Homomorphic Autocomplete

Gizem S. Çetin[1], Wei Dai[1], Yarkın Doröz[1], and Berk Sunar[1]

Worcester Polytechnic Institute
{gscetin,wdai,ydoroz,sunar}@wpi.edu

Abstract. With the rapid progress in fully homomorpic encryption (FHE) and somewhat homomorphic encryption (SHE) schemes, we are witnessing renewed efforts to revisit privacy preserving protocols. Several works have already appeared in the literature that provide solutions to these problems by employing FHE or SHE techniques. These applications range from cloud computing to computation over confidential patient data to several machine learning problems such as classifying privatized data. One application where privacy is a major concern is web search – a task carried out on a daily basis by billions of users around the world. In this work, we focus on a more surmountable yet essential version of the search problem, i.e. autocomplete. By utilizing a SHE scheme we propose concrete solutions to a homomorphic autocomplete problem. To investigate the real-life viability, we tackle a number of problems in the way towards a practical implementation such as communication and computational efficiency.

Keywords: Homomorphic encryption, privacy preserving applications, oblivious keyword search.

1 Introduction

Fully homomorphic encryption (FHE) has gained increasing attention from cryptographers ever since its first plausible secure construction was introduced by Gentry [17] in 2009. FHE allows one to perform arbitrary computation on encrypted data without the need of a secret key, hence without knowledge of original data. That feature would have invaluable implications for the way we utilize computing services. For instance, FHE is capable of protecting the privacy of sensitive data on cloud computing platforms. We have witnessed an amazing number of improvements in fully and somewhat homomorphic encryption schemes (SHE) over the past few years [18, 12, 5, 7, 19, 3]. In [20] Gentry, Halevi and Smart (GHS) proposed the first homomorphic evaluation of a complex circuit, i.e. a full AES block. The implementation makes use of batching [34, 35], key switching [4] and modulus switching techniques to efficiently evaluate a leveled circuit. In 2012 Halevi (and later Shoup) published the HElib [24], a C++ library for HE that is based on Brakerski-Gentry-Vaikuntanathan (BGV) cryptosystem [3]. In [29] a leveled NTRU [25, 37] based FHE scheme was introduced by López-Alt, Tromer and Vaikuntanathan (LTV), featuring much slower growth of noise

during homomorphic computation. Doröz, Hu and Sunar (DHS) [13, 14] used an LTV SHE variant to evaluate AES using windowing in 12 seconds. In early 2015, Gentry, Smart, Halevi (GHS) [21] published significantly improved AES runtime results with a 2 seconds amortized per block runtime. More recently, Ducas and Micciancio [16] presented the FHEW scheme that achieves bootstrapping in half a second for HElib on a common PC.

Applications. These impressive advances motivated researchers to investigate how to best put these new homomorphic evaluation tools to use in privatizing applications. For instance, in [26] Lauter et al. consider the problems of evaluating averages, standard deviations, and logistical regressions which provide basic tools for a number of real-world applications in the medical, financial, and the advertising domains. The same work also presents a proof-of-concept Magma implementation of a SHE for the basic operations. The SHE scheme is based on the ring learning with errors (RLWE) problem proposed earlier by Brakerski and Vaikuntanathan in [6]. Later in [27], Lauter et al. show that it is possible to implement genomic data computation algorithms where the patients' data are encrypted to preserve their privacy. They encrypt all the genomic data in the database and are able to implement and provide performance numbers for Pearson Goodness-of-Fit test, the D' and r^2-measures of linkage disequilibrium, the Estimation Maximization (EM) algorithm for haplotyping, and the Cochran-Armitage Test for Trend. The authors used a leveled SHE scheme which is a modified version of [30] where they get rid of the costly relinearization operation. In [2] Bos et al. show how to privately perform predictive analysis on encrypted medical data. They present an implementation of a prediction service running in the cloud. The cloud server takes private encrypted health data as input and returns the probability of cardiovascular disease in encrypted form. The authors use the SHE implementation of [1] to provide timing results. Graepel et al. in [22] demonstrate that it is possible to execute machine learning algorithms in a service while protecting the confidentiality of the training and test data. The authors design confidential machine learning algorithms using leveled homomorphic encryption. More specifically they implement low-degree polynomial versions of Linear Means Classifier and Fisher's Linear Discriminant Classifier on the Wisconsin Breast Cancer Data set. In [15], Doröz et al. use an NTRU based SHE scheme to construct a bandwidth efficient private information retrieval (PIR) scheme. Due to the multiplicative evaluation capabilities of the SHE, the query and response sizes are significantly reduced compared to earlier PIR constructions. The PIR is generic and therefore any SHE which supports a few multiplicative levels (and many additions) could be used to implement a PIR. The authors also give a leveled and batched reference implementation of their PIR construction including performance figures. Cheon et al. [9] present a method along with implementation results to compute encrypted dynamic programming algorithms such as Hamming distance, edit distance, and the Smith-Waterman algorithm on genomic data encrypted using a SHE scheme. The authors design low depth circuits to compute the distances between two genomic strings using BGV-type leveled SHE schemes. Çetin et al. [8] analyzed the

complexity and provided implementation results for privately sorting an array of integers. The scalability of various sorting algorithms is studied using an LTV [29] variant also proposing new depth optimized sorting algorithms.

All of the aforementioned works develop versatile tools that take us closer to developing practical privacy preserving applications. Although proposed as the prime example of blind computation by Gentry in his very first FHE construction [17], the privatized web search problem, where encrypted keywords are submitted to a server that blindly computes and returns the search result, has not been investigated yet with the new homomorphic tools developed in the last few years. While there are still quite a number of practical challenges that need to be tackled before the search problem can be solved in a practical manner. Here we take the first step by studying a similar yet more tractable problem, i.e. the autocomplete feature. From a privatization point of view, autocomplete provides the perfect micro example of a search. In both applications, we wish the server to privately match keywords in a dictionary and accordingly return the matching results. A search engine compiles the results page from a vast list of precomputed and presorted results optimized for extremely efficient retrieval. Instead, in autocomplete the list is significantly smaller and therefore does not need to be structured and optimized as much. This provides us an opportunity to study the search problem in a simpler setting. Finally we note that, autocomplete is a more generic tool and is also used in other applications such as such as form fills in web browsers, and in e-mail clients etc.

