few sections, we will describe the technical details of the threshold variant of the Damgård-Jurik cryptosystem that enable these two features.

C. Key Generation

Key generation begins by finding two safe primes $p = 2p' + 1$ and $q = 2q' + 1$, where p' and q' are distinct, large primes. Then, we define $n = pq$ and $m = p'q'$. We choose an $s > 0$, which controls the size of the plaintext space Z_{n^s}. The public key is the pair (n, s).

Next, we use the Chinese remainder theorem to select a secret key d such that d satisfies $d = 0 \mod m$ and $d = 1 \mod n^s$. We then use Shamir's secret sharing [12] to split the secret key into l shares in such a way that at least w of those shares are required to reconstruct the secret key. This is accomplished by constructing the polynomial $f(X) = \sum_{i=1}^{w-1} a_i X^i \mod n^s m$, where a_i (for $0 < i < w$) are random values in $\{0, \ldots, n^s m - 1\}$ and $a_0 = d$ is the secret being shared. The ith authority is given the secret key share $s_i = f(i)$ for $1 \leq i \leq l$.

D. Encryption

To encrypt a message M, a random $r \in Z^*_n$ is chosen and the ciphertext is computed as $c = (n+1)^M r^{n^s} \mod n^{s+1}$.

E. Homomorphic Operations

Next, we will demonstrate the operations that make the above encryption scheme additively homomorphic.

1) Adding Encrypted Numbers: First, we will show how to combine two encrypted numbers to obtain the sum of the associated plaintexts. Consider that we have two plaintexts M_1 and M_2 and we encrypt them to obtain ciphertexts c_1 and c_2:

$$c_1 = (n+1)^{M_1} r_1^{n^s} \mod n^{s+1}$$
$$c_2 = (n+1)^{M_2} r_2^{n^s} \mod n^{s+1}$$

Then if we multiply c_1 and c_2, we get

$$\begin{aligned} c' = c_1 * c_2 &= (n+1)^{M_1} r_1^{n^s} * (n+1)^{M_2} r_2^{n^s} \mod n^{s+1} \\ &= (n+1)^{M_1+M_2} (r_1 * r_2)^{n^s} \mod n^{s+1} \end{aligned}$$

which is a valid encryption of $M_1 + M_2$ using $r' = r_1 * r_2$. Thus if we decrypt c', then we will obtain $M_1 + M_2$ as desired.

2) Multiplying an Encrypted Number by a Scalar: Next, we'll show how to multiply an encrypted number by an unencrypted scalar. Consider that we have the plaintext M and the corresponding encryption c:

$$c = (n+1)^M r^{n^s} \mod n^{s+1}$$

Then we can obtain the encryption of the product of M and some scalar x by computing c^x since

$$\begin{aligned} c' = c^x &= \left((n+1)^M r^{n^s}\right)^x \mod n^{s+1} \\ &= (n+1)^{Mx} (r^x)^{n^s} \mod n^{s+1} \end{aligned}$$

which is a valid encryption of Mx using $r' = r^x$. Thus if we decrypt c', then we will obtain Mx as desired.

F. Decryption

Decrypting a ciphertext c requires combining the decryption of c using multiple shares of the secret key.

First, each authority performs part of the decryption using its portion of the secret key. Specifically, the ith authority computes $c_i = c^{2\Delta s_i}$ where $\Delta = l!$, l is the number of secret key shares, and s_i is the ith authority's secret key share.

Once each authority has computed its portion of the decryption, those portions need to be combined to obtain the overall decryption. Since we only need w shares to perform the decryption (where w is the threshold decided during key generation), we will take a subset S consisting of w shares and combine them as follows[2]:

$$c' = \prod_{i \in S} c_i^{2\lambda_{0,i}^S} \mod n^{s+1} \quad \text{where } \lambda_{0,i}^S = \Delta \prod_{i' \in S \setminus \{i\}} \frac{-i'}{i - i'}$$

An astute reader may recognize the $\lambda_{0,i}^S$ term as a slight variant of the polynomial reconstruction expression from Shamir's secret sharing scheme. Thus, we can see that the c' we have computed is actually:

$$c' = c^{4\Delta^2 f(0)} = c^{4\Delta^2 d}$$

Next, recall that d was chosen so that $d = 1 \mod n^s$ and $d = 0 \mod m$. Thus, we can see that $4\Delta^2 d = 4\Delta^2 \mod n^s$ and $4\Delta^2 d = 0 \mod m$. From here, it is possible to show that we can simplify c' to get:

$$\begin{aligned} c' = c^{4\Delta^2 d} &= (n+1)^{4\Delta^2 M d} r^{4\Delta^2 n^s d} \mod n^{s+1} \\ &= (n+1)^{4\Delta^2 M} r^0 \mod n^{s+1} \\ &= (n+1)^{4\Delta^2 M} \mod n^{s+1} \end{aligned}$$

For a detailed derivation, see [2].

