Multiparty Homomorphic Encryption

Alex Padron, Guillermo Vargas

***Abstract*—An interesting and desirable encryption property is homomorphism. A homomorphic encryption scheme is a cryptographic system that allows computation to be executed directly on encrypted data. Homomorphic computation could include a wide series of operations such as addition, multiplication, and quadratic functions. The most powerful class of such schemes is described as fully homomorphic. A fully homomorphic encryption scheme is an encryption scheme that supports arbitrary computation on encrypted data.**

Several partially homomorphic encryption schemes have been developed that support limited operations, such as addition or multiplication. Although these schemes perform relatively well in practice, they have few applications due to their restricted set of operations. Conversely, there exist fully homomorphic encryption schemes that support both addition and multiplication, but run rather slowly in practice. Consequently, homomorphic encryption has found few applications in real world systems, despite its potential to offer confidentiality in a ubiquitous technology: cloud computing.

Cloud computing has the potential to be one the most expansive applications of homomorphic encryption. Unfortunately, it requires a fully homomorphic cryptosystem that performs well in practice. In the interest of this application, we have developed a new fully homomorphic cryptosystem. Our setting strays from the traditional encryption setting in that our primary function is not messaging. In fact, our scheme does not need to support encrypted messaging at all, though it will utilize it. We propose a fully homomorphic encryption scheme under a weakened model, in which the encrypting party is also the decrypting party. The cryptosystem is built on multiparty computation. In particular, we are weakening the traditional homomorphic encryption model as follows. Plaintext messages will be encrypted using multiple keys in a secret sharing mechanism, so that only parties that know every key can decrypt a ciphertext. Our system offers computation on ciphertext by allowing parties with any key to perform some restricted operations. In order for a full operation to be executed, an analogous operation needs to be executed with each key. The encryption and operation methods are inspired by one time pads and symbolic execution. The cryptosystem is inspired by Shamir's secret sharing construction and multiparty computation.

I. Introduction

Encryption is primarily used as a means to keep data confidential and integrous while sharing it with another party. Several such encryption schemes exist, but most of them are only concerned with read and write operations. In particular, common encryption schemes such as padded RSA encryption and encryption through Diffie-Hellman key exchange are only designed to allow parties to encrypt data when trying to $write$ or send messages, and decrypt data when trying to $read$ or receive messages [14] [15]. This is sufficient for simple applications such as sending messages and storing data, but more complex applications could benefit from $modify$ operations that could be applied to encrypted data.

Such encryption schemes are called homomorphic. Specifically, homomorphic encryption schemes support direct computation on ciphertexts, without revealing any information about the underlying plaintext. Given a homomorphic encryption scheme, a client could run computational tasks using encrypted inputs on an untrusted server without sacrificing confidentiality. There is a wide range of applications for homomorphic encryption schemes such as anonymous voting, confidential medical device algorithms, and cloud computing.

Partially homomorphic schemes are encryption schemes that support operation on ciphertexts, but do not support arbitrary computation on ciphertexts. There currently exist several partially homomorphic encryption schemes. Most of these schemes only support either addition or multiplication. Such encryption schemes tend to perform well in practice. Unfortunately, their computational restrictions limit their potential applications.

Fully homomorphic encryption schemes are schemes that support arbitrary computation on ciphertexts. Fully homomorphic encryption schemes are better suited for real world applications, because they support arbitrary computation. However, the few existing fully homomorphic encryption schemes run too slowly to support any practical applications.

We propose a fully homomorphic encryption scheme built on multiparty computation. We have designed this scheme with cloud computing as the potential application in mind. In particular, the system described in the following sections is designed to support arbitrary operations over the integers in a way that places most of the computational load on the servers. Furthermore, note that our proposed encryption scheme does not support messaging between parties, but instead focuses on providing a client the ability to perform computation on untrusted servers.

II. Previous Work

Several partially homomorphic encryption schemes have been developed already. Some example of multiplicative homomorphic encryption schemes are unpadded RSA encryption and the ElGamal cryptosystem. Similarly, the Goldwasser-Micali cryptosystem, the Benaloh cryptosystem, and the Paillier cryptosystem are all examples of additive homomorphic encryption schemes. Implementations of these schemes have even performed well enough to find applications in systems today, such as anonymous voting systems [5]. There also exist some fully homomorphic encryption schemes, but they have yet to be applied to real world systems in an time-efficient manner.

We examined all of the encryption schemes listed above, and even implemented the ElGamal and Paillier cryptosystems. The following subsections discuss a few of the homomorphic encryption schemes that we studied before trying to develop our own scheme. Studying these schemes revealed

several components and attributes that are essential to any homomorphic encryption scheme. In particular, the following sections will summarize four components of each scheme: key generation, encryption, homomorphic computation, and decryption. Additionally, we will analyze how confidentiality is maintained in each scheme.

