Efficient Integer Vector Homomorphic Encryption

Angel Yu, Wai Lok Lai, James Payor

Massachusetts Institute of Technology
77 Massachusetts Avenue, Cambridge, MA 02139, USA
{angelyu, wllai, payor}@mit.edu

May 14, 2015

Abstract

Homomorphic Encryption yields deliberately malleable ciphertext, allowing operations on encrypted data. We outline and implement a homomorphic encryption scheme as suggested by Zhou and Wornell [1] that encrypts integer vectors to allow computation of arbitrary polynomials in the encrypted domain with a bounded degree. This finds applications particularly in cloud computation, when one is interested in learning low dimensional representations of the stored encrypted data.

With our optimizations to the scheme, we find that it is orders of magnitude faster than HELib, an implementation of fully homomorphic encryption, but in turn a few orders of magnitude slower than operations on unencrypted data. This makes the scheme nearly viable for practical use, which we give basic implementations of such applications as search and classification on encrypted data.

1 Introduction

With more and more data available on the cloud, big data analysis becomes ever more promising. However, for many applications, storing sensitive data in third-party sites raises privacy concerns. While clients can store encrypted data remotely and pull a selection of the data when needed, network constrains often make it infeasible for the clients to pull the entire data set from the cloud and perform the computation on their own. Therefore, to fully harness the power of the cloud, the cloud must do more than merely storing the data.

1.1 Motivation

While *non-malleability* is an important aspect for encryption for network security –that no one should be able to feasibly modify ciphertexts to encrypt different but related data– many situations arise in which performing computations on data in encrypted form is valuable, as in the case of cloud computing mentioned above. In particular, for *multiparty computation* in which several parties cooperate to compute a function of their data, it becomes useful to be able to maintain the secrecy of both the data involved and the function being computed.

For instance, suppose we have stored our encrypted data on a third-party cloud storage site ("the cloud"), and we want to compute a low-dimensional function of our data, such as search queries or obtaining features of items. To avoid streaming the entire data set from the cloud, we need the ciphertext to be malleable, so that the cloud can compute our lower-dimensional function and send us only the relevant information. Homomorphic Encryption solves this problem elegantly.

Homomorphic Encryption schemes give methods for encrypting plaintext so that operations on the ciphertext translate into operations on the plaintext. That is, given ciphertext c for plaintext x and a function $f(\cdot)$, we can find a new ciphertext c' for some related plaintext $f(x)$ without decrypting x. We shall discuss this notion further in Section 2.

1.2 Our Contributions

In this paper, we examine the Homomorphic Encryption scheme given by Zhou and Wornell in [1], which we have implemented and on top of which we built several real life applications, as we shall discuss. This scheme allows the encryption of *integer vectors* whilst supporting vector addition, linear transformation, and weighted inner products in the encrypted domain. The scheme provides a mechanism for switching secret keys without revealing the old or new secret key, and uses this observation to implement the supported operations. The infeasibility of recovering secret keys is based on the conjectured hardness of the Learning with Errors (LWE) problem, as we shall discuss in Section 3.3.

In this scheme, the party describing the function to compute on the data needs to know its secret key, while the party performing computations on the ciphertext only learns the types of operations in use: the coefficients of the particular linear transformation or inner product weights are kept hidden. As such, the scheme is suited to applications that involve a cloud server with a huge amount of data, and a client who wants to make hidden queries on the data. The cloud will not learn the nature of the data or the particular query, and network costs are minimized for repeated computation of the same function: the client can send one description of the computation for the cloud to apply to all required data items, and the cloud need only send the encrypted results back, which are typically low-dimensional.

In the following sections, we aim to (1) give a brief overview of current results in Homomorphic Encryption, (2) an in-depth understanding of the scheme in [1], and (3) an analysis of the scheme in practice. We begin with an overview of fully and partially homomorphic encryption schemes in Section 2. Next, in Section 3, we describe Zhou and Wornell's approach and the methods for performing operations in the encrypted domain. Section 4 then gives applications of the scheme. Finally, Section 5 describes our implementation of the scheme and its performance, including optimizations for performing linear transformations and our implementation of search and classification on encrypted data.

2 Homomorphic Encryption

The idea of *Homomorphic Encryption* was first introduced by Rivest et al. [4] in 1978. In the paper, they presented an application of Homomorphic Encryption and posed the problem of creating a secure Homomorphic Encryption scheme that supports a large set of operations. A Homomorphic Encryption scheme is defined to be Fully Homomorphic if it supports computation of arbitrary functions in the encrypted domain, while a Partially Homomorphic Encryption scheme only supports a limited subset of functions that can be computed in the encrypted domain. Because of the complexity of Fully Homomorphic Encryption schemes, Partially Homomorphic Encryption schemes are usually orders of magnitude faster than Fully Homomorphic Encryption schemes. We shall now introduce examples of both types of schemes.

2.1 Partially Homomorphic Encryption

There are many Partially Homomorphic Encryption schemes adapted from popular Public-key cryptography schemes. For example, Unpadded RSA [7] and ElGamal [8] are great examples of Partially Homomorphic Encryption schemes. We present the adaptation of RSA and ElGamal below.

2.1.1 Unpadded RSA

We show that the Unpadded RSA encryption scheme [7] can be used a Partially Homomorphic Encryption scheme supporting multiplication in the encrypted domain. Given an RSA public key (n, e) and two plaintexts x_1 and x_2 we observe,

$$\begin{aligned} \text{Enc}(x_1) \cdot \text{Enc}(x_2) &\equiv x_1^e \cdot x_2^e && \mod n \\ &\equiv (x_1 \cdot x_2)^e && \mod n \\ &\equiv \text{Enc}(x_1 \cdot x_2) && \mod n \end{aligned}$$

Hence, to compute multiplication in the encrypted domain for RSA, we just need to multiply the ciphertexts to obtain our new ciphertext.

2.1.2 ElGamal

We now show that the ElGamal encryption scheme [8] can also be used a Partially Homomorphic Encryption scheme supporting multiplication in the encrypted domain. Assume we have an ElGamal public key (G, q, g, h) and secret key s where G is a cyclic group with order q, generator g and $h = g^s$. For two plaintexts $x_1, x_2 \in G$ encrypted with random $r_1, r_2 \in \{0, 1, \ldots, q-1\}$ we observe,

$$\begin{aligned} \text{Enc}(x_1) \cdot \text{Enc}(x_2) &= (g^{r_1}, x_1 \cdot h^{r_1}) \cdot (g^{r_2}, x_2 \cdot h^{r_2}) \\ &= (g^{r_1+r_2}, x_1 x_2 \cdot h^{r_1+r_2}) \\ &= \text{Enc}(x_1 \cdot x_2) \end{aligned}$$

Hence, to compute multiplication in the encrypted domain for ElGamal, we again just need to multiply the ciphertexts to obtain our new ciphertext.

