[[1]](#footnote-1)

Secure Voting via Homomorphic Secret Sharing

Ben Jones, Ethan Mines, Philip Speegle

*Abstract*—This paper describes a recent alternative to Fully Homomorphic Encryption, called Homomorphic Secret Sharing. Recent developments in homomorphic secret sharing have led to more secure and efficient schemes. These recent developments are simulated in the context of secure voting. The implementation presented in this paper verifies correctness of a probabilistic error bound and demonstrates a low efficiency. There are several improvements that can be made to the implementation, but none that are substantial enough to provide a practical voting software.

# INTRODUCTION

I

n this paper, we implement improvements to Homomorphic Secret Sharing (HSS) schemes to secure voting. HSS Schemes were first introduced in 2016 by [1]. We begin with a high-level comparison of HSS schemes with their predecessors, Fully Homomorphic Encryption (FHE) and Function Secret Sharing (FSS). This comparison provides clarity and context that highlight the benefits that HSS provides in relation to the others. In particular, we consider the improvements that [2] make to existing HSS schemes, and apply them to voting. We construct an implementation of the ElGamal cryptosystem, which will be used as a major basis element in the implementation of the HSS algorithms.

## Homomorphic Encryption

Homomorphic encryption allows a user to perform operations on encrypted data without decrypting it [3]. An example use case of this is secure cloud computation. With homomorphic encryption, Alice could encrypt her financial data and upload it to a remote server, owned by Bob. Bob could then sum the numbers in the financial data, without knowledge of what the numbers are. Bob then sends back the encrypted result, which is finally decrypted by Alice. In this case, Alice is the only entity that knows the plaintext, but Alice was able to outsource her computation to Bob.

In [2], Gentry categorizes homomorphic encryption schemes into several classes. Two examples of these are the partially homomorphic and fully homomorphic [2]. In a partially homomorphic cryptosystem, the user is able to perform precisely one operation on the encrypted data, such as addition or multiplication [2]. In contrast, fully homomorphic encryption (FHE) allows the user to do arbitrary computation [2]. In 2009 and after being an open problem since 1978, Craig Gentry proposed the first viable FHE scheme in [2].

One way to formalize the concept of FHE is to create a scheme ℰ with an algorithm Eval that can evaluate an arbitrary logical circuit C, with algorithms Encrypt and Decrypt for encrypting data and decrypting it, respectively. Gentry accomplished this with a concept called “bootstrapping.” A scheme ℰ is bootstrappable whenever the evaluation algorithm can compute the decryption algorithm Decrypt and an augmented NAND gate. From this, Gentry proves that you can create a complete set of circuits [3]. Finally, he constructs a bootstrappable scheme.

## Function Secret Sharing

This subsection summarizes the seminal paper on function secret sharing in a format that will be used to compare FSS to HSS. The full details can be found in [4].

A function secret sharing scheme splits a function into secure "shares" such that certain subsets of the shares cannot be recombined to give an advantage in computing the original function.

More precisely, an FSS scheme has 6 components. The first is a set of p parties, denoted [p] who will receive the shares. The second is a subset T of [p] that are adversaries. The third is a class of functions, ℱ, containing our eventually secret function. The fourth is an output decoder algorithm, Dec, that takes the shares as inputs and decodes it as a single output. This is usually a sum function. The fifth is a key generation algorithm Gen that generates the p “keys” or “shares” for the input function, according to a security parameter constraint. The sixth and final component is an evaluation function, Eval, that evaluates each function share.

The FSS is also subject to a correctness and security constraint. The correctness constraint rigorously states that with probability 1, the decoder algorithm operating on the shares generated from a function, f, will return the same value as f. The security constraint describes in rigorous terms that a set of adversaries has a probabilistically negligible advantage over randomly guessing the function values [3].

## Overview of Homomorphic Secret Sharing

Homomorphic secret sharing was originally conceived as a dual to function secret sharing, which splits the *program* rather than the inputs into shares [1], [4]. Of the variants of homomorphic secret sharing described in [1], we use *Distributed Evaluation Homomorphic Encryption*, which allows multiple clients to send shares of inputs to two servers [1], [2].