At this point, all that is left is to extract M from the exponent in c'. In [2], the authors prove that the algorithm presented in Algorithm 1 can extract the exponent i from any integer a of the form $a = (n+1)^i \mod n^{s+1}$. Applying Algorithm 1 to c', we obtain the exponent $4\Delta^2 M$. Finally, we multiply this term by $(4\Delta^2)^{-1} \mod n^s$ to obtain the plaintext message M.

Algorithm 1 Damgård-Jurik Reduce

```
1: procedure REDUCE($a$, $n$, $s$)
2:     $i = 0$
3:     for $j = 1$ to $s$ do
4:         $t_1 = (a \mod n^{j+1} - 1)/n$
5:         $t_2 = i$
6:         for $k = 2$ to $j$ do
7:             $i = i - 1$
8:             $t_2 = t_2 * i \mod n^j$
9:             $t_1 = t_1 - \frac{t_2 * n^{k-1}}{k!} \mod n^j$
10:        $i = t_1$
11:    return $i$
```

III. Shuffle-Sum Protocol

The Shuffle-Sum protocol was developed by Benaloh et al. [1] to tally an encrypted STV election. The main idea of the protocol is to perform shuffles in between each threshold decryption by the authorities so as to safeguard the confidentiality of the ballots, even among the authorities themselves. Suppose there are l authorities and w is the minimum number of authorities required to tally an election. The security assumption requires only that at least $l - w + 1$ authority is *honest*. Hence, if $l = w$, only one authority will need to be honest. With such an assumption held, the individual ballots are guaranteed to be safe (i.e., only *necessary* information can be inferred about each individual ballot), which in turn helps prevent coercion. As a preliminary, the protocol assumes a secure homomorphic encryption scheme. The protocol is concerned with single transferable vote elections, although it is worth noting that only very slight modifications are required to further support most other popular social choice functions.

Throughout the protocol, the ballots go through different representations to allow for homomorphic operations on different fields of data (while preserving ballot confidentiality). More precisely, we define four representations for a ballot.

- **Candidate-order ballot**. Columns are ordered with respect to the identification number of the candidates. The preference for each candidate is encrypted. It also contains the encrypted weight of the ballot.
- **Preference-order ballot**. Columns are ordered so that the preferences are from most desired to least desired. The candidate identification number corresponding to each preference is encrypted. It also contains the encrypted weight of the ballot.
- **Candidate-elimination ballot**. The same as preference-order ballot, except that there is an encrypted flag (0 or 1) for each candidate indicating whether he or she is being eliminated in the round being considered. The preferences row is also encrypted, not for confidentiality (since the columns are sorted by preferences anyway), but rather for functionality (to allow for addition or subtraction with other encrypted numbers).
- **First-preference ballot**. Columns are sorted in the order of candidates. There is an encrypted row of weights which contains the current weight of the ballot for the candidate who is the voter's first preference and contains 0 for all other candidates.

Each ballot cast is initially stored in the candidate-order ballot form. Then, given all the ballots in this form, the authorities can tally the election with a tallying process that occurs in multiple rounds, each consisting of the following steps.

1) **Compute first-preference tallies**. Candidate-order ballots are converted into the first-preference representation. Then, the tally for each candidate is simply the homomorphic addition of each ballot's encrypted weight in the corresponding candidate column (note that the unencrypted weight is non-zero for exactly one candidate). The authorities can then work together to decrypt the tallies.
2) **Elect candidates** (if any). In this step, the candidates whose tally is at least the set quota get elected and are announced to have been elected.
3) **Reweight votes** (only if at least one candidate was elected). This process is to re-adjust the weight of the ballots whose most-desired candidate was just elected. To do this, we simply perform the appropriate homomorphic multiplications to the ballots in the first-preference representation.
4) **Eliminate candidates**. If there were candidates elected during this round, *eliminate* them from the ballots.

[2]Note: In [2], the numerator of the $\lambda_{0,i}^S$ expression is $-i$ instead of $-i'$, but we believe this is a mistake.

Otherwise, eliminate the one candidate who received the least votes. This is done by converting candidate-order ballots into the candidate-elimination representation. Then, appropriate homomorphic subtractions with elimination tags are performed, before the ballots are converted back into the candidate-order representation.

More details of each step are available in the original Shuffle-Sum paper [1], although they will also be briefly explained in Section IV.

There are a few ballot conversion subroutines that are integral to our protocol. Namely, we are interested in conversions **(1)** from candidate-order ballot to first-preference ballot, **(2)** from candidate-order ballot to candidate-elimination ballot, and **(3)** from candidate-elimination ballot to candidate-order ballot. While it is true that it is possible to simply decrypt the ballot content and restructure the content accordingly before re-encrypting the appropriate fields in order to obtain the ballot in another representation, it is not secure, as this would mean that any party doing the computation must have access to the fully-decrypted ballot at some intermediate point. Rather, the specially designed conversion subroutines make use of random shuffling so that the content of any specific ballot being converted is never revealed to any authority (when there are multiple authorities and our security assumption holds).