2.2 Fully Homomorphic Encryption

It was not until 2009 that a Fully Homomorphic Encryption scheme was first developed by Gentry [5]. The secrecy behind most existing Fully Homomorphic Encryption schemes is based on the Learning with Errors (LWE) problem, described in detail in Section 3.3. The scheme described in Gentry [5] supports addition and multiplication. This allows the scheme to support arbitrary functions by converting the function into a circuit of additions and multiplications. To ensure that error does not increase too much such that it affects the correctness of decryption, the scheme will periodically perform a bootstrap operation to reset the error. This is done by homomorphically applying the decrypt algorithm on the ciphertext. Hence, this procedure enables the scheme to handle arbitrarily many additions and multiplications in the encrypted domain while still maintaining correctness.

Figure 1 illustrates most Homomorphic Encryption schemes, including Gentry's scheme. Since the function we are interested in computing needs to be converted into a circuit, it is necessary that the function be available to the cloud in unencrypted form.

Schemes based on LWE that do not support bootstrapping are called "Somewhat Homomorphic Encryption schemes," as they can only perform a limited number of operations before the error becomes too large. The scheme examined in this paper is a Somewhat Homomorphic Encryption scheme.

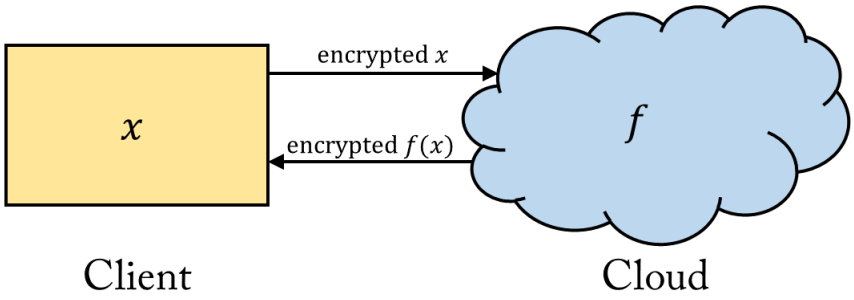

Figure 1: Most Homomorphic Encryption schemes: The cloud has access to the function f, and the client sends encrypted x to the cloud for computation.

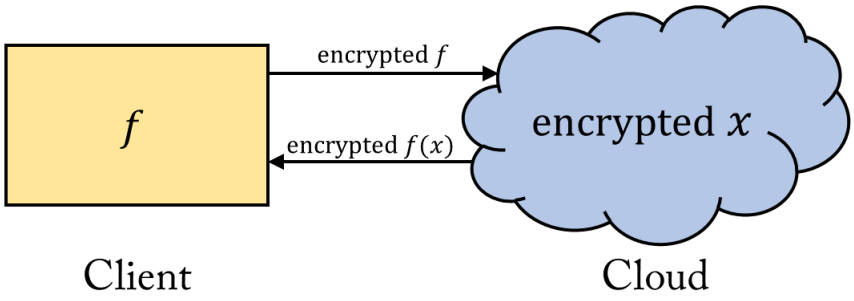

Figure 2: The scheme proposed by Zhou [1]. The cloud computes $f(x)$ without knowing either x or $f(\cdot)$.

3 Homomorphic Encryption for Integer Vectors

As introduced in Section 1.2, the scheme proposed by Zhou and Wornell [1] allows for computation on encrypted data while maintaining the secrecy of the function, approaching Homomorphic Encryption differently from schemes described with Figure 1. This new scheme is illustrated in Figure 2, where the client encrypts both x and $f(\cdot)$, and the cloud computes $f(x)$ without knowing either x or $f(\cdot)$.

We now present the scheme in full detail, describing the set of operations it supports as well as proving its correctness. We first introduce some useful notations:

Definition 3.1. For $a \in \mathbb{R}$, define $\lceil a \rfloor$ to round a to the nearest integer.

Definition 3.2. For vector $\mathbf{a} \in \mathbb{R}^n$, define $\lceil \mathbf{a} \rfloor$ to round each entry a_i of $\mathbf{a}$ to the nearest integer.

Definition 3.3. For vector $\mathbf{a} = [a_1, \ldots, a_n]^T \in \mathbb{R}^n$, define $|\mathbf{a}| := \max_i\{|a_i|\}$.

Definition 3.4. For matrix $A \in \mathbb{R}^{n \times m}$, define $|A| := \max_i\{|A_{ij}|\}$.

Definition 3.5. For matrix $A \in \mathbb{R}^{n \times m}$, define $\text{vec}(A) := [\mathbf{a}_1^T, \ldots, \mathbf{a}_m^T]^T$ where $\mathbf{a}_i$ is the i^{th} column of A.

Now we are ready to present the scheme. For plaintext vector $\mathbf{x} \in \mathbb{Z}^m$ and a secret key $S \in \mathbb{Z}^{m \times n}$, a ciphertext $\mathbf{c} \in \mathbb{Z}^n$ is a vector that satisfies

$$S\mathbf{c} = w\mathbf{x} + \mathbf{e} \qquad \text{(m, 1) = (m, n) x (n, 1) = w x (m, 1) + (m, 1) = (m, 1)} \tag{3.1}$$

where w is a large integer and $\mathbf{e}$ is an error term with elements smaller than $\frac{w}{2}$. We further assume that $|S| \ll w$. This will be important in keeping the error term small later when applying operations in the encrypted domain.

With knowledge of S, decryption is straightforward:

$$\mathbf{x} = \left\lceil \frac{S\mathbf{c}}{w} \right\rfloor \tag{3.2}$$

An important concept used to implement the operations in the encrypted domain is *key switching*. Given S and $\mathbf{c}$, we want to change to new chosen secret key S' and a corresponding ciphertext $\mathbf{c}'$. Given a procedure to do this, encryption becomes straightforward. We take wI to be a "secret key" and $\mathbf{x}$ to be the "ciphertext", as $(wI)\mathbf{x} = w\mathbf{x}$ is of the required form with a zero error term. Then we can perform a key switch from wI to some chosen S, and obtain our ciphertext.

We now describe how to perform key switching, and then how to perform the fundamental operations in the encrypted domain: addition, linear transformation, and weighted inner products.

3.1 Key Switching

Key switching allows changing a secret-key–ciphertext pair to another one with a chosen secret key while still encrypting the original plaintext. As we shall see, key switching simplifies the implementation and analysis of many of our operations.