Our Distributed-Evaluation Homomorphic Encryption scheme consists of 5 parts,

1. An algorithm Gen(1λ) that generates the public and secret keys, pk and (ek0,ek1) respectively, with securityparameter1λ∈{0,1}λ.
2. An algorithm Share that outputs encrypted, additives hares of a ciphertext, using the keys from Gen.
3. An algorithm Eval that each server uses to run a program on the encrypted data. In this case, the programs will be voting programs, although it applies to a much broader class of programs.
4. In algorithm Dec that each client, in this case voters, are able to use to decrypt the result from Eval. In our implementation, this is simply a sum of the additive shares.

They describe three different “levels'' of encoding [2]. The first is ElGamal encryptions of the inputs [2]. The second is additive shares in which, given a variable where . The third are multiplicative shares in which, given a variable where . These notions can of course be generalized to cases with servers where each share vector has components [5].

Restricting ourselves to the current case of , each clients splits its input into additive shares and sends one share to each server [2]. Each server must then perform an evaluation of its share such that given evaluations is equivalent to the computation desired by the clients.

For these “subprograms” executed by the servers, [2] permit only four types of instructions:

* Load input:
* Add variables:
* Multiply variable by input:
* Output result:

The circumflex mark denotes that for each input and variable , an individual server does not have the actual value of this variable, but rather an additive share of it [1]. Note that additive shares are homomorphic over addition [2]; that is, given with and a similar :

This formalism is known as Restricted Multiplication Straight-Line Programs (RMS) [1]. Do note that since we cannot multiply two memory locations , this formalism does not cover arbitrary boolean circuits.

Now, the multiplication instruction temporarily creates multiplicative shares, but these are then converted back to additive ones with some probability of error [1]. During computations by the servers, multiplicative shares of a program variable must sometimes by converted to additive ones [2]. This is done via a probabilistic procedure known as DistributedDLog that relies on a pseudorandom function. If each of the two servers call the procedure on their multiplicative share , they each receive a new additive share such that . Since this is a probabilistic procedure, the outputs fail to be correct additive shares with probability at most , where is a parameter that can tweaked such that lower values of result in longer runtimes of the procedure. This is the primary limiting factor of the size of programs executed by the servers: error may accumulate by repeated calls to DistributedDLogn [1].

Because these instructions maintain the invariant that every variable is represented as an additive share [1], it is easy to reconstruct the actual result from the servers' individual outputs: given sharing , we have

# Gen: Simple ElGamal Cryptosystem

The ElGamal cryptosystem is an asymmetric cryptosystem making use of a public key and a single-use private or `ephemeral key.'

One of its variants is homomorphic with respect to addition, making it a suitable candidate for Homomorphic Secret Sharing.

## Key Generation and Encryption

Consider a voting system where there are questions and two options for each question, {0,1}. Let denote the set of all possible permutations of options chosen given that all the questions were answered. Then, we define the bijective map which maps each coefficient of to the chosen option for question . Note that this bijection implies that there are possible outcomes for a vote. For a quick example, let . Then, the eight possible outcomes are . It will be advantageous to utilize the fact that all elements of can be written in binary by only using their coefficients. As such, we seek a group order such that .

## The Group G

The hypotheses in [1] require that the group G be cyclic of prime order. In this subsection, we compare the usage of different groups G, not necessarily of prime order, and how they may impact the HSS scheme. We consider groups of order pq, p−1, s= 2pq+ 1, and p, where p, q, and s are primes. We note that [2], [6] establish several groups of prime order that are usable in the context of HSS. After investigating these other groups, we intend to move forward with the groups of order p, because [2] gives some examples that are computationally efficient to implement. These will also be discussed in this subsection. In the current work, we use groups of order p−1. We will see that this is moderately successful. Given that there are known attacks on groups with this construction, we plan to adapt to efficient, large groups of prime order. In spite of this, our groups of order p−1are able to successfully handle many computations.

