SHIELD: Scalable Homomorphic Implementation of Encrypted Data-Classifiers

Alhassan Khedr, Glenn Gulak
ECE, University of Toronto
Email: alhassan, gulak@ece.utoronto.ca

Vinod Vaikuntanathan
CSAIL, MIT
Email: vinodv@mit.edu

Abstract

Homomorphic encryption (HE) systems enable computations on encrypted data, without decrypting and without knowledge of the secret key. In this work, we describe an optimized RLWE-based implementation of a variant of the HE system recently proposed by Gentry, Sahai and Waters [21] (henceforth called GSW). Although this system was widely believed to be less efficient than its contemporaries, we demonstrate quite the opposite behavior for a large class of applications.

We first highlight and carefully exploit the algebraic features of the system to achieve significant speedup over the state-of-the-art HE implementation, namely the IBM homomorphic encryption library (HElib) [23]. We introduce several optimizations on top of our HE implementation, and use the resulting scheme to construct a homomorphic Bayesian spam filter, secure multiple keyword search, and a homomorphic evaluator for binary decision trees.

Our results show a factor of 10× improvement in performance (under the same security settings and platforms) compared to HElib for these applications. Our system is built to be easily portable to GPUs (unlike HElib) which results in an additional speedup of up to a factor of 10× to offer an overall speedup of 100×.

Keywords— Homomorphic Encryption, FHE, Ring LWE, Bayesian Filter, Secure Search, Decision Trees, GPU.

1 Introduction

A fully homomorphic encryption scheme (FHE) is an encryption scheme that allows evaluation of arbitrary functions on encrypted data. Starting with Gentry's mathematical breakthrough constructing the first plausible FHE scheme [18, 19], we have seen rapid development in the theory and implementation of homomorphic encryption (HE) schemes. HE schemes can now be based on a variety of cryptographic assumptions – approximate greatest common divisors [13, 15], learning with errors (LWE) [7, 8, 10, 21], and Ring-LWE (RLWE) [9, 20, 28].

We demonstrate that new HE techniques can be used to speed up data-classifier class of applications. In particular, we show that a careful consideration of the mathematics underlying the recently proposed GSW scheme and Brakerski and Vaikuntanathan [10] (henceforth called BV), together with the features of the specific applications and computing platforms results in significant speedups.

Our starting point is a RLWE variant of the framework of GSW and BV. The main feature of this scheme is that the error in these ciphertexts grow with homomorphic operations, but they grow much slower than in previous schemes. In particular, when doing a large number of homomorphic multiplications, the error grows linearly (as opposed to quasi-polynomially, as in previous schemes, e.g. HElib) in the number of multiplications. This enables us to choose smaller parameters than the other HE schemes for the same security level, and therefore, better overall efficiency.

The next key observation is that when multiplying a sequence of numbers using the scheme, if the final result happens to be a zero, then the error level plummets close to zero. Of course, because of the security of the scheme, there is no way for the homomorphic evaluator to tell if and when this happens, but if such an event is guaranteed to happen often during homomorphic evaluation, we are guaranteed to have small error growth. As an illustrative example, consider evaluating an expression of the form

$$F(x_1, \ldots, x_v) = \sum_{(y_1,\ldots,y_v)\in S} \left(\prod_{i=1}^{v} \overline{(x_i \oplus y_i)} \right) \qquad (1)$$

where $x_1,\ldots,x_v$ are v-tuples of input encrypted bits, $y_1,\ldots,y_v$ are v-tuples of bits in some set S, operation $\overline{(x_i \oplus y_i)}$ represents binary XNOR between bits x_i and y_i. Since the form of the expression guarantees us that exactly one of the terms may survive ($F = 1$ when $x_1,\ldots,x_v \in S$, otherwise $F = 0$), the homomorphic evaluator is guaranteed to have a small total error growth (even though the evaluator does not know precisely which term will survive). It can be noticed that (1) is identical to the server's computation in a PIR scheme. Indeed, the same format appears in several applications, including the ones we discuss in Section 4.

In this work we introduce some algorithmic optimizations to the work in [10, 21] as will be discussed in Sections 3.1 and 3.1.1 to reduce computational complexity. We next introduce the notion of *decrypting a flag submerged in noise* described in Section 4.1 which can be used to argue that decryption error in fact gives us a meaningful bit of information! This is unlike all other lattice-based HE schemes that we are aware of where one gives up hope the moment the error exceeds a certain threshold. We also design circuits for spam filtering and secure search applications that reuse certain computations and reduce the number of multiplications, and thus reduce the overall running time, by a factor of at least two. Finally, we carefully exploit the parallelism in the encryption system by implementing it on a GPU platform. In addition, our data-classifier design is completely scalable and its running time can be reduced proportional to the number of GPU cores utilized.

Our GPU implementation of the HE scheme scores a ciphertext (Ctxt) multiplication run time of 0.037 seconds, and the CPU implementation requires 0.372 seconds (See Table 2 for the design environment). Our CPU implementation scores a speedup of $10\times$ over IBM HElib [23]. Our GPU implementation gives us a further $10\times$ speedup, and overall a factor of $100\times$ speedup over HElib (Since HElib is built on the thread-unsafe NTL library, we were not able to natively port it to the GPU setting). Also for the same security level, our ciphertext size is smaller than HElib by a factor of $1.5\times$. Our improvement is realized through a reduction in parameters for the same security level and homomorphic capacity, stemming from our observations about noise growth. This ultimately leads to faster implementation and smaller ciphertext sizes.

We build on top of this to construct an encrypted data classifier that can tag different e-mails with different priorities depending on the encrypted data inside them. Our data-classifier takes the idea of Private Information Retrieval (henceforth named PIR) one step forward, and homomorphically computes on the encrypted data retrieved by the PIR to obtain useful pieces of information. We also build a secure multiple keyword search engine that can search for encrypted keywords inside encrypted files. The performance of the data-classifier as well as the secure search engine depends on the size of the database/file and can be as low as a few seconds. Finally we build a binary decision tree using our HE scheme. The example decision tree described took 0.296 seconds. We present additional details about these results in Section 5.

