[[1]](#footnote-1)

Secure Voting via Homomorphic Secret Sharing

Ben Jones, Ethan Mines, Philip Speegle  
bdjones13@crimson.ua.edu, elmines@crimson.ua.edu, pspeegle@crimsonua.edu  
CS542 - 001

*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) [2], [3]. 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 [4] makes to existing HSS schemes and apply them to voting.

We first construct an implementation of the ElGamal cryptosystem, which will be used as a major basis element in the implementation of the HSS algorithms [1]. We then implement two programs via HSS: vote tallying and unanimous voting.

## Homomorphic Encryption

Homomorphic encryption allows a user to perform operations on encrypted data without decrypting it [2]. An example use case of this is secure cloud computation [4]. 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 [2]. 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 [2].

In [2], Gentry categorizes homomorphic encryption schemes into several classes. Two examples of these are the partially homomorphic and fully homomorphic variants [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]. FHE saw its first viable implementation in 2009, after being an open problem since 1978 [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 [2]. Gentry accomplished this with a concept called “bootstrapping” [2]. A scheme ℰ is bootstrappable whenever the evaluation algorithm can compute the decryption algorithm Decrypt and an augmented NAND gate [2]. From this, Gentry proves that one can create a complete set of circuits [2]. Finally, he constructs a bootstrappable scheme [2].

## 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 [3].

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 [3].

More precisely, an FSS scheme has 6 components [3]. The first is a set of p parties, denoted [] who will receive the shares [3]. The second is a subset of [] that are adversaries [3]. The third is a class of functions, ℱ, containing our eventually secret function [3]. The fourth is an output decoder algorithm, Dec, that takes the shares as inputs and decodes it as a single output [3]. This is usually a sum function [3]. The fifth is a key generation algorithm Gen that generates the “keys” or “shares” for the input function, according to a security parameter constraint [3]. The sixth and final component is an evaluation function, Eval, that evaluates each function share [3].

The FSS is also subject to a correctness and security constraint [3]. The correctness constraint rigorously states that with probability 1, the decoder algorithm operating on the shares generated from a function, , will return the same value as f [3]. 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], [3]. 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], [4].

Our Distributed-Evaluation Homomorphic Encryption scheme consists of 3 parts [4]:

1. An algorithm Gen that generates the public and secret keys, and respectively, with security parameter .
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.

They describe three different “levels'' of encoding [4]. The first is ElGamal encryptions of the inputs , expressed as , where c is the ElGamal secret key [4]. The second is additive shares in which, given a variable where [4]. The third are multiplicative shares in which, given a variable where [4]. 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 client splits its input into additive shares and sends one share to each server [4]. Each server must then perform an evaluation of its share such that given evaluations is equivalent to the computation desired by the clients [4].

# The Decision Diffie-Hellman Problem

The essence of this cryptosystem we use is a complete reliance on the Decision Diffie-Hellman Problem assumption being true. Essentially, given a cyclic group and an element , there cannot be a useful algorithm that can determine the difference between the ordered pair and , where are chosen pseudorandomly [6]. As we will show, this assumption is extremely strong and will prove to be difficult to maneuver around. As such, using an arbitrary cyclic group will not be sufficient for this protocol. This section will explore the importance and usefulness of this problem in more depth and explain why it is necessary that its assumption is true for the cryptosystem to be of any use.

## The Computational Diffie-Hellman Assumption

The first step into checking if the Diffie-Hellman protocol is valid or not is determining the truth of the computational Diffie-Hellman assumption. Indeed, given the Diffie-Hellman function , it must be computationally infeasible to compute in a reasonable number of iterations [6]. If this were possible, then an attacker could easily obtain the secret key, which is . Of course, must be complex and of large group order for this to be true; for example, if , then is trivial to find independent of and . Indeed, this is the case for all additive finite fields, where this assumption is invalid [7]. However, if is too complex, this increases the complexity of an implementation and could even affect the runtime negatively. This highlights the importance of and its structure: the simple fact that it is cyclic is not enough, but the structure of should not be too complex as to interfere with the implementation. Indeed, there is a vast selection of cyclic groups, as given any positive integer , there exists a cyclic group of order [8]. However, almost all groups in which both this assumption and the decision Diffie-Hellman assumption holds are restricted to those of prime order [6].