Suppose the plaintext $\mathbf{x} \in \mathbb{Z}^m$ is currently encrypted as $\mathbf{c} \in \mathbb{Z}^n$ with secret key $S \in \mathbb{Z}^{m \times n}$, and we want a new ciphertext $\mathbf{c}' \in \mathbb{Z}^{n'}$ with a new secret key $S' \in \mathbb{Z}^{m \times n'}$ such that

(m, 1) = (m, n') x (n', 1) = (m, n) x (n, 1) = (m, 1)

$$S'\mathbf{c}' = S\mathbf{c} \tag{3.3}$$

We shall divide this into two step: converting $\mathbf{c}$ and S into an intermediate bit representation $\mathbf{c}^*$ and S^*, then switching the bit representation into the desired secret key. We note that by using the bit representation, we keep the magnitude $|\mathbf{c}^*| := \max\{|c_i|\}$ to be 1. This will prevent the error term $\mathbf{e}'$ from increasing too much, which preserves the correctness of the decryption.

Step 1. We choose ℓ such that $|\mathbf{c}| < 2^\ell$. Then, we can write each c_i in their bit representation $\mathbf{b}_i = [b_{i(\ell-1)}, \ldots, b_{i1}, b_{i0}]^T$ such that $b_{ik} \in \{-1, 0, 1\}$, and let

(nl, 1)

$$\mathbf{c}^* = [\mathbf{b}_1^T, \ldots, \mathbf{b}_n^T]^T \tag{3.4}$$

For example, the vector $\mathbf{c} = [1, -3]$ will be written as $\mathbf{c}^* = [0, 0, 1, 0, -1, -1]$ if $\ell = 3$.

Next, we construct the matrix S^* by changing each S_{ij} in the original secret key into the vector

$$B_{ij} = [2^{\ell-1}S_{ij}, \ldots, 2S_{ij}, S_{ij}] \tag{3.5}$$

For example, a secret key

(m, n)

$$S = \begin{bmatrix} 1 & 2 \\ 5 & 4 \end{bmatrix}$$

will give

(m, nl)

$$S^* = \begin{bmatrix} 4 & 2 & 1 & 8 & 4 & 2 \\ 20 & 10 & 5 & 16 & 8 & 4 \end{bmatrix}$$

such that we preserve

$$S^*\mathbf{c}^* = S\mathbf{c}$$

(m, nl) x (nl, 1) = (m, 1)

Step 2. With the bit vector representation, we are now ready to convert this into the new secret-key–ciphertext pair. We now construct the "key-switch matrix" $M \in \mathbb{Z}^{n' \times n\ell}$ which satisfies

$$S'M = S^* + E \tag{3.6}$$

(m, n') x (n', nl) = (m, nl) + (m, nl)

T: (m, (n'-m))

For ease of implementation, we limit our discussion to new secret keys in the form $S' = [I, T]$, the identity matrix concatenated with some matrix T. Hence, we take M such that

(n' , nl) (m, nl) ((n'-m), nl)

$$M = \begin{bmatrix} S^* - TA + E \\ A \end{bmatrix} \tag{3.7}$$

where $A \in \mathbb{Z}^{(n'-m)\times n\ell}$ is a random matrix and $E \in \mathbb{Z}^{m\times n\ell}$ is a random noise matrix. We then define

$$\mathbf{c}' = M\mathbf{c}^* \tag{3.8}$$

and we can see that

$$S'\mathbf{c}' = S^*\mathbf{c}^* + E\mathbf{c}^*$$

where $E\mathbf{c}^* = \mathbf{e}'$ is the new error with $|\mathbf{e}'|$ kept small, because $|c^*| = 1$ by construction and E can be randomly generated with $|E|$ small.

3.2 Operations

In addition to the three fundamental operations described above, we also supports three operations over the encrypted domain, namely addition, linear transformation, and weighted inner product.

3.2.1 Addition

For plaintexts $\mathbf{x}_1, \mathbf{x}_2$ and their corresponding ciphertexts $\mathbf{c}_1, \mathbf{c}_2$ that are encrypted with the same secret key S, we observe that

$$S(\mathbf{c}_1 + \mathbf{c}_2) = w(\mathbf{x}_1 + \mathbf{x}_2) + (\mathbf{e}_1 + \mathbf{e}_2)$$

Hence to add to plaintexts $\mathbf{x}_1, \mathbf{x}_2$ in the encrypted domain, we add the original ciphertexts $\mathbf{c}_1$ and $\mathbf{c}_2$ given that they are encrypted with the same secret key. Therefore, the new ciphertext is

$$\mathbf{c}' = \mathbf{c}_1 + \mathbf{c}_2 \tag{3.9}$$

and the secret key remains the same. It is clear that the overhead for the addition operation is proportional to the difference between the length of the ciphertext and the length of the plaintext. In our case, since the secret key is going to be the identity concatenated with a thin column matrix T, the ciphertexts should only be 1 bit longer than the plaintexts. This results in negligible overhead.

3.2.2 Linear Transformation

For plaintext $\mathbf{x}$ and its corresponding encryption $\mathbf{c}$ by secret key S, we can compute a linear transformation $G\mathbf{x}$ by an arbitrary $G \in \mathbb{Z}^{m'\times n}$ by observing that

$$(GS)\mathbf{c} = wG\mathbf{x} + G\mathbf{e}$$

S c = w x + e

Hence, we can treat $\mathbf{c}$ as an encryption of $G\mathbf{x}$ by secret key GS. Then, the client can compute the key-switch matrix $M \in \mathbb{Z}^{(m'+1)\times(m'\ell)}$ to switch the key from GS to an $S' \in \mathbb{Z}^{m'\times(m'+1)}$ of their choice, and send M to the cloud. The cloud computes

$$\mathbf{c}' = M\mathbf{c} \tag{3.10}$$

This key-switch procedure reduces the dimension of the ciphertext to $m'+1$, reflecting the dimension of $G\mathbf{x}$.

The overhead in this calculating the linear transformation comes from the key switch operation, which has an overhead of ℓ times the original cost, where ℓ is the number of bits used in the bit vectorization in key switching.