Our Contribution. In this work, we

- first model the autocomplete problem in the client/server setting. Whenever the client starts typing a keyword, the server performs a lookup in the database, predicts the rest of the word and provides a number of popular keywords to the user;
- we convert this model into a homomorphic circuit, so that it can be evaluated using encrypted input/outputs. In this scenario, when a user starts typing a keyword, it is first encrypted under an SHE scheme with the user's own public key on the client side, then the ciphertexts are sent to the server. The server evaluates the autocomplete circuit and returns the predicted words still in encrypted form to the user;
- we consider a realistic scenario where the size of the lookup table, i.e. the database can be fairly large, hence we provide a batched scheme suitable for a real-world application;
- we give a detailed analysis on the noise growth during the evaluation of our circuit depending on different application variables. Furthermore, we explain the parameter selection process with respect to both noise management and the cryptographic security.
- Finally, we provide implementation results of the proposed homomorphic circuit using a C++ library with an SHE instantiation based on LTV. Additionally, given the computation and bandwidth complexity of the SHE scheme, we also implemented a highly parallel, multi-threaded version with

GPU support to accelerate the SHE evaluations. We provide run-time results for the proposed methods for both CPU and GPU implementations.

2 Background

In this work, we use a single-key SHE construction which is a modified version of DHS in [14]. The DHS scheme was derived from the multi-key FHE scheme LTV in [29] and LTV construction uses a variant of NTRU Encryption proposed by Stehlé and Steinfeld in [36] as a base point. (see Appendix)

2.1 The DHS Library

DHS is a customized single-key version of the LTV scheme that was proposed in [14] by Doröz, Hu and Sunar. The source code is available in C++ and the library is implemented using NTL [32] with GMP [23] support. The library contains some special customizations that improve the efficiency in both run-time and the memory requirements. The library can perform 5 main operations as well as a packing method; KeyGen, Encryption, Decryption, Modulus Switch and Relinearization and Batch. Due to the fact that our finalized circuit, that will be defined later in Section3, is shallow enough to execute with a SHE scheme, we will not use Relinearization. By disregarding relinearization, we need to update our decryption key according to the circuit to be evaluated. The details of the DHS customizations, that will be used in our implementation, are as follows:

- **Encryption.** In the previous section 7.1, we defined encryption and decryption primitives as bit operations. But in the library, we can encrypt a message m from a larger message domain $\mathbb{Z}_p$ for the i^{th} level, using $c^{(i)} = h^{(i)}s+pe+m$ as the encryption function. Similarly decryption works as $\mathsf{Decrypt}\left(c^{(i)}\right) = c^{(i)}f^{(i)} \mod p = m$. With word size-$p$ message domains, all homomorphic properties are preserved and we gain the ability to homomorphically multiply and add integers via simple ciphertext multiplication and additions. As stated earlier, we will use a function of the secret key on the decryption step for our SHE implementation. $\mathsf{Decrypt}\left(c^{(i)}\right) = c^{(i)}\left(f^{(i)}\right)^{\ell}$ and the exponent of the secret key ℓ depends on the number of multiplications in the circuit.
- **Batching.** The new domain is defined as $\mathrm{R}_q = \mathbb{Z}_q[x]/\Psi_m(x)$ where $\Psi_m(x)$ is the m^{th} cyclotomic polynomial. The specially selected cyclotomic polynomial is used to batch multiple messages into the same polynomial for parallel evaluations as proposed by Smart and Vercauteren [33, 34] (also see [14]). When $\mathbb{Z}_p$ is the plaintext space, the polynomial $\Psi_m(x)$ can be factorized over $\mathbb{F}_p$ (if we have a prime p), into equal degree polynomials $F_i(x)$. The degree of these factors will be the smallest integer t that satisfies $m|(p^t-1)$. The degree n of $\Psi_m(x)$ is equal to the Euler Totient function of m, i.e. $n = \varphi(m)$. This means that, we have $\mathrm{S} = n/t$ factors and each one of them

defines a separate message slot. As a result, S messages can be embedded into a single polynomial using the Chinese Remainder Theorem (CRT).

– **Modulus Switching.** Previously, we defined q_is as a decreasing sequence of primes for each level i. In this construction, q_i is created as the product of a number of primes for each level. It starts with setting the last level q, q_d to a selected prime ρ, i.e. $q_d = \rho$, then for the rest of the sequence $i = d-1, \cdots, 0$, it generates the new q_i as, $q_i = q_{i+1}\rho_i$. The value ρ_i is a prime number that cuts ($\kappa = \log_{\rho_i}$)-bits of noise in each level.

2.2 CUDA GPU Support

We adopt the cuHE library [11] to implement our design on CUDA GPUs. The library features fast polynomial ring operations and is designed for a homomorphic evaluation of a leveled circuit. A ciphertext on GPU can be represented in three ways: straightforward representation of a polynomial with very large coefficients, a set of polynomials with small coefficients converted with the CRT or a set of vectors achieved with number-theoretic transform. The latter two are essential for efficient computation. They adopt a different method than DHS to generate coefficient moduli, featuring efficient modulus switching. We trimmed the cuHE library to better fit our need, such as removing relinearization.

3 Our Autocomplete Scheme

In autocomplete schemes, expensive computations and massive user-server bandwidth requirements stays a challenge towards a real time SHE based applications. In this section, we present a homomorphic autocomplete algorithm that can handle practical scenarios. Here, we provide optimizations and propose a lightweight algorithm, to push our design closer to a real-life application. In the following we first explain an autocomplete scheme briefly and later give details of the homomorphic autocomplete.

A server holds a database of which each entry contains a sequence of words (i.e. sorted by their searching frequencies) starting with a specific combination of letters or characters. We call those leading characters the header of an entry, as shown in Table 1. A client would type a few characters (e.g. “a”) and would expect suggestions of words starting with those characters, i.e. the entry with a matching header (e.g. “amazon”, “abc”, etc.) from the server. In a regular autocomplete scheme the client has the reply from the server while the server learning the typed information in the mean time. However, if the client does not want to reveal his intention, i.e. “a”, to the server he has to use homomorphic autocomplete scheme to protect his information. The algorithm we build with homomorphic encryption works as follows: the client encrypts “a” and sends ciphertexts to the server; the server process the ciphertexts and return the corresponding entry contents in encrypted form to the client; the client decrypts and gets entry contents in plaintexts. In that way, the client is content to have the

server facing a computationally difficult problem (decrypting without decryption key) to extract any significant knowledge of his intention, which is identical to a computational private information retrieval (cPIR) problem [10]. Based on those definitions we are now able to provide a general model to the problem and explain our solutions.

