Blind Web Search: How far are we from a privacy preserving search engine?

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

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

Abstract.
Recent rapid progress in fully homomorphic encryption (FHE) and somewhat homomorphic encryption (SHE) has catalyzed renewed efforts to develop efficient privacy preserving protocols. Several works have already appeared in the literature that provide solutions to these problems by employing FHE or SHE techniques. In this work, we focus on a natural application where privacy is a major concern: web search. An estimated 5 billion web queries are processed by the world's leading search engines each day. It is no surprise, then, that privacy-preserving web search was proposed as the paragon FHE application in Gentry's seminal FHE paper. Indeed, numerous proposals have emerged in the intervening years that attack various privatized search problems over encrypted user data, e.g. private information retrieval (PIR). Yet, there is no known work that focuses on implementing a completely blind web search engine using an FHE/SHE construction. In this work, we focus first on single keyword queries with exact matches, aiming toward real-world viability. We then discuss multiple-keyword searches and tackle a number of issues currently hindering practical implementation, such as communication and computational efficiency.

Keywords: fully homomorphic encryption, privacy preserving applications, encrypted web search

1 Introduction

The Electronic Frontier Foundation writes "Anonymous communications have an important place in our political and social discourse. The [US] Supreme Court has ruled repeatedly that the right to anonymous free speech is protected by the [US] First Amendment"[1]. The contents of our web searches give a glimpse into our personal lives and the information that can be harvested from a log of such activity has, in several well-publicized instances, led to clear violations of privacy. It is not only the keywords themselves that leak information. When a user chooses from among query responses and chooses to be directed to a particular URL on the list, the contents of the query are often shared with that website. Browsing history can be tracked and shared, and with data mining techniques, the user can end up revealing much more than keywords, such as their Personally Identifiable Information or Sensitive Personal Information. Search engines are also able to infer one's geographic location through one's IP address. When search and other such data is stored, a user profile can be created and this can be shared with third parties such as governments, marketers, and even cyber-criminals. One can easily imagine, for instance, how knowledge of recent queries regarding financial instruments could assist a hacker in composing a more credible phishing email that purports to originate at a bank at which the user is a customer.

To protect users from such malicious players, search engines have introduced encrypted search traffic (keywords and results) over the past few years, employing secure connections and creating

[1] `https://www.eff.org/issues/anonymity`

an encrypted channel between the user device and the search engine server. This option prevents third parties from spoofing responses, but also ended the practice of passing keyword history to the chosen URL (except when that URL was selected through Adwords). Solutions of this sort put the privacy barrier between the search engine and the websites who are their customers, but make no guarantees of private queries. The search engines can still form user profiles and can still share them with third parties for a price. Moreover, such keyword histories can be released by the search engine unintentionally, for example by court order or data breach. Standard search services such as Google require the cleartext query to be handled on the server side revealing to the search company a wealth of information to mine. When mined along with other sources of private information, e.g. e-mail or cloud storage, the search provider can distill a wealth of sensitive information to an unprecedented level of detail. To counter this trend, privacy friendly search services has emerged in a last few years, e.g. DuckDuckGo and StartPage by ixquick. DuckDuckGo, for instance, has even gained customers reaching 10M searches per day. The standard approach taken by these companies is to **promise** to respect the privacy of their customers. While there has been no incident to suspect these products, here too privacy hangs by a thread, i.e. a fragile trust mechanism.

The rapid emergence of a new rich set of homomorphic encryption tools in the past few years provides a new opportunity to revisit the privacy challenges of online search. Our solution, in contrast, leaves the server/search engine completely oblivious not only to the actual keywords but also the corresponding search results. Fully homomorphic encryption (FHE) allows one to perform arbitrary computation on encrypted data without the need of a secret key [1]. If made practical, FHE has tremendous potential to protect user's privacy. For instance, FHE allows for fully blinded cloud computing and retrieval services for applications such as healthcare databases. Indeed Gentry motivated his dissertation using online search: "Fully homomorphic encryption has numerous applications. For example, it enables private queries to a search engine — the user submits an encrypted query and the search engine computes a succinct encrypted answer without ever looking at the query in the clear." [1]. We are inspired to investigate FHE solutions for a privacy-preserving web search engine. Even though there are several works that focus on search over encrypted databases using FHE, the present paper is the first in which both keywords and responses are encrypted in an end-to-end fashion, with only ciphertexts accessible to any observer on the Web. Considering the computational and bandwidth requirements of early FHE schemes, along with the fact that web search is a real-time application, the lack of practical proposals prior to this one is not surprising. However we have witnessed an amazing barrage of improvements in FHE and SHE schemes over the past few years [2–7]. In [8] Gentry, Halevi and Smart (GHS) proposed the first homomorphic evaluation of a complex circuit: a full AES block. In 2012 Halevi (and later Shoup) published the HElib [9], a C++ library for HE that is based on Brakerski-Gentry-Vaikuntanathan (BGV) cryptosystem [7]. In [10] a leveled NTRU-based [11, 12] 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 just 12 seconds. In early 2015, Gentry, Smart, Halevi (GHS) [15] published significantly improved AES runtime results with 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. Recently, a new leveled FHE scheme called $\mathsf{F\text{-}NTRU}$ was proposed in [17] that makes use of the flattening technique from [18] over a variant of NTRU [19]. $\mathsf{F\text{-}NTRU}$ does not use relinearization — no evaluation keys are needed —

and, due to its small parameter sizes, the scheme achieves fast homomorphic multiplication, for instance a depth 30 multiplication circuit can be executed in ≈ 17 msec.

Our Contribution. In this work,

- We present the first end-to-end study of blinded search using homomophic encryption where the client submits encrypted keywords and the server performs a blinded lookup and returns the results again in encrypted form. We separate our problem into two parts; Comparison and Aggregation.
- We cast Comparison into a private information retrieval (PIR) problem and compare various PIR algorithms, i.e. variants of Kushilevitz-Ostrovsky PIR, for suitability in our setting.
- We analyze the depth and bandwidth complexities as well as number of multiplications for each approach.
- We present our Aggregation step first in the single keyword scenario and then extend our construction to consider the queries with multiple keywords. To this end, we perform a homomorphic intersection by encoding URLs as zeros of polynomials. We compare our approach against other with regard to the problem of returning false positives.
- With all the pieces in place, we provide a noise analysis of the proposed methods with respect to F-NTRU parameters and finally give the implementation results of the proposed schemes using a GPU implementation of the scheme. The results show that the bandwidth overhead is in the MBytes while the query requires micro to milliseconds for processing per row to support single and multiple keyword lookups with intersection, i.e. AND operations on keywords.