3.2.3 Weighted Inner Products

To perform weighted inner products, we need the following lemma that allows us to consider the operation as the a matrix-vector product. Let $\text{vec}(M)$ be a vectorized form of a matrix M, as defined in Definition 3.5, where we concatenate the columns of M into a vector with the same number of entries. Then we have the following:

Lemma 3.6. *For any vectors $\mathbf{x}, \mathbf{y}$ and matrix M of matching dimensions,*

$$\mathbf{x}^T M \mathbf{y} = \text{vec}(M)^T \text{vec}(\mathbf{x}\mathbf{y}^T)$$

Proof. We expand out the right hand side to obtain

$$\begin{aligned}
\text{vec}(M)^T \text{vec}(\mathbf{x}\mathbf{y}^T) &= \sum_i \sum_j M_{ij} \mathbf{x}_i \mathbf{y}_j \\
&= \sum_j \left(\sum_i \mathbf{x}_i M_i \right)_j \cdot \mathbf{y}_j \\
&= \sum_j (\mathbf{x}^T M)_j \mathbf{y}_j \\
&= \mathbf{x}^T M \mathbf{y}
\end{aligned}$$

□

Now, suppose we have some $\mathbf{x}_1$ and $\mathbf{x}_2$ encrypted as $\mathbf{c}_1$ and $\mathbf{c}_2$ with secret keys S_1 and S_2. Specifically, $S_1 \mathbf{c}_1 = w\mathbf{x}_1 + \mathbf{e}_1$ and $S_2 \mathbf{c}_2 = w\mathbf{x}_2 + \mathbf{e}_2$. We can compute a weighted inner product $\mathbf{x}_1^T H \mathbf{x}_2$ in the encrypted domain by the following proposition:

Proposition 3.7. *Secret key $S = \text{vec}(S_1^T H S_2)^T$ and ciphertext $\mathbf{c} = \left\lceil \frac{\text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)}{w} \right\rfloor$ correspond to plaintext $\mathbf{x}_1 H \mathbf{x}_2$. That is, for some new error term $\mathbf{e}$:*

$$\text{vec}(S_1^T H S_2)^T \left\lceil \frac{\text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)}{w} \right\rfloor = w\mathbf{x}_1 H \mathbf{x}_2 + \mathbf{e}$$

Proof. First, we will ignore the division by w and directly evaluate the product above. Using Lemma 3.6,

$$\begin{aligned}
\text{vec}(S_1^T H S_2)^T \text{vec}(\mathbf{c}_1 \mathbf{c}_2^T) &= \mathbf{c}_1^T S_1^T H S_2 \mathbf{c}_2 \\
&= (S_1 \mathbf{c}_1)^T H (S_2 \mathbf{c}_2) \\
&= (w\mathbf{x}_1 + \mathbf{e}_1)^T H (w\mathbf{x}_2 + \mathbf{e}_2) \\
&= w^2 \mathbf{x}_1^T H \mathbf{x}_2 + w(\mathbf{x}_1^T H \mathbf{e}_2 + \mathbf{e}_1^T H \mathbf{x}_2) + \mathbf{e}_1^T H \mathbf{e}_2
\end{aligned}$$

By assumption, we can choose w large such that the sum of the entries of H is much less than w, so $\mathbf{e}_1^T H \mathbf{e}_2 \leq |\mathbf{e}_1||\mathbf{e}_2| \sum H_{ij} \ll w$. Thus, dividing the whole expression by w and rounding gives us

$$\begin{aligned}
\left\lceil \frac{\text{vec}(S_1^T H S_2)^T \text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)}{w} \right\rfloor &= w\mathbf{x}_1^T H \mathbf{x}_2 + \mathbf{x}_1^T H \mathbf{e}_2 + \mathbf{e}_1^T H \mathbf{x}_2 \\
&= w\mathbf{x}_1^T H \mathbf{x}_2 + \mathbf{e}' \qquad (3.11)
\end{aligned}$$

where $\mathbf{e}'$ is an error term with $|\mathbf{e}'| \ll w$ because $|\mathbf{x}_1|, |\mathbf{x}_2| \ll w$.

We now consider the left hand side of (3.11). Let $a_1, a_2, \cdots, a_k \in \mathbb{Z}$ and $b_1, b_2, \cdots, b_k \in \mathbb{Z}$ be such that $[a_1, a_2, \cdots, a_k]^T = \text{vec}(S_1^T H S_2)$ and $[b_1, b_2, \cdots, b_k]^T = \text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)$. Then,

$$\begin{aligned}\left\lceil \frac{\text{vec}(S_1^T H S_2)^T \text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)}{w} \right\rfloor &= \left\lceil \frac{\sum_i a_i b_i}{w} \right\rfloor \\ &= \sum_{i=1}^{k} \left\lceil \frac{a_i b_i}{w} \right\rfloor + \mathbf{e}'' \end{aligned} \tag{3.12}$$

where $\mathbf{e}''$ is an error term with $|\mathbf{e}''| \leq k \ll w$. Since the entries in S_1, S_2, and H are all $\ll w$, we can deduce that $a_i \ll w$ for all $i \in \{1, 2, \ldots, k\}$.

Let $q_1, q_2, \ldots, q_k \in \mathbb{Z}$ and $r_1, r_2, \ldots, r_k \in \mathbb{Z}$ be such that $b_i = q_i w + r_i$ with $0 \leq r_i < w$ for all $i \in \{1, 2, \ldots, k\}$. Hence, we have

$$\begin{aligned}\sum_{i=1}^{k} \left\lceil \frac{a_i b_i}{w} \right\rfloor &= \sum_{i=1}^{k} \left\lceil \frac{a_i (q_i w + r_i)}{w} \right\rfloor \\ &= \sum_{i=1}^{k} \left(a_i q_i + \left\lceil \frac{a_i r_i}{w} \right\rfloor \right) \\ &= \sum_{i=1}^{k} a_i q_i + \mathbf{e}''' \\ &= \sum_{i=1}^{k} a_i \left\lceil \frac{b_i}{w} \right\rfloor + \mathbf{e}''' \end{aligned} \tag{3.13}$$

where e''' is an error term with $|e'''| < \sum_i a_i \ll w$. Combining (3.11), (3.12), and (3.13),

$$\begin{aligned} w \mathbf{x}_1^T H \mathbf{x}_2 &= \left\lceil \frac{\text{vec}(S_1^T H S_2)^T \text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)}{w} \right\rfloor - \mathbf{e}' \\ &= \sum_{i=1}^{k} \left\lceil \frac{a_i b_i}{w} \right\rfloor + \mathbf{e}'' - \mathbf{e}' \\ &= \sum_{i=1}^{k} a_i \left\lceil \frac{b_i}{w} \right\rfloor + \mathbf{e}''' + \mathbf{e}'' - \mathbf{e}' \\ &= \text{vec}(S_1^T H S_2)^T \left\lceil \frac{\text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)}{w} \right\rfloor - \mathbf{e} \end{aligned}$$

where $\mathbf{e} = \mathbf{e}' - \mathbf{e}'' - \mathbf{e}'''$ is the new error term, with $|\mathbf{e}| \leq |\mathbf{e}'| + |\mathbf{e}''| + |\mathbf{e}'''| \ll w$ by the triangle inequality. Hence, the smallness of the error term is satisfied, and the following holds:

$$\text{vec}(S_1^T H S_2)^T \left\lceil \frac{\text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)}{w} \right\rfloor = w \mathbf{x}_1^T H \mathbf{x}_2 + \mathbf{e}$$

S c = w x + e □

Since $\text{vec}(S_1^T H S_2)^T$ is a row vector, we can group m' such row vectors, each with different H's, into a matrix S', and we can calculate multiple inner products with different weight matrices $\{H_j\}_{j=1}^{m'}$ with one operation. The height m' of the secret key S' is thus the number of different weighted inner products we are interested in computing, and the width of S' is n^2 because of the vectorize operation.