### The case for Group Order pq:

Ideally, we would like to be flexible in terms of size. Finding large primes is nontrivial, and if can be expressed in terms of the product of two primes, this would alleviate some of the runtime necessary to generate large primes. As such, a good first attempt at determining a group order would be to let be a cyclic group of order , where and are primes. In this section, we give propositions explaining the desirability of such a group order, but ultimately, a group order of this form has a fundamental flaw that will render the technique unusable.

Recall that , and the amount of generators of is , where each element in is all for which . Then, the proposed generation algorithm is as follows. First, we randomly generate two distinct prime numbers and , preferably making the product large. Using the Miller-Rabin primality test, large values of primes can be found easily. Without loss of generality, assume . Compute and let . Randomly choose an integer until is sufficient such that . The following results show that, for sufficiently large and , the time this takes is essentially negligible.

**Proposition II.1.** Let . Then,

where is Euler’s Totient function.

*Proof.* It is well-known that is a multiplicative homomorphism, so we have . The primality of both and implies and , and the result trivially follows.□

**Remark:** Indeed, is quite close to itself. This notion is rigorized in the following theorem:

**Theorem II.1.** Let where for two distinct primes . Then, with probability.

*Proof.* Consider . Since and are both prime, . For all integers . So the amount of possible generators for is surely since we require We sample a total of elements, so the probability of choosing a generator is , as desired.□

We claim that this probability is sufficiently large to guarantee a generator very quickly. The probability that the algorithm fails times in succession is

, (1)

since each trial is independent. Indeed, even if is minimized (), the probability of failing to generate 3 times is less than . After this, we randomly choose and let . Since is cyclic, the possible values for is (we remove the identity). We denote as our public key, and the value is the private key. It is important to note, however, that keeping close to will achieve more desirable probablistic results than fixing and making large. The reason for this is due to the fact that and vice versa.

After the key generation, a second party will encrypt a message using the public key generated by the above algorithm. We choose an integer and compute the *shared secret* denoted by . The cyphertext is computed by and . The ciphertext is then sent back to the first party. Finally, it is important to note that this method of encryption is fully reliant on the Diffie-Hellman problem being NP-hard, as if this were not the case, an attacker could quickly compute given and [7].

### Drawbacks to Group Order pq:

Despite all the benefits of having a group order of , there is a fundamental flaw which completely negates its usage. While encryption is done in an additive group, decryption is done in a multiplicative group: this multiplicative group must be isomorphic to the additive group.

The implications that these groups must be isomorphic are huge. Since is cyclic, there must be some that is ALSO cyclic. However, this group cannot be of the form because this requires the order of the group to be . So, we could try setting the additive group to . This way, at least and are candidates for an isomorphism. The reader might wonder if this simply means that and must be chosen carefully as to allow this. However, this is quickly undermined by the following theorem:

**Theorem II.2.** Let be odd primes. Then, is not cyclic.

*Proof.* First, we will show that all cyclic groups of even order have a unique element of order 2. Let , and assume by contradiction that we have more than one element of order two. Choose from these elements and consider . It must be the case that ; however, is not cyclic, a contradiction. This is a contradiction due to the well-known fact that all subgroups of a cyclic group must also be cyclic.

We will now show that there are two elements of order 2 in Since . Clearly, , so has order 2. Now, consider an element such that

and . Since , there exists a solution for this set of constraints. However, , so this element is not because . However, and , so has order 2 as well. As such, by the above lemma, cannot be cyclic, and the proof is complete.

If , this is essentially the same as using a single prime, except the number of generators is not as easy to calculate and could be zero. Although seems like a great option to choose the group order in theory, this crucial drawback makes its case fall completely apart. However, there exist other candidates that could potentially have similar benefits while not having the same drawbacks.

### The case for Group Order p-1:

Here, we discuss using the group , the multiplicative group modulo a prime p. This group has order , since is prime and for all nonzero . We now discuss how to find a candidate generator.

