LEAF: A Faster Secure Search Algorithm via Localization, Extraction, and Reconstruction

Rui Wen
CISPA Helmholtz Center
for Information Security

Yu Yu
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute

Xiang Xie
PlatON

Yang Zhang
CISPA Helmholtz Center
for Information Security

ABSTRACT

Secure search looks for and retrieves records from a (possibly cloud-hosted) encrypted database while ensuring the confidentiality of the queries. Researchers are paying increasing attention to secure search in recent years due to the growing concerns about database privacy. However, the low efficiency of (especially multiplicative) homomorphic operations in secure search has hindered its deployment in practice. To address this issue, Akavia et al. [CCS 2018, PETS 2019] proposed new protocols that bring down the number of multiplications in the search algorithm from $O(n^2)$ to $O(n \log^2 n)$, and then to $O(n \log n)$, where n is the size of the database. In this paper, we present the first secure search protocol – LEAF and its variant LEAF$^+$– which only requires $O(n)$ multiplications. Specifically, at the core of LEAF are three novel methods we propose, referred to as *Localization*, *Extraction*, and *Reconstruction*. In addition, LEAF enjoys low communication complexity and only requires the client to perform decryption, which adds its advantage in deployment on weak-power devices such as mobile phones.

KEYWORDS

Secure Search; (Leveled) Fully Homomorphic Encryption.

ACM Reference Format:

Rui Wen, Yu Yu, Xiang Xie, and Yang Zhang. 2020. LEAF: A Faster Secure Search Algorithm via Localization, Extraction, and Reconstruction. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372297.3417237

1 INTRODUCTION

With the advancement of cloud computing technologies, more and more companies and individuals (*client*) start to hand over their data to third-party cloud storage companies (*server*). In this context, a client gives away her data to a server, who has full access to and operates on the data in plaintext following the client's instructions, such as database queries and operations. In many such cases, privacy concerns arise especially when a client's data contains highly sensitive information, such as biomedical records, and the server is not fully trustworthy.

Many advanced cryptographic techniques, such as multi-party computation (MPC) and searchable encryption, can be leveraged to perform data searching in the cloud in a privacy-preserving manner. Among these techniques, fully homomorphic encryption (FHE) based secure search has attracted a lot of attention in recent years. Compared to MPC, secure search is more efficient communication wise as it only requires a single round of interaction independent of the matching function. Moreover, secure search does not reveal any valid information contrary to searchable encryption, which provides a stronger data privacy guarantee.

Secure search roughly contains two steps, i.e., *matching* and *searching*. In the matching step, the server compares the encrypted search query (from the client) with all encrypted items in the database, and returns another encrypted array of 0s and 1s with 1 indicating the corresponding database item satisfying the query. The searching step returns all 1's indexes and corresponding items to the client. Secure search suffers from high computation cost due to the expensive FHE operations required, in particular, in the searching step. In this paper, we focus on the searching step, and will refer to the searching step as secure search without ambiguity. The most intuitive method, namely Folklore, requires $O(n^2)$ times of homomorphic multiplication which is not practical in real-world scenarios. In recent years, some solutions have been proposed to make secure search more practical. For example, Akavia et al. [1] proposed SPiRiT, which leverages the multi-ring technique to reduce the required multiplication number to $O(n \log^2 n)$. More recently, Akavia et al. [2] proposed a new algorithm which adopts a low-degree approximation method for the OR operation to further reduce the number of multiplication to $O(n \log n)$.

1.1 Our Contributions

In this paper, we propose the first FHE-based secure search algorithm which only requires $O(n)$ times multiplication and, at the same time, does not add depth in the asymptotic sense. As we will explain in more detail in Section 2, depth is a unique and vital concept we will meet when designing FHE-based algorithms. Higher depth leads to more time per homomorphic operation costs. Our algorithms, namely (LEAF) and its variant (LEAF$^+$), rely on three novel techniques: *Localization*, *Extraction*, and *Reconstruction*.

*Part of this work was done while the first author was doing his final year project at Shanghai Jiao Tong University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '20, November 9–13, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417237

Table 1: Complexity comparison. In the Full LEAF scheme, the bootstrapping technique is applied after every homomorphic encryption, therefore the cost of each multiplication operation is independent of n, and the total complexity of $O(n)$. We use blue to represent the best complexity in practice, and use red to represent the optimal complexity in theory. As multiplication takes much more time than addition, here the time complexity only considers multiplication.

Algorithm Name	Degree of Function	Number of Multiplications	Time Complexity
Folklore	$O(n)$	$O(n^2)$	$O(n^2 \log^{\omega} n)$
SPiRiT Det.	$O(\log^3 n)$	$O(n \log^2 n)$	$O(n \log^2 n (\log\log n)^{\omega})$
AGHL	$O(\log n)$	$O(n \log n)$	$O(n \log n (\log\log n)^{\omega})$
LEAF	$O(\log^2 n)$	$O(n)$	$O(n (\log\log n)^{\omega})$
LEAF$^+$	$O(\log n)$	$O(n)$	$O(n (\log\log n)^{\omega})$
Full LEAF	/	$O(n)$	$O(n)$

Localization technique is used to localize the first matched item in a smaller interval, it divides the original array into many equal length's smaller intervals, and returns the encrypted indexes of the interval containing the matched item.

Extraction is designed to extract the interval containing the desired item. Since both indexes and contents are homomorphically encrypted, we cannot use indexes to extract the target interval directly. Therefore, we use extraction technique to extract the interval while do not need to decrypt the index or increase depth.

Localization and extraction are able to reduce non-necessary operations on non-target intervals, and allow us only apply the search algorithm on the target interval. This results in LEAF and LEAF$^+$only requiring $O(n)$ times multiplication.