This paper is organized as follows: In Section 2, we give a brief description of the underlying leveled FHE scheme that is used in our construction. Within the same section, we describe an encoding technique to represent large messages which helps with the noise growth when used alongside a special plaintext modulus. In the next section 3, we give our definition of an encrypted search engine by providing an example and real world numbers. In Section 4, we start building our encrypted search engine by using homomorphic properties and investigating different algorithms and their asymptotic performances with respect to the FHE scheme in use and finally we give the implementation details of the proposed methods and provide a comparison of their run time performances as well as their bandwidth requirements in the last section.

2 Background

In this section, we will first define the leveled FHE scheme [17] that we employ in our application. Then, we will define an encoding technique that was previously used in [20] which we will use later in our construction to represent large integers.

2.1 FHE Scheme: F-NTRU

A new FHE scheme [17] F-NTRU adopts the flattening technique proposed in GSW to derive an NTRU based scheme that (akin to GSW) does not require evaluation keys or key switching. This scheme eliminates the decision small polynomial ratio (DSPR) assumption but relies only on the standard R-LWE assumption. The scheme uses wide key distributions, and hence is immune to the subfield lattice attack.

The F-NTRU scheme makes use of the following operations in the ring $\mathbb{Z}_q/(x^n+1)$ for key setup, encryption/decryption and the homomorphic evaluations:

- **KeyGen:** The same parameters and the keys from NTRU scheme are used in this scheme. For a plaintext modulus p, and a security parameter λ, we choose a ring modulus q and a ring degree n which is a power of 2. We also fix the Gaussian distributions $\chi_{\text{err}} = \chi_{\text{err}}(\lambda)$ and $\chi_{\text{key}} = \chi_{\text{key}}(\lambda)$ using the same security parameter. We compute the public key $h = 2gf^{-1}$ and the secret key $f = 2f' + 1$, where g, f' are sampled from χ_{key}.
- **Encrypt:** The encryption function from NTRU, $\mathsf{Enc}(m) = hs + pe + m$ where $m \in [0, p-1]$, is used with a difference in the ciphertext structure. We must first define an operation called $\mathsf{BitDecomp}$ that splits a ciphertext polynomial $c(x)$, into ℓ binary polynomials and we show it as follows:

$$\begin{aligned}\mathsf{BitDecomp}(c(x)) &= \langle \tilde{c}_{\ell-1}(x), \cdots, \tilde{c}_1(x), \tilde{c}_0(x) \rangle \\ &= \tilde{\boldsymbol{c}},\end{aligned}$$

  and given ℓ $\tilde{c}_i(x)$s, computing $c(x)$ is called the inverse bit decomposition, $\mathsf{BitDecomp}^{-1}$.

$$\begin{aligned}\mathsf{BitDecomp}^{-1}(\tilde{\boldsymbol{c}}) &= \sum_{i=0}^{\ell-1} 2^i \cdot \tilde{c}_i(x) \\ &= c(x).\end{aligned}$$

  In this scheme we have a vector of NTRU encryptions, as our ciphertext of a single encrypted message μ. The length of the ciphertext vector is $\ell = \log q$ and we start by placing an encryption of zero in every element of this vector.

$$\begin{aligned}\boldsymbol{c} &= \langle \mathsf{Enc}_{\ell-1}(0), \mathsf{Enc}_{\ell-2}(0), \ldots, \mathsf{Enc}_0(0) \rangle \\ &= \langle c_{\ell-1}, c_{\ell-2}, \ldots, c_0 \rangle,\end{aligned}$$

  where $c_i = \mathsf{Enc}_i(0) = hs_i + pe_i$. By taking the transpose of $\boldsymbol{c}$, we first place each encryption of zero in a single row and then using bit decomposition over each row, we build an $\ell \times \ell$ matrix $c = \mathsf{BitDecomp}(\boldsymbol{c}^\top)$.
  Finally, using this matrix, we encrypt the message μ by computing,

$$C = \mathsf{Flatten}(I_\ell \cdot \mu + c)$$

  where I_ℓ is the identity matrix of order ℓ, $\mathsf{Flatten}$ is the special technique from [18] which is an inverse bit decomposition, followed by a bit decomposition operation:

$$\mathsf{Flatten}(\tilde{\boldsymbol{c}}) = \mathsf{BitDecomp}(\mathsf{BitDecomp}^{-1})(\tilde{\boldsymbol{c}}).$$

- **Decrypt:** To decrypt a ciphertext, we take the first row of the matrix, which is the vector $\tilde{\boldsymbol{c}}_0$, and apply $\mathsf{BitDecomp}^{-1}$ to form an NTRU ciphertext $\mathsf{BitDecomp}^{-1}(\tilde{\boldsymbol{c}}_0) = c_0$. Once we compute the NTRU ciphertext, we apply the decryption method from the NTRU scheme as $\lfloor c_0 f \rceil \bmod p$ and retrieve the message μ.
- **Eval:** The homomorphic XOR and AND operations are matrix addition and multiplication operations, followed by a $\mathsf{Flatten}$ operation as below.

$$C' = \mathsf{Flatten}(C + \tilde{C}) \quad , \quad C' = \mathsf{Flatten}(C \cdot \tilde{C}).$$

2.2 Encoding Large Integers

Similar to the approach used in [20], we will use a small polynomial for our plaintext modulus. For this method, we set the plaintext parameter $p = (x - 2)$. Whenever we want to encrypt a k-bit (integer) message α, we write $\alpha = \sum_{i=0}^{k-1} \alpha_i 2^i$ and place its bit decomposition in a plaintext polynomial $\mu(x) = \sum_{i=0}^{k-1} \alpha_i x^i$ which is then encrypted. Upon decryption, α can be retrieved by simply evaluating[2] $\mu(x)$ at $x = 2$. Note that, provided the ring modulus exceeds M, this provides for carry-free addition of M ciphertexts; i.e., if $\alpha_h = \sum_{i=0}^{k-1} \alpha_{h,i} 2^i$ is encoded as $\mu_h(x) = \sum_{i=0}^{k-1} \alpha_{h,i} x^i$, then the polynomial $\nu(x) = \sum_{h=1}^{M} \mu_h(x)$ satisfies $\nu(2) = \sum_h \alpha_h$.