The number of generators in this group is [8]. This means that there is always a generator for this group. A randomized approach to searching for generators has a probability of succeeding of . It has been shown that

and hence that in infinitely many cases φ(n)is arbitrarily close to n. This suggests that the desired probability is quite large, and a randomized approach may be successful.

However, even when $, the probability is not small. In this case, even though is bounded by , many values of hover around that value, meaning a randomized generator test would only have to run a few times for a generator to be found. In fact, there exist many sequences $ such that for each , but stays constant (which will be proven in this paper). A very simple sequence, , is proven below to have a fixed success rate of :

**Proposition II.2.** for all integers .

*Proof.* This is an inductive argument, but not mathematical induction per se. Note , since , our base case holds. (Note that we cannot simply say because these numbers are not relatively prime). Now, also only has factors of and . Because of this, by the divisibility rules, any number ending in or cannot be in . Since these are the only factors, all other numbers are in . This is equivalent to saying , which means

which is the desired result. Note that, in many cases, is prime, so this could, in theory, be a result of one of these groups.

A table showing values of , and is shown in the above figure for emphasis. These values were easily computed by factoring . Coincidentally, all values chosen had a success rate of 40%. This figure will stay in that range for even larger numbers.

TABLE I

Values of Euler’s Totient Function

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 10 | 4 | 40% |
| 100 | 40 | 40% |
| 500 | 200 | 40% |
| 10000 | 4000 | 40% |
| 25000 | 10000 | 40% |

However, an iterative solution may be similarly efficient and is deterministic. It is shown in [9] that an upper bound on the smallest generator modulo p is , which is very fast asymptotically.

First, we factor . While factoring large numbers is difficult, we are guaranteed that , since is odd. This allows us to more rapidly find the prime factors of .Then, given the prime factors , we consider . If generates the group, then , and so since , we have that

Now if does not generate the group, it must have some finite order dividing , by Lagrange’s Theorem. Then there exist such that , and so . This implies that for some Then since , we have . Therefore, if is not a generator, then will be equivalent to for some prime dividing . This gives us a test for if any given element is a generator of .

### Drawbacks to Group Order p-1

Unfortunately, this relies on the ease of factoring . However, if is factored easily, then [10] shows that we have violated the primary cryptographic assumptions of HSS, called Decision Diffie Hellman (DDH) [2], and hence we have lost all security of the protocol.

### The case for

The case where, for some prime and are prime is similar to the above case where we take . This approach, however, seeks to combat the issues raised with DDH. Since we start with primes , we know that the only small factor of is , and so it will be much more computationally difficult to factor .

### Drawbacks to

Unfortunately, while we can guarantee that it is more difficult to factor than the general group , we cannot guarantee that it is asymptotically more difficult. In particular, we still violate Decision Diffie Hellman. This case falls into Boneh’s case 1.1.2 [10], where we have , prime and arbitrary, and . In our case, we have a prime , and hence we require both and . This would imply , which is false for as small as . Hence, we still do not have security.

### The case for Group Order p

It is explained in [6] and [10] that there are multiplicative groups of prime order that do not necessarily violate DDH, but they must be chosen very carefully. The suggested case is , where and are both special primes, and in particular is a pseudo-Mersenne prime such that , where a pseudo-Mersenne prime takes the form [6]. Such primes arecalled “conversion friendly” in [2] and [6], since they make it computationally easy to compute the convert shares procedure. Then by Sylow’s theorems from Group Theory, we know that there exists a Sylow-subgroup of order in. In this group, every non-identity element is of order , and hence generates the group.

By Lagrange’s Theorem, we can see that the order of is or . It cannot be , so it must be or . If it were , then would generate . However, it is impossible for whenever is even and , so it is not possible for . Hence, . Sylow’s theorems also guarantee that all Sylow −subgroups are conjugate to each other, or that for some . However, since ||= , we can have at most two such Sylow −subgroups, simply by counting elements. Then must be in one of them, and furthermore, must generate it. Therefore, we have a such that is prime and a generator, namely .