Finally, we utilize *Reconstruction* technique to combine two position information together to generate the final output. This method accepts two encrypted position indexes: one indicates the interval the matched item located, the other indicates the offset the matched item in the target interval. Without decryption, *Reconstruction* is able to output the encrypted actual index in the original database.

We combine these three techniques to construct LEAF. Moreover, on the basis of LEAF, we use lazy bootstrapping to refresh depth at a specific stage to construct its variant LEAF$^+$. We compare the state-of-the-art secure search algorithms in Table 1, where n is the number of items and $\omega < 2.3727$ is the matrix multiplication exponent. As we can see, LEAF$^+$ has reached the most advanced level both in degree of function and number of multiplications.

Asymptotically, LEAF$^+$ algorithm performs best. But compared with LEAF, when n is small, the optimization effect is less satisfactory due to the increased time overhead and computational depth associated with introducing the bootstrapping step. However, when n is large, the optimization efficiency of LEAF$^+$ algorithm is better.

Our algorithms (LEAF, LEAF$^+$) support negligible error probability and do not need pre- and post-processing, which means our algorithm has very low computational power requirements for the client. In our protocol, the client only needs to do one encryption and decryption operation, which is particularly useful when the client is limited in its computational power, e.g., the client is a mobile device. This feature greatly expands the application scenarios of the algorithm. Meantime, our algorithms require computations solely over $GF(2)$, which makes them better optimized as smaller plaintext modulus requires lower depth and makes the bootstrapping procedure cost fewer multiplication operations. Our algorithms also support unrestricted search function, e.g., exact match operation, compare operation, and range limited operation.

To illustrate our algorithms' correctness and efficiency, we give a correctness proof and derive the complexity specifically. Moreover, we implement our algorithm and compare with the prior state-of-the-art proposal to show our algorithm's efficiency.

In summary, our key contributions are as follows:

- We propose a new secure search algorithm which requires only $O(n)$ times of homomorphic multiplications, while the state-of-the-art requires $O(n \log n)$ times of multiplications.
- We keep our algorithm's depth invariant even after applying retrieving method, which further speeds up our algorithm's efficiency.
- We give a concrete complexity analysis of our algorithm, which allows potential users and researchers to estimate the practical efficiency.

1.2 Organization

The rest of this paper is organized as follows: we present the whole framework of secure search in Section 2, preliminary definitions and notations are given in Section 3. The algorithm description (LEAF) and its correctness proof is described in Section 4. Efficiency analysis is provided in Section 5. A variant of the algorithm (LEAF$^+$) is given in Section 6. The experimental results are given in Section 7. Further optimization is given in Section 8. Related works are given in Section 9. Finally we conclude the paper in Section 10.

2 SECURE SEARCH: OVERVIEW

Due to the nature of cloud services, privacy has been a concern since the day the cloud service appeared. Clients do not want the server to learn anything sensitive. Thus, encrypt clients' data would be a natural solution. Privacy-preserving solutions are methods that meet the requirements, including MPC, searchable encryption and FHE. Among them, FHE has multiple advantages, such as low communication complexity and stronger privacy guarantee. Using FHE to realize searching on encrypted data is called *secure search.*

For the sake of simplicity, we assume there is only one client (*single client*) and one server (*single server*) when we describe the secure search process in our paper.

In theory, it is feasible to use FHE to securely compute any (polynomial-time) algorithm including secure search. To use FHE,

one needs to represent the algorithm using a Boolean circuit. Following many FHE-relevant works (e.g., [10, 13]) by depth we refer to the "multiplicative depth" of the circuit throughout this paper, as illustrated in Figure 1. In homomorphic encryption, the time required for the multiplication operation is much larger than the addition operation, and the time consumption of the single multiplication operation is related to the circuit *depth*. How to design an algorithm with fewer number of multiplications and low computation *depth* remains a huge challenge.

2.1 Secure Search Framework

Before presenting the secure search algorithm, we need to first introduce some initial operations:

- **Key Generation:** Run the homomorphic encryption key generation algorithm, generate a public key (pk) and a private key (sk), publish the public key (pk), and keep the private key (sk) by the client.
- **Upload:** The client encrypts items using the public key (pk), as shown in Figure 2. in the client part. Then the client uploads the encrypted items. It is worth noting that in this step, the client can encrypt the newly added item with the same public key and upload it to the server-side at any time.
- **Data Structure:** We utilize *array* as the data structure used in the protocol. The items in the array are *unsorted*, which eliminates the need for our protocol to preprocess uploaded data and there is no need to guarantee the order in which the data is stored in the database.

That is, we need to first use the public key to encrypt our data and then upload them to the server. We use array as the data structure to store data, every data has its own index, we only need to do these steps once. If new data needs to be added, we can use the public key to encrypt the new data and upload it, which also reflects the benefits of not preprocessing the data, then we can execute our search process:

- **Input:** The client selects the lookup value as needed, encrypts it using the public key (pk) generated in the Key Generation step and sends it to the server.

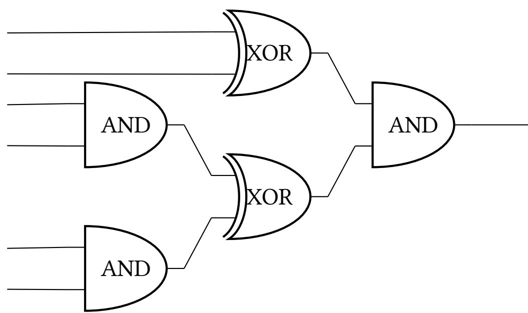


Figure 1: A function is represented by a Boolean circuit composed of addition (XOR) and multiplication (AND), where the multiplicative depth is the maximal number of ANDs along the paths of the circuit (i.e., omitting XORs).