Although this assumption is strong, it is still not enough to justify the cryptosystem’s success. Even though it could be infeasible to compute f, it could still be possible to predict a reasonably large percentage of the bits of the output of f [6]: the computational Diffie-Hellman assumption does not guarantee exemptions from informational attacks. As such, an even stronger assumption is needed to justify the security of the algorithm [6].

## The Decision Diffie-Hellman Assumption

The justification needed in the previous section is revealed in the form of the decision Diffie-Hellman Assumption. In [6], a formal definition of this assumption is given and is presented in this paper for clarity. We define a DDH Algorithm on a group as a polynomial time algorithm such that, for all   
for some fixed α > 0 and n sufficiently large. Essentially, this means, given and , is indistinguishable from another element [6]. It is important to note that the assumption states that no algorithm exists for a fixed group . This does not mean that no algorithm exists for any . Indeed, in additive groups such as , a Cayley table is more than sufficient to construct such an algorithm. Again, the choice of is extremely important; a list of valid groups appears in [6].

The key difference between this assumption and the computational assumption is the fact that the computational assumption only guarantees no polynomial-time algorithm such that

for some fixed and sufficiently large [6]. From this definition, two things should be apparent. First, now the already-established fault of the computational assumption is rigorized: nowhere in the definition does it say cannot be well-predicted. Similarly, since in the decision assumption is arbitrary, it is much stronger than the computational assumption.

Despite there being evidence that the decision assumption is true, its proof still remains an open problem [9]. That being said, it is shown in [6] that there can only be an algorithm that satisfies the assumption if and only if the probability that predicts correctly is , where is negligible. As such, for large enough n, there can only exist an if it can be fine-tuned into an algorithm that is as close to perfect as possible. Certainly, that makes finding much harder, but a proof that even such an cannot exist remains elusive [9].

## Connection to the Discrete Logarithm

Our claim that the computational assumption for Diffie-Hellman is much weaker than the decision assumption is illustrated with comparisons to another hard problem, the discrete logarithm one. In many groups, even if the decision assumption is computationally easy, the computational assumption is just as hard as the discrete logarithm problem [7]. In [7], more specific definitions for these problems in additive groups were formed:

1. The Discrete Logarithm Problem—Given an additive group G+ and two elements , find an such that
2. CDH—Given an additive group G+ and three elements , find an element such that
3. DDH—Given an additive group G+ and four elements determine if

Of course, in the discrete logarithm problem, does not necessarily exist. However, if is finite and cyclic, then indeed, an between and exists [8]. In [7], they present families of elliptic curves for which CDH and DL are both hard, but DDH is easy. This shows the difficulty of finding a suitable group for our cryptosystem—showing CDH is hard is not nearly enough to imply DDH is hard as well.

# 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 . A similar constraint is given in [6] for security purposes.

## The Group G

The hypotheses in [1] and [6] 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 , and , where , , and are primes. We note that [4], [10] 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 , because [4] gives some examples that are computationally efficient to implement. These will also be discussed in this subsection.

### 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 III.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 III.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

,

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 [11].

### 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 III.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 [12]. 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 III.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. It should be noted that this assumes the Generalized Riemann Hypothesis to be true [9].

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 [6] shows that we have violated the primary cryptographic assumptions of HSS. Mainly, the problem with discrete factoring is known to be harder than the DDH assumption explained in the second section of the paper [4], 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 [6], 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.

### Alternative Groups Not of Order p:

One might also try to form the group denoted by U(p^n). By [8], this group is cyclic for all odd primes p and natural numbers n. However, this also violates the security constraints set in [6] as this group order is just as hard to factor as the previous cases. The reason for this is because φ(p^n) = (p−1)(pn−1) [8], and as such, the only difficult part of factoring this is finding p

### 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 [10]. Such primes are called “conversion friendly” in [4] and [10], 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 [8].

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, [4] 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 [10] 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 [13]. 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 [4], [10].

We then create the group , and refer to the generator 2 as g as needed. This is our multiplicative group [1].