A. Unpadded RSA

Key Generation: A party begins key generation by randomly selecting two prime integers p and q.

$$p \xleftarrow{\$} \mathbb{P}$$
$$q \xleftarrow{\$} \mathbb{P}$$

The party then computes n and $\phi(n)$ as follows.

$$n = p \cdot q$$
$$\phi(n) = (p-1) \cdot (q-1)$$

Finally, the party randomly selects an positive integer e in the less than $\phi(n)$ that is coprime with $\phi(n)$, and computes its modular multiplicative inverse.

$$e \xleftarrow{\$} \mathbb{Z}_n^+ \quad | \quad gcd(e, phi(n)) = 1$$
$$d = e^{-1} \pmod{\phi(n)}$$

Ultimately, the party's public key is (n, e), and its private key is (n, d) [14].

Encryption: A message m can be encrypted under a party's RSA public key (n, e) as follows [14].

$$\varepsilon(m) = m^e \pmod{n}$$

Homomorphic Computation: The RSA algorithm's multiplicative homomorphic property is built on the power of a product property. In particular, encrypting two messages m_1 and m_2 would produce the following ciphertexts.

$$\varepsilon(m_1) = m_1^e \pmod{n}$$
$$\varepsilon(m_2) = m_2^e \pmod{n}$$

It follows that the product of these ciphertexts can be derived as follows.

$$\varepsilon(m_1) \cdot \varepsilon(m_2) \pmod{n}$$
$$m_1^e \cdot m_2^e \pmod{n}$$

By the power of a product property, it is clear that encrypting the product of m_1 and m_2 produces a ciphertext equivalent to the product of $\varepsilon(m_1)$ and $\varepsilon(m_2)$.

$$\varepsilon(m_1 \cdot m_2) = (m_1 \cdot m_2)^e \pmod{n}$$
$$\varepsilon(m_1 \cdot m_2) = m_1^e \cdot m_2^e \pmod{n}$$
$$\varepsilon(m_1 \cdot m_2) = \varepsilon(m_1) \cdot \varepsilon(m_2) \pmod{n}$$

This shows that unpadded RSA encryption satisfies the multiplicative homomorphic property.

Decryption: A party can decrypt a message encrypted under its public key (n, e) using its corresponding private key (n, d) as follows.

$$m = \varepsilon(m)^d \pmod{n}$$
$$m = m^{ed} \pmod{n}$$
$$m = m \pmod{n}$$

The correctness of the decrypted output follows from the fact that d and e are inverses in n's multiplicative group, implying that $e \cdot d = 1 \pmod{n}$ [14].

Confidentiality: Unfortunately, unpadded RSA encryption does not satisfy confidentiality to a high degree. It is vulnerable to both chosen ciphertext attacks and chosen plaintext attacks. The vulnerability to chosen plaintext attacks is obvious from the fact that the encryption scheme is deterministic. Adversaries could easily distinguish between the encryption of two known plaintexts by encrypting the plaintext themselves. Worse yet, if the message space is too small, adversaries could simply brute force search the set of messages and build a rainbow table. Unpadded RSA really only satisfies confidentiality in the case where the message space is too large to search, and plaintexts are selected in a uniform distribution to make repeats unlikely. This weaker degree of confidentiality is enforced by the difficulty of prime factorization [16], which is used to prevent an adversary from compromising a party's private key.

B. ElGamal

Key Generation: Under the ElGamal encryption scheme, a party generates a public key (G, q, g, h) and private key x by the following procedure. First, the party selects an cyclic group G of order q with generator g. The party can then randomly select its private key x.

$$x \xleftarrow{\$} \mathbb{Z}_q^+$$

Next, the party can compute the last component of its public key, h.

$$h = g^x$$

Finally, the party can publish $G, q, g, h)$ as its public key and retain x as its private key [1].

Encryption: A message m can be encrypted under an ElGamal public key as follows. First, the encrypting party randomly selects some integer y less than q, and uses it to compute the first part of the ciphertext, c_1.

$$y \xleftarrow{\$} \mathbb{Z}_q^+$$
$$c_1 = g^y$$

The encrypting party can then compute a secret to be shared with the decrypting party using the decrypting party's public key. The shared secret is used to encrypt the plaintext m and compute the second part of the ciphertext, c_2.

$$s = h^y$$
$$c_2 = m \cdot s$$

Lastly, the encrypting party's ciphertext is compiled as follows [1]

$$\varepsilon(m) = (c_1, c_2)$$

Homomorphic Computation: The multiplicative homomorphic property of the ElGamal cryptosystem is built on product of powers property. To demonstrate this, consider that encrypting two messages m_1 and m_2 would produce the following ciphertexts.

$$\varepsilon(m_1) = (g^{y_1}, m_1 \cdot h^{y_1})$$
$$\varepsilon(m_2) = (g^{y_2}, m_2 \cdot h^{y_2})$$

It follows that the product of these tuple ciphertexts can be derived as follows.

$$\varepsilon(m_1) \cdot \varepsilon(m_2)$$
$$(g^{y_1}, m_1 \cdot h^{y_1}) \cdot (g^{y_2}, m_2 \cdot h^{y_2})$$
$$(g^{y_1+y_2}, m_1 \cdot m_2 \cdot h^{y_1+y_2})$$

Note that this exactly equivalent to encrypting $m_1 \cdot m_2$ with a y_3, where $y_3 = y_1 + y_2$. This implies that $\varepsilon(m_1 \cdot m_2) = \varepsilon(m_1) \cdot \varepsilon(m_2)$, and shows that the ElGamal cryptosystem satisfies the multiplicative homomorphic property.