3 Encrypted Web Search

In this section, we will define a privacy-preserving web search engine that evaluates encrypted search queries.

To fully execute a typical web search, a server carries out four fundamental tasks: web crawling, indexing, ranking and retrieval. For an encrypted web search engine, the first three tasks are unchanged as these do not involve encryption; it is only the task of retrieval in response to an encrypted query from the user that we need to address here. Modern search engines have become quite sophisticated, accounting for spelling errors, synonyms or word meanings, sometimes applying ranking after retrieval. But this is beyond the scope of the present work: we deal only with exact matches here. Thus we assume that pre-processing on the server side provides us with a rank-ordered output list of URLs attached to each keyword.

In our construction, we have two parties: a user $\mathcal{U}$ who submits the (encrypted) search query and a server $\mathcal{S}$ who is the owner of the database and the entity that performs the retrieval look up. Server $\mathcal{S}$ has a table consisting of the dictionary words and their corresponding rank-ordered search results. In a regular search engine, $\mathcal{S}$ has to know the user keyword κ in order to perform the look up. In order to be able to process encrypted keywords, we convert this standard comparison/aggregation model into a homomorphic circuit so that it can be evaluated using encrypted input(s) and it can output encrypted results.

In a simple scenario, when $\mathcal{U}$ submits a query, the input keyword κ is first encrypted on the client side under client's own encryption key, then the ciphertext(s) are sent to the server and the server performs the retrieval step obliviously using homomorphic circuits over the encrypted user input and the server's own database which is in cleartext form. For instance, in order to search for the keyword "otomycosis", $\mathcal{U}$ encrypts κ = "otomycosis" using his own encryption key and sends the encrypted data[3] $[\![\text{otomycosis}]\!]$ to the server. $\mathcal{S}$ then evaluates the homomorphic circuit using only the encrypted input(s) and the index table it owns. During the homomorphic evaluation, all intermediate results will still be encrypted under the client's encryption key. After the circuit evaluation, $\mathcal{S}$ returns the corresponding output ciphertext(s) to $\mathcal{U}$. Finally, $\mathcal{U}$ decrypts the resulting ciphertext(s) using its own decryption key and gets the matched query results (if any). Note that user-specific filtering, ranking and formatting of results can then be performed on the client side working with plaintexts obviating the need to share certain user preferences with an untrusted server.

[2] Note that this is same as computing $\mu \mod p$ and in case of a multiplication, as long as the resulting polynomial has a degree less than $n = n(\lambda)$, there will be no overflow.

[3] Throughout this paper, we will use $[\![x]\!]$ to denote an encrpytion of x.

In reality, there are around 60 trillion web pages on the web[4] and the number of pages indexed by Google is around 47 billion[5]. However, this number can be much smaller for a custom search engine that is designed for a specific target group of clients, for example a medical search engine would only index the health care related pages. Another example can be an internal search engine for a company that only crawls the subnet of that company. Hence, in practice the indexed pages may contain a small subset of the overall web, especially for a proof-of-concept design. We denote the set of all indexed URLs under consideration by $\mathcal{L}$ and denote the size of $\mathcal{L}$ by T. We will use a short URL representation where each element $u \in \mathcal{L}$ is a $\lceil \log T \rceil$ bit integer. In case of a universal search engine such as Google, this number can be as high as 36 bits. This forces us to make a sacrifice by placing an upper limit t on the number of "hits" for any given keyword; that is, we assume that our Search Engine Results Page (SERP) contains only the t top-ranked URLs for each simple query. This is obviously quite a bit simpler than modern search engines. However, since our outputs will be encrypted under an FHE scheme, our plaintext space -total number of bits that we can decrypt at the end of homomorphic computations- is limited due to chosen FHE parameters. Therefore the number of output URLs t depends on several factors like plaintext space, bit-size of a single URL and the number of search keywords and the details of this relation will be given in Section 4.3.

The other component of the database is the keyword list. The Oxford English Dictionary includes over a half million English words whereas a target-oriented vocabulary can have fewer number of items, for example Stedman's medical dictionary has around a hundred thousand medical terms. Thus, the size of the keyword list changes according to the search engine functionality. To accommodate a realistic scenario where the database can be fairly large, we choose N as one million, which is our bound on the number of entries in the lookup table. We will make use of a local dictionary to encode each keyword κ as a 20-bit integer ($\log N$) using their row index in the table and reserving index 0 for the keywords not found in the dictionary. These parameters determining the number of possible input/output values to be transferred between the server and the client have an effect on both the circuit size and the bandwidth, thus they have significant bearing on the runtime of the application. We will design our circuits for generic values of these parameters and return to specific values only later, in Section 5.3, where we observe their effect on bandwidth and execution time.

4 Our Construction

In this section, we will construct a homomorphic circuit to provide our solution to the encrypted search problem. First we will have a look at well-known search algorithms and where they stand when used in the encrypted domain. Then, we will separate our design into two steps and analyze them in detail.

4.1 Associative Array and Search Algorithms

The dictionary we defined in Section 3 consists of a pair of objects — a *key* from the alphabetically sorted keyword list; and a list of *values*, each from the URL list — in each row. The association between the two objects will be referred to as binding. Whenever we want to find the value which is bound to a given keyword, we will apply an operation called Aggregation. This Aggregation

[4] `https://www.google.com/insidesearch/howsearchworks/thestory/`
[5] `http://www.worldwidewebsize.com/`

operation requires a Comparison operation to find the keyword location in the array. To this end, there are a number of search algorithms that we can use such as serial search, binary search and search by hashing. From these three serial search and hash methods have a worst case complexity of $\Theta(N)$ whereas binary search has $\Theta(\log(N))$. But logarithmic complexity for binary search is not achievable in a blind search. Since homomorphic execution requires the method to explore all paths regardless of outcome, even the binary search has linear complexity in this setting.

4.2 The Construction for Comparison

In encrypted search schemes, expensive computations and massive user-server bandwidth requirements remain as serious obstacles to achieving real-time FHE-based applications. In this section, we present a homomorphic construction that can handle practical scenarios. We start with a primitive scheme with low circuit depth and then provide optimizations and propose a lightweight algorithm, to push our design closer to a real-life application. This improvement comes with a price: the more we decrease bandwidth, the deeper the circuit required.