Taking the weighted inner product of two vectors increases the dimension of the the ciphertext from n to n^2. However, we can use the key-switching technique to bring the dimension down to a more manageable

size, eg. $m'+1$. Thus, the client can compute the key-switch matrix $M \in \mathbb{Z}^{n^2 \times (m'+1)}$ which switches from secret key S' to some S'', making the final ciphertext

$$\mathbf{c}'' = M \left\lceil \frac{\text{vec}(\mathbf{c}_1 \mathbf{c}_2^T)}{w} \right\rfloor \tag{3.14}$$

with secret key $S'' \in \mathbb{Z}^{m' \times (m'+1)}$.

The runtime of the operation is is $O(n^2 m' \ell)$ for both the client and the cloud, where ℓ is the number of bits needed to represent the ciphertext and m' is the number of different weight matrices we are considering. This runtime corresponds to an ℓ time overhead, the same as that of linear transformation.

3.2.4 Polynomial

The three fundamental operations (addition, linear transformation, weighted inner product) are sufficient to produce arbitrary polynomials of our choice. The key component to synthesizing polynomials is the weighted inner product operation, which we shall apply with a slight modification to allow for polynomials with inhomogeneous degrees. Consider the plaintext vector $\mathbf{x} = [x_1, \cdots, x_n]^T$, which we extend to $\mathbf{x}' = [1, x_1, \cdots, x_n]^T$. This new plain text correspond to the new ciphertext

$$\mathbf{c}' := [w, \mathbf{c}^T]^T$$

with secret key

$$S' := \begin{bmatrix} 1 & 0 \\ 0 & S \end{bmatrix}$$

Hence, to calculate polynomials of degree 2, we simply need to take the inner product

$$(\mathbf{x}')^T H(\mathbf{x}')$$

for appropriately chosen H, and we can operate on with minimal modifications the the existing data, the method for which is described above. For example, calculating the polynomial $f(\mathbf{x}) = x_2^2 + 3x_1x_2 - 2x_1 + x_2 + 3$ can bre expressed as an weighted inner product $(\mathbf{x}')^T H(\mathbf{x}')$ with $\mathbf{x}' = [1, x_1, x_2]$ and

$$H = \begin{bmatrix} 3 & -2 & 1 \\ 0 & 0 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

Further, to calculate a polynomial of degree d, the cloud can parallelize the operations, taking $O(\log d \cdot n^3)$ time for n variables.

We note that the overhead of this operation comes mainly from calculating the key-switch matrices on the client side for each of the inner products, proportional to the degree of the polynomial. However, the key-switch matrices only needed to be calculated once for each secret key, and subsequently the cloud can operate with the same matrices on different vectors. Thus, this client-side overhead is justifiable if the same polynomial is to be evaluated on many different values.

In addition, applying weighted inner products operations causes the biggest increase in the error term which needs to have magnitude less than $\frac{w}{2}$ in order for successful decryption. Because of the lack of an error resetting operation, this scheme does not allow for arbitrarily many operations on the data. Hence, we have shown that we are able to compose arbitrary polynomials using a the three fundamental operation that this scheme supports. However, w needs to be chosen to be sufficiently large taking into account the number of weighted inner product operations required for the application.

3.3 Secrecy

An important feature of the homomorphic encryption scheme we are considering is the secrecy of f, the function that the clients send to the cloud. As we have seen, the encryption of the function f is the key-switch matrix M that the client sends to the cloud, and it is important that M does not reveal any information of the coefficients of the linear transformation or the weighted inner product, other than their degree. The secrecy of the scheme relies on the conjectured hardness of an open problem called "Learning with Errors" [6], which we shall formulate now.

Definition 3.8. Learning with Errors (LWE) Problem.
Given polynomially many samples of $(\mathbf{a}_i, b_i) \in \mathbb{Z}_q^m \times \mathbb{Z}_q$ with

$$b_i = \mathbf{v}^T \mathbf{a}_i + \varepsilon_i \tag{3.15}$$

where the error term $\varepsilon_i \in \mathbb{Z}_q$ is drawn from some probability distribution χ, it is infeasible to recover the vector $\mathbf{v} \in \mathbb{Z}_q^m$ with non-negligible probability.

Theorem 3.9. *If Learning with Errors (LWE) is hard, then it is infeasible to recover S' from Equation 3.6*

$$S'M = S^* + E$$

given access to M and S^*.

Proof. We shall reduce this problem to LWE. Suppose that we have a solver for (3.6). We treat each of the elements in the matrices, which are in $\mathbb{Z}$, to be elements of $\mathbb{Z}_q$ for some prime $q \gg \max\{|S'|, |M|, |S^*|, |E|\}$. Then, consider the special case when $S' = (\mathbf{s}')^T$ is a row vector with dimension n, meaning $S^* = \mathbf{s}^*$ and $E = \mathbf{e}$ are also row vectors of dimension n. Then translates to having access to $\mathbf{m}_i$ and $\mathbf{s}^*$ in

$$(\mathbf{s}')^T \mathbf{m}_i = \mathbf{s}^* + \mathbf{e}$$

ie. we have access to n samples of $(\mathbf{m}_i, s_i^*)$ in

$$s_i^* = (\mathbf{s}')^T \mathbf{m}_i - e_i \tag{3.16}$$

Then, we can apply our solver to solve (3.16) for $\mathbf{s}'$, which is equivalent to solving (3.15) for $\mathbf{v}$.

Hence, if LWE is hard, solving (3.6) for S' is also hard. □

Therefore, given the conjectured hardness of LWE, it is infeasible to recover the new secret key S' with the key-switch matrix M. Thus the cloud, as well as any adversaries who intercepted our communications with the cloud, will not be able to recover the secret key S, the plaintext $\mathbf{x}$, as well the G of our linear transformation or the H of our weighted inner products. This completes the proof of the secrecy of the scheme.