Decryption: A party can decrypt a ciphertext (c_1, c_2) encrypted under its public key (G, q, g, h) by using its corresponding private key as follows. The correctness of the shared secret is built on the power of a power property. In particular, the decrypting party must begin by computing the shared secret from the ciphertext as follows.

$$s = c_1^x$$
$$s = g^{xy}$$
$$s = h^y$$

As shown above, the decrypting party will always compute the same shared secret computed by the encrypting party, by the power of a power property. Next, the decrypting party can use the shared secret to recover the plaintext from the second part of the ciphertext.

$$m = c_2 \cdot s^{-1}$$
$$m = m \cdot s \cdot s^{-1}$$
$$m = m$$

Finally, because the shared secret can always be correctly computed, the correct plaintext can always be recovered from the ciphertext [1].

Confidentiality: ElGamal does manage to offer stronger confidentiality guarantees than unpadded RSA. In particular, ElGamal is secure against chosen plaintext attacks. This is achieved because of the randomized shared secret selection. By selecting a new secret key for each message, encrypting parties ensure that plaintexts do not always produce the same ciphertexts. This prevents an adversary from listening for repeated messages, or even from being able to distinguish the encryption of any two plaintexts. Note though that ElGamal is still vulnerable to chosen ciphertext attacks due to its malleability. Ultimately, ElGamal offers IND-CPA security by encrypting plaintexts with randomly selected shared secrets. The confidentiality of the shared secrets follows from the difficulty of the discrete logarithm problem [16].

C. Paillier

Key Generation: A party begins key generation by randomly selecting two large prime integers p and q, such that pq and $(p-1)(q-1)$ are coprime.

$$p \xleftarrow{\$} \mathbb{P}$$
$$q \xleftarrow{\$} \mathbb{P}$$
$$s.t. \quad gcd(pq, (p-1)(q-1)) = 1$$

Next, the party computes the values n and λ, to be used in its public and private keys as follows.

$$n = pq$$
$$\lambda = lcm(p-1, q-1)$$

Finally, the party randomly selects a random integer g in the multiplicative group of n^2, and calculates a carefully constructed modular multiplicative inverse μ.

$$g \xleftarrow{\$} \mathbb{Z}_{n^2}^*$$
$$\mu = \frac{(g^\lambda \mod n^2) - 1}{n}^{-1} \pmod n$$

The party's public key can then be published as (n, g), and its private key can be retained as (λ, μ) [4].

Encryption: A plaintext m can be encrypted under a party's Paillier public key (n, g) using a randomly selected integer r as follows [4].

$$r \xleftarrow{\$} \mathbb{Z}_n^*$$
$$\varepsilon(m) = g^m \cdot r^n \mod n^2$$

Homomorphic Computation: To demonstrate the additive homomorphic property of the Paillier cryptosystem, consider that encrypting two plaintexts m_1 and m_2 would produce the following ciphertexts.

$$\varepsilon(m_1) = g^{m_1} \cdot r_1^n \mod n^2$$
$$\varepsilon(m_2) = g^{m_2} \cdot r_2^n \mod n^2$$

It follows that the product of these ciphertexts can be derived as follows.

$$\varepsilon(m_1) \cdot \varepsilon(m_2) \pmod{n^2}$$
$$g^{m_1} \cdot r_1^n \cdot g^{m_2} \cdot r_2^n \pmod{n^2}$$
$$p^{m_1+m_2} \cdot (r_1 \cdot r_2)^n \pmod{n^2}$$

Note that this is exactly equivalent to encrypting $m_1 + m_2$ with an $r_3 = r_1 \cdot r_2$. This implies that $\varepsilon(m_1 + m_2) = \varepsilon(m_1) \cdot \varepsilon(m_2)$, and that the ElGamal cryptosystem satisfies the additive homomorphic property.

Decryption: A party can decrypt a ciphertext encrypted under its public key (n, g) by using its corresponding private key as follows.

$$m = \frac{\varepsilon(m)^\lambda \mod n^2 - 1}{n} \cdot \mu \pmod{n}$$

Correctness follows from the fact that μ was selected to be the aforementioned modular multiplicative inverse, and that λ is the least common multiple of $p-1$ and $q-1$. In particular, μ and λ cancel out all variables in the expression above except for m [4].

Confidentiality: Much like ElGamal, Paillier is secure against chosen plaintext attacks. This is achieved because of the randomized selection of g, and consequently μ. By selecting a new secret μ for each message, encrypting parties ensure that plaintexts do not always produce the same ciphertexts. This prevents an adversary from listening for repeated messages, or even from being able to distinguish the encryption of any two plaintexts. Note though that Paillier is still vulnerable to chosen ciphertext attacks due to its malleability. Ultimately, Paillier offers IND-CPA security by encrypting plaintexts with randomly selected shared secrets. The confidentiality of μ is protected by the confidentiality of λ, which follows from the difficulty of the discrete logarithm problem [16].

D. Fully Homomorphic Encryption Schemes

Several fully homomorphic encryption schemes have been developed and implemented. The first proposed fully homomorphic cryptosystem, developed by Craig Gentry, was based on ideal lattices [6]. Unfortunately, the first implementation of the scheme took about 30 seconds to execute a single bit operation [7].