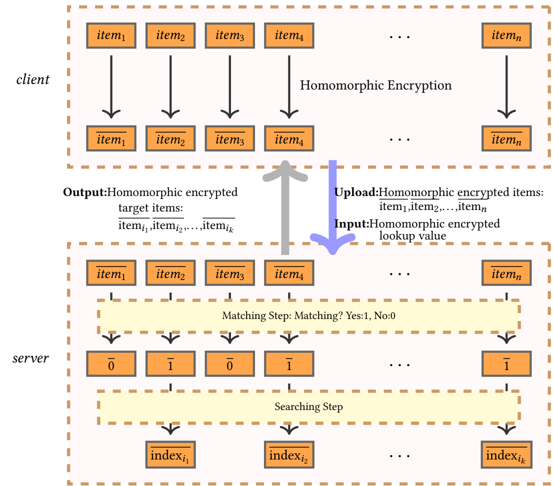


Figure 2: The procedure of secure search using homomorphic encryption, the client encrypts its lookup value, and the server returns the indexes and content of the corresponding item. $\overline{x}$ represents the ciphertext corresponding to plaintext x.

- **Matching:** The server runs the specified *matching function* to match the encrypted items in the database and returns an array whose element is encrypted 0 or 1 to represent a mismatch or match based on the matching results.
- **Searching:** Search function takes an array of length n whose elements are encrypted 0 or 1 (n is the number of elements) as input, outputs the encrypted indexes of all non-zero elements in the array entered.
- **Output:** According to the output of the searching step, the server returns ciphertext corresponding to the index. After the client decrypts it with the private key (sk), the required plaintext can be obtained.

This gives a *single round* protocol with *low communication*, the client only needs to input the encrypted lookup value, and the server will return matched encrypted items to the client, as shown in Figure 2. Specifically, the communication complexity is proportional only to the sizes of the input and output ciphertexts, while using prior secure multi-party computation (MPC) techniques (see [18, 19, 32]), the cost is proportional to the size of the search function.

It should be noted that this protocol can easily be extended to multi-client and multi-server situation. All clients should have access to the private key, and thus can upload their encrypted lookup value and decrypt returned results. We can apply our algorithm on every server, then gather the outputs together.

In the matching step, *matching function* refers to determining whether two plaintexts corresponding to two ciphertexts satisfy the certain condition when given two ciphertexts. There's a lot of research on matching function, see Section 9 for details. In this paper, we mainly focus on improving the searching part.

Intuitively, the whole process seems very simple, however, due to the inefficient operation of homomorphic multiplication, there are still many difficulties and requirements in the design of a specific

algorithm. Specifically, since the time that homomorphic multiplication costs are absolutely dominant compared with other operations, the goal of our algorithm design is to reduce the number of multiplications as much as possible.

At the same time, when utilizing the leveled fully homomorphic algorithm, we must consider the depth parameter L of the algorithm. Because choosing a larger depth parameter will increase the time consumption of a single homomorphic operation as mentioned before, we need to reduce the depth required by the entire algorithm as much as possible while reducing the multiplication number.

Note that we may describe the searching polynomial in terms of its (multiplicative) depth or degree depending on the context. One should not be confused with these two notions, and in general $depth \approx \log degree$ for bounded fan-in circuits. That is, for AND gates with fan-in bounded by B, the depth of $\Pi_{i=1}^{degree} x_i$ is minimized using the balanced B-ary tree evaluation such that $depth \approx \log_B degree$, where B is a small constant (typically 2) and thus often omitted, see Figure 10 in Appendix B as an example.

We now show the most relevant works: *Secure Search on FHE Encrypted Data*, which addresses the same formulation as considered in our work:

Folklore: This is the most direct and intuitive solution for secure search on FHE encrypted data. For an encrypted array $v \in \{0,1\}^n$ to be searched, the search polynomial is ($p > n$):

$$\text{Folklore}(v) = \sum_{i=1}^{n} v[i] \cdot \prod_{j=1}^{i-1} (1 - v[j]) \cdot i \mod p$$

This method will iterate through each item, resulting in inefficient server runtime due to evaluating $O(n)$ degree polynomials, in addition, the protocol also requires $O(n^2)$ multiplications, for n the number of items.

SPiRiT: A breakthrough work by Akavia, Feldman, and Shaul appeared in CCS18 [1], only needs to evaluate a polynomial of degree $O(\log^3 n)$. The authors use Fermat's theorem as a normalized function which leads to a high degree. This algorithm requires post-processing and requires $O(n \log^2 n)$ multiplications.

AGHL: The prior state-of-the-art for secure search on FHE encrypted data appeared in a recent work of Akavia, Gentry, Halevi and Leibovich [2], where the server can evaluate a polynomial only with logarithmic degree $O(\log n)$ to get the results without post-processing and can be implemented on $GF(2)$. Their work uses a low-degree polynomial to compute the OR operations of the first i items, and put the result in the i-th item. Their work requires $O(n \log n)$ multiplications.

In this paper, our main motivation is to find a more efficient search algorithm which reduces the number of multiplications and computational depth as much as possible.

2.2 Overview of Our Techniques

As described in Section 2.1, the search algorithm will receive an array of which elements are 0 or 1 as input and output all non-zero elements' indexes and corresponding contexts. Without loss of generality, we will only consider returning the *first* non-zero item's coordinate in this paper as previous works [1, 2]. In summary,

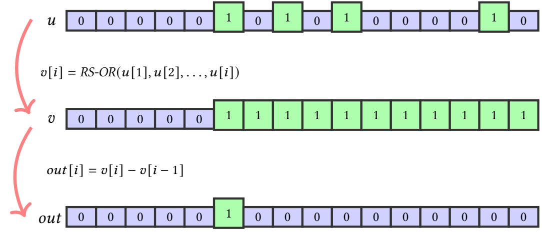


Figure 3: Schematic diagram of AGHL algorithm, first we change the value of each number in the array to the OR result of this number and all the numbers before this number in the array to obtain a *stepped array*, then calculate pairwise difference to get the *pulse array*.

our goal is to find the first non-zero item in an *unsorted* FHE encrypted *array* in the searching step, while minimizing the number of multiplications and required depth.