The rest of the paper is organized as follows. Section 2 presents related work. In Section 3 we introduce the improved encryption scheme. The encrypted data-classifier design, secure multiple keyword search engine and encrypted binary decision trees are introduced in Section 4. Performance results are introduced in Section 5. Finally we conclude in Section 6.

2 Related Work

Previous constructions of RLWE-based FHE schemes include [7, 8, 20]. One of the drawbacks of these schemes is the need to maintain a so-called "modulus chain" which increases the size of the prime number and consequently increases the ring dimension for the same security level. They also need to perform expensive modulus and key switching operations.

Based on [7], Halevi and Shoup designed a homomorphic encryption library [20, 23], but due to the need of some additional large data structures and functions, the performance of their library was diminished. A performance comparison between our library and the IBM library is presented in Section 5. In [28] they implemented a variant of the RLWE FHE scheme. Our results also show considerable speedups over their implementation. Another homomoprhic library was developed by Rohloff, Cousins, and Peikert [14]. In their paper they implement primary building blocks in hardware to accelerate their system. There are no results available yet publicly to compare our library with theirs. Other implementation attempts were made but they were either incomplete implementations of HE scheme capable only on performing one multiplication operation [35], or based on other cryptographic assumptions [11,29,33,34].

Applications analyzed in this paper were primarily inspired from [28, 31]. We extended their ideas and developed full algorithms. The work on CryptDB [30] used a combination of very simple HE schemes to implement a subset of encrypted SQL queries, and the work on "ML Confidential" [22] implemented simple classification tasks on encrypted data. Searching an encrypted database was previeously addressed by [3, 5, 6, 12]. One drawback in [3, 5] was the need for a special key to aid the server in performing the search request. They achieve a weaker security notion, namely one where partial information about the data access pattern is leaked. In particular, in the work of [6], the same server requests would generate the same tags. For a simple and general overview of homomorphic encryption concepts the reader is encouraged to read [1, 4, 24].

3 The Encryption System

Notation For an odd prime number q we identify the ring $\mathbb{Z}/q\mathbb{Z}$ (or $\mathbb{Z}_q$) with the interval $(-q/2, q/2) \cap \mathbb{Z}$. The notation $[x]_q$ denotes reducing x modulo q. Our implementation uses polynomial rings defined by the cyclotomic polynomials $R = \mathbb{Z}[X]/\Phi_m(X)$, where $\Phi_m(X) = x^n + 1$ is the irreducible mth cyclotomic polynomial, in which n is a power of 2 and $m = 2n$. We let $R_q = R/qR$. Any type of multiplication including matrix and polynomial multiplication is denoted by the multiplication operator '$\cdot$'. Rounding up is denoted by $\lceil a \rceil$. Matrices of rings are defined as $A_{M\times N}$, where $A_{ij} \in R_q$ and M, N are the matrix dimensions. Row vectors are represented as $[a \ \ b]$, where a and b are the vector elements. Column vectors on the other hand are represented as $[a \ ; \ b]$.

Ring Learning With Errors The ring learning with errors problem (RLWE) was introduced in [26]. It is the mapping of the LWE problem from the vectors over $\mathbb{Z}_q$ to polynomial rings over R_q. The RLWE problem is to distinguish between the following two distributions. The first distribution is to draw (a,b) uniformly from R_q^2. The second is to first draw t uniformly from R_q. Then sample (a,b) as follows. Draw a uniformly from R_q, sample e from a discrete Gaussian error distribution $e \leftarrow D_{R_q,\sigma}$, and set $b = a \cdot t + e$.

3.1 The Encryption Scheme

The parameters of the system are n, the degree of the number field; q, the modulus; σ_k and σ_c, the standard deviation of the discrete Gaussian error distribution in the keyspace and ciphertext space respectively; $\ell \triangleq \lceil \log q \rceil - 1$; and $N = 2\ell$ that governs the number of ring elements in a ciphertext. The setting of these parameters depends on the security level λ (say, $\lambda = 80$ or 128 bits) as well as the complexity of functions we expect to evaluate on ciphertexts.

Let the *bit decompose* function $\mathsf{BD}(d)$ transform the polynomial d to the ℓ-dimensional vector $[d(0), \ldots, d(\ell-1)]$, which are the bitwise decomposition of d. That is, $d(0), \ldots, d(\ell-1)$ are polynomials with 0-1 coefficients such that $d = \sum_{\tau=0}^{\ell-1} d(\tau) \cdot 2^\tau$, which represents the *bit decompose inverse* function $\mathsf{BDI}(d)$. Note that $A_{N\times N} = \mathsf{BD}(B_{N\times 2})$, inversely $B_{N\times 2} = \mathsf{BDI}(A_{N\times N})$, and that $\mathsf{BD}(B_{N\times 2}) \cdot \mathsf{BDI}(A_{N\times N}) = A_{N\times N} \cdot B_{N\times 2}$.

We introduced some algorithmic optimizations to the encryption system in [10, 21] in order to reduce computational complexity and to speedup our operations, as will be detailed below. Our encryption system works as follows.