Soon after, several new techniques and schemes were developed to offer more efficient fully homomorphic encryption. In particular, between 2011 and 2013, the Brakerski-Gentry-Vaikuntanathan cryptosystem [8], Brakerski's scale-invariant cryptosystem [9], the NTRU-based cryptosystem [10], and the Gentry-Sahai-Waters cryptosystem [11] all emerged in attempts to develop efficient fully homomorphic cryptosystems.

The technical details of these cryptosystems are beyond the scope of this project, but we will briefly mention that the Brakerski-Gentry-Vaikuntanathan cryptosystem, Brakerski's scale-invariant cryptosystem, and the Gentry-Sahai-Waters cryptosystem are all based on the learning with errors problem [8] [9] [11], which is conjectured to be computationally hard machine learning problem [12]. The NTRU-based cryptosystem is based on lattice-based cryptography and multiparty computation [10].

Different implementations of these new schemes have emerged since their development, but none of them have proven to be efficient enough for arbitrary cloud computing. HElib, one of the more popular fully homomorphic implementation, is an open source implementation of the Brakerski-Gentry-Vaikuntanathan cryptosystem. In our experience, even this implementation took about 2-5 seconds per operation, which is still much too slow for general cloud computing. Even still, each of the aforementioned schemes is interesting in its confidentiality guarantees and its supported homomorphic properties.

III. Threat Model

Our system has a weaker threat model than typical homomorphic encryption schemes. We use two servers to compute homomorphically, and we assume that an adversary controls at most one of them. We also assume that the client and servers have a secure method of sending data between each other. Furthermore, note that our scheme is not designed to support secure communication between parties. In our model, only one party ever sees the plaintexts. Since our main focus is on cloud computing, it is fair to expect that the encrypting party and decrypting party are always the same.

Three goals in security are to provide confidentiality, integrity, and availability. However, our system only seeks to provide confidentiality. Homomorphic encryption can never provide integrity, as the point is to allow meaningful manipulation of ciphertexts. Additionally, our scheme can not provide availability, since both servers are required to compute, and the adversary could control one of them.

We give a game based definition for providing confidentiality, modeled after IND-CCA2 [18]. A description follows.

- The adversary has access to the entire system, both servers and the client. It can compute any function any inputs it likes using the system.
- The adversary selects a function f and two inputs arrays i and j.
- The adversary chooses one of the servers to control during computation.
- The client randomly selects one of the input arrays, and computes f on it homomorphically.
- After the client's computation is completed, the adversary can again access the entire system and compute any sets of inputs on it, including i and j

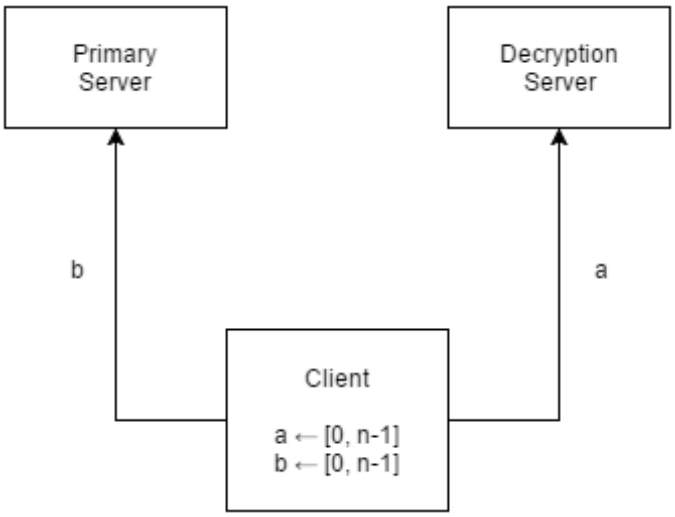

Fig. 1. Client sending keys to the primary and decryption servers.

- The adversary guesses which of i and j the client computed on homomorphically. If he guesses correctly with greater than 50% probability, he wins the game.

We choose this threat model because it provides a very strong requirement for confidentiality. After the design section, Section IV, we will prove that our system satisfies these criteria.

IV. System Design

Our system contains a single client and two servers. The client has an input vector $\vec{m} = [m_0, ..., m_{x-1}]$, and a function f, which takes in an input vector of length x, and outputs a vector of length y. The client wants to be able to compute $f(\vec{m})$ homomorphically.

Of the two servers, the first is called the primary server. This server will be responsible for actually performing homomorphic computation on input from the client. The second server is the decryption server. This server is responsible for simplifying the output from the primary server.

A. Key Generation

The client starts by picking a number n that defines the message space and key space $\mathbb{Z}_n$. The client then generates two random pad vectors, one corresponding to the input vector $\vec{m}$ and one corresponding to the homomorphic output vector $f(\vec{m})$. The pad vector $\vec{a}$ corresponding to the input vector $\vec{m}$ can be defined as follows.

$$\vec{a} = [a_0, ..., a_{x-1}]$$

Each pad a_i is randomly selected from $\mathbb{Z}_n$. Similarly, the pad vector $\vec{b}$ corresponding to the output vector $f(\vec{m})$ can be defined as follows.

$$\vec{b} = [b_0, ..., b_{y-1}]$$

Each b_i is again randomly selected from $\mathbb{Z}_n$.