The first step of the blind search is encrypting the user input κ. Hence the first decision to make is about the representation of the keywords. As the keywords are arbitrary length strings and we will be working in an homomorphic setting, binary representation seems like a natural choice. But binary representation of a string would leak information about its length. And particularly too short or too long keywords can be easier to guess. The second option is to encrypt the string as a word by setting a large plaintext domain. This requires a word-wise homomorphic comparison method which is possible, but costly. Lastly, we can use the index w of the input keyword in the dictionary. We consider the following four algorithms for encrypted keyword search. In the following algorithms, all w are $\log N$ bits, where N is the dictionary size.

– **Standard Comparison Algorithm.** In this method, the bits of the index w are encrypted. Using an equality check circuit $\prod_{j=1}^{\log N}(w_j \oplus i_j \oplus 1)$, we can compare the input to every single possible index value $i = 1, \cdots, N$ with bits $i_1 i_2 \cdots$. The bandwidth of this approach is equal to the number of bits of the index w, hence it is bounded by $s = \log N$. The number of multiplications will be $s-1$ for each i, thus in total there are $N(s-1)$ ciphertext multiplications in this method.
– **Kushilevitz-Ostrovsky (KO) Algorithm.** In this case the input index w is divided into two parts, $w = (w_1, w_2)$ where $w = w_1\sqrt{N} + w_2$ and both w_1 and w_2 are one-hot encoded using the above approach. This reduces the bandwidth to $2^{\log(N)/2+1} = 2\sqrt{N}$, and the depth becomes 1. This method can be applied recursively on the new index values w_1, w_2 and we can have partitions into four subwords $w = (w_1, w_2, w_3, w_4)$ which would reduce the bandwidth to $4N^{1/4}$ and increase the depth to 2. If we partition into k pieces, we end up with a bandwidth of $kN^{1/k}$.In the case of multiplications, we are not able to set up a regular multiplication tree which is optimized for KO constructions because of the limitations of the F-NTRU scheme. In order to compute the comparison, we perform k-dimensional multiplication in a serial manner. First, we multiply along two axes, contributing $N^{1/k} \cdot N^{1/k}$ multiplications. Next, the results are multiplied with values along a third axis which results in $N^{1/k} \cdot N^{2/k}$ multiplications. After $k-1$ iterations, the total number of multiplications is $N^{1/k}\sum_{i=1}^{k-1} N^{i/k}$.
– **Hybrid Algorithm.** We will divide the input index w into two parts $w = (w_1, w_2)$ where $w = w_1 N/2^s + w_2$, i.e. w_1 is the first s bits of the index. We will perform the standard comparison on the first part w_1 of length s and encode the second part w_2 of length $\log N - s$ using the KO

Table 1: Comparison of bandwidth requirements and number of multiplications for different Comparison algorithms

Algorithms	Bandwidth	Multiplications
Standard Comparison	$\log N$	$N(\log N - 1)$
KO Construction	$kN^{1/k}$	$\sum_{i=1}^{k-1} N^{\frac{i+1}{k}}$
Hybrid Method	$s + k(N/2^s)^{1/k}$	$2^s(s-1) + \sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}} + N$

algorithm. Then, the bandwidth of scheme is summarized as $s + k(N/2^s)^{1/k}$, where s is coming from the first part and the rest is coming from the KO algorithm. The number of multiplications are $2^s(s-1)$ and $\sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}}$ for the first and second parts, respectively. Since we need to multiply the results of first and seconds parts with each other to form the decisions, the total number of multiplications results in $2^s(s-1) + \sum_{i=1}^{k-1} (N/2^s)^{\frac{i+1}{k}} + N$.

The overall bandwidth and multiplicative complexity comparison is given in Table 1. We quickly observe that the optimal choice of encoding depends heavily on the properties of the FHE scheme in use.

4.3 Construction for Aggregation

After the Comparison step is completed for a user input, we have the intermediate values which we call the *decision vector* and denote by $\boldsymbol{d}$. This vector has N encrypted bit values; $\boldsymbol{d}[i] = [\![1]\!]$ if $i = w$ and $\boldsymbol{d}[i] = [\![0]\!]$ otherwise. After constructing our circuits, we will see that we do not need the Comparison results all at once, but instead we can have one result at a time. This means that we do not have to store N encrypted values, but we will compute the $\boldsymbol{d}[i]$ values in an iterative way.

The second step of our application is the actual Aggregation step, where we will compute our final output(s). Note that for a more practical engine, we consider the multi keyword search (e.g., a conjunction of m terms). This means that whenever there are two or more input keywords, we will perform the Comparison multiple times. As a result, we will have one decision vector corresponding to each input keyword. For an m-keyword search, we will have decision vectors: $\boldsymbol{d}_1, \boldsymbol{d}_2, \ldots, \boldsymbol{d}_m$ corresponding to the inputs $w_1, w_2, \ldots, w_m$.

The size of each URL list L_i that is bound to the i^{th} keyword in the database is $|L_i| = t_i$, for $i \in [1, N]$. Since the server is oblivious to the input value(s) w, it is also oblivious to the list size t_w. This means that the output size $|L_{\text{out}}|$ will be determined by the longest list in the database, given by $t = \max\{t_i | i \in [1, N]\}$. In the next part of this section, we will describe the algorithm for a single keyword query (i.e., $m = 1$). In the following part, we will give the details for our multi-keyword aggregation problem ($m > 1$).

Single Keyword First let us assume that we have a single input keyword with dictionary index w. We have the URL list[6] $L_i = [u_{i,1}, u_{i,2}, \ldots, u_{i,t}]$ for each keyword index i. After computing the

[6] In order to maintain uniformity, we define $u_{i,j} = 0$ for $t_i < j \leq t$.

decision vector $\boldsymbol{d}$, we can find the final outputs by simply computing

$$\begin{aligned} L &= \sum_{i=1}^{N} \boldsymbol{d}[i] L_i \\ &= \left[\sum_{i=1}^{N} \boldsymbol{d}[i] u_{i,1}, \sum_{i=1}^{N} \boldsymbol{d}[i] u_{i,2}, \ldots, \sum_{i=1}^{N} \boldsymbol{d}[i] u_{i,t}\right] \\ &= [\llbracket u_{w,1} \rrbracket, \llbracket u_{w,2} \rrbracket, \ldots, \llbracket u_{w,t} \rrbracket] \,. \end{aligned}$$

If each URL index $u_{i,j}$ is $\lceil \log T \rceil$ bits in length as we defined in Section 3, at the end the server will have $t\lceil \log T \rceil$ bits to return to the client. By using the large integer encoding technique from Section 2.2, we can decode a single ciphertext and retrieve n bits where n is the FHE ring degree. This means that we can have $\lceil t\lceil \log T \rceil / n \rceil$ output ciphertexts or alternatively, if we limit the t value such that the inequality $t\lceil \log T \rceil \leq n$ holds, then all outputs can be encoded into a single n degree polynomial. It means that the server will send back only a single ciphertext.