### Drawbacks to Group Order p:

In general, it may be difficult to find a pseudo-Mersenne prime and also verify that for some prime . To this end, [2] gives explicit conversion-friendly primes and groups that work such as and . A smaller case, yields and , which is useful for verifying correctness.

There are also other multiplicative groups of prime order, namely some groups over elliptic curves. It is noted in [6] that these are not subject to as many optimizations as there are with conversion-friendly primes, and hence you lose multiple orders of magnitude of efficiency. They did, however, leave open the problem of optimizing the elliptic curve implementation, which could prove useful under further research. Given the slow speed and difficult implementation, we will not be using elliptic curves.

### Selecting Primes and generating the groups

Now that we have a target set of groups we wish to generate, we now need to actually generate our primes and the resulting group.

Based on the input , the security parameter, we choose candidate primes q less than . Then, we use the Miller-Rabbin primality test, as described in \cite{textbook}. Then, we set p=2q+1, and check that p is a prime and that . If either of these conditions fails, we calculate a new q and a new p. This procedure repeats until suitable conversion-friendly primes are found \cite{boyle2017homomorphic}, \cite{boyle2017group}.

We then create the group , and refer to the generator 2 as g as needed. This is our multiplicative group \cite{boyle2016\_first\_Reference}. We also must frequently calculate powers of the base in our scheme, so we describe our efficient algorithm for computing . We implement a recursive, memoized algorithm based on the binary representation of and repeated squaring. First, we calculate , then , and so on, until we have . Then, for each bit , if the bit is 1 we multiply our cumulative product by . This algorithm will then run in O() time.

All other operations are done in the standard way, except calculating modular inverses, which is discussed in detail in the following section on decryption.

## Decryption

Decryption is done in the multiplicative group with order . This means that for any element in this group, . This will be useful in the decryption algorithm.

**Proposition II.2.** The shared secret .

*Proof.* Routine. and the result follows. □

As such, with the private key , the first party can find the shared secret. Next, we compute with the main result of this section:

**Theorem II.2.** The inverse **.**

*Proof.* By Proposition II.1, . This implies that . By extension, . As such, . Since , . Finally, since , quick substitution gives . □

With , one can compute the original message by computing , which can be mapped back to the original plaintext.

TABLE II

ElGamal Simulation Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| p | q | Votes | Result | Duration (s) |
| 1433 | 1553 | 0101101111 | 7 | 52.17 |
| 1439 | 2039 | 1000110010 | 4 | 123.89 |
| 1103 | 1801 | 1101111110 | 8 | 76.47 |
| 1447 | 1573 | 0001000011 | 3 | 71.35 |
| 1471 | 1571 | 0111100100 | 5 | 87.18 |
| 1847 | 1879 | 1010101010 | 5 | 228.65 |
| 1033 | 1867 | 0110101011 | 6 | 58.74 |
| 1327 | 1721 | 0100000100 | 2 | 77.43 |
| 1193 | 1553 | 0110011000 | 4 | 76.00 |
| 1187 | 1619 | 1111010100 | 6 | 61.62 |

## Additive Homomorphism

The key aspect that makes this variant of ElGamal useful is that multiplication in the ciphertext space is equivalent to addition in the plaintext space [7]. Given encryptions of messages and , we have

which is a valid ElGamal encryption of input .

# Simple ElGamal Results

Before implementing a full homomorphic secret sharing scheme, we pilot a simple voting application using an ElGamal cryptosystem, reminiscent of that of [7]. We implement the simulation in Python3, with all keys and primes being generated by the “secrets” module. We have clients each cast a binary vote . We perform ten iterations of this, noting the correctness of the output and the runtime in Table II.

# Share

## AdditiveShare

This routine generates a tuple denoted such that . This is done by choosing a cryptographically random and setting .

# Evaluation Algorithm eval

In this section, we describe in more detail the procedures in Eval as presented in [1]. We will assume for this section that we have access to a multiplicative cyclic group of prime order and generator . The Homomorphic Secret Sharing scheme also requires a security parameter, , which we will use liberally. There are primary functions, additiveShare, multShares, and convertShares. These primary functions make frequent use of a subroutine called the Distributed Discrete Logarithm, or DistributedDLog and a Pseudo-Random Function (PRF).