After generating the two pad vectors, the client then sends $\vec{a}$ to the decryption server, and $\vec{b}$ to the primary server for future computation. Figure 1 depicts the client distributing keys.

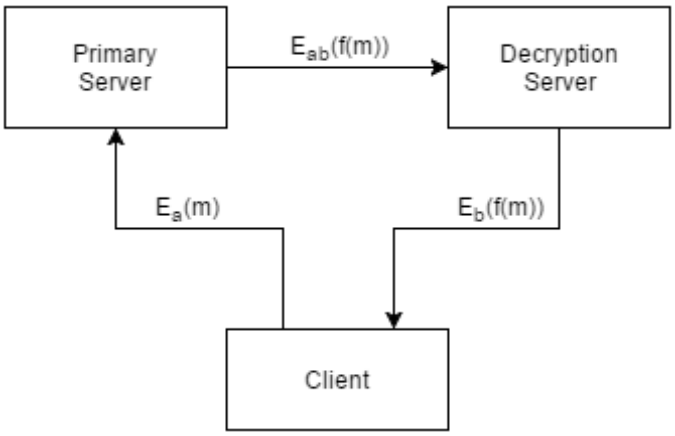

Fig. 2. Encrypted values sent between client and servers

B. Encryption

Our system uses a simple encryption scheme, based on the One Time Pad [19]. In this scheme, we encrypt all values modulo n, a public value which the client chose during the key generation phase. To encrypt a value d, we start by picking a random value $r \in \mathbb{Z}_n$. The encryption is then $E(d, r) = d + r \bmod n$. To encrypt an array in this scheme, we start by generating a random pad array that the same length as the input vector. The encrypted array is the pairwise sum of the input and random pad vectors modulo n.

To encrypt the input array, the client uses the described encryption scheme to compute $\vec{C} = E(\vec{m}, \vec{a})$. The client then sends $\vec{C}$ to the primary server. After performing its computation, the primary server will send an output array encrypted under both $\vec{a}$ and $\vec{b}$ to the decryption server. The decryption server will then use its secret key array $\vec{a}$ to simplify the output. The result is an output array encrypted under $\vec{b}$. Finally, the result is returned to the client. Note that the primary server only sees the input and output arrays encrypted under a, and the decryption server only sees the output array encrypted under b. A diagram of these interactions can be found in Figure 2.

C. Homomorphic Computation

When the primary server receives the ciphertext vector $\vec{C}$ from the client, it computes f on the ciphertext, storing the offset between the encrypted and decrypted results.

The key insight here is that we can use symbolic execution to compute f on $\vec{m}$ to generate a result in terms of $\vec{a}$ and the unencrypted output of the function. Each term on the primary server is stored as a $\langle$*value, offset*$\rangle$ pair. The value is the actual ciphertext values for that term. The offset is an algebraic expression in terms of $\vec{a}$ such that the unencrypted value of that term can be computed by subtracting the offset from the value after plugging in for $\vec{a}$.

When the primary server first receives the ciphertext vector $\vec{C}$ from the client, the terms on the server will each have an offset of a single element of $\vec{a}$. Specifically, since $\vec{C} = \vec{m} + \vec{a} \bmod n$, each term will have the form:

$$\langle C_i, \boldsymbol{a_i} \rangle$$

We can see here that subtracting the offset from each of these terms will yield the corresponding term of the input vector.

Now that the primary server has initial offsets from the client, it can perform computations on the terms, propagating the offsets such that the relationship between value, offset, and the decrypted result is preserved. Below, we will discuss how terms can be both added and multiplied while preserving offset. These two operations are sufficient for fully homomorphic encryption [20]. As such, the primary server can use these two operations to compute f on the ciphertext.

1) Adding Terms: To add terms, we simply add the offsets of the respective terms. Specifically, if we let

$$t_1 = \langle v_1, \boldsymbol{o_1} \rangle, t_2 = \langle v_2, \boldsymbol{o_2} \rangle$$

Then we can add the terms to get

$$t_1 + t_2 = \langle v_1 + v_2, \boldsymbol{o_1} + \boldsymbol{o_2} \rangle$$

Such that subtracting the offsets produces the result of the unencrypted computation, as desired.

2) Multiplying Terms: Multiplying terms is slightly more complicated, and requires multiplying the values and offsets together. Again, we define:

$$t_1 = \langle v_1, \boldsymbol{o_1} \rangle, t_2 = \langle v_2, \boldsymbol{o_2} \rangle$$

And then

$$t_1 \cdot t_2 = \langle v_1 \cdot v_2, \boldsymbol{o_2} \cdot v_1 + \boldsymbol{o_1} \cdot v_2 - \boldsymbol{o_1} \cdot \boldsymbol{o_2} \rangle$$

We derive this quantity by realizing that the underlying plaintext result will be:

$$(v_1 - \boldsymbol{o_1}) \cdot (v_2 - \boldsymbol{o_2})$$

Furthermore, the value of the resulting term will be $v_1 \cdot v_2$, so we want to find an offset such that

$$v_1 \cdot v_2 - \text{offset} = (v_1 - \boldsymbol{o_1}) \cdot (v_2 - \boldsymbol{o_2})$$

We can expand this equation to get

$$v_1 \cdot v_2 - \text{offset} = v_1 \cdot v_2 - \boldsymbol{o_1} \cdot v_2 - \boldsymbol{o_2} \cdot v_1 + \boldsymbol{o_1} \cdot \boldsymbol{o_2}$$

Which simplified yields

$$\text{offset} = \boldsymbol{o_1} \cdot v_2 + \boldsymbol{o_2} \cdot v_1 - \boldsymbol{o_1} \cdot \boldsymbol{o_2}$$

Which is the desired result. Hence, multiplying terms while propagating the offset in this manner will preserve the underlying plaintext values.