As described above, Shuffle-Sum requires voters to provide a full ranking of all candidates. However, in many elections, voters do not have to rank all candidates, and in some cases there is a even cap on the number of candidates that a voter is allowed to rank. A slight modification to the Shuffle-Sum protocol makes it possible to handle these cases. In such an election, an additional "stop" candidate is added to each ballot and is given the preference immediately following the lowest preference indicated by the voter. All un-ranked candidates are then given lower preferences than the "stop" candidate in a random permutation. The Shuffle-Sum protocol is then modified so that the "stop" candidate is never elected or eliminated. The existence of the "stop" candidate thus prevents the voter's vote from counting towards any (originally un-ranked) candidate who has a lower preference than the "stop" candidate.

IV. Our Implementation

In this section, we describe our Python implementations of the Damgård-Jurik cryptosystem and the Shuffle-Sum protocol.

A. Damgård-Jurik

Our implementation of the Damgård-Jurik cryptosystem is available on GitHub at https://github.com/cryptovoting/damgard-jurik. We have also published it to the Python Package Index (https://pypi.org/project/damgard-jurik/), meaning it can be installed via pip: `pip install damgard-jurik`. Below we describe some of the key components of the package. The only dependency outside of built-in Python functions is `gmpy2` [13].

1) Key Generation: Key generation occurs in a function called `keygen`. `keygen` takes as input the number of bits that the public and private keys should have, the public key parameter s, the threshold number of shares for decryption w, and the total number of shares to generate l. It uses `gmpy2`'s `next_prime` and `is_prime` functions to find two safe primes p and q with the appropriate number of bits and then computes $n = pq$. It then uses a Chinese remainder theorem function we implemented to find the secret key d. Next, it applies Shamir's secret sharing, which we implemented as a separate module within the Damgård-Jurik package, to split d into the appropriate number of shares. Finally, it returns the public key and private keys as instances of the `PublicKey` and `PrivateKeyRing` classes, which will be discussed in the next section.

2) Public Keys and Private Keys: A `PublicKey` consists of the parameters n and s along with two methods: `encrypt`, which encrypts a single integer, and `encrypt_list`, which is a convenience method that applies `encrypt` to every integer in a list.

A private key is more complicated and is stored using two different classes. Each share of the private key lives in an instance of the `PrivateKeyShare` class, which holds i and $f(i)$, where $f(X)$ is the polynomial defined by Shamir's secret sharing. It also holds a reference to the associated `PublicKey`. The `PrivateKeyShare` class also contains a `decrypt` method which performs that share's portion of the decryption, i.e., $c_i = c^{2\Delta s_i} \mod n^{s+1}$. All the `PrivateKeyShare`s for a particular keypair are then stored in an instance of the `PrivateKeyRing` class. This class contains a `decrypt` method which calls each `PrivateKeyShare`'s decrypt method and then combines all of the partial decryptions into the final decryption. As with the `PublicKey`, the `PrivateKeyRing` class has a convenience method called `decrypt_list` which decrypts every encrypted number in a list.

3) Encrypted Numbers: While plaintexts are simply represented as basic Python integers, ciphertexts are represented as instances of the `EncryptedNumber` class. An `EncryptedNumber` contains an integer which is the actual encrypted value along with a reference to the `PublicKey` which produced that encryption.

To make it easy to apply homomorphic operations to encrypted numbers, we overrode the $+$, $-$, $\star$, and $/$ operators for the `EncryptedNumber` class to automatically apply the operation in ciphertext space that is equivalent to the desired operation in plaintext space. For instance, if `c_1` and `c_2` are `EncryptedNumbers` containing encryptions of the integers `m_1` and `m_2`, then executing `c = c_1 + c_2` actually performs the operation $c = c_1 * c_2 \mod n^{s+1}$ so that the decryption of `c` is equal to `m_1 + m_2`.

4) Optimizations: To increase the speed of all encrypted operations, we used `gmpy2`'s `mpz` class, which optimizes operations on large integers. Additionally, we pre-computed and stored many constants such as n^s and $\Delta = l!$ in the `PublicKey`, `PrivateKeyShare`, and `PrivateKeyRing` classes. Furthermore, since the reduction function in Algorithm 1 computes n^j and $k!$ for the same values of j and k

many times, we use Python's built-in `lru_cache` decorator to cache those operations to avoid redundant computation.

B. *Shuffle-Sum*

Our implementation of the Shuffle-Sum protocol is available at https://github.com/cryptovoting/shuffle-sum. Below we describe the overall architecture of our implementation.

First, we discuss ballot representation and conversion between different representations.