## Distributed Discrete Logarithm

We begin by discussing the Distributed Discrete Logarithm, or DistributedDLog. For this, we first require a pseudo-random function (PRF) that can be seeded by elements of .

1) Pseudo-Random Function: We require a pseudo-random function (PRF) . It is laid out explicitly in [12] how to construct a Pseudo-Random Function given any Cryptographically Secure Bit (CSB) generator. For our CSB, we take Python’s PRNG library, Random, which is built on the Mersenne Twister algorithm [13]. This is notably a hole that should be solved with a more advanced CSB, since this PRNG is not cryptographically secure. The reason for this conscious choice is that Python’s Cryptographically Secure Pseudo-Random Number Generator does not allow seeding, by design, which is an essential functionality of our PRF. We now suppose that we have an input and security parameter . Mimicking the notation of [12], we say that our CSB generator is a function . If , we also say that and , which can be thought of as the left and right halves of the image of . Now let be the -th bit of the security parameter , where is the most significant bit. We then define the PRF inductively.

We first let and . That is, if the MSB of is 1, we take to be the right half of . If the MSB is 0, we take the left half. Then for every , we take . Finally, we call . Thus, the PRF is a function We will commonly use the notations and . Essentially, the PRF is indexed by , so we can refer to the PRF by its id.

2) DistributedDLog: The purpose of the DistributedDLog is to help pass from multiplicative to additive shares. This procedure is probabilistic with a controllable error tolerance and relies heavily on the PRF from above [1]. We consider a “distinguished” subset of , those elements with the most significant bits being . This will be determined later. Each call to DistributedDiscreteLog outputs the power such that for a secret share , is the nearest element in the distinguished set.

DistributedDLog takes the following inputs: . The parameters and are used to control the error bound and maximum number of steps taken, as well as the number of leading zeros for elements in the distinguished set. This procedure iteratively computes until either a maximum number of steps has been taken or the first binary digits of are . It then outputs the number of steps taken to achieve this.

The proof that DistributedDLog gives the desired result and the upper bound on the steps in DistributedDLog is covered in greater detail in [1].

In the first construction of a homomomorphic secret sharing scheme, the DistributedDLog function searched for the pattern , as opposed to the current [1]. This change to the form is one of the main optimizations presented in [4], which improves the theoretical running time of the DistributedDLog by a factor of two. Since the search space is effectively cut in half at the cost of a very small operation, checking the initial bit, we need to only find a sequence of n-1 0-bits.

Two other optimizations are presented in parallel with the above pattern switch\cite{boyle2017homomorphic}. Although these optimizations do not appear in our implementation, their impacts are relevant in describing the limitations and potential for improvement. These improvements are only necessary because of a potential error due to the context of the ConvertShares procedure, this improvement will be discussed later \cite{boyle2017homomorphic},\cite{boyle2017group}. A third major optimization of the DistributedDLog procedure is presented in \cite{boyle2017homomorphic}, which will also be discussed later.

## MultShares

This routine takes as inputs an ElGamal-encrypted input with secret key and ciphertext , additive shares and and outputs a multiplicative share denoted ,which is a sharing of . We then compute . This is the multiplicative share.

For the proof of correctness, see [1].

In later papers, the DistributedDLog function is optimized by factors of or more [2]. This is a potential future improvement we can make.

## ConvertShares

This routine takes as inputs a parameter identifying which server the share will go to, a multiplicative share , a randomly generated execution identifier , an error parameter , and the maximum size bound . It outputs the additive share for the server .

The maximum size bound controls how large the value of any computation can be. By enlarging , we increase the runtime.

We denote the first bits of . This length is the required number of zeroes in the output of DistributedDLog for the required fault tolerance, per [1]. Also, denote ’s share of by . We are now ready to describe the actual algorithm. If , we set . Then, we let be the output of DistributedDLog. Finally, we output either or , depending on . This is the additive share we desire [1].