- $\mathsf{Keygen}(1^\lambda)$: Choose polynomial $t \leftarrow D_{R_q,\sigma_k}$. The secret key $sk = s_{2\times 1} \leftarrow [1; -t] \in R_q^2$. Uniformly sample $a \leftarrow R_q$, $e \leftarrow D_{R_q,\sigma_k}$, set $b = a \cdot t + e$, The public key $pk = A_{1\times 2} = [b \ \ a]$. Note that
$$A_{1\times 2} \cdot s_{2\times 1} = b - a \cdot t = e \tag{2}$$
(as opposed to $sk = v = \mathsf{PO2}(s)$ in [10, 21], where *power of two* $\mathsf{PO2}(x)$ is defined as $[x, \ 2x, \cdots, \ 2^{\ell-1}x]$. We have a smaller secret key by a factor of ℓ times)
- $\mathsf{Enc}(\mathsf{pk}, \mu)$: The message space of our encryption scheme is R_q. Sample a uniform vector $r_{N\times 1} \in \{0,1\}$, $E_{N\times 2} \leftarrow D_{R_q^{N\times 2},\sigma_c}$, encrypt the plain text polynomial $\mu \in R_q$ by calculating
$$C_{N\times 2} = \mu \cdot \mathsf{BDI}(I_{N\times N}) + r_{N\times 1} \cdot A_{1\times 2} + E_{N\times 2} \tag{3}$$
(as opposed to $C_{N\times N}$ in [10, 21], we have a smaller ciphertext by a factor of ℓ times)
- $\mathsf{Dec}(\mathsf{sk}, C)$: Given the ciphertext C, the plaintext $\mu \in R_q$ is restored by multiplying C by the secret-key s as follows :
$$\begin{aligned} C_{N\times 2} \cdot s_{2\times 1} &= (\mu \cdot \mathsf{BDI}(I_{N\times N}) \\ &\quad + (r_{N\times 1} \cdot A_{1\times 2} + E_{N\times 2})) \cdot s_{2\times 1} \\ &= \mu \cdot \mathsf{BDI}(I_{N\times N}) \cdot s_{2\times 1} \\ &\quad + r_{N\times 1} \cdot A_{1\times 2} \cdot s_{2\times 1} + E_{N\times 2} \cdot s_{2\times 1} \\ &= \mu \cdot \mathsf{BDI}(I_{N\times N}) \cdot s_{2\times 1} \\ &\quad + r_{N\times 1} \cdot e + E_{N\times 2} \cdot s_{2\times 1} \\ &= \mu \cdot \mathsf{BDI}(I_{N\times N}) \cdot s_{2\times 1} + error \end{aligned} \tag{4}$$
(as opposed to $\mathsf{Dec}(\mathsf{sk}, C) = C_{N\times N} \cdot v_{N\times 1}$ in [10, 21], we have fewer operations in Dec by a factor of ℓ times)

Observe that the first ℓ coefficients in the first term of the last equation in (4) are in the form $\mu, 2\mu, \cdots, 2^{\ell-1}\mu$.

This means that the element at location $i \in [0, \ell-1]$ is in the form $\mu \cdot 2^i + error$. That is, the most significant bit of each entry carries a single bit from the number μ assuming that $error < q/2$ and there is no wrap-around mod q as was described in [21].

3.1.1 Homomorphic Operations

Homomorphic operations are described next.

- $\mathsf{ADD}(C,D)$: To add two ciphertexts $C_{N\times 2}$ and $D_{N\times 2} \in R_q^{N\times 2}$ encrypting μ_1 and μ_2 respectively, simply output $C_{N\times 2} + D_{N\times 2}$, which is an entry-wise addition.

- $\mathsf{MULT}(C,D)$: To multiply two ciphertexts $C_{N\times 2}$ and $D_{N\times 2} \in R_q^{N\times 2}$, output $\mathsf{BD}(C_{N\times 2}) \cdot D_{N\times 2}$.

  (as opposed to $\mathsf{MULT}(C,D) = \mathsf{FLATTEN}(C_{N\times N} \cdot D_{N\times N})$ in [10, 21], where $\mathsf{FLATTEN}(A)$ is defined as $\mathsf{BD}(\mathsf{BDI}(A))$. We have fewer operations in MULT by a factor of at least ℓ times)

Correctness of homomorphic addition is immediate, however it is not that obvious for the homomorphic multiplication. It is clear that the multiplication algorithm is asymmetric in the input ciphertexts C and D. That is, we treat the components of D as a whole, whereas the components of C are broken up into their "bit-wise decompositions". This is a "feature" that is inherited from the work of BV [10]. It is shown below that this multiplication method is not only correct, it also gives a slow noise-growth rate.

The correctness of the multiplication operation can be noticed from the decryption operation. Matrix dimensions are removed for clarity.

$$\begin{aligned}
\mathsf{BD}(C)\cdot D\cdot s &= \mathsf{BD}(C)\cdot(\mu_2\cdot\mathsf{BDI}(I)+r_2\cdot A+E_2)\cdot s\\
&= \mathsf{BD}(C)\cdot(\mu_2\cdot\mathsf{BDI}(I)\cdot s+r_2\cdot e+E_2\cdot s)\\
&= \mu_2\cdot C\cdot s+\mathsf{BD}(C)\cdot(r_2\cdot e+E_2\cdot s)\\
&= \mu_2\cdot(\mu_1\cdot\mathsf{BDI}(I)\cdot s+r_1\cdot e+E_1\cdot s)\\
&\quad +\mathsf{BD}(C)\cdot(r_2\cdot e+E_2\cdot s)\\
&= \mu_2\cdot\mu_1\cdot\mathsf{BDI}(I)\cdot s+\mu_2\cdot(r_1\cdot e+E_1\cdot s)\\
&\quad +\mathsf{BD}(C)\cdot(r_2\cdot e+E_2\cdot s)\\
&= \mu_2\cdot\mu_1\cdot\mathsf{BDI}(I)\cdot s+\mu_2\cdot error_1\\
&\quad +\mathsf{BD}(C)\cdot error_2\\
&= \mu_2\cdot\mu_1\cdot\mathsf{BDI}(I)\cdot s+error
\end{aligned} \tag{5}$$

which is the encryption of $\mu = \mu_2 \cdot \mu_1$.

Function 1: Multiply "v" Ciphertexts Function

```
Input: "v" Ctxts C_1, C_2, ··· , C_v
Output: C_accum
        The multiplication result of "v" input Ctxts.

C_accum = C_1
For i from 2 to v {
    C_accum = C_accum × C_i
}
Return C_accum
```

Noise Analysis Correct decryption depends crucially on the ciphertext noise being bounded. Thus, it is crucial to understand how homomorphic operations increase ciphertext noise. Let C be a fresh ciphertext. We make the following observations, following [10].