1) Ballot representation: We represent each ballot with an instance of the `Ballot` class as defined in `ballots.py`. The `Ballot` class contains three sub-classes, corresponding to each of the three representations of the ballot: `FirstPreferenceBallot`, `CandidateOrderBallot`, and `CandidateEliminationBallot`. Note, however, that there is no subclass explicitly representing preference-order ballots—it is unnecessary and was defined in Section III only as a substructure of a candidate-elimination ballot.

2) Ballot conversion: Also defined in `ballots.py` are the conversion protocols needed for tallying. Each conversion protocol requires a private key ring as a parameter to enable threshold decryption of the ballot content. The details of how these protocols work are outlined in the original Shuffle-Sum paper [1] and will be omitted here. Note, however, that the ballot conversion protocols that we implemented are not completely secure as is: a protocol that takes as input a private key ring basically requires that the party doing the computation has all the private key shares necessary to decrypt any ballot content, allowing them to maliciously decrypt and see the content of any ballot they want. Ideally, we would need a communication protocol between all the authorities that would allow for a secure threshold decryption (i.e., where each individual authority computes their partial decryption and sends it to a server running the Shuffle-Sum protocol without ever revealing their portion of the private key). Additionally, as per the current implementation, shuffling is not completely secure. These will be described in more detail as future work in Section VII.

With the ballot structure constructed, we implement the main protocol for tallying: `stv_tally`, which resides under `protocols.py`. This protocol takes as an input **(1)** a list of all ballots in the candidate-order representation, **(2)** the number of seats, **(3)** the identification number of the "stop" candidate, **(4)** a private key ring, **(5)** the public key, and **(6)** the list in which the elected candidates should be stored. The procedure mainly makes use of three subroutines, defined in the same source file:

1) `compute_first_preference_tallies` – Determines who gets elected and converts candidate-order ballots into the first-preference representation.
2) `reweight_votes` – Re-weights each ballot whose top choice candidate was just elected.
3) `eliminate_candidate_set` – Eliminates the recently elected candidates from the ballots. In case no one was elected, the candidate who received the least votes is eliminated.

At the end, the elected candidates are stored in the list initially passed into the protocol.

Note that a private key ring is an argument to the protocol, which, as discussed earlier, is not secure. Section VII contains more details on how this security hole may be fixed.

V. Performance

After verifying the correctness of our implementation on small test cases, we proceeded to evaluate its performance on real election data. Since encryption, decryption, and homomorphic operations on encrypted numbers are relatively slow, we were particularly interested in how the overhead of encryption affects the practicality of using our implementation for tallying real-world election results.

A. *Data*

We evaluated our implementation on San Francisco's November 2016 Board of Supervisors election in District 1 [14], which featured 12 candidates vying for 1 open seat. We downloaded anonymized ballots for the election from the Ranked Choice Voting Resource Center [15]. We then generated a public key and a private key, where the private key consisted of 3 private key shares with a threshold of 3 to decrypt (i.e., all 3 keys are needed for decryption). Next, we converted the ballots from ballot image format to Shuffle-Sum's candidate-order ballot format, with the voter's preferences encrypted using the public key. Since voters were allowed to rank at most 3 of the 12 candidates, we used the stop candidate variant of Shuffle-Sum by adding a 13th "stop" candidate that was given the rank immediately following the voter's lowest marked preference. After filtering out invalid ballots, we were left with 31,787 ballots.

B. *Experimental Setup*

All of the following experiments were run on a 2018 Macbook Pro with two 6-core 2.2GHz Intel Core i7 processors[3]. Except where otherwise noted, the public and private keys use 32 bits of encryption and $s = 1$.

C. *No Encryption*

To establish a baseline for performance, we ran our Shuffle-Sum implementation on the San Francisco election data without any encryption (i.e. using a public key and a private key ring that both act as an identity function). All other aspects of the algorithm, including shuffling, were unchanged. Running the election without encryption took 1.97 minutes.

D. *Bits of Encryption*

With this baseline established, we then experimented with different levels of encryption by varying the number of bits in the public and private keys. The results are presented in Table I and Figure 1. As can be seen, even minimal 32-bit encryption

[3]Note that the number of cores is important since we implemented an optimization that performs most of the Shuffle-Sum operations in parallel across the ballots.

TABLE I: Time vs number of bits of encryption.