# RMS Programs

Mimicking most of the examples given in [2], [1], we let the number of servers be with indices . As in an ElGamal cryptosystem, we find a cyclic group generated by [1]. Here, we choose a single prime as our divisor; thus, .

We choose a secret key of bits and let the public key be

where is used for encryption, and the latter componentsare the ElGamal encryptions of the individual bits of the secret key (still using the entire as the key) [1]. Likewise, each of the two servers received an evaluation key

## Vote Counting

We first do a simple tally of binary votes of clients, with representing “yes” and representing “no”. Each server receives a share from every one of the voters and performs its own local summation. Each server receives inputs , each of which corresponds to one of the clients. We first load each input into memory:

In the public-key variant of HSS, rather than loading the input directly into memory, we multiply it by a symbolic constant [1]. This constant is shared between the servers in the same way that any variable is [1]. Since a given memory value is represented as , this constant is obtained from the values and in the local evaluation key. Following this, we perform a cumulative sum over the variables:

Finally, each server outputs its local tally:

The complete tally of votes is reconstructed by adding the two outputs of the servers: .

## Unanimous Vote

We also perform a vote in which only unanimity is determined, 1 being output for unanimous agreement and 0 otherwise. Thus, it is an AND gate over the votes [2]. A single vote is loaded into memory initially, followed by a cumulative product over all the votes:

TABLE III

HSS Simulation results

|  |  |  |
| --- | --- | --- |
| Program |  | Accuracy |
| Counting |  | 100% |
| Unanimity |  | 17.3% |

# Other Applications of HSS

# HSS SIMULATIONS

We simulate each of the two programs 1000 times. In the case of the unanimous votes, we ensure exactly 50% of the inputs are unanimous. Otherwise, all inputs are random. We take votes and set the error bound . For the counting program, we set the magnitude bound . For the unanimity program, only one bit is needed to encode any memory value, so . Results are shown in Table III. While the small group orders used resulted in a remarkable perfect accuracy for the counting program, the greater use of multiplication in the unanimity program led to a significantly higher error rate. After all, it is in the multiplication of shares that the potential for error occurs [2].

# Limitations

In this section, we discuss the drawbacks of our implementation, particularly those dealing with efficiency and security.

Due to the limited scope of this project, we were unable to implement several of the improvements as done in [4]. While they are not in our implementation, it is still important to discuss their potential benefits and impacts they could have on our running time and security.

As mentioned in the description of the DistributedDLog routine, there are several improvements that were made in [4]’s implementation.

The first such improvement is related to the so-called “danger zone” of distinguished points [4]. The ConvertShares algorithm will fail if there is a distinguished point between the shares that party b= 0 and b= 1 compute DistributedDLog with, and so the space between them is called the danger zone [10]. This improvement just marks all the distinguished points in the danger zone as non-distinguished. They note that this only has a marginal impact on the running time [4].

The second improvement is to randomize the conversion algorithm, in a way that lowers the average failure probability

10 decreases by a factor of 8, leading to an overall efficiency increase by a factor of 16 [4].

The third improvement is a low-level optimization of the way to search for distinguished group elements [4]. Because the distinguished points all contain a nice structure, namely some large number of consecutive 0-bits, they implement a “window” algorithm that can eliminate a large number of candidate distinguished points quickly. This window algorithm, first described in [10], splits an input into strips of length d/2 and checks if any of those strips are zero. If none of the strips are zero, then that input element several others following it cannot be a distinguished point, so they can be skipped [10]. The window size was doubled from 32 to 64 bits in [4], which improves speed. This optimization cannot be used in our implementation because it is too slow to support groups of size larger than 64 bits.

Another limitation of our design is that we chose not to reference any libraries external to Python. While this improves the readability and cohesion of our software, the drawbacks of this can be seen in several places. First, we were not able to take advantage of the GNU Multiple Precision library as was done in [4], [20]. Second, we could not use an efficient, black-box approach to our PRF, which could save substantial computation time [21], [22]. We also could have tested our results with elliptic curves if we had used external libraries [23].