At present, the prior state-of-the-art algorithm (AGHL) is proposed by Akavia et al.[2], the schematic diagram of their algorithm is shown in Figure 3. Roughly speaking, AGHL algorithm's key idea is changing the value of each element in the array to the OR result of this element and all the elements before this element in the array. Since any number OR 1 will result in 1, thus if the first non-zero element appears, all the elements after this element will be changed to 1, then compute pairwise differences of adjacent indexes, we can obtain a *pulse array* containing only one 1 without revealing any information, which could lead to a binary representation of the first non-zero item's coordinate. To reduce the depth, a low-degree approximation method for OR called RS-OR (see definition in Section 3.3) is applied.

In this paper, we utilize the *Localization*, *Extraction* and *Reconstruction* techniques to propose a more efficient algorithm, whose multiplication complexity is consistent with plaintext search asymptotically, more precisely, reduce the number of multiplications from $O(n \log n)$ to $O(n)$.

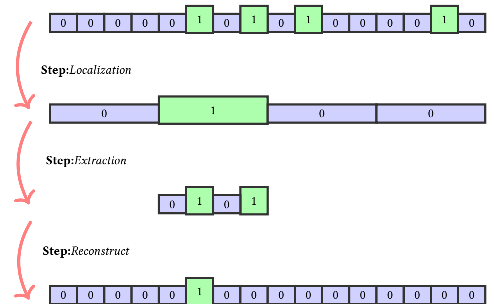


Figure 4: The basic framework of LEAF, after the *localization* step, it is determined the interval in which the target item is located, then applied *extraction* step to extract this interval, and finally the *reconstruction* step is used to obtain the array we need.

Key Intuition: We derive the intuition behind our algorithm from two observed facts:

(1) In AGHL algorithm, RS-OR operation is the main source of multiplication.
(2) In OR operation, if at least one of the elements involved in the operation is 1, the result of the operation is 1.

According to (1), to improve the efficiency of the algorithm, we hope to reduce *RS-OR* operations as much as possible. Utilizing (2), we can locate the first non-zero item to a smaller interval and thus reduce *RS-OR* operations on the non-target interval. Then we use AGHL algorithm to find the first non-zero item in this small interval. Combining the above two position information, we can obtain the position information of the first non-zero item in the original array.

For the sake of easy understanding, we take Figure 4 as an example: the length of the array is 16 ($n = 16$), and we divide it into four equal length intervals, each with a length of 4, we create a new array of length 4, where the i-th item of the array indicates whether the i-th interval of the original array contains the first non-zero item, the second element of the array in the figure is 1, indicating that the first non-zero item in the original array appears in the interval of 5 to 8. Next, we extract the interval containing the first non-zero item, and use AGHL algorithm to get the offset of the first non-zero item in this interval which is 2. Combined with the previous position information, we can get the coordinate of the first non-zero item in the original array, which is $4 + 2 = 6$.

One major difficulty is how to extract the target interval while don't increase degree too much. In plaintext case, we can directly utilize indexes to get the target interval, however, it will not work in ciphertext case since the indexes are encrypted. PIR could be applied to solve this problem without further interaction, but will introduce additional $O(\log n)$ function degree, even worse, PIR can only retrieve one item at a time, we have to utilize PIR several times, which will be time-consuming.

In this paper, we propose a technique call *Extraction* to solve this problem, while only increasing a constant to the depth of the circuit. The key idea behind this technique is that we can obtain more position information in our protocol compared with only know the encrypted indexes.

Later in this paper, to complete our protocol, we propose a retrieval algorithm that will not further increase our full protocol's depth, by adjusting the multiplication structure.

We also provide some suggestions on algorithm optimization, so that the efficiency could be further improved in the implementation, see details in Section 8.

3 PRELIMINARIES

In this section, we introduce notations, security model and necessary building blocks for establishing our algorithm in Section 4.

3.1 Notations

Denote $[n] = \{1, \dots, n\}$. For an array v, we denote $v[i]$ the i-th element in v. We enumerate array entries starting from entry number 1, unless stated otherwise.

We use $\overline{x}$ to represent the ciphertext of x with some homomorphic encryption scheme. For a field $\mathbb{F}$, vectors $v, u \in \mathbb{F}^n$ and $k \in [n]$, denote $\langle v, u\rangle = \sum_{i=1}^{n} v[i] \cdot u[i] \mod 2$, $\text{prefix}_k(v) = (v_1, \dots, v_k) \in \mathbb{F}^k$, $\text{suffix}_k(v) = (v_{k+1}, \dots, v_n) \in \mathbb{F}^{n-k}$, and $|v|$ the size of v.

In this paper, RS-OR refers to Razborov-Smolenski method as described in definition 3.1, and PPT denotes probabilistic polynomial time. We define *pulse array* as an array in which only one element is 1 and the others are 0. *Stepped array* is defined as an array where elements are all 0 before the first 1 appears, and others are all 1 after the first 1 appears.

3.2 Security Model

In our scenario, the server is compromised by a *semi-honest* and *computationally bounded* adversary who will not deviate from the protocol but try to learn additional information in polynomial-time. The use of semantically secure homomorphic encryption ensures that the adversary learns nothing substantial more than the scale of computation and length of ciphertext.