Homomorphic addition of v ciphertexts increases the noise by a factor of v in the worst case. In practice, since the coefficients of the error polynomials follow a Gaussian distribution, the factor is closer to $O(\sqrt{v})$.

Homomorphic multiplication is significantly more interesting. Multiplication of two ciphertexts $C = \mathsf{Enc}(\mu_1)$ and $D = \mathsf{Enc}(\mu_2)$ with error magnitudes B_1 and B_2 respectively, increases the error to $O(B_1 \cdot \|\mu_2\|_1 + B_2 \cdot n\log q)$ in the worst case, and $O(B_1 \cdot \|\mu_2\|_1 + B_2 \cdot \sqrt{n\log q})$ in practice. Here, $\|\mu\|_1$ denotes the ℓ_1 norm of the message polynomial μ. *The key fact to note here is that the error dependence on the two ciphertexts is asymmetric.*

Better Error in Homomorphic Multiplication To multiply v ciphertexts it is crucial to pay attention to the order of multiplication. In the applications, μ will typically be 0 or 1, meaning that the growth is simply additive with respect to B_1. Thus, the best way to multiply v ciphertexts with (the same) error level B is through an accumulator-like algorithm as in Function 1, rather than using a binary tree of multiplications (which grows the error at superpolynomial rates). The resulting error growth is $O(B\cdot\ vn\log q)$ in the worst case, and $O(B\cdot\ \sqrt{vn\log q})$ in practice.

For example consider (1), the noise grows to $O(B \cdot vn\log\ q\cdot|S|)$ in the worst case, or $O(B\cdot\sqrt{vn\log\ q|S|})$ in the typical case. This is in contrast to $O(B\cdot\sqrt{(n\log q)^{\log v}|S|})$ when using the Brakerski-Gentry-Vaikuntanathan [7] encryption scheme, implemented in IBM HElib. Indeed, such expressions are far from atypical – they occur quite naturally in evaluating decision trees and PIR-like functions as will be discussed in Section 4.

Zero Plaintext, Zero Error Yet another source of improvement is evident when we inspect the error term $B_1 \cdot \|\mu_2\|_1 + B_2 \cdot n \log q$. When we multiply using an accumulator as in Function 1, B_2 represents the smaller error in the fresh ciphertexts C_i, and B_1 represents the larger error in the accumulated ciphertext C_{accum}. We see that if C_i encrypts $\mu_2 = 0$, then the larger error term B_1 vanishes in the error expression!

This phenomenon manifests itself in evaluating the expression in (1) as well. When evaluating each of the products in (1), the error grows proportional not to v, the total number of multiplications, but rather with k, the longest continuous chain of 1's starting from the end. This is because the last time a zero is encountered in the multiplication chain, the error vanishes, by the observation above. Assuming that S is a "typical set", the expected length of a continuous chain of trailing 1's is $\sum_{i=1}^{v} i \cdot 2^{-i} < 2$. In other words, the multiplicative factor of v vanishes from the error expression as well, and we get error growth close to $O(B \cdot \sqrt{n \log q |S|})$. This is the same effect as if one were merely adding $|S|$ ciphertexts.

How to Set Parameters Let f be the function that we are evaluating, for example the expression in (1). Let $\mathsf{error}_f(B,n,q)$ denote how much the error grows when evaluating a function f on ciphertexts in R_q with an initial error of magnitude B. For correct decryption, we need

$$\mathsf{error}_f(B,n,q) < q/2 \tag{6}$$

Since errors grow slower in our scheme, q can be set to be correspondingly smaller. Following the analysis of Lindner and Peikert [25], for a security level of λ bits, we need

$$n > \log q(\lambda + 110)/7.2 \tag{7}$$

Since our $\log q$ is smaller, we can set our n to be smaller, for the same security level λ. In turn, since we now have a smaller n, our new $\mathsf{error}_f(B,n,q)$ is smaller, leading to an even smaller q, and so on. In other words, we are in a virtuous cycle of shrinking parameters. The optimal parameters are obtained by solving both the above inequalities together. Table 1 summarizes our final parameter selection.

Table 1: Parameter Selection and Keys/Ctxt Sizes. "units" here refers to the size of the operand used to store each element, which is equal to ℓ bits.

Parameter	**RLWE** (This work)
λ	80
n	1024
ℓ	31
N	$2 \cdot \ell = 62$
σ_k, σ_c	10
SK size (units)	$2 \times n = 2,048$
PK size (units)	$2 \times n = 2,048$
Ctxt size (units)	$N \times 2 \times n = 126,976$

4 Candidate Applications

4.1 Homomorphic Spam Filter

We implement a homomorphic version of Bayesian spam filters [32]. The main idea behind a Bayesian classifier is that words have certain probabilities of occurrence in *authentic* emails (sometimes called *ham* emails) and in *spam* emails. Since the filter doesn't know these probabilities in advance, email training sets are used to estimate these probabilities. The training phase is assumed to take place on unencrypted training sets, and results in a database of words together with probabilities associated to each word arising in spam e-mails. Once this database is created, the word probabilities are used to classify new emails.

Let p_w denote the probability that a word w occurs in spam e-mails. Given an e-mail with key words $(w_1, \ldots, w_K)$, there are many techniques to combine the probabilities of each word to compute a final estimate of whether the e-mail should be classified as spam. The simplest perhaps is to use Bayes rule. This results in the following expression for p, the probability that the e-mail will be classified as spam.

$$p = \frac{p_{w_1} p_{w_2} \cdots p_{w_K}}{p_{w_1} p_{w_2} \cdots p_{w_K} + (1-p_{w_1})(1-p_{w_2}) \cdots (1-p_{w_K})} \tag{8}$$

At a high level, the email server will receive encrypted words w_i, and map them, homomorphically, into the numbers p_w. Once we obtain these numbers p_w, we wish to compute the expression above to obtain p.