A further economy can be found by noting that we do not need to compute all $\boldsymbol{d}[i]$ values in advance. Instead, in the first iteration we can compute $\boldsymbol{d}[1]$ and after initializing L to $\boldsymbol{d}[1]L_1$, we can then compute $\boldsymbol{d}[2]$ and update $L = L + \boldsymbol{d}[2]L_2$. Iterating in this way, we only need to store $t+1$ encrypted polynomial coefficients at a time.

For reasons that will become clear shortly, we propose an alternative way to encode the list $L_i = [u_{i,1}, u_{i,2}, \ldots, u_{i,t}]$ as a polynomial

$$\ell_i(x) = \prod_{j=1}^{t} (u_{i,j} - x).$$

After computing the decision vector $\boldsymbol{d}$, we can find the final outputs by simply computing

$$\ell(x) = \sum_{i=1}^{N} \boldsymbol{d}[i] \ell_i(x).$$

If each URL index $u_{i,j}$ is again $\lceil \log T \rceil$ bits in length, hence the coefficient of x^h in $\ell_i(x)$ is about $\lceil \log T \rceil (t-h)$ bits. Total number of bits to be returned to the user will be $\lceil \log T \rceil \sum_{h=0}^{t} (t-h) = \frac{(t^2+t)}{2} \lceil \log T \rceil$ which is more compared to the first approach where we only have $t\lceil \log T \rceil$ bits. Hence, we will use this polynomial representation of the URL lists for multiple keyword construction in the next section.

Multiple Keywords We discuss here only conjunctive queries; while the product of two polynomials gives a natural solution to the disjunction of keywords (union of URL lists), this changes polynomial degrees and, since disjunctive queries are uncommon, we choose not to address this added level of complexity here. In Aggregation, if a user wants to find all URL values that are bound to multiple keys, we are faced with a set intersection problem [21]. Suppose, for simplicity the query takes the form $w \wedge w'$ and the above procedures generate corresponding polynomials $\ell_w(x)$ and $\ell_{w'}(x)$, respectively. The roots of $\ell_w(x)$ (resp., $\ell_{w'}(x)$) encode the top-ranked URLs for keyword w (resp., w') so the conjunction would naturally correspond to the gcd of the two polynomials. But this is difficult to compute homomorphically. So, as previously noted by [21], we can instead take a random polynomial in the ideal generated by $\ell_w(x)$ and $\ell_{w'}(x)$. Clearly, if $d = \gcd(\ell_w, \ell_{w'})$ and

$$f(x) = g(x)\ell_w(x) + g'(x)\ell_{w'}(x),$$

then $d(x)$ divides $f(x)$ and the probability that $f(x)$ has any additional "spurious" roots is negligible assuming reasonable parameters. In practice, it may even suffice to take $f(x) = \ell_w(x) + \ell_{w'}(x)$. But we prefer to take a more careful approach.

We can, in fact, reduce this probability of spurious roots to zero with careful choice of coefficients. All valid URLs are known to lie in the range $[1, T]$. If r and r' are primes just a bit larger than T, then it is impossible for

$$f(x) = r\ell_w(x) - r'\ell_{w'}(x)$$

to have any root in range $[1, T]$ which is not a common root of ℓ_w and $\ell_{w'}$. Indeed, if $f(u) = 0$, then

$$r\ell_w(u) = r'\ell_{w'}(u).$$

But $\ell_w(x) = \prod(u_j - x)$ for some roots u_j all lying in $[1, T]$. So $\ell_w(u) = \prod(u_j - u)$ has no prime factor exceeding T. Therefore $f(u) = 0$ for $u \in [1, T]$ forces both $\ell_w(u) = 0$ and $\ell_{w'}(u) = 0$. This naturally generalizes to a conjunction of m keywords, as we outline below. But we point out here that, with $m = 2$ being the most common conjunction, we see some economy by choosing r and r' close together — twin primes, for example, with $r = r' + 2$ — and noting

$$\begin{aligned} f(x) &= r\ell_w(x) - r'\ell_{w'}(x) \\ &= r'\left[\ell_w(x) - \ell_{w'}(x)\right] + 2\ell_w(x) \end{aligned}$$

sometimes allows us to control growth of coefficients.

Lemma 1. *Let $w_1, \ldots, w_m$ be m distinct keywords with corresponding polynomials $\ell_1(x), \ldots, \ell_m(x)$ where the roots of $\ell_i(x)$ encode the top-ranked URLs bound to keyword w_i. Let $s_1, \ldots, s_m$ be m distinct primes with $s_i > T$ for all i and, for $1 \le i \le m$, define $r_i = \frac{1}{s_i}\prod_{h=1}^{m} s_h$. Then, for $f(x) = \sum_{i=1}^{m} r_i \ell_i(x)$, $\gcd\left(f(x), \prod_{u=1}^{T}(x - u)\right) = \gcd\left(\ell_1(x), \ldots, \ell_m(x)\right)$.*

Proof. If $u \in [1, T]$ and $f(u) = 0$, then reduction modulo s_i gives $\ell_i(u) = 0 \pmod{s_i}$. But $\ell_i(u)$ has no large prime factors, so $\ell_i(u) = 0$ and, since this holds for all i, u is a common root. □