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 III.3.** 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 III.3.** 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 |

## Homomorphic Encryption

In this section, we detail the definitions and benefits of homomorphic encryption. We say a scheme is homomorphic if, given and , we have:

for some instances [14]. Indeed, 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, which is proven below [14].

**Theorem III.4.** ElGamal encryption is homomorphic with representing addition in an additive field and representing component-wise multiplication.

*Proof*. Let and be encryptions of and , respectively. Then, applying the multiplication operation to and , we see this equals

Applying Proposition III.3,

Finally, we apply component-wise multiplication:  
  
which is a valid ElGamal encryption of input , as

desired.

**Remark:** Note that even though decryption is difficult and provides a potential for error, this will not be a problem for small inputs [14].

# 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 [14]. 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.

# HSS Scheme

## Subroutines

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 G of prime order and generator . The Homomorphic Secret Sharing scheme also requires a security parameter, . There are three primary functions: Additive Share, Mult Shares, and Convert Shares. These primary functions make frequent use of a subroutine called the Distributed Discrete Logarithm, or DistributedDLog and a Pseudorandom Function (PRF).

### Additive Share

The simplest of the routines, the Additive Share procedure takes an element and returns a randomly generated additive share [1]. In our context, this is used to help generate the local evaluation keys for each of the servers [1].

### MultShares

This routine takes as inputs an ElGamal-encrypted input and additive shares ,computing an output multiplicative share denoted [1]. We then compute ; this is the multiplicative share [1].

To see this is correct, note where is the ephemeral key [1]. Let , and . We then have

as shown in [1]. In later papers, the DistributedDLog function is optimized by factors of 2 or more [4]. This is a potential improvement we can make.

### Pseudorandom Function

We require a pseudorandom function (PRF) [1]. It is laid out explicitly in [15] how to construct a Pseudorandom 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 [16]. 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 Pseudorandom 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 [15], we say that our CSB generator is a function . If , we also say that and , i.e. the left and right halves of the output of G. Now let be the i-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 , and if the MSB is 0, we take the left. 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 .

### Distributed Discrete Logarithm

The purpose of the Distributed Discrete Logarithm, or DistributedDLog, is to aid the conversion 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 G, those elements whose most significant bit is one and whose following bits are 0 [4]. This will be determined later. Each call to DistributedDLog outputs the power such that for a secret share , is the nearest element in the distinguished set [1].

DistributedDLog takes the following inputs: [1]. is used to denote the maximum number of bits that can be used to encode a shared memory value [1]. serves as an upper probability bound on whether an error occurs. The maximum number of steps taken and the number of bits n examined from our pseudorandom function [1].

This procedure iteratively computes until either a maximum number of steps has been taken or the output of is [1]. It then outputs the number of steps taken

to achieve this [1].

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 0-bits.

Two other optimizations are presented in parallel with the above pattern switch [4]. 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 [4],[10]. A third major optimization of the DistributedDLog procedure is presented in [4], which will also be discussed later.

### Convert Shares

Convert Shares serves as thin veneer over DistributedDlog for the purposes of converting a multiplicative share to an additve one [1]. It first takes as input an error parameter δ and a magnitude bound M, which are in turn passed to DistributedDLog [1]. It also takes a pseudorandom function ϕ, a server index , an element of the multiplicative share , and an identifier denoting our current location in the shared program [1]. It outputs the additive share for the server b [1].

We denote the first bits of as [1]. This length is the required number of 0s in the output of DistributedDLog for the required fault tolerance, plus one extra bit for the leading 1, per [1], [4]. We also denote ’s share of by . We are now ready to describe the actual algorithm.

If , we set ; otherwise, remains the same [1]. We pass in as and as into DistributedDLog. Then we let be the output of DistributedDLog [1]. Finally, we output either or , depending on [1]. Indeed, this is the additive share we desire [1].

## Setup

Mimicking most of the examples given in [4], [1], we let the number of servers be 2 with indices . We choose a secret key of bits and let the public key be

where and the latter components are 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

where additive shares are randomly generated via the Additive Share procedure [1]. The clients provide to Eval an error parameter denoting the maximum acceptable error probability for the entire program and a magnitude bound denoting the maximum number of a bits a memory value may represent [1]. Since the program will make multiple calls to Convert Shares, passing in the client-provided to Convert Shares would cause a much higher error than expected [1]. Thus, we compute where is the number of instructions in the program and use as our error bound for Convert Shares [1].