4 Applications

We now introduce a few application scenarios of this scheme. The basic format we use is that there is a client who stores data on the cloud in the form of integer vectors, as illustrated in Figure 2. The client then wants to know some function of the vectors without revealing the function to the cloud. This is directly supported by the fundamental operations of the scheme. Further, the data on the cloud can be encrypted with a secret key known only to the client such as in cloud storage applications, thereby hiding user data from third parties. We shall now look at three such applications.

4.1 Classification

A scenario for which this scheme is useful is when the client stores encrypted integer vectors on the cloud and would like to perform a classification task. The client is then able apply polynomial classifiers using weighted inner products as described in Section 3.2.4. Using this scheme, we are able to support some of the most popular machine learning algorithms such as Naive Bayes or Support Vector Machines with a polynomial kernel.

4.2 Search

Another scenario would be that the client stores encrypted integer vectors representing features of documents on the cloud. In addition, the client has a relevance function that is a polynomial in each vector which they send to the cloud for computation. This way, the client's search query remains hidden from the cloud as it evaluates it on each document.

4.3 Feature Extraction

We can generalize classification to give a low dimensional representation of data vectors, which will conserve bandwidth over simply querying all the files. For example, the client stores encrypted long vectors representing images on the cloud and would like to get a feature of the image. The client could then perform a linear transformation on the data using a feature matrix. This allows them to learn a low dimensional representation of the data while not letting the cloud know anything about the data or the feature that they are interested in.

5 Implementation

To examine how applicable the scheme of [1] is to real-world applications, we implemented a library in C++ that handles the described operations in the encrypted domain: addition, linear transformation, and weighted inner products. We used the NTL library for linear algebraic operations, coupled with GMP for large integer arithmetic. We will give performance analysis of the operations in practice, and describe our implementation of two applications on top of the scheme: search and classification. A copy of our code is available on `https://github.com/jamespayor/vector-homomorphic-encryption` for reference.

5.1 Performance

In evaluating the performance of the scheme, we were primarily interested in the overhead that operations on ciphertext introduces over operations on plaintext, and how this overhead compares to that of fully homomorphic encryption. We will start with a brief theoretical analysis, and then give results of our tests.

5.1.1 Theoretical Overhead

When describing each of the operations, we noted that the ciphertext can be just slightly larger than the plaintext (e.g. one or two entries larger), and that key-switch matrices force us to use matrices that are a factor of ℓ larger for linear transformation and inner products. So after the key-switch matrices required for a series of operations are computed, we should expect an $O(\ell)$ slowdown over operations on plaintext (and no slowdown for addition).

Computing the key-switch matrices is another matter entirely though. To perform an $n \times n$ linear transformation G in the encrypted domain, we need to find the new secret key GS through matrix multiplication, which cannot be done in better time than $\Omega(n^{2.8})$ in practice at the moment. Similarly, to compute the key-switch matrix for a weighted inner product $\mathbf{x}_1^T H \mathbf{x}_2$, we need to find $S_1^T H S_2$. So if we deal with dimension in the thousands, we should expect at least two orders of magnitude of slowdown in computing key-switch matrices (over just performing the computation on plaintext).

In practical applications, a series of key-switch matrices for a series of operations can be reused to apply the same operations to many different ciphertext vectors. For instance, we ran the operations on 200 different vectors in the applications we implemented, and we can expect this number to be much higher in realistic cloud applications. Referencing the view of the scheme of Figure 2, the key-switch overheads are paid as a one-off cost by the client, and then the cloud pays the factor of ℓ overhead for each linear transformation or inner product operation it executes. This division of work should be kept in mind in designing applications, but note that the key-switch overhead is quickly amortized away as the number of ciphertext vectors processed by the cloud increases.

5.1.2 Test Results

First, note that we found that addition introduces negligible overhead, and that the time to perform an inner product was virtually equivalent to linear transformation by a square matrix. Bearing this in mind, we examine the performance of applying an $n \times n$ linear transformation to an n-dimensional vector to characterize the performance of our implementation.

As discussed before, we will typically use a single key-switch matrix to apply an operation to several ciphertext vectors in practice, so to replicate this we consider the time that the vector encryption scheme takes to generate a key-switch matrix for a linear transformation and to apply it to 50 ciphertext vectors. So we amortize the initial overhead over 50 operations in the following analysis. In practice we would expect to be dealing with at least hundreds of ciphertext vectors, so the assumption is reasonable.

To test the relative performance of our somewhat homomorphic scheme against fully homomorphic encryption, we tried comparing our running times to that of HElib, a library hosted on Github that implements the BGV scheme [9] with optimizations, as well as optimized primitive operations. HElib uses NTL and GMP internally (open source numeric libraries), as does our implementation.

It was very difficult to tailor the size of the vectors of HElib (as this depended implicitly on the group parameters given as inputs), and the largest size we could achieve was 88 element vectors. (HElib is far from practical applications, but does give an interesting reference point to fully homomorphic encryption.) The parameters we gave to HElib allowed for plaintext size of 4 bits, and the security parameters were low. To compare to Zhou's scheme, we set $w = 2^{12}$ and $\ell = 16$, and bounded the error matrix entries to be $0 - 1$. The performance comparison is given in Figure 3, in which we compare the time to apply random $n \times n$ linear transformations using each scheme.

We were also interested in the overhead our scheme introduces over operations on plaintext. This comparison is given in Figure 4, where we compare the time that an $n \times n$ linear transformation takes NTL to compute against operations in the encrypted domain (which also use NTL for linear algebra). We give times for parameters that support 4-bit and 32-bit plaintext. To handle 32-bit plaintext values, we used $w = 2^{40}$ and $\ell = 100$, with error matrix entries at most 10. We expected to take $O(\ell)$ longer in the encrypted domain (100 times longer for 32-bit numbers, and 16 times longer for 4-bit numbers), so we give the same plot scaled by the relevant values of ℓ in Figure 5.

As shown, we found that HElib has much poorer constant factors than our implementation, appearing to be roughly an order of magnitude slower (assuming it would scale quadratically if we could work out how to have larger vectors). Further, our implementation takes almost exactly ℓ times longer to perform operations in the encrypted domain than it would take to perform them on plaintext.