We do not implement the authors’ concept of leakage pads [4].

A fundamental requirement in most voting applications is confidentiality [24]. The cryptographic assumptions of DDH and the related Discrete Logarithm problem are challenged by quantum computing [25]. Because of this, a voting system built under any quantum-vulnerable cryptographic assumption could lose confidentiality in the future.

Fundamentally, this implementation only provides security and correctness on the vote-counting servers, which also leaves open the problem of authentication and authorization of voters. That is, Homomorphic Secret Sharing can protect against malicious vote-counting servers, but not against voter fraud. Therefore, a different layer of security would be necessary.

# Conclusions and future directions

We utilize Homomorphic Secret Sharing in the practical context of secure voting. We initially obtain a valid implementation of the ElGamal cryptosystem, then use this as a component in our larger HSS scheme. We simulate two programs whose output accuracy falls within the provided error bound. We recognize that λ=14 is too small a security parameter to proof the scheme against a brute-force attack and plan to move to a faster implementation based on C or C++ that will allow for much larger security parameters while still maintaining a reasonable runtime.

We also note that it would be possible to run HSS programs “in parallel” over vectors of inputs . Each component of would be encrypted just as the scalar inputs were in our simulations. For add instructions, addition would simply be performed over rather than [1]. Output instructions would be performed similarly, and it would even be possible to compute a different random offset for each component of the vector by passing in extra bits to the pseudorandom function denoting the index of the component of the vector. Multiplication instructions present the additional complexity of DistributedDLog, but the servers could all elect to only pass in the first component of their vector to the pseudorandom function φ at each iteration. Load instructions are of course again performed via multiplication instructions.

This would allow for “ballots” containing multiple questions or elections in which clients can cast their votes. That said, it does present the possibility of multiple questions be ruined by an error in DistributedDLog, as opposed to just one question in a scalar scenario.

The final future goal of our implementation is to incorporate actual networking to simulate the votes and server computation. This omission allowed for more rapid development within the given time constraints and faster execution time. However, it is a necessary step that would need to be taken for practical use in any voting application.

References

1. E. Boyle, N. Gilboa, and Y. Ishai, “Breaking the circuit size barrier for secure computation under ddh,” *Advances in Cryptology – CRYPTO 2016 Lecture Notes in Computer Science*, p. 509–539, 2016.
2. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orr`u, “Homomorphic secret sharing: optimizations and applications,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*,2017, pp. 2105–2122.
3. C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing,* ser. STOC ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 169–178.
4. E. Boyle, N. Gilboa, and Y. Ishai, “Function secret sharing,” in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 337–367.
5. E. Boyle, N. Gilboa, Y. Ishai, H. Lin, and S. Tessaro, “Foundations of homomorphic secret sharing,” *Innovations in Theoretical Computer Science,* 2018.
6. D. R. L. Brown and R. P. Gallant, “The static diffie-hellman problem,” 2004.
7. R. Cramer, R. Gennaro, and B. Schoenmakers, “A secure and optimally efficient multi-authority election scheme,” *European Transactions on Telecommunications*, vol. 8, 10 2000.
8. D. M. Burton, *Elementary Number Theory.* McGraw-Hill College; 4th edition, 1997.
9. V. Shoup, “Searching for primitive roots in finite fields”. *Mathematics of Computation,* vol. 59, pp. 369-380, 1992.
10. D. Bonch, “The Decision Diffie-Hellman problem,” in *Third Algorithmic Number Theory Symposium, Lectur Note in Computer Science,* vol. 1423. Springer-Verlag, 1998, pp.48-63.
11. R. Cramer, R. Gennaro, and B. Schoenmakers, “A Secure and Optimally Efficient Multi-Authority Election Scheme,” *European Transactions on Telecommunications,* vol. 8, 10 2000.
12. S. G. Oded Golreich and S. Micali, “How to Construct Random Functions,” vol. 33, No. 4, pp. 792-807, 1986.
13. M. Mastumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,” vol. 8, No.1, pp.3-30, 1998.

1. [↑](#footnote-ref-1)