Headers	**Words**
a	amazon, abc, abc news ...
b	bank of america, best buy ...
⋮	⋮
z	zillow, zappos, zipcar ...
0	0 to 100, 0 divided by 0 ...
1	1800flowers, 1800contacts ...
2	2048, 21 day fix ...
⋮	⋮

1100001	97	109	97	122	111	...
1100010	98	97	110	107	32	...
⋮						
1111010	122	106	108	108	111	...
0110000	48	32	116	32	49	...
0110001	49	56	48	48	102	...
0110010	50	48	52	56	0	...
⋮			⋮			

Table 1. An example database where headers have a single character and the corresponding words are ordered according to their search ranks. Their ASCII representation on the right.

3.1 Setup

We first setup an LTV instance for our application as the base homomorphic encryption module with the following preliminaries:

Plaintext space	: $\mathcal{P} = \mathbb{Z}_{p}$[1]
Depth of the circuit to be evaluated	: d
Ciphertext space	: $\mathcal{C} = \mathbb{Z}_{q_i}[x]/\langle \Psi_m(x) \rangle, i \in \mathbb{Z}_{\mathrm{d}+1}$
Number of message slots	: S
Batching domain	: $\mathcal{B} : \mathbb{Z}_p^S \rightarrow \mathbb{Z}_p[x]/\langle \Psi_m(x) \rangle$

This generic setup will be used throughout this section for building our homomorphic circuits. Later in Section 4, we will give detailed information about selecting the parameters (p, q, m, S) with respect to the depth (d) of the circuit. Before constructing our homomorphic circuit, we also need to define some

[1] p does not necessarily have to be a prime number for encryption/decryption, but it must be a prime for batching.

terminology and set the following preliminaries for the design:

Bit size of the user input/headers	:	L
Number of rows in the database	:	E
Number of $\log p$-bit chunks to be returned from the word list	:	R
User input as an array of bits	:	$\mathrm{U_L}$
Database headers as a matrix of bits	:	$\mathrm{H_{E\times L}}$
Database words as a matrix of $\log p$-bit chunks	:	$\mathrm{W_{E\times R}}$

The parameters, $\mathrm{L}, \mathrm{E}, \mathrm{R}$, determines the number of input/output data that is transfered between the server and the client. Therefore they have an effect on both the circuit size and the bandwidth, thus they are significantly important for the runtime of the application. We will design our circuits for generic cases and later in Section 5, we will choose different values and observe their effect on the LTV setup and consequently on execution time.

Database elements are accessed using an array notation, hence we have $\mathrm{H}[i][j] \in \mathbb{Z}_2,\ \forall i \in \mathbb{Z}_\mathrm{E},\ \forall j \in \mathbb{Z}_\mathrm{L}$ for database headers and $\mathrm{W}[i][j] \in \mathbb{Z}_p,\ \forall i \in \mathbb{Z}_\mathrm{E},\ \forall j \in \mathbb{Z}_\mathrm{R}$ for the word list. For example, for the database in Table 1, $\mathrm{H}[0][0] = 1$ which is the first bit of the first row, $\mathrm{H}[0][1] = 0$ which is the second bit of the first row, and so on. Similarly, $\mathrm{W}[0][0] = 97$ which is the first character of the word list on the first row. User input is hold in the same structure, i.e. we have $\mathrm{U}[i] \in \mathbb{Z}_2, \forall i \in \mathbb{Z}_\mathrm{L}$ and for example for a user input 'b' = 1010110 in ASCII representation, we will have $\mathrm{U}[0] = 1$, $\mathrm{U}[1] = 0$. Similarly, whenever we have a vector $\boldsymbol{v} = \langle v_0, v_1, \cdots v_{k-1} \rangle$ we access its elements using the notation, $\boldsymbol{v}[i] = v_i$ for any i. If we have an array of such vectors, $A = [\boldsymbol{v_0}, \boldsymbol{v_1}, \cdots, \boldsymbol{v_l}]$, we will use $A[i] = \boldsymbol{v_i}$ and $A[i][j] = \boldsymbol{v_i}[j]$ for some i and j.

3.2 The Autocomplete Circuit

The autocomplete problem is identical to PIR problem, thus we can contract the same circuit to evaluate the autocomplete. The algorithm will be consisting of a number of comparisons for the lookup and a multiplexer to generate the end result is given in Algorithm 1.

Algorithm 1 Auto-Complete

```
1: function AUTOCOMPLETE (U)
2:     for i ← 0 to E-1 do
3:         if U equals to H[i] then
4:             return W[i]
5:         end if
6:     end for
7: end function
```

The user input U is encrypted in the algorithm, thus to perform the if statement on line 3 of Algorithm 1, we need a sub-routine that can compare encrypted

inputs, i.e. a homomorphic circuit. We build a boolean comparison circuit using a binary tree of XNOR gates, as can be seen in the first line of Algorithm 2. Since we need to access each bit of the inputs we encrypt each of the user input bits separately. After performing the comparison step, the intermediate results will still be in an encrypted form, the server will still be blind. Hence, it cannot perform any branch operation. Thus, we need to implement a multiplexer circuit for the branching step. The overall circuit is summarized in Algorithm 2.

Algorithm 2 Auto-Complete Circuit

```
1: function COMPARE(U, H[i])
2:     output ← 1
3:     for j ← 0 to L-1 do
4:         output ← output ∗ (U[j] + H[i][j] − 1  mod 2)
5:     end for
6:     return output
7: end function
8: function AUTOCOMPLETE (U)
9:     output ← 0
10:    for i ← 0 to E-1 do
11:        output ← output + (COMPARE (U, H[i]) ∗ W[i])
12:    end for
13:    return output
14: end function
```

Even though the circuit we defined in Algorithm 2 is shallow enough to operate on a homomorphic domain, considering a real world scenario where we have a really large database in which we perform the lookup, we cannot expect a practical application unless we have parallel evaluations. Batching, as described in Section 2.1, is one the most useful techniques in order to reduce the cost of homomorphic applications. Placing multiple data into message slots and packing them into a single polynomial using CRT allows us to do parallel addition/multiplications on each message slot. In our scheme, since we have 2-dimensional arrays to represent the database, we have two options, packing row-wise or column-wise. Packing database matrix row-wise means that we are batching bits of an header, therefore we also need to batch the user input by placing each bit into a message slot. But, notice that our comparison circuit in Algorithm 2 takes the product of bitwise comparisons, i.e. XNOR operations. Hence, even if we perform the bitwise XNORs in parallel, we will need to apply multiplication across different message slots in order to compute the product. Permuting the message slots by rotating and manipulating the packed polynomial is possible, yet inconveniently expensive for a real-time application. Therefore, the latter batching method, i.e. packing the database elements column-wise is a better approach for our scheme. Since we only have S message slots, we need $\lambda = \lceil \frac{\mathrm{E}}{\mathrm{S}} \rceil$ packed vectors per column in order to cover every row of the database. At the end, we will have a matrix of vectors $[\boldsymbol{h}]_{\lambda \times \mathrm{L}}$ for headers and $[\boldsymbol{w}]_{\lambda \times \mathrm{R}}$

for word lists. How we construct the batched database elements can be seen in Equation 1 and Equation 2.