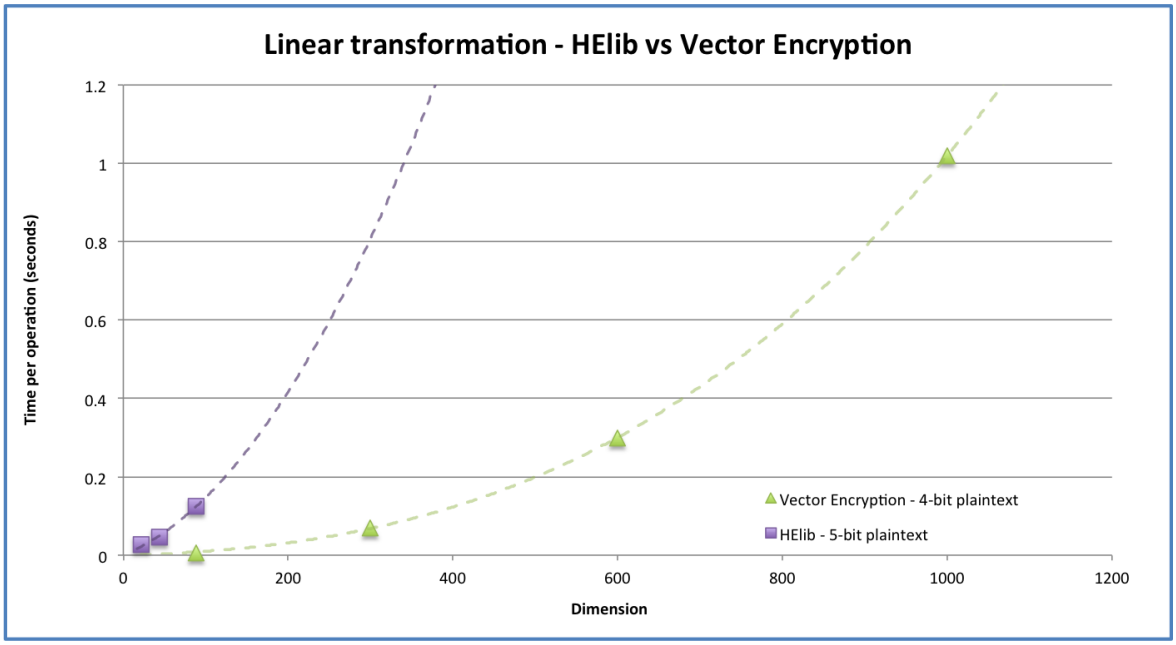

Figure 3: Performance comparison of matrix-vector multiplication in the encrypted domain for HElib and the Vector Encryption scheme.

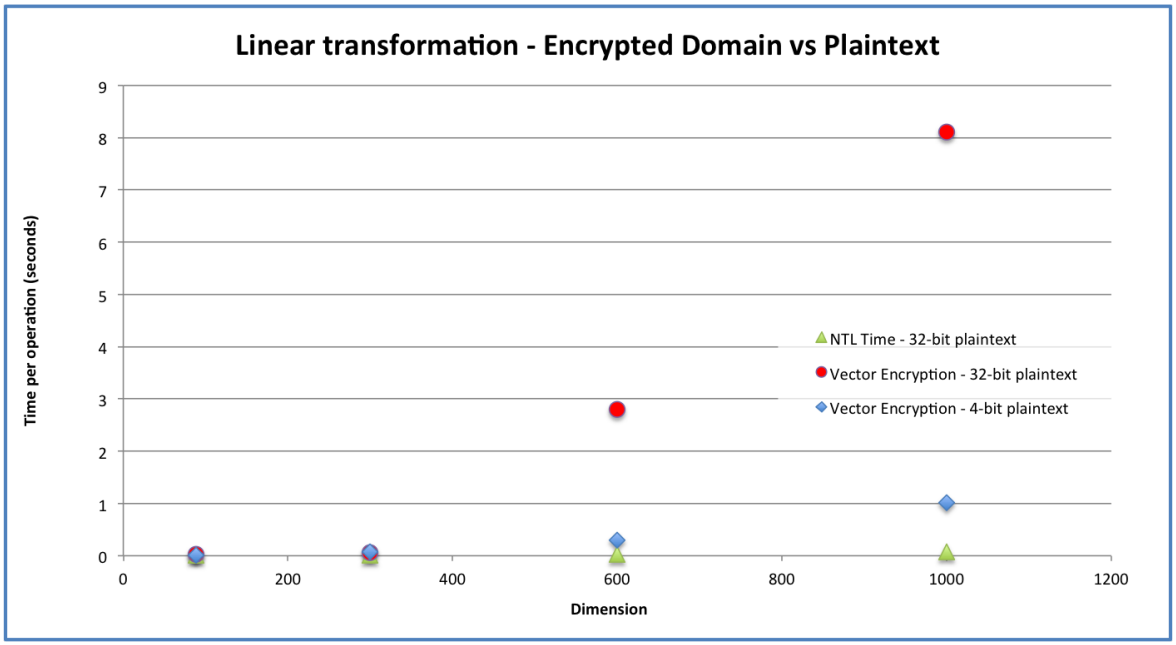

Figure 4: Performance comparison of matrix-vector multiplication in the encrypted domain with in the unencrypted domain.

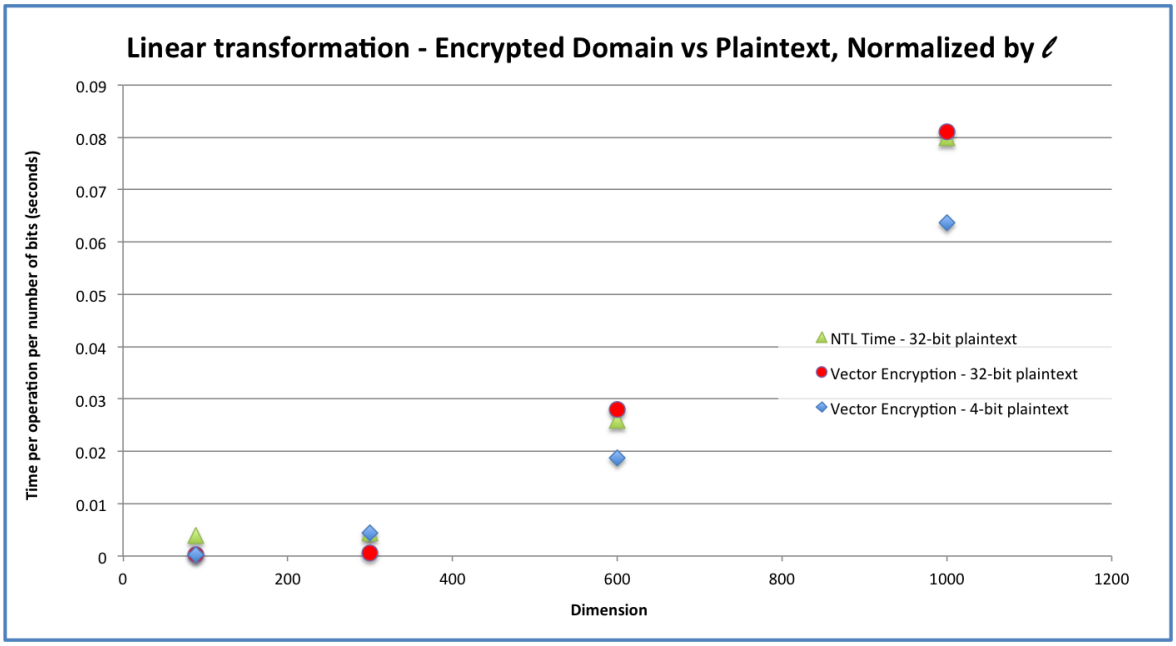

Figure 5: Performance comparison of encrypted versus unencrypted operations, scaled by ℓ, the theoretical overhead.