Since the protocol itself returns all matches, allowing the server to know the number of matches, we can let the client decides whether to continue the next match search after finding a certain number of matches in the subsequent interaction design to solve this problem.

3.3 Razborov-Smolenski Method

The Razborov-Smolenski method [26, 29] is a low-degree approximation algorithm for OR, which reduces the degree of OR from n to $\log(\frac{n}{\varepsilon})$ by introducing an error parameter ε. This technique is applicable in $GF(q)$ for any $q \geq 2$, but we only consider the case of $q = 2$ in this paper.

Definition 3.1 (Razborov-Smolenski Method). *For any k-bit prefix $(v[1], \dots, v[k]) \in \{0,1\}^k$ of the vector of n binary indicator values, we can calculate the approximation OR result of these k values in the following way: Select $N(\varepsilon) = \lceil \log(\frac{n}{\varepsilon}) \rceil$ independent uniformly random $r_1, \dots, r_{N(\varepsilon)} \in \{0,1\}^n$, compute the parity of the corresponding random subset of entries,*

$$p(r_j) = \sum_{i=1}^{k} r_j[i] \cdot v[i] \mod 2$$

Next, compute the OR of these parity values using the standard degree $N(\varepsilon)$ polynomial for the logical-OR of $N(\varepsilon)$ binary values:

$$\begin{aligned} \textit{RS-OR}(v[1], \dots, v[k]) &= OR\big(p(r_1), \dots, p(r_{N(\varepsilon)})\big) \\ &= 1 - \prod_{j=1}^{N(\varepsilon)} \big(1 - p(r_j)\big) \mod 2 \end{aligned}$$

The parity bit $p(r_j)$ is always zero when $v = 0^k$ and it is one with probability half when $v \neq 0^k$. Therefore, when $N(\varepsilon)$ random variables r_i are selected, the error probability of RS-OR is:

$$Pr\big[OR(v[1], \dots, v[k]) \neq \textit{RS-OR}(v[1], \dots, v[k])\big] < 2^{-N(\varepsilon)}$$

The above equation indicates that when $N(\varepsilon) = \lceil \log(\frac{n}{\varepsilon}) \rceil$, the probability that RS-OR$(v[1], \dots, v[k])$ and OR$(v[1], \dots, v[k])$ are equal is $1 - \frac{\varepsilon}{n}$, but reduced the calculation degree to $N(\varepsilon) = \lceil \log(\frac{n}{\varepsilon}) \rceil$.

Since the error parameter ε will directly affect the degree of RS-OR operation, choosing a small error probability will reduce the efficiency of the algorithm. Therefore, from a practical perspective,

clients need to choose the appropriate parameter ε according to their requirements for accuracy, and we usually choose $\varepsilon = 2^{-80}$. In this paper, we will set n to be the whole database's size, so every RS-OR operation's error probability is equal to $\frac{\varepsilon}{n}$ in this paper.

3.4 Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) [4, 5, 15, 16] allows server to receive encrypted data from client and perform arbitrarily complex computations on that data while it remains encrypted.

Definition 3.2. *A leveled fully homomorphic encryption (FHE) scheme is defined by a quadruple of PPT algorithms FHE = (KGen, Enc, Dec, Eval) as follows:*

- ***Key Generation:*** *$(pk, sk) \leftarrow KGen(1^\kappa, 1^L)$ takes a security parameter κ and a circuit depth upper bound L, and outputs the public key pk and secret key sk.*
- ***Encryption:*** *$\bar{b} \leftarrow Enc_{pk}(b)$ takes as input the public key pk and a message $b \in \{0,1\}$, and outputs a ciphertext $\bar{b}$. For $x \in \{0,1\}^n$, we denote its bit-by-bit encryption $\overline{x[i]} \leftarrow Enc_{pk}(x[i])$ by $\overline{x} = \overline{x[1]}, \ldots, \overline{x[n]}$.*
- ***Decryption:*** *$x \leftarrow Dec_{sk}(\overline{x})$ takes the secret key sk and a ciphertext $\overline{x}$, and outputs a message $x \in \{0,1\}^*$. When $\overline{x}$ is an array of ciphertexts, decryption is ciphertext-by-ciphertext. Correctness says that:*
$$Dec_{sk}\big(Enc_{pk}(x)\big) = x.$$
- ***Evaluation:*** *$\overline{y} \leftarrow Eval_{pk}(f, \overline{x[1]}, \ldots, \overline{x[t]})$ takes as input the pk, a function $f : \{0,1\}^t \rightarrow \{0,1\}$ represented as an arithmetic circuit over $GF(2)$ and a set of t ciphertexts $(\overline{x[i]})_{i=1}^t$ outputs a ciphertext $\overline{y}$ such that $Dec_{sk}(\overline{y}) = f(x[1], \ldots, x[t])$. As a shorthand notation we write $f(\overline{x})$ in place of $Eval_{pk}(f, \overline{x})$.*

3.5 Bootstrapping

The bootstrapping technique introduced by Gentry [16] is a fascinating framework to transfer leveled fully homomorphic encryption into fully homomorphic encryption. All the existing leveled fully homomorphic encryption schemes involve complicated error management, which means after some homomorphic operations, the error in the ciphertext increases rapidly, and will finally destroy the plaintext and incur decryption failure. In order to support arbitrary homomorphic operations, the bootstrapping technique enables to "refresh" the ciphertext into a new one with the same plaintext while reducing the error.

Theoretically, one could refresh the ciphertext after every homomorphic operation (especially multiplication) using bootstrapping to manage the error in the ciphertext. However, bootstrapping is one of the heaviest computations in the FHE scheme, this will dramatically slow down the performance.