D. Decryption

After the primary server has finished computing f on the ciphertext $\vec{C}$, it will end with an output vector $\vec{u}$ where each element is a term: $\langle u_{i,v}, \boldsymbol{u_{i,o}} \rangle$. It then encrypts the values from $\vec{u}$ under $\vec{b}$ to get a new encrypted vector $E(\vec{u}, \vec{b})$ where each element in $E(\vec{u}, \vec{b})$ is of the form $\langle u_{i,v} + b_i \bmod n, \boldsymbol{u_{i,o}} \rangle$. The primary server then sends $E(\vec{u}, \vec{b})$ to the decryption server.

The decryption server uses $\vec{a}$ from the client to simplify the offsets given by the primary computation server. Each of the offsets in $E(\vec{u}, \vec{b})$ will only be in terms of $\vec{a}$, and hence the decryption server can simplify the offset to a numerical value, and subtract it from the current the current value on the term. After this is complete, $E(\vec{u}, \vec{b})$ will have all of the offset terms be 0. The decryption server will then send this simplified result to the client.

When the client gets this result, they can determine the unencrypted result by simply subtracting $\vec{b}$ from the values in $E(\vec{u}, \vec{b})$. This will be the unencrypted result of the homomorphic computation.

E. Further Improvements

There are many optimizations that can be made to our scheme to improve both security and speed.

1) Offset Removing: The size of the offsets in each term in the primary computation server can grow exponentially in the size of the program, bounded exponentially by the the number of inputs x. If x is small, the maximum size of the offsets in a term will be as well, and our system will run quickly. Note that the system can still take large inputs by increasing n, the size of each input.

It may be the case that x is too large to perform computation on the primary server. If this is the case, the primary server can use a technique called offset removing to reduce the amount of computation needed. In this technique, when the offsets of the terms on the primary server get too large, the primary server encrypts the terms under $\vec{b}$ and sends them to the decryption server. The decryption server then simplifies the offsets of the terms to a single value each. Then, the primary computation server can start computing again with offsets back to a single value. The decryption server stores the new single value offsets for either the next time this technique is used, or for decryption.

Determining when to use offset removing is implementation dependent. Each call will require a back and forth of the network, during which the primary server will not be computing.

2) Parallelized Decryption: This technique is related to offset removing. It provides faster computation by simplifying offsets as fast as possible, but increases network overhead.

Here, the primary encryption server repeatedly sends offset removing calls to the decryption server in parallel with its computation. The realization here is that the primary server does not have to wait for the decryption server to finish the offset removing calls in order to be able to continue computation. However, the primary server can only have one outstanding offset removing request at a time.

Therefore, the primary server will constantly be sending a offset removing call to the decryption server in parallel with

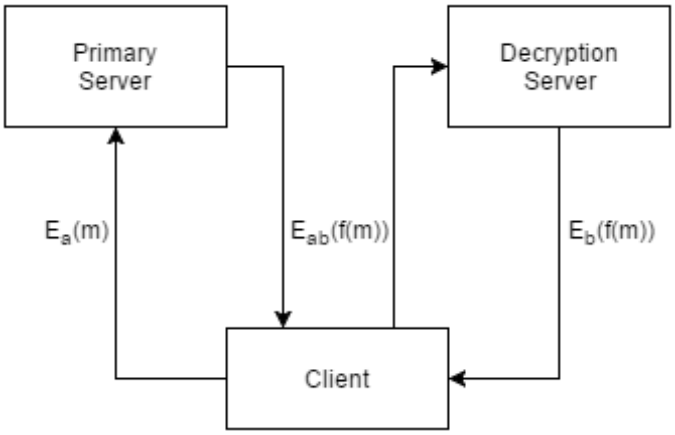

Fig. 3. Encrypted values send between client and servers with the Hiding Servers optimization

the computation it is doing. Each time it gets a response from an offset removing call, it simply sends another. This technique will keep very small offsets on the primary server, and will not hurt computation time since it is parallelized. However, it will drastically increase network overhead.

3) Hiding Servers: The current system assumes that both the primary and decryption servers know each others' identities and can talk to each other. However, it is possible to implement our system such that the servers are not aware of each other. This technique is at odds with offset removing and parallelized decryption, as those in conjunction with hiding servers would require the client to perform lots of work during the computation. Even without those optimizations, this technique requires the client to perform a small amount of extra work.