5.2 Search and Classification Applications

Having implemented the operations of the scheme, we built two applications on top of it: email search and spam classification. We used the Enron email dataset, storing on our cloud server encrypted email and encrypted *feature vectors* (integer vector representations of each email), with the client holding the secret keys. We allowed the client to query the server for functions of the feature vectors, and built example query functions for search relevance and spam classification. The client was also able to query the server for raw email data, to be able to act on results it finds.

To choose a feature vector representation of emails, we need to choose features that can effectively be used in polynomial models (so that we can make use of them with the allowed operations), and we also need them to be low-dimensional, due to the quadratic dependence the operations have in dimension. We made use of simple linear models, and so chose feature vectors to have the relative word counts of the most common 3000 words (that were not "stop words" like "and"). (This is a TF-IDF representation, and we scaled up the values to be integers in $[0, 1000]$.)

Although having 3000-dimensional feature vectors made it expensive to compute key-switch matrices on the client side, we were able to keep the final key-switch matrices small in size because we only needed low-dimensional output. The overall network traffic in each application was around 5 integers per email on the server. It takes the client somewhere between 10 and 20 seconds to generate the key-switch matrices for its query, and then the server around the same time to apply them to all of the feature vectors sequentially. (The server allowed queries on 200 emails.)

To compute search rankings, for each email we added together the weighted counts of words in our search query, implemented with a 1×3000 linear transformation. (This is equivalent to a naive-bayes theoretical model.) The client takes a given query, computes the key-switch matrix necessary to apply the transformation, and sends this to the server. The server applies the transformation to all feature vectors, sending back a list of the low-dimensional results, which the client decrypts. The client then has scores for each document, and can proceed to show the user the ordering or make further queries. More complicated query functions are possible, but this worked nicely.

For spam classification, we took a similar approach, training a linear model (again theoretically equivalent to naive-bayes) to find likelihoods that a given feature vector corresponds to a legitimate or spam email. This was sufficient to correctly classify 95% of spam, but in instances where a linear model is not enough we can begin to rely on precomputed features, or on machine learning methods that learn polynomial models instead (e.g. using a polynomial kernel for a SVM).

We learn from implementing these applications that given good models which do not require large feature vectors, it is feasible to run queries on encrypted data in practice. Any realistic cloud application will particularly benefit from evaluating queries in parallel, allowing the server to run the desired operations on many items, and also benefit from the ability to train stronger models that can tolerate an order of magnitude fewer dimensions than ours, so we can expect faster query times than in our implementation. It is also interesting to note is that in addition to the server being unable to learn about the data or the specifics of the queries, it cannot even distinguish between search and spam queries - all the server can learn is the degree of the polynomial being evaluated.

6 Conclusion

The ability to compute on encrypted data holds the future of cloud computing. By taking advantage of the conjectured hardness of the Learning with Errors problem, Zhou and Wornell [1] have devised a "Somewhat Homomorphic Encryption scheme" that enables computation of encrypted functions on encrypted data.

The approach of the scheme naturally lends itself to modern cloud storage, where we expect users to have their data stored across servers and to want to be able to make private queries. The viability of this scheme, hiding both queries and data whilst being almost feasible in practice, certainly warrants further investigation. That this scheme can achieve orders of magnitude less slowdown than Fully Homomorphic Encryption highlights an interesting point about the trade-off between performance and the computation supported. It seems likely that when Fully Homomorphic Encryption is not strictly required in practice, it may be more effective to rely on restricted schemes, like the "Somewhat Homormophic" examined in this paper. It is certainly the case that a lot in practice can be achieved just with the operations supported. Even with just support for linear transformations we can achieve viable search and classification over encrypted data, and we can imagine using more complicated models that rely on more general polynomial computation using this scheme.

It would be interesting to see Zhou's scheme improved to have better dependence on ℓ, currently having a linear slowdown in the number of bits, and it would be exciting if this scheme could be extended to support Fully Homomorphic Encryption, for the scheme is definitely a promising candidate for real-world application of Homomorphic Encryption.

Acknowledgments. The authors would like to thank Professor Ron Rivest for offering 6.857, through which the authors were exposed to the fascinating subject of cryptography; Professor Vinod Vaikuntanathan for giving a guest lecture on Homomorphic Encryption which inspired the authors to investigate this topic in more depth through this final project; and the TAs for offering valuable feedback throughout the project, which we incorporated into many of our decisions.

References

[1] Zhou, Hongchao, and Gregory Wornell. "Efficient homomorphic encryption on integer vectors and its applications." *Information Theory and Applications Workshop (ITA)*. IEEE, 2014.

[2] Lagendijk, Reginald L., Zekeriya Erkin, and Mauro Barni. "Encrypted signal processing for privacy protection: Conveying the utility of homomorphic encryption and multiparty computation." *Signal Processing Magazine*, IEEE 30.1 (2013): 82-105.

[3] Riggio, Roberto, and Sabrina Sicari. "Secure aggregation in hybrid mesh/sensor networks." *Ultra Modern Telecommunications & Workshops, ICUMT*. IEEE, 2009.

[4] Rivest, Ronald L., Len Adleman, and Michael L. Dertouzos. "On data banks and privacy homomorphisms." *Foundations of secure computation* 4.11 (1978): 169-180.

[5] Gentry, Craig. *A fully homomorphic encryption scheme*. Diss. Stanford University, 2009.

[6] Brakerski, Zvika, Craig Gentry, and Shai Halevi. "Packed ciphertext in LWE-based homomorphic encryption." *Public Key Cryptography* vol. 7778, 2013.

[7] Rivest, Ronald L., Adi Shamir, and Len Adleman. "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM* 21.2 (1978): 120-126.

[8] ElGamal, Taher. "A public key cryptosystem and a signature scheme based on discrete logarithms." *Advances in Cryptology*. Springer Berlin Heidelberg, 1985.

[9] Brakerski, Zvika, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping." *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 2012.