Number of Bits	Time (minutes)
32	6.66
64	8.76
128	15.12
256	68.63

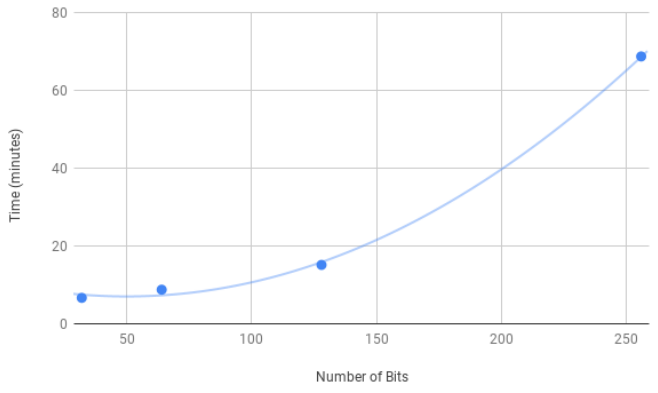

Fig. 1: Time vs number of bits of encryption.

TABLE II: Time vs exponent s.

Exponent s	Time (minutes)
1	6.66
2	6.97
3	8.26
4	10.01

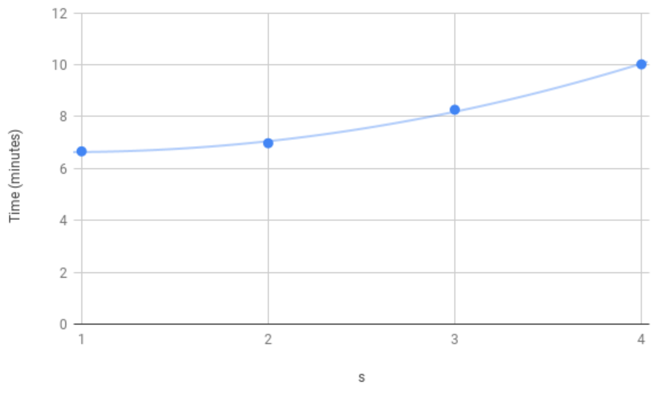

Fig. 2: Time vs exponent s.

takes about 3.4x longer than the unencrypted baseline, indicating the amount of time taken by encrypting, decrypting, and performing homomorphic operations. Furthermore, the time taken appears to grow quadratically with the number of bits of encryption. Extrapolating using quadratic regression to the 2048 bits typically used in RSA encryption (which is secure due to the same difficulty of factoring a product of large primes), we would expect that our implementation would take about 5,795 minutes, or about 4 days, to tally the election[4]. Fortunately, most of the operations are run in parallel, meaning the time could be significantly reduced on a machine with more CPU cores.

E. Exponent s

In addition to the number of bits used in the public and private keys, the Damgård-Jurik exponent s, which controls the size of the plaintext space Z_{n^s}, also affects encryption time. The results are presented in Table II and Figure 2. As with the number of bits of encryption, the time to tally the election seems to depend quadratically on s[5]. However, for many applications, including the San Francisco election, the largest plaintext ever used is less than n and so $s = 1$ produces a plaintext space that is large enough for our purposes.

F. Number of Ballots

Another important question is how the time required to tally an election depends on the number of ballots. Table III and Figure 3 shows that the time is roughly linear in the number of ballots[6], as would be expected. Extrapolating to an election on the scale of the US presidential election in 2016, which saw about 158 million votes, it would take our implementation approximately 32,073 minutes, or about 22 days, to tally the election using 32-bit encryption. Extrapolating to both 158 million votes and 2048-bit encryption, it would take about 28,804,543 minutes, or about 55 years, to tally the election. Again, since most of the operations can be performed in parallel, this time could be significantly reduced on a more powerful machine. Furthermore, there may be other optimizations, such as the Table-Sum algorithm from the Shuffle-Sum paper [1], that could further reduce this time.

[4]The quadratic regression is $f(x) = 0.00145x^2 - 0.145x + 10.6$, which has an R^2 of 0.999.

[5]The quadratic regression is $f(x) = 0.359x^2 - 0.66x + 6.94$, which has an R^2 of 0.998.

[6]The linear regression is $f(x) = 0.000203x - 0.0485$, which has an R^2 of 1.000.

TABLE III: Time vs number of ballots.

Number of Ballots	Time (minutes)
5,000	0.97
10,000	1.94
20,000	4.05
30,000	6.00

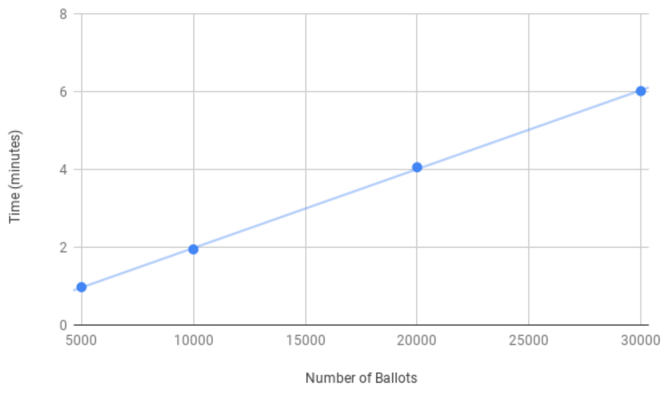

Fig. 3: Time vs number of ballots.