This in-time bootstrapping technique is only used in the Full LEAF algorithm. In LEAF$^+$, we will use it in a "lazy" way. This means the bootstrapping technique is only applied when the error will reach the decryption error bound.

3.6 Match Functions

In this section, we list the two most commonly used matching functions as a reference.

3.6.1 Exact Match. The standard equality operator, for $a, b \in \{0,1\}^\mu$, is:

$$\mathsf{IsEqual}_\mu(a,b) = \prod_{i \in [\mu]} (1 + a[i] + b[i]) \mod 2$$

with degree μ and $\mu - 1$ overall multiplications.

It is worth noting that, unlike PIR, *secure search* does **not** require the keyword to be searched to be a *unique identifier*. That is to say, under this setting, the length of the keyword is a constant *independent* of the number of elements n. In contrast, the length of a *unique identifier* is at least $\log n$.

In this paper, we select Exact Match as the matching function, the content to be matched could be the name of the disease (in medical scenario) or company valuation (in business scenario), and the length (μ) of the matching keywords is a *constant* independent of the database's size (n). Therefore, it takes μn multiplications to do an exact match on the entire database.

3.6.2 Greater Than. The greater than operator $(a > b)$, for $a, b \in \{0,1\}^\mu$, is:

$$\begin{aligned}\mathsf{IsGrt}_\mu(a,b) = \sum_{i \in [\mu-1]} &\big((a[i] \cdot (b[i]+1)) \\ &\cdot \mathsf{IsEqual}_{\mu-i}(\mathsf{suffix}_i(a), \mathsf{suffix}_i(b))\big) \\ &+ \big(a[\mu] \cdot (b[\mu]+1)\big) \mod 2\end{aligned}$$

with degree $\mu + 1$ and 2μ overall multiplications.

4 OUR SECURE SEARCH SCHEME (LEAF)

We introduce LEAF in this section. For ease of description, all of our operations are done by default under homomorphic encryption, unless otherwise specified.

Following previous works [1, 2], we mainly focus on locating the first non-zero item, and once it is done, all the matches can be retrieved via similar processing. We defer the details on how to retrieve all records to Appendix A for completeness. Further, we only focus on the algorithm of returning the index i instead of the value array(i). As shown in Section 4.4, with the encrypted index i we just need to pay $O(n)$ multiplications and without incrementing the depth to get the encrypted value.

4.1 Overview

We will briefly introduce the three main steps of the algorithm, i.e., *Localization*, *Extraction*, and *Reconstruction*, following which we obtain an encrypted array with only a single 1 whose coordinate locates the first non-zero term. For completeness, we explain how to get the encrypted binary representation of the non-zero term's index in this encrypted array without increasing the number of multiplications in Section 4.2.4.

In the *Localization* step, our goal is to find the interval containing the first non-zero item. We divide the original array v into t smaller intervals, each of which has length k, we create a new array ind to indicate whether the first i intervals contain a non-zero item, which could be implemented as follows: Let $ind[1] = \text{RS-OR}(v[1], v[2], \ldots, v[k]), ind[2] = \text{RS-OR}(v[1], v[2], \ldots, v[2k])$, and so on. Suppose that the first non-zero item in the original array has coordinate between $(j-1)k+1$ and jk, then we have $ind[1] = ind[2] = \cdots = ind[j-1] = 0$ and $ind[j] = ind[j+1] =$

$\cdots = ind[t] = 1$ for any $j \in [t]$, by computing pairwise differences of adjacent indexes, we can get an array $flag \in \{0,1\}^t$ with only one 1 at $flag[j]$, which indicates that the first non-zero item's index in the original array v is between $(j-1)k+1$ and jk.

In the *Extraction* step, our goal is to extract the interval that contains the first non-zero item. The difficulty of the problem is that although we have coordinates of the target interval, these coordinates are homomorphically encrypted, so we cannot directly use these coordinates to get the target interval. The good news is that we can use PIR to solve this problem without further interaction with the client under this situation, but using PIR will introduce an additional $\log n$ degrees. In this paper, we propose a new technique for extracting complete interval, which only increases the depth by 1 to get the target interval. The basic idea of this technique is to make use of the richer position information in the interval than the encrypted coordinates. More specifically, we can change all the elements in the non-target interval to 0 through a method we put forward. Finally, we add the elements in the corresponding position to get the target interval. See Section 4.2.2 for details.

In the *Reconstruction* step, our goal is to integrate the position information from the two steps above into one final output. The above two steps output the starting coordinate of the target interval (index_1) in which the first non-zero item is located and the offset of the non-zero item within the target interval (index_2), respectively. In theory, we can output the two position information to the client, and then calculate the coordinate of the first non-zero item in the original array by the client after decrypting ($\text{index} = (\text{index}_1 - 1)k + \text{index}_2$). However, we want to output the final result directly for two purposes:

(1) We hope the client only needs to decrypt, thus reducing the requirement of the protocol on the client's computing power;
(2) Output the coordinate of the output non-zero item in the whole array can increase the compatibility of our algorithm, as this output could be adapted to any existing retrieval algorithm.

4.2 Algorithm Description

4.2.1 Localization. The goal of this step is to locate the target item into a smaller interval, then we can ignore non-target intervals and only apply search operation on target interval, thereby reducing the number of RS-OR operations, which is the main source of multiplication operation.

Step 1: The process is shown in Figure 5. We first divide the original array into t smaller intervals, the size of the partition interval is k (determined by the parameter in section 5.2). By calculating RS-OR result of all elements in one chunk, we can determine whether this chunk contains the non-zero element. Specifically, we apply the RS-OR method to the first ik elements in array v and put the result at the position of the i-th element in array ind.

$$ind[i] \leftarrow \text{RS-OR}(v[1], v[2], \ldots, v[i \times k])$$

Depending on the nature of the RS-OR operation, if the i-th element in ind is 1, all elements after this element are 1 (Because the result of 1 OR any number is 1). We don not calculate OR of each interval separately since we want to derive an array with only one 1 indicating the interval where the first non-zero item appears.