To implement this, the client has data that would be routed directly between the primary and computation server routed through itself instead. Rather than giving the primary server the ID of the decryption server, the client instead has the primary server send it the result of the computation. Then, the client directly passes the result to the decryption server to simplify the offset. This is shown in Figure 3.

This improvement makes it challenging for an adversary to control both servers. Assuming the adversary controls one of the servers, he can not then determine the other server to take control of it. This improvement will not protect against traffic analysis, where the adversary can look at the client's packets to determine which servers they are sending to. However, to protect against this attack, the client can route data through Tor or a similar system [17].

4) Additional Servers: Another way of increasing security for this system is to use many servers, rather than just two, such that the adversary would still need to control each server to reveal the client data. This would drastically increase security by increasing the number of servers that need to be controlled by an adversary in order to access client data.

We have not yet come up with a scheme to implement this improvement. It is one of the areas in which we would like to improve our system.

5) Additional Function Methods: In Section IV-C we saw how add and multiply are implemented in our system. This is sufficient to provide fully homomorphic encryption, but provides an inconvenient interface for programmers. It is possible to easily also support adding and multiplying by constants. Here, the new value of the term is just the updated value from the constant, and the offset remains the same. It is also possible to use control statements with constant inputs, such as a for loop in the range $(0, 100)$. This can drastically reduce the size of programs, but providing convenience for the programmer and reducing the amount of data sent from the client to the primary server.

V. Proof of Security

To prove security of our system, we want to show that for any server the adversary can control, the inputs to that server can correspond to any plaintext with equal probability. Therefore, the adversary's initial probability distribution over possible inputs $\vec{m}$ will be the same as his probability distribution after seeing the computation.

We break the proof down into two sections, one where the adversary controls the primary server, and one where the adversary controls the decryption server.

One of our assumptions stated above is that the client and servers are using a secure and authenticated encryption scheme to communicate, such that the adversary can learn nothing from watching the network. As such, the only data that the adversary sees will be the data that is directly sent to the server it controls.

For this proof, it doesn't matter that the adversary can control the whole system in the sections of the game before and after the stage where the client computes on the randomly selected input. If we can show that all the input that the adversary sees during this computation can correspond to any plaintext with equal probability, then he will not be able to distinguish the plaintext that was selected.

A. Adversary Controls Primary Server

In this case, the input the adversary receives is the ciphertext $\vec{C} = E(\vec{m}, \vec{a})$ from the client. They also receive the vector $\vec{b}$ from the client. We want to show that any plaintext $\vec{m}$ can be underlying these values with equal probability. From there,

First, we note that $\vec{b}$ has no relation to m, and therefore gives no information about $\vec{m}$.

We want to show that the adversary gains no knowledge by seeing $\vec{C} = E(\vec{m}, \vec{a})$. We can do this as follows. First, we show that any ciphertext $\vec{C}$ is equally likely. For each element in $\vec{C}$:

$$Pr[C = c] = \sum_{m' \in \mathbb{Z}_n} Pr[C = c | M = m'] \cdot Pr[M = m']$$

$$Pr[C = c] = \sum_{m' \in \mathbb{Z}_n} Pr[a = c - m'] \cdot Pr[M = m']$$

$$Pr[C = c] = \sum_{m' \in \mathbb{Z}_n} \frac{1}{n} \cdot Pr[M = m']$$

$$Pr[C = c] = \frac{1}{n}$$

Then, we can use Bayes' Rule to show that the adversary gains no knowledge about the message from seeing a ciphertext.

$$Pr[M = m|C = c] = \frac{Pr[C = c|M = m] \cdot Pr[M = m]}{Pr[C = c]}$$
$$Pr[M = m|C = c] = \frac{Pr[a = c - m] \cdot Pr[M = m]}{\frac{1}{n}}$$
$$Pr[M = m|C = c] = \frac{\frac{1}{n} \cdot Pr[M = m]}{\frac{1}{n}}$$
$$Pr[M = m|C = c] = Pr[M = m]$$

Since the adversary gains no knowledge about the underlying message from the information he receives, he will not be able to guess which message the client is computing on over 50% of the time. Therefore, he will not be able to win the game.

B. Adversary Controls Decryption Server

In this case, the adversary receives the encrypted result $\vec{E} = E(\vec{u}, \vec{b})$ from the primary server. It also receives the pad vector $\vec{a}$ from the client. We note here that knowing the values of $\vec{u}$ and $\vec{m}$ are equivalent for the adversary. Therefore, we will show that the adversary can not determine $\vec{u}$, and therefore can not determine $\vec{m}$.

First, we note that $\vec{a}$ is independent of $\vec{u}$, and therefore gives the adversary no information about $\vec{u}$.