The second problem we must deal with is the representation of the polynomials. Assuming the polynomials ℓ_i have degree t, we have $\|\ell_i\|_\infty = t \log T$. Therefore the large integer encoding technique from Section 2.2 turns these ($t \log T$)-bit numbers into polynomials of degree $t \log T$ with 0/1 coefficients. The computations we must perform on the ℓ_i are polynomial additions and constant multiplications; so we can perform the same operations over each coefficient separately, send the results to the user without further processing, and the user can reconstruct the polynomial encoding the conjunction of m keywords. If we write

$$\ell_i(x) = \sum_{k=0}^{t} \alpha_{i_k} x^k,$$

then constant multiplication by integer b is computed

$$b\ell_i(x) = \sum_{j=0}^{t} b\alpha_{i_k} x^k$$

and similarly, the sum of m polynomials is computed coefficient by coefficient:

$$\sum_{i=1}^{m} \ell_i(x) = \sum_{j=0}^{t} \left(\sum_{i=1}^{m} \alpha_{i_k} \right) x^k.$$

We have $(t \log T)$-bit coefficients and the constant multiplication with the integers r_i in the above lemma will add $(m-1)(\log T + 1)$ bits to the end result, so we will have approximately $(m + t)\log T$ bit values to be encoded/decoded. If we have an FHE ring of degree n, we can afford $t < \lfloor n/\log T \rfloor - m$.

We have $\ell_i(x) = \sum_{k=0}^{t} \alpha_{i_k} x^k$ and it will be convenient to likewise write $\ell_{w_j}(x) = \sum_{k=0}^{t} \alpha_{(w_j)_k} x^k$. These are stored as lists of coefficients and when the server processes a decision vector $\boldsymbol{d}_j$ encrypting the one-hot encoding of the j^{th} keyword of the query, it may compute $\sum_{i=1}^{N} \boldsymbol{d}_j[i]\ell_i(x)$ one coefficient at a time. But we can do better. The user needs access to the polynomial $\ell(x) = \sum_{j=1}^{M} r_j \ell_{w_j}(x)$ and, if we write $\ell(x) = \sum_{k=0}^{t} \beta_k x^k$, we have

$$\begin{aligned} [\![\ell(x)]\!] &= \sum_{j=1}^{m} r_j [\![\ell_{w_j}(x)]\!] \\ &= \sum_{j=1}^{m} r_j \sum_{i=1}^{N} \boldsymbol{d}_j[i] [\![\ell_i(x)]\!] \end{aligned}$$

so that,

$$[\![\beta_k]\!] = \sum_{j=1}^{m} r_j \sum_{i=1}^{N} \boldsymbol{d}_j[i] [\![\alpha_{i_k}]\!] \ .$$

This list $\{[\![\beta_k]\!]\}_{k=0}^{t}$ is passed to the user who, upon decryption, recovers all β_k, reconstructs $\ell(x)$ and applies standard root-finding techniques to obtain the desired list of URLs arising from the conjunctive query. Note that the constant term of $\ell(x)$ has no large prime factors, so its roots lying within $[1, T]$ can be recovered rather efficiently by standard techniques.

4.4 Noise Analysis

We take same approach as authors did in [17], since we are using the F-NTRU scheme with a small modification. Our scheme changes only the message space from 2 to $x - 2$. Using $||g||_\infty = ||f'||_\infty = B_{\text{key}}$, $||s||_\infty = ||e||_\infty = B_{\text{err}}$, a fresh ciphertext has the following noise bound:

$$\begin{aligned} ||y_{(0)} f||_\infty &\leq 4nB_{\text{key}}B_{\text{err}} + 4nB_{\text{err}}(4B_{\text{key}} + 1) \\ &\leq 4nB_{\text{err}}(5B_{\text{key}} + 1). \end{aligned}$$

At each multiplication, using the single sided multiplication approach as in [17], the noise bound is equal to

Table 2: The values of index i in noise bound B_i to compute decisions for each entry in Comparison algorithms.

Algorithms	**# Multiplications (i)**
Standard Comparison	$s-1$
KO Construction	$k-1$
Hybrid Method	$s+k-1$

$$\begin{aligned} B_i = ||fy_i||_\infty \leq & [4n^2 B_{\text{key}} B_{\text{err}}(2^w-1)\ell \\ & + 4n^2 B_{\text{err}}(4B_{\text{key}}+1)(2^w-1)\ell] \\ & + [4nB_{\text{err}}B_{\text{key}} + 4nB_{\text{err}}(4B_{\text{key}}+1)] \\ & + [B_{i-1}] + [(4B_{\text{key}}+1)] \end{aligned}$$

The number of multiplications for the decision depends on the Comparison algorithm. Thus, it changes the required bitsize for each algorithm. In Table 2 we give the maximum length of multiplicative chain to compute a decision for an entry, i.e. it is the index i used in noise bound B_i.

In addition to the base noise which is occurring from the decisions, we have additional noise occurring from the latter operations. We can list the additional noise occurrences as follows

- multiplication of decision with the message (encoded as binary polynomial): $\log\log n$
- summation of all the entries: $\log N$
- multiplication with prime number to eliminate spurious roots (encoded as binary polynomial): $\log\log r$
- number of search results which we add together (number of keywords): $\log m$

These additional noise should be added to the base noise bound so that the modulus is large enough to support the operations.

5 Implementation and Performance

Either single-keyword or multi-keyword search requires a tremendous amount of computation power. Taking advantage of previous research in [22–24], we believe an implementation on CUDA-enabled GPUs offers high efficiency. We compared our proposed hybrid algorithm to a recursive KO algorithm in terms of bandwidth requirements and computation time. All the timing results in this paper are measured on the machine described in Table 3.

5.1 Polynomial Multiplications

During the evaluation steps, the performance of ciphertext multiplications dominates the total execution time. Our implementation focuses on optimizing a ciphertext multiplication (recall Section 2.1) which is the product of a vector of polynomials and a matrix of polynomials: $R_{2^w}^{1\times l} \times R_{2^w}^{l\times l}$. Note that we perform ciphertext multiplications in a chain and keep only the last row of the left-hand multiplier all the time, which explains why one of the multiplier is in vector form.

Table 3: Testing Environment

Item	Specification
CPU	Intel Core i7-3770K
CPU Freq.	3.50 GHz
System Memory	32 GB DDR3
GPU	NVIDIA GeForce Titan X
GPU Core Freq.	1.20 GHz
GPU Memory	12 GB
# of CUDA Cores	3072

Starting with polynomial multiplications, we compare the efficiency of Karatsuba algorithm and the Number-theoretic transform (NTT) based algorithm proposed in [22]. We ignore overhead caused by additions or binary operations such as shifting/or/and/xor to draw a simpler yet fair comparison. Assume polynomials have degree slightly smaller than 2048. We need to perform 4096-sample NTT conversions.