## RMS Formalism

For these share programs executed by the servers, [1] permit only four types of instructions. This formalism is known as Restricted Multiplication Straight-Line (RMS) [1]:

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

is the number of the current instruction [1]. A ciphertext is represented as

where is the th plaintext and is a single bit of [1]. Additionally, a memory value is represented as , where is the “true” value being shared across servers [1].

The add instruction simply adds two memory values and component-wise, providing a result = [1]. This is possible since these additive shares are both additively and multiplicatively homomorphic over [4]. The multiply instruction, covered in greater detail in [1], takes as input a ciphertext and a memory value . This is why multiplication in RMS programs is “restricted:” we cannot multiply two arbitrary memory values [1]. The ElGamal ciphertext and, are passed into Mult Shares to obtain a multiplicative share . It is then converted back to an additive share via Convert Shares [1].

To compute , the procedure first repeats the above process with each ElGamal ciphertext to obtain the additive shares ,. It then computes as the result where is number of bits needed to represent the secret key c [1]. Finally, we let

The load instruction, under this variant of HSS, simply performs a multiplication instruction between the input and one: [1]. This symbolic constant is obtained from the server’s evaluation key [1].

Because the above instructions maintain the invariant that every memory value has an additive share of the shared value, it is easy for the servers to each output their shares of and have the clients reconstruct the final result via modular addition [1]. As an extra security measure, a random offset is computed using the pseudorandom function: where is the number of the current instruction. For server 1, we let , and we similarly let [1]. When these shares are added, we still have , so we let each server output its share of [1].

# Simulated RMS Programs

We simulate two RMS programs, one for tallying votes to determine whether a majority or similar threshold has been reached, and the other for determining whether there is unanimous agreement. These are simulations in the sense that any data sharing is done via subroutine calls rather than actual networking.

In order to achieve a fast development time, we implemented the HSS scheme in Python 3.6 rather than a (significantly) faster compiled language such as C++. We use no external mathematical libraries such as Numpy, instead using overloaded operators to implement our algebraic groups.

## Vote Counting

We first do a simple tally of binary votes of n clients, with 1 representing “yes” and 0 representing “no”. Each server receives n inputs , each of which corresponds to one of the n clients. We first load each input into memory:

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: mod .

## 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 [4]. A single vote is loaded into memory initially, followed by a cumulative product over all the votes:

Ironically, this program requires the same number of multiplication operations as the counting program, since load

instructions are performed via multiplication.

## Results

We simulate each of the two programs 100 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 . For purposes of speed in our relatively slow Python implementation, we set the security parameter to a low fourteen bits. Results are shown in Table III. Note that the error does not exceed the bound:

TABLE III

HSS Simulation results

|  |  |  |
| --- | --- | --- |
| Program | Accuracy | Average Runtime (s) |
| Counting | 98% | .84 |
| Unanimity | 91% | .76 |

# Other Applications of HSS

## Private Information Retrieval

The notion of secure data access has always been a challenge in many facets of computing. In particular, private information retrieval from a server allows one to access a database and obtain information while concealing which items were retrieved [17]. [4] describes an application of the HSS algorithm to split a query to servers . Again, this is similar to the voting model because each item has attributes , with two options for each attribute: 0 or 1. If the attribute vector completely matches the query, then each attribute will have a 1, and it does not match otherwise [4]. In this way, we see the same bijection which was described in the voting system with slightly different restrictions. The voter model has “classifications” of votes (as each vote is different), while a match vs. non-match only corresponds to two classifications.

As with all such schemes, the problem lies in reporting back to the user a single answer after the query is split into multiple shares [4]. [18] suggests reconstruction of shares using error correcting codes, but there is still a chance of failure. However, as is common, repetitions of this particular subroutine will cause the chance of failure to drop with a cost of runtime [4].