$$\begin{aligned}
\boldsymbol{h}[i][j] &= \langle \mathrm{H}[iS+0][j], \mathrm{H}[iS+1][j], \cdots, \mathrm{H}[iS+S-1][j] \rangle \\
h[i][j] &= \textsc{Batch}(\boldsymbol{h}[i][j]), \\
&\quad \boldsymbol{h}[i][j] \in \mathrm{Domain}(\mathcal{B}),\ h[i][j] \in \mathrm{Image}(\mathcal{B}),\ \forall i \in \mathbb{Z}_\lambda,\ \forall j \in \mathbb{Z}_\mathrm{L} \qquad (1) \\
\boldsymbol{w}[i][j] &= \langle \mathrm{W}[iS+0][j], \mathrm{W}[iS+1][j], \cdots, \mathrm{W}[iS+S-1][j] \rangle \\
w[i][j] &= \textsc{Batch}(\boldsymbol{w}[i][j]), \\
&\quad \boldsymbol{w}[i][j] \in \mathrm{Domain}(\mathcal{B}),\ w[i][j] \in \mathrm{Image}(\mathcal{B}),\ \forall i \in \mathbb{Z}_\lambda,\ \forall j \in \mathbb{Z}_\mathrm{R} \qquad (2)
\end{aligned}$$

In case of client side message batching, we want to compare an arbitrary user input bit $\mathrm{U}[i]$ with i^{th} bit of every header. Therefore, we place the same bit $\mathrm{U}[i]$ in each message slot of a single vector, so that comparison can be performed in parallel. When we batch a vector consisting of the same elements, the mapping will be to a constant polynomial with the corresponding element being the constant term. This means that, we will not spend any time for batching in the runtime and this is the perquisite of the column-wise packing method. The operations in the user side are

$$\begin{aligned}
\boldsymbol{u}[i] &= \langle \mathrm{U}[i], \mathrm{U}[i], \cdots, \mathrm{U}[i] \rangle \\
u[i] &= \textsc{Batch}(\boldsymbol{u}[i]) = U[i] \quad c[i] = \textsc{Encrypt}(u[i]), \\
&\quad \boldsymbol{u}[i] \in \mathrm{Domain}(\mathcal{B}),\ u[i] \in \mathrm{Image}(\mathcal{B}),\ c[i] \in \mathcal{C},\ \forall i \in \mathbb{Z}_\mathrm{L} \qquad (3)
\end{aligned}$$

After retrieving the encrypted user bits $c[i]$, the server can follow the steps of the homomorphic circuit in Algorithm 2. First step will be the comparison as in Equation 4. After this step, the server will have an array of ciphertext polynomials r which holds the comparison results in distinct message slots for all header entries. And we know that there is only one entry in the database that is equal to user input, hence when decrypted only one message slot will have 1 or $p-1$ and the rest will be 0. The reason we may have $p-1$ is due to the fact that in the plaintext domain we may use a prime p larger than 2. There are three possible outcomes for bitwise comparison as shown in Table 5. (See Appendix) Focusing on the particular case where the $\mathrm{H}[i]$ equals to U, for some $i \in \mathbb{Z}_\mathrm{E}$, note that whenever there is a bit 0 in the user input U, there will be a (-1) in the product. Let the number of zero bits in the user input be ω, then we will have $\prod_{j=0}^{\mathrm{L}-1} (\mathrm{H}[i][j] + \mathrm{U}[j] - 1) = (-1)^{\omega} (1)^{\mathrm{L}-\omega}$. If there are odd number of zeros in the user input, then after the decryption this result must be multiplied by -1, otherwise it will have the value $-1 = p-1 \in \mathbb{Z}_p$. For other cases where $\mathrm{H}[i] \neq \mathrm{U}$, there will be at least one zero in the product vanishing the "-1"s as well as "1"s, thus the output will be zero nonetheless.

$$r[i] = \prod_{j=0}^{\mathrm{L}-1} \left(h[i][j] + u[j] - 1 \right),\ r[i] \in \mathcal{C},\ \forall i \in \mathbb{Z}_\lambda \qquad (4)$$

Now that we know r consists a single ∓ 1 in a particular message slot, i.e. the one that corresponds to a single row $i \in \mathbb{Z}_\mathrm{E}$ of the database, where $\mathrm{H}[i] = \mathrm{U}$,

we need to AND it with the word lists. Then we can take the cumulative sum of the rows. Since only one of them will hold a value different than 0 the sum will evaluate a row lookup.

The server sends R encrypted word chunks i.e. for each column j of the word list, it sends a $d[j]$ to the user:

$$d[j] = \sum_{i=0}^{\lambda-1} \left(r[i] * w[i][j]\right), \ d[j] \in \mathcal{C}, \ \forall j \in \mathbb{Z}_{\mathrm{R}} \tag{5}$$

Let $k \in \mathbb{Z}_{\mathrm{S}}$, be the offset of the row i in the packed vector, again for the case $\mathrm{H}[i] = \mathrm{U}$. This means that each $d[j]$ will have $(-1)^{\omega}\mathrm{W}[i][j]$ in the k^{th} message slot and it will be the only non-zero element of the output vector unless the word list is not empty, for the corresponding header. Otherwise all outputs/message slots will be zero and this can be checked by seeking a non-zero value in the first vector. The operations for the decryption process in the client side can be seen in Equation 6 and the final output, i.e. the autocompleted word list can be seen in Equation 7.