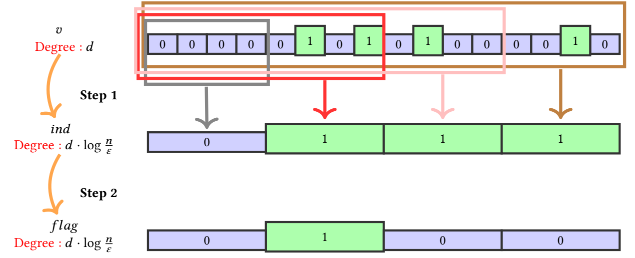


Figure 5: Localization step: divide the array to be searched into many intervals to determine the specific interval in which the first non-zero entry occurs

Step 2: For a stepped array containing only 0s and 1s (they're all 0 before the first 1 and 1 after the first 1), we do the difference operation on the array, that is, we change the value of the i-th element in the array to the value at i minus the value at $i-1$:

$$\forall i \in [2,t] : flag[i] \leftarrow ind[i] - ind[i-1]$$

$$flag[t+1] \leftarrow 1 - ind[t]$$

After this operation, $flag$ only have one 1. If its index is $j (\neq t+1)$, it means that the first 1 is located in $(v[(j-1)\cdot k+1], \ldots, v[j \cdot k])$, if its index is $t+1$, it means there is no 1 in the original array.

Through this process, we locate the position of the first non-zero item in a smaller interval.

4.2.2 Extraction. The purpose of this subroutine is to extract the interval containing the first non-zero element for subsequent search operations on the interval. The reason for this step is that the location information obtained in the previous step is homomorphically encrypted, so we need an extraction method while increasing the computing depth as less as possible.

Step 1: The second part is shown in Figure 6, the result of the previous step is the output of a new array $flag$, this array only contains one 1 represents the interval in which the first non-zero element located, the purpose of step 1 is to use the array $flag$ to build a new array $shield$ of the same length as the original array v, where all the elements in the interval without the first non-zero item are 0, and all the elements in the interval where the first non-zero element appears are 1:

$$\forall j \in [t], \forall i \in [k] \; shield[(j-1)\cdot k + i] \leftarrow flag[j]$$

We call this new array "*shield*", as we'll see later, it acts as a shield.

Step 2: We multiply array $shield$ and initial array v bit by bit:

$$\forall i \in [n], \; v[i] \leftarrow v[i] \cdot shield[i]$$

It is like covering a newspaper with a piece of paper with a hole in it, all we can see is the text under the hole. That is, after this operation, only the interval contained the first 1 in v is not all $0's$.

Step 3: In order to extract the target interval, we define a new array $fin \in \{0,1\}^k$:

$$\forall j \in [k], \; fin[j] \leftarrow v[j] + v[k+j] + \cdots + v[(t-1)\cdot k + j]$$

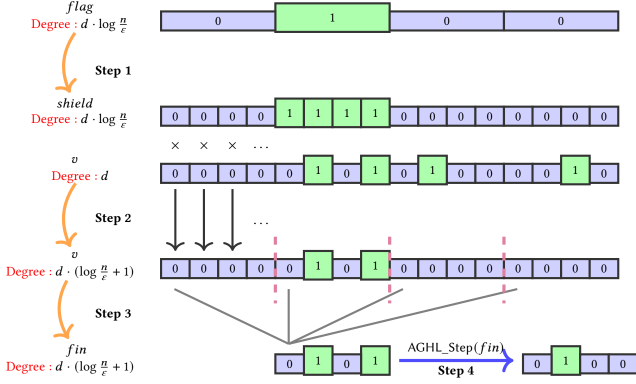


Figure 6: Extraction step: we turn elements in the non-target interval to 0, then add the corresponding position together to obtain the target interval.

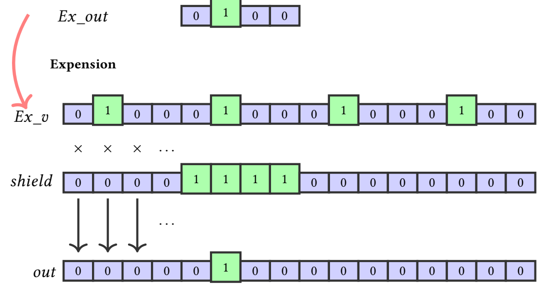


Figure 7: Reconstruction step: we want to get a pulse array with the same length as the original array, while the only 1's index is corresponding to the first non-zero term in the original array.

Adding the elements in the corresponding position of each interval, because all the elements in the interval except the interval with the first 1 are all 0, so the result is the interval with the first non-zero item. So far, we have completed the goal of extracting the target interval.

Step 4: After extracting the target interval, we only need to call AGHL_Step algorithm on this interval to obtain a *pulse array*, then we assign the result to array Ex_out. There are two things we want to point out about AGHL_Step algorithm:

(1) Compared with AGHL algorithm mentioned above, this algorithm (AGHL_Step) eliminates the step of finally converting pulse array into binary representation;
(2) The parameter selection of RS-OR operation in this algorithm is the same as we mentioned in Section 3.3, which are all $N(\varepsilon) = \lceil \log(\frac{n}{\varepsilon}) \rceil$, means that the probability of failure of all RS-OR operations in this paper is $\frac{\varepsilon}{n}$

Compared to the previous algorithm, LEAF algorithm does not need to apply RS-OR on every elements in the array, instead, we utilize RS-OR in a coarse-grained manner, that is, we partition the original array, evaluate only one RS-OR operation in an interval, and replace the result of the whole interval with this result, so as to determine the first non-zero element into a smaller interval with less expensive operations. Although the *localization* and *extraction* steps bring extra time overhead, through choosing suitable parameters, we obtained, by analysis, the ascension of the whole performance of the algorithm will be relatively large. All the way through, we get approximate interval coordinates of the first non-zero element and the offset within that interval. In the *Reconstruction* step, we'll explain how to build the final result with the two position information.