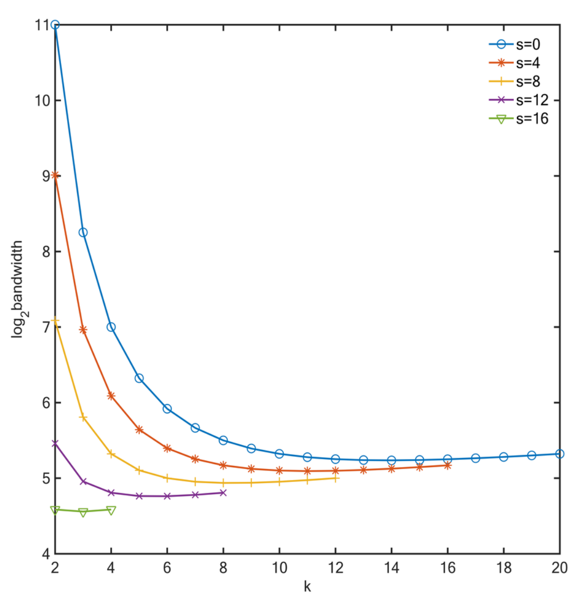
In [22] a special finite field $\mathbb{F}_P$ where $P = \texttt{0xffffffff00000001}$ is chosen. Also the NTT or inverse-NTT (INTT) conversions of 4096 samples can be factorized into smaller size (e.x. 64 sample) following the CooleyTukey FFT algorithm [25]. Conversions of no larger than 64 samples are implemented with shifting, addition and fast modular reduction over P, which takes advantage of properties of P. The NTT-based algorithm only requires 4096 integer multiplications per conversion when multiplying twiddle factors. One polynomial multiplication requires two NTT conversion, a coefficient-wise multiplication of two NTT domain vectors (4096 integer multiplications) and one INTT conversion, which add up to a total of 16384 integer multiplications. However, the Karatsuba algorithm that requires $3^{\log 2048} = 177147$ multiplications would be much slower.

Plus, the NTT-based algorithm is highly parallelizable hence more suitable for a GPU implementation. In conclusion, the NTT-based algorithm outcomes the Karatsuba algorithm on a GPU. Following the ideas in [22], we developed 4096-point NTT/INTT conversions for a GPU. Each NTT conversion takes 0.75 μs and each INTT conversion takes 1.45 μs. Every ciphertext multiplication contains $l^2 + l$ NTT converions and l INTT conversions. The overhead of multiplications and additions in NTT domain is negligible.

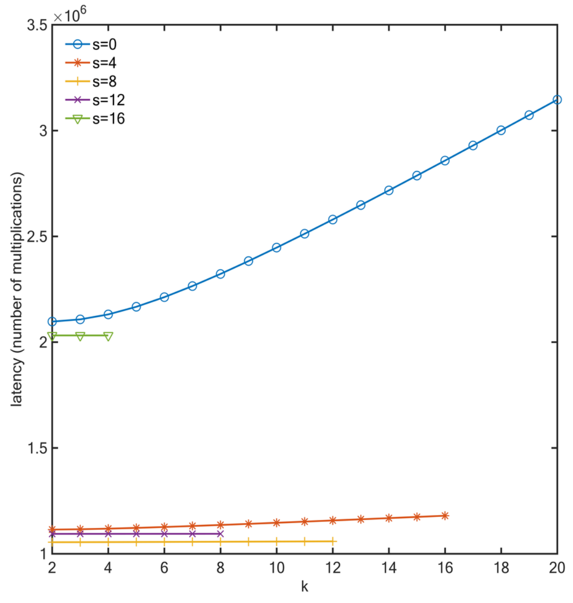
5.2 Select Ring for $\mathsf{Flatten}$

Ciphertext multiplications, although taking 16-bit norm polynomials as input, produce polynomials with ln times larger norm (less than 64-bit) as output. A $\mathsf{Flatten}$ operation is expected to reformat the ciphertext coefficients back to 16-bit. The $\mathsf{Flatten}$ includes (e.x. 64-bit) integer additions with shifting and reductions modulo q. We decide to choose q as a power of 2 and polynomial degree n as a prime, which is used in the classic NTRU cryptosystem. The benefit is obvious: modulo reduction over a power of 2 is lightning fast.

A ciphertext multiplication takes $l+1$ polynomials in R_q as input. The $\mathsf{BitDecomp}$ takes no time. Then the multiplication algorithm produces l polynomials with 64-bit coefficients. $\mathsf{BitDecomp}^{-1}()$ or $\mathsf{Flatten}() = \mathsf{BitDecomp}(\mathsf{BitDecomp}^{-1}())$ is achieved with a sequence of 16-bit additions with carries, which adds 2.66 μs.



(a) Bandwidth (base 2 logorithm)



(b) Latency (number of multiplications)

Fig. 1: A comparison of hybrid and KO algorithms in bandwidth and latency with respect to parameter choices. The database has $N = 2^{20}$ entries. Standard comparison is applied on the first s bits of input index. A KO algorithm with k iterations is applied on the rest. KO algorithm is adopted when $s = 0$.

5.3 A Comparison of Algorithms in Performance

In our experiments, we first fix the dictionary size $N = 2^{20}$ and the FHE parameters according to the noise analysis given in Section 4.4 and the security parameter of $\mathsf{F-NTRU}$ scheme. We use the ring $\mathbb{Z}_q/(x^n + 1)$ where the coefficient modulus is $q = 2^{192}$ and the degree is $n = 2039$. Following the security analysis from [17], the Hermite factor for the chosen values is 1.00525 which provides 128-bit security.

Based on previous analysis of bandwidth requirements and computation time, Fig. 1 illustrates a comparison of several algorithms with selected yet typical parameters. The hybrid algorithm becomes a KO construction when $s = 0$. And when $k = 2$, it is a basic (non-recursive) KO construction. With the help of those figures, we may find optimal parameters that has low bandwidth and latency.

As shown in Table 1, a standard comparison algorithm has minimum bandwidth requirement. The same fact is reveal in Fig. 1a: a hybrid algorithm, when applying standard comparison on more bits, i.e. when choosing a larger s, has a lower bandwidth requirement. Then as we apply more iterations of KO, i.e. as k increases from 2, bandwidth requirement drops significantly. The value k, although affects latency of KO, has insignificant influence on hybrid schemes. Hence, a optimal hybrid scheme with a certain s would choose k with minimal bandwidth.