The first downside of the equation above, when it comes to homomorphic computations, is that integer divisions are extremely expensive to carry out using current homomorphic encryption schemes. In order to overcome this, we make a number of reformulations of the equation above, as follows.

$$\eta \triangleq \log(1-p) - \log p = \sum_{i=1}^{K} (\log(1-p_{w_i}) - \log p_{w_i}) \tag{9}$$

and we will let

$$\eta_{w_i} \triangleq \log(1-p_{w_i}) - \log p_{w_i} \tag{10}$$

In other words, the email training phase will store the numbers η_w for each word w in the dictionary (rather than the numbers p_w). The numbers η_w are represented as binary fixed-point numbers, whose bits are encoded into the coefficients of polynomial π_w. For example, $\eta_w = 101_b$ is represented as the polynomial $\pi_w = x^0 + x^2$. The addition of two binary polynomials will not generate a carry between adjacent polynomial elements, rather polynomial elements will grow individually and will be appropriately reconstructed after decryption (e.g. $101_b + 111_b = 212$, which will be constructed back after decryption to 1100_b). The encrypted spam filter computation will take as input an encrypted word w, map it first into an encrypted η_w as will be described in Function 5, and then simply perform a homomorphic addition of the η_w to get an encrypted η. This is then sent back to the client who decrypts, recovers η using her secret key, and computes $p = 1/(2^\eta + 1)$ in the clear.

The only remaining question is how to map encrypted words w into encrypted η_w. When input e-mails and words are not encrypted, matching a certain word is an easy task. Each email word can be searched across the database. If the word is found, the corresponding number η_w is fetched. On the other hand, when input emails are encrypted, matching words become much harder. This "lookup problem" is the same as the problem of private information retrieval (PIR) [16, 36]. Our data-classifier takes the idea of PIR one step forward, it homomorphically computes on the encrypted data retrieved by the PIR to obtain useful pieces of information. Other PIR constructions [2, 8, 16, 17, 27, 36] cannot implement data-classifiers the way we do because they either: (a) cannot compute with the PIR responses, or, (b) their plaintext field is only mod 2 (or modulo a small prime, for efficiency purposes) and thus they cannot do integer addition as required by (9). Our HE on the other hand has the advantage of being able to use the full modulo-q domain for plaintext additions. As should be clear from the description above, spam filtering is just an example of a class of "lookup-and-compute" type of applications for which we can use our HE scheme.

Function 2 shows how to encrypt individual words in a given list (email). Function 3 shows how we match an input encrypted word versus another unencrypted word from the database. The matching function in Function 3 can be used to construct our encrypted-email spam-filter by simply multiplying the database word probabilities by the "match" output as in Function 5. Only the words that find a match in the database will contribute towards the final probability. It is also possible to keep the database encrypted to protect it from outside attackers. To do this, the matching function should be replaced by `EncWordMatch` presented in Function 4 which performs bit matching for two encrypted inputs, but at an extra cost of two extra Ctxt multiplications to implement the XNOR operation.

Function 2: Word List Encryption

Input: Set of words in a list (email)
Output: Encrypted words using HE

For each word in the list {
  a = Hash(word).
  For each bit i in a {
    C_i = Encrypt(a_i).
    Store C_i to the output list.
  }
}

Function 3: Word Matching `WordMatch`

Input$_1$: Encrypted bits of Word$_1$ (C_i)
Input$_2$: Plaintext Word$_2$
Output: Binary bit "$match$" = 1 if words match, 0 otherwise

$match = 1$
a = Hash(Word$_2$).
For each bit i in a {
  If ($a_i = 1$)
    $B_i = C_i$
  Else
    $B_i = 1 - C_i$
  $match = match \times B_i$
}
Return $match$

Function 4: Enc. Word Matching `EncWordMatch`

Input$_1$: Encrypted bits of Word$_1$ (C_i)
Input$_2$: Encrypted bits of Word$_2$ (D_i)
Output: Binary bit "$match$" = 1 if words match, 0 otherwise

$match = 1$
For each bit i {
  $B_i = \overline{C_i \oplus D_i}$
  $match = match \times B_i$
}
Return $match$

Function 5: Homomorphic Email Spam Filter

Input$_1$: Encrypted Email
Input$_2$: Spam Database (DB)
Output: Email Spam/Ham Probability

$prob = 0$
For each encrypted word "i" in the email {
  For each word "j" in the database {
    $match = \texttt{WordMatch}(EmailWord_i \ , \ DBWord_j)$
    $prob \ \ += match \times WordProbability_j$
  }
}
Return *prob*

Function 6: Secure Multiple Keyword Search In Encrypted Files

Input$_1$: Set of encrypted keywords
Input$_2$: Encrypted file
Output: Keywords Found "$Result = 1$", otherwise "0"

$result = 1$
For each encrypted keyword "i" {
  For each encrypted word "j" in the file {
    $match = \texttt{EncWordMatch}(FileWord_j, Keyword_i)$
    $result \ \ += match$
  }
}
Return *result*

In order to increase the performance and efficiency of the spam filter, several optimizations are introduced:

Optimization 1**:** By storing probability numbers in a single polynomial entry (ex. $\eta = 5, \pi = 5x^0$), the other polynomial entries will be unused. This will also lead to the rapid growth of the final result. We decided to store a probability number as binary bits in adjacent polynomial entries (e.g. $\eta = 5 = 101_b, \pi = x^0 + x^2$). By doing so, we will benefit from the unused slots and also when the adjacent slots are added without a carry propagate, values in individual slots will grow much slower than before (grows logarithmically). By having individual polynomial slot values grow logarithmically, we will also have a logarithmic growth in the Ctxt noise as was discussed in Section 3.1.1.