$$\begin{aligned} v[j] &= \textsc{Decrypt}(d[j]), \\ \boldsymbol{v}[j] &= \textsc{Batch}^{-1}(v[j]), \\ \boldsymbol{v}[j] &= \langle 0, 0, \cdots, \mp\mathrm{W}[i][j], \cdots, 0, 0\rangle \\ & \quad v[j] \in \mathrm{Image}(\mathcal{B}), \ \boldsymbol{v}[j] \in \mathrm{Domain}(\mathcal{B}), \ \forall j \in \mathbb{Z}_{\mathrm{R}} \\ & \quad \exists i \in \mathbb{Z}_{\mathrm{E}}, \text{ s.t. } \mathrm{U} = \mathrm{H}[i]. \end{aligned} \tag{6}$$

$$\mathrm{V} = \begin{cases} [(-1)^{\omega}\, \boldsymbol{v}[0][k], \cdots, (-1)^{\omega}\, \boldsymbol{v}[\mathrm{R}-1][k]] & \text{if } \exists k \in \mathbb{Z}_{\mathrm{S}} \text{ s.t. } \boldsymbol{v}[0][k] \neq 0 \\ [\emptyset] & \text{otherwise.} \end{cases} \tag{7}$$

Notice that the multiplicative depth of the scheme comes from the comparison operation (Equation 4), where we compare bits of the user input to database entries, because it is the only operation that involves ciphertext-ciphertext multiplication. Even though Equation 6 also has multiplications, it is only between a ciphertext and a plaintext which is a constant multiplication. For further optimization, we will try to drop this equality check in the next section by switching from using ASCII representation to a special encoding.

3.3 A Simpler Circuit

The circuit presented in the previous section is consisting a comparison tree and a multiplexer. This comparison tree significantly decreases the performance. Although the multiplication between the ciphertexts are expensive, it is not the main reason that decreases the performance. The algorithm consist a lot of multiplications between ciphertext and plaintexts in the multiplexer. The overhead of a few expensive operations in the comparison tree is only a small portion of the whole computational burden. The comparison gives the circuit a depth along with which the noise growth needs to be handled (otherwise,

decryption will fail). Also we abandoned relinearization, because it would be impractical to require clients sending huge evaluation keys to the server. We choose to let the client perform decryption with a power of the secret key, e.g. f^{24}, consequently this significantly increase the coefficient moduli q_i's.

Autocomplete without Comparison With a small assumption on client's side, provided that there is a unique index in the database for each possible input combinations, an abbreviated circuit solves the problem. For example, the database has 26 entries with headers: "a", "b",..., "z"; the entry with header "p" has index "15". The client, acknowledge of indexing rules, is then able to send an encrypted index to the server. With one-hot encoding, the encrypted index can be used as input to the multiplexer. Assume that the client intents to retrieve the entry with index $\alpha \in \mathbb{Z}_E$. Create a matrix of bits R_E such that

$$\forall i \in \mathbb{Z}_E,\ R[i] = \begin{cases} 1 & \text{if } i = \alpha \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Apply batching and encrypt batched vectors to ciphertexts:

$$r[i] = \prod_{j=0}^{\mathrm{L}-1} \left(\textsc{Encrypt}(\textsc{Batch}(\langle R[i \cdot S], R[i \cdot S + 1], \ldots, R[i \cdot S + S - 1]\rangle))\right),$$

where $r[i] \in \mathcal{C},\ \forall i \in \mathbb{Z}_\lambda$. The client sends those ciphertexts to the server. The process follows the previous circuit from Equation 5 to enter the multiplexer. The abbreviated circuit consists of multiplications only between ciphertext and plaintext. More importantly, we work with much smaller q_i's, of which an detailed analysis is provided in the next section.

4 Coping with Noise

In this section, we explain our method to manage the growing noise along the homomorphic evaluation of levels of the circuit. Our goal is to minimize computational burden by minimizing ciphertext size, i.e. degree n and coefficient moduli q_i's, so that correct decryption is ensured with very high probability. In our homomorphic encryption scheme, the decryption key is $f = pf' + 1$ and the encryption key is $h = pgf^{-1}$, where $g, f' \in \chi_B$ and p denotes the message space.

Autocomplete with Comparison The comparison step has multiple levels. Therefore noise in ciphertexts grows rapidly. Recall the construction. Let L denote the number of input bits, and let $d = \lceil \log_2 L \rceil$ denote the depth of circuit given L batched plaintexts $M_j \in R_p$ where $j \in \mathbb{Z}_L$. The plaintexts are encrypted as $C_j^{(0)} = \mathsf{Enc}(M_j) = hs_j + pe_j + M_j$, where s_j and e_j are sampled from χ_B.

To estimate the noise growth, we track operations in each level and denote the norm of decrypted ciphertexts entering the i-th level as B_i. Before entering the multiplicative circuit, there is bit-wise comparisons of inputs C_j and batched

dictionary headers H_j: $C'_j = C_j + H_j - 1 = hs + pe + M + H - 1$, where $j \in \mathbb{Z}_L$ and $H_j \in \chi_p$. Those are the ciphertexts entering the 0-th level of circuit (for simplicity assuming $c = C'_0 = C'_1 = \cdots = C'_{L-1}$), then

$$B_0 \leq \|cf\|_\infty \leq n[(B^2 + 2B)p^2 + (B^2 + 2B + 2)p + 1].$$

Then in each level, we perform multiplication on ciphertexts. And remember, since we don't have relinearization in our scheme, to decrypt a ciphertext in the i-th level (c_i), we perform $\mathsf{Dec}(c_i) = c_i \cdot f^{2^i} \mod p$. Moreover, after each level of circuit, a modular switching is performed on ciphertexts to reduce the noise growth. As proposed by Doröz et al. in [14], assume $q_{i+1}/q_i \approx \kappa$, $\forall i \in \mathbb{Z}_d$. A decrypted ciphertext entering the i-th level is presented as:

$$c^{2^i} f^{2^i} = (\ldots((c^2\kappa + \epsilon_1)^2\kappa + \epsilon_2)^2 \ldots \kappa + \epsilon_i) f^{2^i},$$

where $\epsilon_1, \epsilon_2, \ldots, \epsilon_i$ are parity fixing values in χ_p. This yields

$$B_i \leq \|c^{2^i} f^{2^i}\|_\infty = \|(c^{2^{i-1}} f^{2^{i-1}})^2\kappa + \epsilon_i f^{2^i}\|_\infty \leq \kappa n B_{i-1}^2 + pn^{2^i}(pB+1)^{2^i}.$$

If we set $1/\kappa = \varepsilon(nB^2) + pn(pB+1)$ where $\varepsilon > 1$ is a small constant, the growth of the norm is nearly constant over the levels of evaluation.