VI. Web Interface

While developers are able to make use of our Python implementation of the Shuffle-Sum protocol, we recognize that many people hoping to administer electronic elections may not have the time nor technical capabilities to build a pipeline of data into a set of Python modules. Therefore, we also built a web interface that allows the easy creation and administration of electronic elections[7].

The goal of the web interface is to provide a method of collecting encrypted ballots that is easy to use, secure against malicious actors, and capable of authenticating voters properly. Additionally, the web interface allows the tabulation of elections results and the public posting of those results.

A. Election Creation

In our web implementation, elections are initially created by one of the authorities who will be administering the election. They begin the election creation process by visiting the homepage of the Cryptovote web app and providing their name, email address, and proposed election name. After verifying the authority's email address, the authority is prompted to select a form of authentication, which could be either via WebAuthn or a password (see Section VI-C). After registering authentication credentials, a new public/private keypair is automatically generated for the authority. In the current implementation, the public and private keypairs are both stored on the server in a database (see Section VII for future improvements on this implementation). Finally, the authority is prompted to add candidates to the ballot by name and voters to the election by email address following the initial election creation process.

After an election has been created, a subdomain with the name of the election is generated on the web application. This subdomain is a micro-website that is the central location to access anything regarding its corresponding election for authorities, voters, and members of the public. Through this subdomain, members of the public have the ability to see the list of candidates, the list of authorities, and the public bulletin associated with the Shuffle-Sum protocol [1]. Additionally, authenticated authorities have the ability to add new voters to the election and end the election by tabulating the election results.

B. Voting

Voters are invited to cast ballots in the election by email. After an authority adds the voter to the election by email address, the voter will immediately receive an email asking them to cast their ballot in the election. The email jointly serves as an invitation to vote in the election and as a tool for email address verification.

Once the voter has followed the link to vote in the election, the voter is asked to supply their name for listing on the voter role. After this, the voter is taken to an electronic ballot which lists all candidates. The voter indicates their preferences by dragging and dropping candidates to sort them in the order of their preference. By design of the input mechanism, all candidates must be sorted as is required by the stop-candidate-free Shuffle-Sum algorithm. Every time the ballot is loaded, it initially displays the candidates in a random order to prevent ranking bias based on order of appearance.

After a voter has cast their electronic ballot, they cannot modify that ballot and cannot vote in the election again as the same voter.

C. User Authentication

As previously mentioned, voters and authorities are verified using their email address when voting or creating their authority account, respectively. In essence, both voters and authorities are identified on the site by their email address, since that is the only currently verified piece of information.

Voters do not have traditional "accounts" on the Cryptovote website which would allow them to log back in with a password after voting; this was a design decision to decrease the time and effort required for users to cast a ballot. Authorities, however, do have credentials that enable them to log back into the site after their initial account creation. This is necessary for authorities to be able to manage the election after its creation; i.e., to add additional voters and tabulate the final results of the election.

Authorities have the choice of two different methods for authenticating with the Cryptovote web application: WebAuthn and passwords.

WebAuthn is a browser-native protocol that enables public key authentication of users via JavaScript for websites implementing the WebAuthn standard [16]. WebAuthn works by requiring that the server store the public key of a user instead of a hashed password. When a user requests authentication via WebAuthn, the server submits a challenge to the client. The client completes that challenge using their secret key. Secret keys may be stored by the user in a variety of fashions, but typically involve some kind of security hardware. For example, on a MacBook, WebAuthn security keys can be placed into local storage and protected with the fingerprint reader built into the computer's hardware. For more portability, USB security keys can also be used on most computer hardware. The WebAuthn workflow is visualized[8] in Figure 4.

The initial design of the Cryptovote web application used WebAuthn as the only form of authentication for authorities because of its desirable security properties. However, we later chose to add password authentication as a secondary option because user testing revealed that a substantive portion of user hardware does not natively support WebAuthn. Rather than requiring some users to purchase USB security keys in order to use the web application, we opted to allow a more traditional form of authentication in addition to WebAuthn.

D. Implementation

The Cryptovote web application backend was written in Python using the Flask microframework. The frontend of the web application is built using the Bootstrap library. Database

[7]The web interface is accessible at https://cryptovote.ml.

[8]Graphic from https://auth0.com/blog/introduction-to-web-authentication/.