Optimization 2**:** The matching Function 3 was stated naively for simplicity. This is done by matching bits one-by-one to get the matching flag. Another more clever way to do the same task is to notice that since database words are in the clear, we can rearrange database entries in ascending order. By doing so, we can infer consecutive matching bits in adjacent plaintext entries in the database to skip redundant computations. As a simple example, assume the following two 4-bit database entries: 1001 and 1011, both those entries share the left-most two bits "10". Instead of doing 6 multiplication operations to match an input encrypted word with those two entries as in Function 3, we can store partial matching results of the left-most two bits "10" and reduce these multiplication operations to 4 operations. Experimental results for a database of size 10^5 and hash numbers of size 32-bits show that the number of multiplications needed for matching one word across the entire database decrease from $32 \cdot 10^5$ to $16 \cdot 10^5$ which is a factor of 2 reduction in the number of multiplications.

Optimization 3**:** The interesting property of *zero plaintext zero error*, described in Section 3.1.1, can be used for applications where we can correctly decrypt a *binary flag* even when it is totally submerged in noise! For example, if the application in hand needs many multiplication operations to be done to match one entry as in (1), this may lead to the rapid growth of the noise in the Ctxt to the limit that it may not be decrypted correctly. On the other hand, as discussed in Section 3.1.1, in the case of non-matching items, the result will have much less noise. This means that when the resulting flag is "0", it will most probably be decrypted correctly (otherwise, if we get an error in decryption, this most probably means that the resulting flag was a "1").

4.2 Secure Multiple Keyword Search

Another interesting problem is the problem of searching for a set of input encrypted keywords in encrypted files [31]. Consider an application at an airport where an agent can encrypt passenger names and search for them in an encrypted watchlist present in the cloud. This would be crucial to preserve the security of the watchlist without compromising the privacy of the passengers. Another useful security application would be in monitoring encrypted emails for keywords without compromising the privacy of users. This problem is somewhat parallel to the problem of the data classifier discussed in Section 4.1. The only difference is that Function 5 will be replaced by Function 6 which will compute the number of matched keywords in a given file. The computational complexity of this search problem can be decreased if the input keywords are not encrypted (plaintext keywords). In this case, `EncWordMatch` can be replaced by `WordMatch` defined in Function 3, which is computationally less expensive.

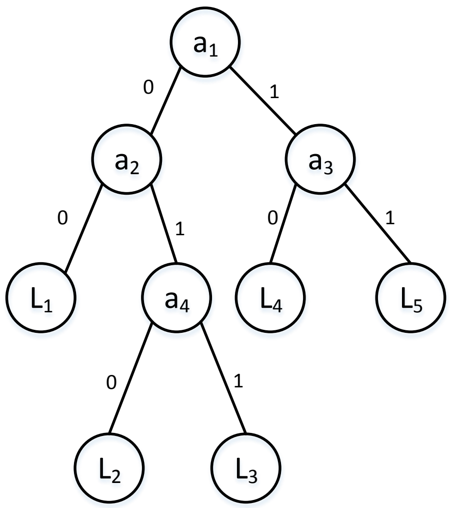

Fig. 1: Binary Decision Tree with nodes a_i and leafs L_i.

4.3 Binary Decision Trees

Binary decision trees are classifiers consisting of interior nodes and leaf nodes. Interior nodes are decision nodes which decide which direction the tree should follow. Leaf nodes are the final tree decision. Binary decision trees are considered as a simplified version of the spam filter described previously, which is considered as a complete decision tree. Fig. 1 shows an example of a binary decision tree with 4 nodes and 5 leafs.

The decision tree in Fig. 1 can be expressed as polynomial equation as in (11). Such a polynomial equation can be efficiently implemented using our HE scheme.

$$\begin{aligned} T(a_1,a_2,a_3,a_4) = {} & a_1(a_3 \cdot L_5 + (1-a_3)\cdot L4) \\ & + (1-a_1)(a_2(a_4 \cdot L_3 + (1-a_4)\cdot L_2) \\ & + (1-a_2)\cdot L_1) \end{aligned} \tag{11}$$

5 Performance Results

Design Environment A summary of the specifications of the system used to implement our work for the purpose of benchmarking is found in Table 2.

5.1 Ctxt Multiplication

Ctxt multiplication is considered the main bottleneck for most of the homomorphic applications. Thus, we next report the performance of the Ctxt multiplication operation on different platforms.

Table 2: Design Environment.

Item	Specification
CPU	Intel Core-i7 4770K
# of CPU Cores	4
# of Threads	8
CPU Frequency	3.5 GHz
Cache Size	8 MB
System Memory	8 GB DDR3
Operating System	Windows 8.1 Ultimate 64-bits
Programming IDE	Visual Studio 2012 Ultimate edition
GPU	NVIDIA GeForce GTX750 Ti
Maxwell Version	GM107
# of CUDA Cores	640
GPU Core Frequency	1020 MHz
GPU Memory	2 GB
GPU L2 Cache	2 MB

Performance using CPU The performance of the Ctxt multiplication in our library compared to the IBM HElib library across different circuit depths, when running on a single CPU core, is shown in Fig. 2. Our library scores speedups up to 10× across all circuit depths.

Performance using GPU To explore parallelization in our work, we partitioned the polynomial operations across GPU cores. The downside of the IBM HElib is that it is not parallelizable. The performance of the Ctxt multiplication in the GPU implementation of our library compared to the IBM HElib library running on a CPU, across different circuit depth, is shown in Fig. 3. Again, our library scores speedups up to 100× across all circuit depths.

Table 3 summarizes the performance results of the complete homomorphic operations for our library compared to the [23, 28] at a circuit depth equal 10. It can be seen from this table that we have a 9.5× speedup for the multiplication operation of our CPU implementation compared to IBM HElib library. By additionally exploring the parallelizable properties that our HE library has, we get another 10× speedup by distributing the HE computations on the GPU cores. This resulted in an overall ≈ 100× speedup for the multiplication operation compared to IBM HElib library and a 3412× compared to [28]. The comparison between the Ctxt size of this work and the Ctxt in the IBM library is shown in Table 4.