At last, the multiplication tree yields a ciphertext as output of comparison. However, so far we only focus on a single batched output ciphertext out of a total number of λ. Taking the number of batched results λ into consideration, we have $y = y_0 = y_1 = \cdots = y_{\lambda-1}$ and $B_{d-1} \geq \|y\|_\infty$. Given the suggested words (batched in polynomials) $w = w_0 = w_1 = \cdots = w_{\lambda-1} \in R_p$, the final step before decryption computes r as: $r = \sum_{i=0}^{\lambda-1} y_i w_i = \lambda y w$.

For a general multiplication tree with d levels, if $q_d/2 > \|rf^{2^d}\|_\infty$ is satisfied, there will be no wraparound on the ciphertext coefficients during decryption. Hence the scheme will be correct. To be precise, $y = c^L$ and $2^{d-1} \leq L < 2^d$. $q_d/2$ would only need to be larger than the upper bound of norm $\|rf^L\|_\infty$. Let $[L_0, L_1, \ldots, L_{d-1}]$ be the bitwise expression of L, such that $L = \sum_{i=0}^{d-1} L_i 2^i$ and $L_i \in \mathbb{Z}_2$ $\forall i \in \mathbb{Z}_d$ and $L_{d-1} = 1$. Then we have

$$\begin{aligned}\|rf^L\|_\infty &= \|\lambda w c^L f^L\|_\infty \\ &\leq \lambda pn \prod_{i=0}^{d-1} \|c^{2^i} f^{2^i}\|_\infty^{L_i} \cdot \prod_{i=0}^{d-2} n^{L_i} \leq \lambda pn \prod_{i=0}^{d-1} B_i^{L_i} \cdot \prod_{i=0}^{d-2} n^{L_i}.\end{aligned}$$

In conclusion, to ensure the correctness of our scheme, the following condition must be satisfied:

$$q_d/2 > \lambda pn^{2^d - L + 1}(pB+1)^{2^d - L} \prod_{i=0}^{d-1} B_i^{L_i} \cdot \prod_{i=0}^{d-2} n^{L_i}.$$

Note that this is the analysis of the worst case. The scheme will work with high probability even with a smaller q_d.

Autocomplete without Comparison This construction is derived from the previous one after we computed comparison results (y). The circuit is as simple as $r = \sum_{i=0}^{\lambda-1} y_i w_i = \lambda y w$. And to decrypt r we compute rf. Note that y is the index encoded and encrypted by user, hence, $y = hs + pe + m$ where $m \in R_p$. We can estimate norm of rf in the worst case as:

$$\|rf\|_\infty = \|\lambda w(hs + pe + m)\|_\infty \leq \lambda n^2 p^2 (pB^2 + pB + B^2 + B + 1).$$

In conclusion, to ensure correct decryption, the following condition must stand in the worst case $q_d/2 > \lambda n^2 p^2 (pB^2 + pB + B^2 + B + 1)$.

Average Case Behavior However, a smaller q_d results in less computational burden, of which we would like to take advantage. When viewed as a distribution, the norm of polynomial multiplication $\|uv\|_\infty$ grows much more slowly. And the probability of the norm reaching the worst case is exponentially small. Doröz et al. explained in [14] that $\|uv\|_\infty \leq \sqrt{n} \cdot \|u\|_\infty \cdot \|v\|_\infty$ is a sufficiently good approximation of the expected norm. We obtain new formulas by substituting n with $\sqrt{n}$. The construction with comparison decrypts correctly if (set $1/\kappa = \sqrt{n}B^2 + p\sqrt{n}(pB + 1)$ to stabilize the norm growth):

$$\begin{aligned} q_d/2 &> \lambda p \sqrt{n} \prod_{i=0}^{d-1} B_i^{L_i} \cdot \prod_{i=0}^{d-2} \sqrt{n}^{L_i} \\ B_i &\leq \kappa \sqrt{n} B_{i-1}^2 + p\sqrt{n}^{2^i} (pB + 1)^{2^i} \\ B_0 &\leq \sqrt{n}[(B^2 + 2B)p^2 + (B^2 + 2B + 2)p + 1]. \end{aligned}$$

The construction decrypts correctly if $q_d/2 > \lambda n p^2 (pB^2 + pB + B^2 + B + 1)$.

5 Parameter Selection

We provide the selected parameters for polynomial dimension and coefficient size in Table 2. In the parameter selection process, we use the noise analysis in Section 4 to determine the coefficient size $\log(q)$ and the cutting size $1/\kappa$ for different levels of circuit evaluations. In evaluation of the polynomial sizes we based our selection of parameter on the work by Lepoint and Naehrig [28]. Their work is a revisit of a previous work by van de Pol and Smart [31] which demonstrates that larger lattice dimensions can achieve the same security level with different Hermite factors. Basically, their claim is that the assumption of choosing a secure Hermite factor $\delta_\mathcal{B}$ for all dimensional lattices with basis $\mathcal{B}$ is not true. In the parameter selection process, the Hermite factor should be chosen according to the hardness of lattice basis reduction in higher dimensions. Specifically, the Hermite factor value gets higher for higher lattice basis in order to achieve same security level. In our work, we used the Hermite factor value evaluated by Lepoint and Naehrig which is a larger value then the one used in the earlier works, i.e. $\delta = 1.0064$.

As explained in Doröz et al. [14], the LTV scheme has a lattice dimension of $2n$ for a polynomial degree n. For polynomial degree n being equal to 8190, 16384 and 21504, the lattice dimensions are roughly 15000, 30000 and 40000 and the Hermite factors δ_B are equal to 1.00826, 1.00846 and 1.00851 respectively for 80-bit security. In our case we have the Hermite factors as 1.0037, 1.0061 and 1.0083 for degrees 8190, 16384 and 21504 respectively. The evaluated Hermite factors gives us more than 80-bit of security in all the cases.

(p, n)	With Comparison			Without Comparison		
	(L, E, λ)	$(\log q_d, \log 1/\kappa)$		(E, λ)	$\log q_d$	
		Worst	Average		Worst	Average
(2, 8190)	(8, 52, 1)	(137, 16)	(76, 10)	(26, 1)	32	19
(2, 8190)	(16, 784, 2)	(256, 16)	(142, 10)	(676, 2)	33	20
(2, 16384)	(24, 5473, 6)	(948, 17)	(531, 10)	(17576, 28)	37	24
(257, 21504)	(8, 52, 1)	(211, 31)	(146, 24)	(26, 1)	55	41
(257, 21504)	(16, 784, 1)	(390, 31)	(268, 24)	(676, 1)	55	41
(257, 21504)	(24, 5473, 1)	(1341, 31)	(912, 24)	(17576, 2)	56	42

Table 2. Coefficient modulus size $\log q_d$ to ensure correct decryption (based on cutting size $\log 1/\kappa$), in worst and average cases, for both constructions with different message spaces.