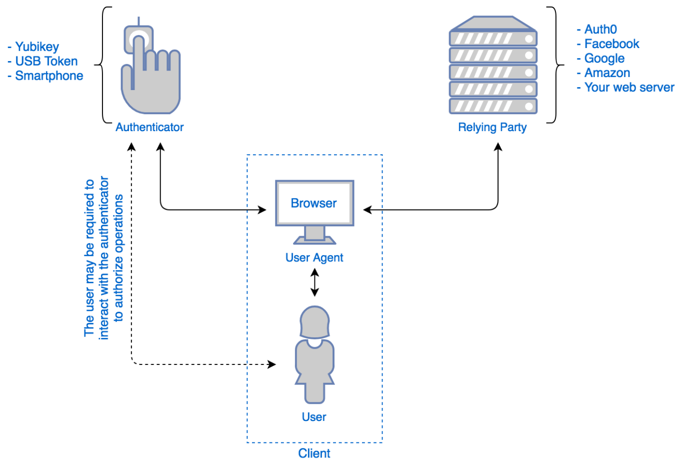

Fig. 4: Visualization of WebAuthn protocol.

functionality is abstracted by the SQLAlchemy toolkit, which enables support for most major SQL-like database hosts.

The source code for the Cryptovote web application can be found at https://github.com/cryptovoting/cryptovote.

VII. Future Work

A. Zero-knowledge proofs

The Shuffle-Sum paper [1] describes multiple protocols that serve as zero-knowledge proofs for validity of different stages of the STV tally protocol. They include, but are not limited to, proving that a certain candidate should be eliminated, proving that a certain candidate does indeed have enough votes for the quota, and proving that some candidate has the lowest tally in a round.

Due to time constraints, we have not implemented these zero-knowledge proofs, but they would be a very useful feature both in the Python package and on the website, as they can make the election results more verifiable for the voters.

B. Table-Sum Protocol

A faster alternative for the Shuffle-Sum protocol, called Table-Sum, is proposed in the same paper as the Shuffle-Sum protocol [1]. The main idea behind Table-Sum is that it uses an $m \times m$ matrix instead of a ballot with candidates and preferences. A matrix entry at row i, column j is an encryption of -1 if candidate i is preferred to candidate j, and an encryption of 0 otherwise.

This way, elimination of candidates does not require converting between three types of ballots but rather just requires ignoring a column of the matrix. On the other hand, it comes with a spatial complexity of $\mathcal{O}(m^2)$, where m is the number of candidates.

C. Multi-Authority Tallying

The Damgård-Jurik cryptosystem was chosen due to its homomorphic properties and because it can be structured as a threshold cryptosystem, meaning decryption requires multiple authorities, as described in Section II-F. Furthermore, those authorities can all perform their part of the decryption by themselves and combine their results without ever revealing their own secret keys to each other.

The latter property enables us to have multiple election authorities. While our implementation does use threshold decryption with multiple shares of the secret key, it is not yet done in a secure way. Specifically, our current implementation has a single process running the Shuffle-Sum protocol, and this one process has access to the entire private key ring with all the private key shares. This is not desirable, as it means that one party holds all the secret key shares at the same time.

That said, supporting secure threshold decryption would only require a few infrastructural changes to the code. There would still be a single process computing the tally, but this process would not have access to any of the private key share. Instead, any time the process requires a decryption, it would send the ciphertext to each of the election authorities, who would compute and send their portion of the decryption using their share of the private key. The election tallying process would then collect these shares and compute the full decryption. This can all be done without any election authority revealing their portion of the private key. Note that measures will need to be taken in order to ensure that the server requests a valid decryption for the tallying process (e.g. not a malicious ciphertext like an individual ballot). Alternatively, the authorities could do all the computations in parallel. This way, each authority knows which decryptions are actually necessary to tally the vote, allowing the authorities to determine whether any requested decryptions are malicious. One downside of this approach, however, is that it will require more computation power.

Although this added infrastructure would slow down the computation slightly due to network latency, each authority can perform their portion of the decryption independently and simultaneously, meaning this design can support an arbitrary number of election authorities with only a minimal increase in runtime.

D. Shuffling Protocol

A secure shuffling protocol is vital for ballot confidentiality. Currently, our shuffling protocol only generates a pseudo-random permutation. As such, the authority doing the computation can reconstruct the permutation and learn about the individual ballots after threshold decryption, breaking confidentiality of the ballots. In fact, even a random oracle that takes as an input a ballot and outputs the ballot with the columns randomly shuffled would not address this security vulnerability. For full security, all authorities should influence the shuffle outcome, and the encrypted ballot content must be re-randomized.

E. Web Interface

There are multiple enhancements that we would like to make to the current Cryptovote web application implementation.

First, the secret keys of authorities are currently stored by the server in the web application database. In order to provide the full security guarantees of Shuffle-Sum, future work will involve removing the storage of any authority's secret key from the server. Instead, authorities would need to store secret keys locally on their own machines such that the server does not have the ability to decrypt ballots without action on the behalf of the authority.

Similarly, future work will involve moving all Shuffle-Sum operations involving secret keys to client-side JavaScript instead of server-side Python code. This, too, will help ensure the secrecy of authority secret keys.

Additionally, future work will involve extending the web interface to allow multiple authorities per election. While the modifications to the underlying protocol will be minimal, the largest challenge will involve creating a secure channel in which all authorities can directly communicate to tabulate election results in the browser and share decryption information as described in a previous subsection.