Table 3: Performance comparison between our work and IBM HElib. Running time is in seconds.

	Our Work		IBM	GPU Speedup	Work in [28]	GPU Speedup
	CPU	GPU	HElib	over IBM HElib		over [28]
Startup	0.27	0.27	85.3	316×	5	18.5×
Encrypt	0.383	0.043	0.59	13.7×	4.8	111.6×
Decrypt	0.3	0.043	0.39	9×	2.27	52.8×
Add	0.006	0.001	0.002	2×	0.013	13×
Multiply	0.372	0.037	3.6	97.3×	126.25	3412×

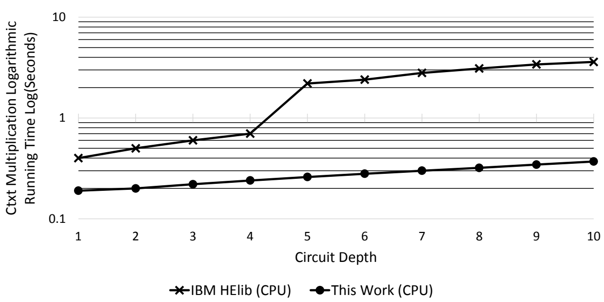

Fig. 2: Ctxt multiplication time of our work compared to the IBM HElib when running on a single CPU core across different circuit depths. The running time is plotted on log scale.

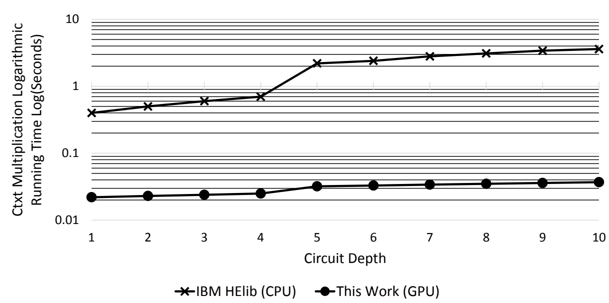

Fig. 3: Ctxt multiplication time of the GPU implementation of our work compared to the IBM HElib running on a single CPU core across different circuit depths. The running time is plotted on log scale.

Table 4: Ctxt Size Comparison.

	This Work	IBM HElib
$SecurityLevel(\lambda)$	80	80
$Depth(L)$	10	10
Width (bits)	$log(q_1) = 31$	301
Poly. Degree [25]	$n > log(q)(\lambda + 110)/7.2$	
n	$n_1 = 1024$	$n_2 = 13981$
$Ctxt$ size (bits)	$4 \cdot n_1 \cdot log^2(q_1) = 3,936,256$	$2 \cdot n_2 \cdot log(q_2) = 6,021,120$

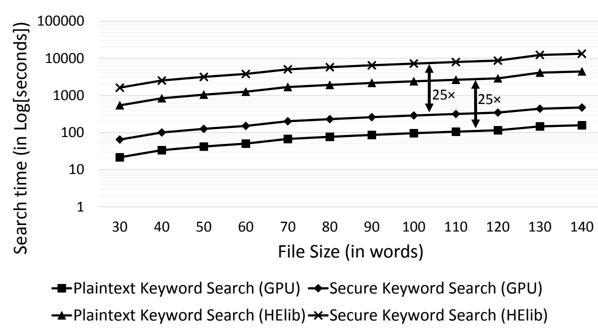

Fig. 4: Secure and plaintext search running time versus different file sizes compared to IBM HElib (Plaintext search means plaintext keyword search in encrypted files).

5.2 Secure Multiple Keyword Search Performance

The performance of the secure and plaintext search engines, described in Section 4.2, compared to IBM HElib versus different file sizes is shown in Fig. 4. We observe a 25× speedup for the secure keyword search on a GPU compared to IBM HElib (and 100× speedup for circuit depth ≥ 5 as indicated in Fig. 3). It is worth mentioning that our implementation is totally scalable and parallelizable. Increasing the number of GPUs inside the server by a factor G, will automatically scale down the running time of the our search engine by the same ratio.

5.3 Binary Decision Tree Performance

The performance of the decision tree depends on the tree structure and the number of nodes and leafs, which will affect our parameter selection and Ctxt operation running times. The decision tree running time depends mainly on the number of multiplications needed. For example, the polynomial equation (11) that describes the tree in Section 4.3 has 8 multiplication operations, each multiplication operation takes 0.037 seconds, which results in a total running time of 0.296 seconds.

6 Conclusion

We described, optimized, and implemented an RLWE-based variant of the HE scheme of [10, 21] which achieves much slower growth of noise, and thus much better parameters than previous HE schemes. We implement three representative applications, namely encrypted spam filters, secure multiple keyword search, and encrypted binary decision trees, using this HE scheme. Compared to the IBM HElib library, our GPU implementation scores a speedup of $100\times$ in Ctxt multiplication, which represents the bottleneck for most HE schemes. Our secure search engine application runs in a few seconds on small to moderate file sizes, and our decision tree computations run in under a second for moderate size decision trees.

Acknowledgments

We want to thank NSERC for financial support.

References

[1] Aguilar-Melchor, C., Fau, S., Fontaine, C., Gogniat, G., and Sirdey, R. Recent Advances in Homomorphic Encryption: A Possible Future for Signal Processing in the Encrypted Domain. vol. 30. 2013, pp. 108–117.

[2] Aguilar-melchor, C., and Gaborite, P. A Lattice-Based Computationally-Efficient Private Information Retrieval Protocol. In *WEWoRC 2007* (2007), pp. 1–13.

[3] Baek, J., Safavi-Naini, R., and Susilo, W. Public Key Encryption with Keyword Search Revisited. In *Computational Science and Its Applications – ICCSA 2008*, O. Gervasi, B. Murgante, A. Laganà, D. Taniar, Y. Mun, and M. Gavrilova, Eds., vol. 5072 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 1249–1259.