We want to show that the adversary gains no knowledge about $\vec{u}$ by observing a given set of values from $\vec{E}$. We start by showing that each value of $\vec{E}$ is equally likely given a value for $\vec{u}$. For each element in $\vec{E}$:

$$Pr[E = e] = \sum_{u' \in \mathbb{Z}_n} Pr[E = e|U = u'] \cdot Pr[U = u']$$
$$Pr[E = e] = \sum_{u' \in \mathbb{Z}_n} Pr[b = c - m'] \cdot Pr[U = u']$$
$$Pr[E = e] = \sum_{u' \in \mathbb{Z}_n} \frac{1}{n} \cdot Pr[U = u']$$
$$Pr[E = e] = \frac{1}{n}$$

Now, we can use this to show that the adversary gains no information about $\vec{u}$ from seeing $\vec{E}$. For each element in $\vec{u}$:

$$Pr[U = u|E = e] = \frac{Pr[E = e|U = u] \cdot Pr[U = u]}{Pr[E = e]}$$
$$Pr[U = u|E = e] = \frac{Pr[b = e - u] \cdot Pr[U = u]}{\frac{1}{n}}$$
$$Pr[U = u|E = e] = \frac{\frac{1}{n} \cdot Pr[U = u]}{\frac{1}{n}}$$
$$Pr[U = u|E = e] = Pr[U = u]$$

Therefore, the adversary will gain no information about $\vec{u}$. Therefore, since $\vec{E}$ is the only piece of information the adversary has that relates to $\vec{m}$, the adversary will not be able to determine any information about $\vec{m}$, and will lose the game.

VI. Conclusion

Homomorphic encryption has many potential applications, most notably cloud computing. Unfortunately, no fully homomorphic encryption schemes have performed well enough in practice to be applied in this field. Our scheme relaxes the security model of homomorphic encryption, allowing for two computing servers, and assuming that an adversary controls at most one of them. Furthermore, our proposed encryption scheme is not concerned with securing communications across parties. A single client is both the encrypting party and the decrypting party. These relaxed assumptions allow us to build a fully homomorphic encryption scheme that uses far simpler encryption than previous schemes.

While our system provides many benefits over previous fully homomorphic schemes, it does have drawbacks. First, the current design only uses two servers, which is less secure than if many servers were used. While we suspect that our scheme can be extended to include many servers, we have not yet found such an extension. Additionally, symbolic execution on the offsets is the main bottleneck for our system. Assuming a large number of inputs, the size of the symbolic execution the servers perform can scale exponentially with the depth of the circuit. While we have many optimizations to reduce this workload, they come at the cost of anonymity between the servers. We hope in the future to devise a scheme that does not require this trade off.

We have designed our system to satisfy confidentiality, while allowing a client to dispatch arbitrary computation to two servers. By focusing solely on the cloud computing application, we believe we have developed a simpler alternative to current fully homomorphic cryptosystems. Furthermore, we believe that our scheme could perform reasonably well if it were implemented using highly parallelized symbolic execution.

References

[1] ElGamal, T. (1984, August). A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in cryptology* (pp. 10-18). Springer Berlin Heidelberg.

[2] Goldwasser, S., & Micali, S. (1982, May). Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing* (pp. 365-377). ACM.

[3] Benaloh, J. (1994, May). Dense probabilistic encryption. In *Proceedings of the workshop on selected areas of cryptography* (pp. 120-128).

[4] Paillier, P. (1999, May). Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptologyEUROCRYPT99* (pp. 223-238). Springer Berlin Heidelberg.

[5] Kifayat, K., Merabti, R., Shi, Q., & Llewellyn-Jones, D. (2007, August). Applying secure data aggregation techniques for a structure and density independent group based key management protocol. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on* (pp. 44-49). IEEE.

[6] Gentry, C. (2009, May). Fully homomorphic encryption using ideal lattices. In *STOC* (Vol. 9, pp. 169-178).

[7] Gentry, C., & Halevi, S. (2011). Implementing Gentrys fully-homomorphic encryption scheme. In *Advances in CryptologyEUROCRYPT 2011* (pp. 129-148). Springer Berlin Heidelberg.

[8] Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2012, January). (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (pp. 309-325). ACM.

[9] Brakerski, Z. (2012). Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in CryptologyCRYPTO 2012* (pp. 868-886). Springer Berlin Heidelberg.

[10] Lpez-Alt, A., Tromer, E., & Vaikuntanathan, V. (2012, May). On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing* (pp. 1219-1234). ACM.

[11] Gentry, C., Sahai, A., & Waters, B. (2013). Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in CryptologyCRYPTO 2013* (pp. 75-92). Springer Berlin Heidelberg.

[12] Leyton-Brown, K., Nudelman, E., & Shoham, Y. (2002, September). Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Principles and Practice of Constraint Programming-CP 2002* (pp. 556-572). Springer Berlin Heidelberg.

[13] Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(11), 612-613.

[14] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM, 21(2)*, 120-126.

[15] Diffie, W., & Hellman, M. E. (1976). New directions in cryptography. *Information Theory, IEEE Transactions on, 22(6)*, 644-654.

[16] Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review, 41*(2), 303-332.

[17] Dingledine, R., Mathewson, N., & Syverson, P. (2004). Tor: the second generation onion router. *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*

[18] C. Rackoff and D. Simon. Non-interactive zero-knowledge proof of knowledge and the chosen ciphertext attack. *Proceedings of Crypto+ 91*, pp. 433444.

[19] Miller, Frank (1882). *Telegraphic code to insure privacy and secrecy in the transmission of telegrams*. C.M. Cornwell.

[20] Wu, D. J. (2015). Fully homomorphic encryption: Cryptography's holy grail. *XRDS: Crossroads, The ACM Magazine for Students, 21*(3), 24-29.