6 Implementation

CPU Implementation. We implemented both of the proposed algorithms in C++ using DHS-SHE Library [14]. All simulations were performed on an Intel Xeon @ 2.9 GHz server running Ubuntu Linux 13.10. We compiled our code using Shoup's NTL library version 9.4.0 and with GMP version 5.1.3.

GPU Implementation. We introduced some optimization to achieve an efficient implementation on CUDA GPUs. We basically adopt the cuHE library [11] to build the SHE scheme and both autocomplete algorithms, yet make modifications to accommodate our parameter settings.

- **Pre-computation.** The cuHE library [11] handles polynomial multiplications in NTT domain. The overhead of NTT conversions on those plaintexts can be spared if they are performed during pre-computation. We then store those data in either CPU or GPU memory. There is no difference in terms of performance, since the memory transfer between CPU and GPU can be fully hidden behind computation tasks, as we monitored. Hence, the requirement on GPU memory size is minimized.
- **Modulus Switching.** In [11] the authors generated a series of prime numbers to make use of the double-CRT method proposed in [21]. We modified the double-CRT method and CRT prime number generation in the library to work with a homormophic encryption scheme with message space larger than 2.

We use an NVIDIA GeForce GTX 690 Graphic Card for development and test. The card comes with two GTX 680 GPUs. Therefore, we have the implementation designed to test either on a single GPU or on both GPUs. The specification of the graphic card is listed in 3.

Table 3. Testing Environment

Item	Specification	Item	Specification
CPU	Intel Core i7-3770K	GPU	NVIDIA GeForce GTX690
# of Cores	4	# of CUDA Cores	1536 × 2
# of Threads	8	GPU Core Frequency	1020 MHz
CPU Frequency	3.50 GHz	GPU Memory	2 GB × 2
Cache	8 MB	NTL Library	v 9.2.0
System Memory	32 GB DDR3	GMP Library	v 6.0.0a

6.1 Timing Results

We offer two encryption modes for 1-bit ($p = 2$) or 8-bit ($p = 257$) message sizes. Theoretically, encryption mode with $p = 257$ would save 7/8 memory required for the database, reduce the number operations by 8 times and consume only 1/8 bandwidth, compared to those of encryption mode with $p = 2$. The increase of p leads to a larger coefficient size and a higher polynomial degree, the consequence of which is a significant drop of performance. Fortunately the polynomial degree 21504 we selected creates more message slots S for batching. Remember the number of batched polynomials λ matters in terms of computational complexity. When the number of entries E is fairly small, $\lambda = \lceil E/S \rceil$ is 1 in both encryption modes. However, when E is large, a larger S yields a smaller λ, which reduce the number of computations greatly. Therefore, setting $p = 257$ only improves performance when the number of entries is sufficiently large.

The timing results in Table 4 are measured with respect to parameters selected for each case in Table 2. For autocomplete without comparison, we monitored that 85% of overhead is caused by launching CUDA kernels. Hence, due to the inefficiency of cuHE for this case, we recorded exactly the same timing results for $(p = 257, n = 21504)$ cases.

7 Conclusion

In this work, to the best of our knowledge for the first time we designed and implemented a privatized autocomplete scheme. We develop two variants of the autocomplete scheme and develop techniques to efficiently evaluate the homomorphic autocomplete schemes, i.e. autocomplete with comparison, and autocomplete without comparison. To this end, we analyzed noise growth in the both approaches and developed simple formula for each approach to evaluate the autocomplete circuit to support any database size with the required parameter sizes to provide an adequate level of security. Furthermore, in order to study

Table 4. Performance of Auto Completion with At Most 20 Suggested Characters

(p, n)	#letters	With Comparison			Without Comparison		
		CPU	1 GPU	2 GPUs	CPU	1 GPU	2 GPUs
(2, 8190)	1	45 sec	267 ms	176 ms	20 sec	299 ms	168 ms
(2, 8190)	2	1.1 min	843 ms	466 ms	35 sec	304 ms	175 ms
(2, 8190)	3	1.8 min	5.42 sec	2.95 sec	58 sec	484 ms	299 ms
(257, 21504)	1	1.45 min	1.71 sec	1.00 sec	40 sec	130 ms	80 ms
(257, 21504)	2	2.2 min	2.73 sec	1.98 sec	58 sec	130 ms	80 ms
(257, 21504)	3	3.4min	8.39 sec	5.26 sec	1.3 min	130 ms	80 ms

the performance (hence the practical relevance) of our schemes, we implemented both approaches on both CPU and GPU platforms. With the GPU implementation, for certain database sizes, we achieve a performance of less than a second which is close to what is desired in real-time interactive applications.

References

1. Bos, J.W., Lauter, K., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: Stam, M. (ed.) Cryptography and Coding, Lecture Notes in Computer Science, vol. 8308, pp. 45–64. Springer Berlin Heidelberg (2013)
2. Bos, J.W., Lauter, K., Naehrig, M.: Private predictive analysis on encrypted medical data. Tech. Rep. MSR-TR-2013-81 (September 2013), `http://research.microsoft.com/apps/pubs/default.aspx?id=200652`
3. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. Electronic Colloquium on Computational Complexity (ECCC) 18, 111 (2011)
4. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ACM (2012)
5. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-lwe and security for key dependent messages. In: Advances in Cryptology–CRYPTO 2011, pp. 505–524. Springer (2011)
6. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Rogaway, P. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 6841, pp. 505–524. Springer (2011)
7. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) lwe. SIAM Journal on Computing 43(2), 831–871 (2014)
8. Çetin, G.S., Doröz, Y., Sunar, B., Savaş, E.: Low Depth Circuits for Efficient Homomorphic Sorting. In: LatinCrypt (2015)
9. Cheon, Jung, H., Miran, K., Kristin, L.: Secure dna-sequence analysis on encrypted dna nucleotides. (2014), `http://media.eurekalert.org/aaasnewsroom/MCM/\-FIL\_000000001439/EncryptedSW.pdf`
10. Chor, B., Gilboa, N.: Computationally private information retrieval. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. pp. 304–313. ACM (1997)
11. Dai, W., Sunar, B.: cuHE: A homomorphic encryption accelerator library (2015)

12. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 6110, pp. 24–43. Springer (2010)
13. Doröz, Y., Hu, Y., Sunar, B.: Homomorphic AES evaluation using NTRU. IACR Cryptology ePrint Archive, Report 2014/039 2014, 39 (2014)
14. Doröz, Y., Hu, Y., Sunar, B.: Homomorphic AES evaluation using the modified LTV scheme. Designs, codes and cryptography, Springer Verlag (2015), https://eprint.iacr.org/2014/039.pdf
15. Doröz, Y., Sunar, B., Hammouri, G.: Bandwidth efficient pir from ntru. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) Financial Cryptography and Data Security, Lecture Notes in Computer Science, vol. 8438, pp. 195–207. Springer Berlin Heidelberg (2014)
16. Ducas, L., Micciancio, D.: Fhew: Bootstrapping homomorphic encryption in less than a second. In: Advances in Cryptology–EUROCRYPT 2015, pp. 617–640. Springer (2015)
17. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford University (2009)
18. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC. pp. 169–178 (2009)
19. Gentry, C., Halevi, S.: Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. IACR Cryptology ePrint Archive 2011, 279 (2011)
20. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. IACR Cryptology ePrint Archive 2012 (2012)
21. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit (updated implementation) (2015)
22. Graepel, T., Lauter, K., Naehrig, M.: Ml confidential: Machine learning on encrypted data. In: Kwon, T., Lee, M.K., Kwon, D. (eds.) Information Security and Cryptology ICISC 2012, Lecture Notes in Computer Science, vol. 7839, pp. 1–21. Springer Berlin Heidelberg (2013)
23. Granlund, T., the GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library (2012), http://gmplib.org/
24. Halevi, S., Shoup, V.: HElib, homomorphic encryption library. Internet Source (2012)
25. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Algorithmic number theory, pp. 267–288. Springer (1998)
26. Lauter, K., Naehrig, M., Vaikuntanathan, V.: Can homomorphic encryption be practical. Cloud Computing Security Workshop pp. 113–124 (2011)
27. Lauter, K., Lopez-Alt, A., Naehrig, M.: Private computation on encrypted genomic data. Tech. Rep. MSR-TR-2014-93 (June 2014), http://research.microsoft.com/apps/pubs/default.aspx?id=219979
28. Lepoint, T., Naehrig, M.: A comparison of the homomorphic encryption schemes fv and yashe. In: Progress in Cryptology–AFRICACRYPT 2014, pp. 318–335. Springer (2014)
29. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: STOC (2012)
30. López-Alt, A., Naehrig., M.: Large integer plaintexts in ring-based fully homomorphic encryption. in preparation (2014)
31. van de Pol, J., Smart, N.: Estimating key sizes for high dimensional lattice-based systems. In: Stam, M. (ed.) Cryptography and Coding, Lecture Notes in Computer Science, vol. 8308, pp. 290–303. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-45239-0_17

32. Shoup, V.: NTL: A Library for doing Number Theory, `http://www.shoup.net/ntl/`
33. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Nguyen, P.Q., Pointcheval, D. (eds.) Public Key Cryptography. Lecture Notes in Computer Science, vol. 6056, pp. 420–443. Springer (2010)
34. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. IACR Cryptology ePrint Archive 2011, 133 (2011)
35. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. Designs, codes and cryptography 71(1), 57–81 (2014)
36. Stehlé, D., Steinfeld, R.: Faster fully homomorphic encryption. Cryptology ePrint Archive 2010/299 (2010)
37. Stehlé, D., Steinfeld, R.: Making NTRU as secure as worst-case problems over ideal lattices. In: Advances in Cryptology–EUROCRYPT 2011, pp. 27–47. Springer (2011)

Appendix

7.1 The LTV Scheme

In 2012 López-Alt, Tromer and Vaikuntanathan proposed a multi-key FHE scheme, LTV [29]. It is based on a variant of NTRU encryption proposed by Stehlé and Steinfeld [36]. The introduced scheme uses relinearization and modulus switching techniques on top of NTRU for noise control and in order to make the scheme somewhat homomorphic and leveled fully homomorphic, respectively. The operations are performed in $\mathrm{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ where n is the polynomial degree and q is the prime modulus depending on a security parameter κ. The scheme also defines an error distribution χ, which is a truncated discrete Gaussian distribution, for sampling random polynomials that are B-bounded. These parameters n, q and χ are public. The term B-bounded means that the coefficients of the polynomial are selected in range $[-B, B]$ with χ distribution.

- **KeyGen.** A sequence of primes $q_0 > q_1 > \cdots > q_d$ is to use a different q_i in each level. A public and secret key pair is computed for each level: $h^{(i)} = 2g^{(i)}(f^{(i)})^{-1}$ and $f^{(i)} = 2u^{(i)} + 1$, where $\{g^{(i)}, u^{(i)}\} \in \chi$.
- **Encrypt/Decrypt.** To encrypt a bit b for the i^{th} level we compute: $c^{(i)} = h^{(i)}s + 2e + b$, where $\{s, e\} \in \chi$. In order to compute the decryption of a ciphertext $c^{(i)}$ for a particular level i we compute: $m = c^{(i)}f^{(i)} \mod 2$.
- **Eval.** The gate level logic operations XOR and AND are done by computing the addition and multiplication of the ciphertexts. In case of $c_1^{(i)} = \mathsf{Encrypt}(b_1)$ and $c_2^{(i)} = \mathsf{Encrypt}(b_2)$; XOR is equal to $\mathsf{Decrypt}\left(c_1^{(i)} + c_2^{(i)}\right) = b_1 + b_2$ and, AND is equal to $\mathsf{Decrypt}\left(c_1^{(i)} \cdot c_2^{(i)}\right) = (b_1 \cdot b_2)$.
- **Modulus Switch.** The multiplication creates a significant noise in the ciphertext and to cope with that a technique called modulus switching was introduced. From the i^{th} level ciphertext $c^{(i)}$, we compute the new level ciphertext $c^{(i+1)}$ with the reduced noise performing $c^{(i+1)}(x) = \lfloor \frac{q_i}{q_{i-1}} c^{(i)}(x) \rceil_2$.

This operation decreases the noise by $\log(q_i/q_{i-1})$ bits by dividing and multiplying the new ciphertext with the previous and current moduli, respectively. The operation $\lfloor \cdot \rceil_2$ refers to rounding and matching the parity bits.

$\mathrm{H}[i][j]$	$\mathrm{U}[j]$	$(\mathrm{H}[i][j] + \mathrm{U}[j] - 1)$
0	0	-1
0	1	0
1	0	0
1	1	1

Table 5. Truth table for bitwise comparison for arbitrary i, j,