[4] Benaloh, J., and Vaikuntanathan, V. Homomorphic Encryption: What, How, and Why . `people.csail.mit.edu/vinodv/Homomorphic-TECHFEST-Final.ppsx`.

[5] Boneh, D., Crescenzo, G. D., Ostrovsky, R., and Persiano, G. Public Key Encryption with Keyword Search. pp. 506–522.

[6] Boneh, D., Gentry, C., Halevi, S., Wang, F., and Wu, D. Private Database Queries Using Somewhat Homomorphic Encryption. In *Applied Cryptography and Network Security*, M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, Eds., vol. 7954 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 102–118.

[7] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (New York, NY, USA, 2012), ITCS '12, pp. 309–325.

[8] Brakerski, Z., and Vaikuntanathan, V. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on* (2011), pp. 97–106.

[9] Brakerski, Z., and Vaikuntanathan, V. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In *Advances in Cryptology – CRYPTO 2011*, P. Rogaway, Ed., vol. 6841 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 505–524.

[10] Brakerski, Z., and Vaikuntanathan, V. Lattice-based FHE As Secure As PKE. In *Proceedings of the 5th Conference on Innovations in Theoretical Computer Science* (New York, NY, USA, 2014), ITCS '14, pp. 1–12.

[11] Brenner, M., Perl, H., and Smith, M. Scarab library , 2011. `hcrypt.com/scarab-library/`.

[12] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., and Steiner, M. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. Garay, Eds., vol. 8042 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 353–373.

[13] Coron, J.-S., Mandal, A., Naccache, D., and Tibouchi, M. Fully Homomorphic Encryption over the Integers with Shorter Public Keys. In *Advances in Cryptology – CRYPTO 2011*, P. Rogaway, Ed., vol. 6841 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 487–504.

[14] Cousins, D., Rohloff, K., Peikert, C., and Schantz, R. An update on SIPHER (Scalable Implementation of Primitives for Homomorphic EncRyption) ; FPGA implementation using Simulink. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on* (2012), pp. 1–5.

[15] Dijk, M., Gentry, C., Halevi, S., and Vaikuntanathan, V. Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed., vol. 6110 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 24–43.

[16] Doroz, Y., Sunar, B., and Hammouri, G. Bandwidth Efficient PIR from NTRU. In *Cryptology ePrint Archive* (2014), pp. 1–12.

[17] Fotiou, N., Trossen, D., Marias, G., Kostopoulos, A., and Polyzos, G. Enhancing information lookup privacy through homomorphic encryption. 2013, pp. 1–11.

[18] Gentry, C. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[19] Gentry, C. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2009), STOC '09, pp. 169–178.

[20] Gentry, C., Halevi, S., and Smart, N. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 850–867.

[21] Gentry, C., Sahai, A., and Waters, B. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. Garay, Eds., vol. 8042 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 75–92.

[22] Graepel, T., Lauter, K., and Naehrig, M. ML Confidential: Machine Learning on Encrypted Data. In *Information Security and Cryptology – ICISC 2012*, T. Kwon, M.-K. Lee, and D. Kwon, Eds., vol. 7839 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 1–21.

[23] HALEVI, S., AND SHOUP, V. Design and Implementation of a Homomorphic-Encryption Library, 2013. `researcher.ibm.com/researcher/files/us-shaih/he-library.pdf`.

[24] HAYES, B. Alice and Bob in Cipherspace. vol. 100. 2012, pp. 362–367.

[25] LINDNER, R., AND PEIKERT, C. Better Key Sizes (and Attacks) for LWE-based Encryption. In *Proceedings of the 11th International Conference on Topics in Cryptology: CT-RSA 2011* (Berlin, Heidelberg, 2011), CT-RSA'11, Springer-Verlag, pp. 319–339.

[26] LYUBASHEVSKY, V., PEIKERT, C., AND REGEV, O. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed., vol. 6110 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 1–23.

[27] MELCHOR, C. A. High-Speed Single-Database PIR Implementation. In *The 8th Privacy Enhancing Technologies Symposium (PETS 2008)* (2008), pp. 1–12.

[28] NAEHRIG, M., LAUTER, K., AND VAIKUNTANATHAN, V. Can Homomorphic Encryption Be Practical?. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2011), CCSW '11, pp. 113–124.

[29] PERL, H., BRENNER, M., AND SMITH, M. Poster: An Implementation of the Fully Homomorphic Smart-vercauteren Crypto-system. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, pp. 837–840.

[30] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, pp. 85–100.

[31] RUSLI, E. M. A Darpa Director on Fully Homomorphic Encryption (or One Way the U.S. Could Collect Data) , 2014. `http://blogs.wsj.com/digits/2014/03/09/a-darpa-director-on-fully-homomorphic-encryption/-or-one-way-the-u-s-could-collect-data/`.

[32] SAHAMI, M., DUMAIS, S., HECKERMAN, D., AND HORVITZ, E. A Bayesian Approach to Filtering Junk E-Mail. In *Learning for Text Categorization: Papers from the 1998 Workshop* (Madison, Wisconsin, 1998), AAAI Technical Report WS-98-05.

[33] WANG, W., HU, Y., CHEN, L., HUANG, X., AND SUNAR, B. Accelerating fully homomorphic encryption using GPU. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on* (2012), pp. 1–5.

[34] WANG, W., HU, Y., CHEN, L., HUANG, X., AND SUNAR, B. Exploring the Feasibility of Fully Homomorphic Encryption. vol. 99. IEEE Computer Society, Los Alamitos, CA, USA, 2013, pp. 1–10.

[35] YASUDA, M., SHIMOYAMA, T., KOGURE, J., YOKOYAMA, K., AND KOSHIBA, T. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2013), CCSW '13, pp. 65–76.

[36] YI, X., KAOSAR, M. G., PAULET, R., AND BERTINO, E. Single-Database Private Information Retrieval from Fully Homomorphic Encryption. vol. 25. May 2013, pp. 1125–1134.