The number of multiplications in a hybrid method is expressed as a formula in Table 1. The fact that one part of the formula involves s while another part involves k makes Fig. 1b more

Table 4: A comparison of hybrid and KO algorithms with different parameters. Database has $N = 2^{20}$ entries.The bandwidth is calculated for 576 KB input ciphertexts and 48 KB output ciphertext. The first column gives the number of input keywords in each scenario. Time includes the latencies of Comparison and Aggregation, and is normalized per database entry.

#	Algorithm	s	k	Bandwidth		Time (μs)
				Input	Output	
1	KO	0	2	1,152 MB	48 KB	304
1	KO	0	9	24 MB	48 KB	341
1	Hybrid	8	8	17 MB	48 KB	168
1	Hybrid	12	5	15 MB	48 KB	173
2	KO	0	9	48 MB	2.40 MB	3,544
2	Hybrid	8	8	34 MB	2.40 MB	3,198
2	Hybrid	12	5	30 MB	2.40 MB	3,207
3	Hybrid	8	8	51 MB	2.35 MB	4,722

complicated (rather than increasing/decreasing along with s). We can see that when $s = 8$, the latency is lower than all other cases. And choosing $s = 4, 8, 12$ does not gives a critical difference in latency. However, when comparing $s = 4, 8, 12$ in Fig. 1a, one may easily notice the remarkable difference in bandwidth.

The hybrid methods outcome the (recursive) KO construction on bandwidth requirements and computation time. Within the hybrid methods, if the priority is to reduce bandwidth requirement, select $s = 12$ and $k = 5$; if the priority is efficiency, select $s = 8$ and $k = 8$. These two parameter sets are adopted in Table 4. For single-keyword search, the hybrid scheme with $s = k = 8$ requires 37.7% less bandwidth and costs half the time, comparing to those of the KO construction with $k = 9$ iterations. However, for multi-keyword scenarios, the Aggregation weights most of the latency. Therefore the advantage of hybrid methods is clear.

Finally, the size of the output ciphertexts depends on the number of URLs that we return t in the multikeyword scenario with respect to the inequality from Section 4.3, $t < \lfloor n/\log T \rfloor - m$ where n is 2039. We set $\lceil \log T \rceil = 40$ so that the indexed URL set $\mathcal{L}$ can have up to a billion URLs. Therefore for 2 keywords, we afford sending back at most $t = 49$ URLs and for 3 keywords, $t = 48$ at most. In each case the server sends back $t + 1$ encrypted coefficients of the resulting t-degree URL polynomial and each coefficient is represented in a separate ciphertext. In order to increase t, we have to choose a larger degree for the FHE setup, which would end up increasing both the bandwidth and the computation time. In the single keyword scenario, we choose to limit the number of URL outputs $t = 50$, so that the result fits into a single ciphertext following the relation $t\lceil \log T \rceil \leq n$ from Section 4.3. Note that in this case, if we want to increase the number of output URLs t, we can do so by increasing the bandwidth and sending back $\tilde{n}$ ciphertexts each carrying n bits, as long as $t \leq \frac{\tilde{n}n}{\lceil \log T \rceil}$ holds.

References

1. C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.
2. ——, "Fully homomorphic encryption using ideal lattices," in *STOC*, 2009, pp. 169–178.
3. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, H. Gilbert, Ed., vol. 6110. Springer, 2010, pp. 24–43.
4. Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Advances in Cryptology–CRYPTO 2011*. Springer, 2011, pp. 505–524.
5. ——, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014.
6. C. Gentry and S. Halevi, "Fully homomorphic encryption without squashing using depth-3 arithmetic circuits," *IACR Cryptology ePrint Archive*, vol. 2011, p. 279, 2011.
7. Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 18, p. 111, 2011.
8. C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," *IACR Cryptology ePrint Archive*, vol. 2012, 2012.
9. S. Halevi and V. Shoup, "HElib, homomorphic encryption library," Internet Source, 2012.
10. A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *STOC*, 2012.
11. J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic number theory*. Springer, 1998, pp. 267–288.
12. D. Stehlé and R. Steinfeld, "Making NTRU as secure as worst-case problems over ideal lattices," in *Advances in Cryptology–EUROCRYPT 2011*. Springer, 2011, pp. 27–47.
13. Y. Doröz, Y. Hu, and B. Sunar, "Homomorphic AES evaluation using NTRU," *IACR Cryptology ePrint Archive, Report 2014/039*, vol. 2014, p. 39, 2014.
14. Y. Doröz, Y. Hu, and B. Sunar, "Homomorphic AES evaluation using the modified LTV scheme," *Designs, codes and cryptography, Springer Verlag*, 2015. [Online]. Available: https://eprint.iacr.org/2014/039.pdf
15. C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit (updated implementation)," 2015.
16. L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology–EUROCRYPT 2015*. Springer, 2015, pp. 617–640.
17. Y. Doröz and B. Sunar, "Flattening ntru for evaluation key free homomorphic encryption," Cryptology ePrint Archive, Report 2016/315, 2016, http://eprint.iacr.org/2016/315.
18. C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *CRYPTO*. Springer, 2013, pp. 75–92.
19. D. Stehlé and R. Steinfeld, "Faster fully homomorphic encryption," *Cryptology ePrint Archive 2010/299*, 2010.
20. K. Lauter, A. Lopez-Alt, and M. Naehrig, "Private computation on encrypted genomic data," Tech. Rep. MSR-TR-2014-93, June 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=219979
21. *Privacy-Preserving Set Operations*, 2005. [Online]. Available: http://dx.doi.org/10.1007/11535218_15
22. W. Dai, Y. Doröz, and B. Sunar, "Accelerating NTRU based homomorphic encryption using GPUs," *2014 IEEE High Performance Extreme Computing Conference (HPEC'14)*, 2014.
23. ——, "Accelerating SWHE based PIRs using GPUs," in *Workshop on Applied Homomorphic Computing – WAHC 2015*, 2015.
24. W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," 2015.
25. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. Comput*, vol. 19, no. 90, pp. 297–301, 1965.

Acknowledgment

Funding for this research was in part provided by the US National Science Foundation CNS Award #1319130.