[[1]](#footnote-1)

Significantly improving the homomorphic secret sharing scheme in the voter model

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.

# 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 [3], Gentry categorizes homomorphic encryption schemes into several classes. Two examples of these are the partially homomorphic and fully homomorphic. In a partially homomorphic cryptosystem, the user is able to perform precisely one operation on the encrypted data, such as addition or multiplication. In contrast, fully homomorphic encryption (FHE) allows the user to do arbitrary computation. In 2009 and after being an open problem since 1978, Craig Gentry proposed the first viable FHE scheme in [3].

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 for

## 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].

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

# 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 .

Let be a cyclic group of order , where and are primes. 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 [6].

## 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 I

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 I.

# Next Steps

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

,

where is used for encryption 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 the additive share indicates the index of the server and is the server's additive share of the secret key.

We first intend to do a simple tally of binary votes of clients, with representing “yes” and representing “no.” Consequently, if one client votes “no,” his additive share would be , since [2]. Likewise, if another client votes “yes,” his additive share would be or [2]. As with our earlier test of the ElGamal cryptosystem, we take advantage of the fact that multiplication in the ciphertext space is equivalent to addition in the plaintext space [7]. This time, however, each of the two servers performs a separate product.

Each server receives inputs , each of which corresponds to one of the n clients. We first load each input into memory:

In this precise variant of HSS, rather than load input directly into memory, we multiply it by a symbolic constant . This constant is shared between the servers in the same way that any other input or variable is [1].

Following this, we perform a cumulative sum over the variables:

Finally, each server outputs its personal product:

Thus, the program is exactly instructions according to the RMS formalism.

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.

1. [↑](#footnote-ref-1)