## Generating Bilinear Form Correlations

In this context, as was done in [4], we can extend the definition of a bilinear map M between two vector spaces to abelian groups such that . In this scenario, one party X holds a random , another party Y holds a random , and both hold additive secret shares (as each group is abelian) over (since is a homomorphism) [8]. This is a special case of generalized correlation randomness, and it can be generated using HSS in the following manner described in [4]. The following is a simplified version of this algorithm. Both parties sample a random bit-string of size m: . Then, both parties encode their inputs bitwise. Each party then runs a homomorphic evaluation algorithm that multiplies each by and generates some error . For each and , each and are set to zero. The output for party is:

and the corresponding where each is a randomly chosen, public parameter. The output for party is similar:

where again, each is randomly chosen. [4] gives examples of applications for this method of generation; namely, it can aid in generation of Beaver triples over a ring , which are three elements such that [19]. These triples can help with simplifying multiplication given similar common inputs [19]. It should be noted that this presents a special case of the above algorithm as, in this case, [4].

## Linear Algebra

While we did not simulate this program specifically, we note that is possible to use RMS to perform dot products and similar linear algebra computations. If we let the inputs be two vectors , we would first load all the components of the first vector:

.

We would then multiply each component of a with the corresponding component of b:

We finally perform a cumulative sum over all the intermediate products:

That said, this is limited in that we may only perform the dot product on vectors in a finite field.

# 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 and 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

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 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 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. 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.
3. 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.
4. 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.
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. Boneh, “the decision diffie hellman problem,” in Third Algorithmic Number Theory Symposium, Lecture Notes in Computer Science, vol.1423. Springer-Verlag, 1998, pp. 48–63.
7. A. Joux and K. Nguyen, “Separating decision diffie–hellman from computational diffie–hellman in cryptographic groups,” Journal of Cryptology, vol. 16, pp. 239–247, 2003.
8. J. Rotman, An Introduction to the Theory of Groups. Springer, 1959.
9. V. Shoup, “Searching for primitive roots in finite fields”. *Mathematics of Computation,* vol. 59, pp. 369-380, 1992.
10. E. Boyle, N. Gilboa, and Y. Ishai, “Group-based secure computation: optimizing rounds, communication, and computation,” in Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 2017, pp. 163–193.
11. D. R. L. Brown and R. P. Gallant, “The static diffie-hellman problem,” 2004.
12. D. M. Burton, *Elementary Number Theory.* McGraw-Hill College; 4th edition, 1997.
13. W. Stallings, Cryptography and Network Security: Principles and Practice, 5th ed. Prentice Hall, 2011.
14. R. Cramer, R. Gennaro, and B. Schoenmakers, “A secure and optimally efficient multi-authority election scheme,” *European Transactions on Telecommunications*, vol. 8, 10 2000.
15. S. G. Oded Golreich and S. Micali, “How to Construct Random Functions,” vol. 33, No. 4, pp. 792-807, 1986.
16. M. Mastumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,” vol. 8, No.1, pp.3-30, 1998.
17. B. C. et al., “Private information retrieval,” Proceedings of the 36thAnnual IEEE Conference on Foundations of Computer Science, pp. 41–50, 1995.
18. R. Ostrovsky and W. E. S. III., “Private searching on streaming data,” 2005, pp. 223–240.
19. P. Pullonen, “Actively secure two-party computation: Efficient beaver triple generation,” 2013.
20. T. Granlund and T. G. D. Team, “The gnu multiple precision arithmetic library.”
21. B. Applebaum, “Pseudorandom generators with long stretch and low locality from random local one-way functions,” SIAM Journal on Computing, vol. 42, 2013.
22. J.-P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein, “Blake2: simpler, smaller, fast as md5,” 2013. [Online]. Available: https://blake2.net/blake2.pdf
23. SECG, “Sec 2: Recommended elliptic curve domain parameters, version 2,” 2010. [Online]. Available: <http://www.secg.org>
24. M. R. Clarkson, S. Chong, and A. C. Myers, “Civitas: Toward a secure voting system.” USA: IEEE Computer Society, 2008.
25. V. Mavroeidis, K. Vishi, M. D. Zych, and A. Jøsang, “The impact of quantum computing on present cryptography,” 2018.

1. [↑](#footnote-ref-1)