In addition to these larger extensions, other future work on the web implementation will involve implementing two-factor authentication and having voters sign their encrypted ballots to verify ballot integrity.

VIII. Conclusion

Although the Shuffle-Sum protocol works on the specific single transferable vote type of ranked choice voting, it should be possible to apply it to any ranked choice election with minor modifications of the protocols. We hope that our Python implementation will be a good start for anyone who wants to implement a slightly different algorithm, and they could always use a different type of homomorphic encryption as our code is designed in a modular fashion. Nevertheless, we believe that Damgård-Jurik should do the job in most cases thanks to its many useful properties.

We decided to set up a website to make it more convenient to use, as it would be very convenient for voters to vote and for authorities to compute results. Hopefully, it will enable people to run secure ranked choice elections online.

Although our current implementation needs a lot of computational power for an election on a scale of a state or nation, it may still be more efficient than in-person voting as a lot of personnel is required to both collect and tally votes. Election results also do not need to be available immediately, so a couple of hours on a large computing cluster should be acceptable.

Acknowledgment

The authors would like to thank the staff of MIT course 6.857 for their contributions to this paper. Particularly, we would like to thank Professor Ron Rivest and Professor Yael Kalai for providing course instruction on many of the topics addressed in this paper and for providing suggestions on how to approach the problems we chose to address. Additionally, we are grateful for the support given by TA Leo de Castro, who was the assigned advisor for this paper.

We are further grateful for the input of Josh Benaloh, whose input into this project led us to his own work on Shuffle-Sum, which ended up being the basis for this paper. We are also grateful for the input of Vanessa Teague.

References

[1] J. Benaloh, T. Moran, L. Naish, K. Ramchen, and V. Teague, "Shuffle-sum: Coercion-resistant verifiable tallying for stv voting," *Trans. Info. For. Sec.*, vol. 4, no. 4, pp. 685–698, Dec. 2009, ISSN: 1556-6013. DOI: 10.1109/TIFS.2009.2033757. [Online]. Available: http://dx.doi.org/10.1109/TIFS.2009.2033757.

[2] I. Damgård and M. Jurik, "A generalisation, a simplification and some applications of paillier's probabilistic public-key system," *Basic Research in Computer Science*, Dec. 2000. [Online]. Available: https://www.brics.dk/RS/00/45/BRICS-RS-00-45.pdf.

[3] "Electronic voting," *Wikipedia*, [Online]. Available: https://en.wikipedia.org/wiki/Electronicvoting.

[4] X. Yang, X. Yi, S. Nepal, A. Kelarev, and F. Han, "A secure verifiable ranked choice online voting system based on homomorphic encryption," *IEEE*, pp. 20 506–20 519, 2018.

[5] A. Azougaghe, M. Hedabou, and M. Belkasmi, "An electronic voting system based on homomorphic encryption and prime numbers," *11th International Conference on Information Assurance and Security*, 2015.

[6] Y. Zhao, Y. Pan, S. Wang, and J. Zhang, "An anonymous voting system based on homomorphic encryption," *10th International Conference on Communications*, 2014.

[7] "First-past-the-post voting," *Wikipedia*, [Online]. Available: https://en.wikipedia.org/wiki/First-past-the-post_voting.

[8] "Single transferable vote," *Wikipedia*, [Online]. Available: https://en.wikipedia.org/wiki/Single_transferable_vote.

[9] "Spoiler effect," *Wikipedia*, [Online]. Available: https://en.wikipedia.org/wiki/Spoiler_effect.

[10] E. Watkins, "How gary johnson and jill stein helped elect donald trump," Nov. 2016. [Online]. Available: https://www.cnn.com/2016/11/10/politics/gary-johnson-jill-stein-spoiler/index.html.

[11] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," *EUROCRYPT'99*, pp. 223–238, 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.4035&rep=rep1&type=pdf.

[12] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, pp. 612–613, 11 Nov. 1979. [Online]. Available: https://dl.acm.org/citation.cfm?doid=359168.359176.

[13] [Online]. Available: https://gmpy2.readthedocs.io/en/latest/#.

[14] "2016 san francisco board of supervisors election," *Wikipedia*, [Online]. Available: https://en.wikipedia.org/wiki/2016_San_Francisco_Board_of_Supervisors_election.

[15] [Online]. Available: https://www.rankedchoicevoting.org/data_clearinghouse.

[16] Dirk Balfanz, Alexei Czeskis, Jeff Hodges, J.C. Jones, Michael B. Jones, Akshay Kumar, Angelo Liao, Rolf

Lindemann, and Emil Lundberg, “Web authentication: An API for accessing public key credentials level 1,” W3C, Mar. 4, 2019. [Online]. Available: https://www.w3.org/TR/webauthn/ (visited on 05/15/2019).