4.2.3 *Reconstruction.*

In this step, our goal is to integrate the position information from the two steps above into one final output. The process is shown in Figure 7.

Expansion: In this step, we extend the pulse array (Ex_out) obtained by AGHL_Step algorithm to an array with the same length as the original array (v), $\forall i \in [k]$:

$$Ex_v[i] = \cdots = Ex_v[i + (t-1)k] = Ex_out[i]$$

Integration: We multiply the expanded array Ex_v by the array *shield* we got from the previous section to get a new array *out*, $\forall i \in [n]$:

$$out[i] = Ex_v[i] \times shield[i]$$

Since the array *shield* is only non-zero in the target interval, the result of the multiplication is the final reconstructed array.

4.2.4 *Transform array to number.*

This subroutine takes as input an encrypted array, which consists of a single 1 and multiple 0s for the rest, and outputs the ciphertext of the index of the non-zero term in the array.

As a slightly abuse of notation, $out \in \{\overline{0}, \overline{1}\}^n$ represents a vector of encrypted bits as input, where $\overline{x}$ denotes x's ciphertext. We obtain the binary representation of the index (of the non-zero term in *out*) in encrypted form, i.e., by outputing $B \cdot out$, where $B \in \{0,1\}^{\lceil \log(n+1) \rceil \times n}$ is a $\lceil \log(n+1) \rceil \times n$ matrix whose every i-th column is the binary representation of integer i in cleartext.

While the process seemingly involves multiplications, it actually requires only homomorphic additions since B consists of 0-1 valued entries in plaintext, and therefore multiplying B with a vector *out* of encrypted entries first selects the corresponding ciphertexts in *out* (based on the values in B) and then outputs their encrypted sum using homomorphic additions. We provide an example below for $n = 4$ and thus B is a 3×4 matrix and we let the encrypted vector $out = (\overline{0}, \overline{1}, \overline{0}, \overline{0})^T$. Multiplying B and *out* yields:

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \overline{0} \\ \overline{1} \\ \overline{0} \\ \overline{0} \end{pmatrix} = \begin{pmatrix} \overline{0} \\ \overline{1} + \overline{0} \\ \overline{0} + \overline{0} \end{pmatrix} = \begin{pmatrix} \overline{0} \\ \overline{1} \\ \overline{0} \end{pmatrix},$$

where the result $(\overline{0}, \overline{1}, \overline{0})^T$ corresponds to binary representation of 2 (i.e., the non-zero term's index in *out*).

Core Idea of Our Algorithm: In our algorithm, we use the *Localization* technique to narrow down the scope of the first 1 entity, then use the *Extraction* technique to extract the desired interval, so that we do not need to apply expansive operations on non-target

Algorithm 1 LEAF

1: **Input**: Array $v \in \{0,1\}^n$;
2: **Output**: $i^* = min\{i \in [n] \mid v[i] == 1\}$;
3: Set $t = \lfloor \sqrt{n} \rfloor$, $k = \lceil \frac{n}{t} \rceil$, pad the input array by $t \times k - n$ zero entries;
4: Create a new array $ind \in \{0,1\}^t$, $\forall i \in [t]$, set $ind[i] \leftarrow$ RS-OR$(v[1], v[2], \ldots, v[i \times k])$;
5: Create a new array $flag \in \{0,1\}^t$, $\forall i \in [2,t]$: $flag[i] \leftarrow ind[i] - ind[i-1], flag[t+1] \leftarrow 1 - ind[t]$;
6: Create a new array $shield \in \{0,1\}^n$: $\forall j \in [t], \forall i \in [k]$ $shield[(j-1) \cdot k + i] \leftarrow flag[j]$;
7: $\forall i \in [n]$, $v[i] \leftarrow v[i] \times shield[i]$;
8: Create a new array $fin \in \{0,1\}^k$: $\forall j \in [k]$, $fin[j] \leftarrow v[j] + v[k+j] + \cdots + v[(t-1) \times k + j]$;
9: $Ex_out \leftarrow$ AGHL_Step(fin);
10: Create a new array $Ex_v \in \{0,1\}^n$: $\forall i \in [k]$, $Ex_v[i] = Ex_v[i+k] = \cdots = Ex_v[i+(t-1)k] = Ex_out[i]$;
11: Create a new array $out \in \{0,1\}^n$: $\forall i \in [n]$, $out[i] \leftarrow Ex_v[i] \times shield[i]$;
12: index $\leftarrow B \cdot out$, $for\ B \in \{0,1\}^{\lceil \log(n+1) \rceil \times n}$;
13: **return** index;

intervals, significantly reducing the number of multiplication operations. Finally, we use *Reconstruction* technique to construct the final output based on the two position information.

The reason why we need to use new techniques to extract the target interval instead of directly using array index is based on the fact that the coordinates representing the general range are homomorphically encrypted, and the server cannot get the general position of the target items.

Combining the above operations, we get Algorithm 1 (The selection of t, k is detailed in Section 5.2).

4.3 Correctness of LEAF Algorithm

In this paper, a low-degree approximation method for OR called RS-OR is adapted to construct our algorithm, this method can significantly reduce the degree of the algorithm, but at the same time introduce some error probability. We will prove that the error probability of our algorithm can be controlled at a very low level.

As mentioned above, all RS-OR operations in this paper select the same parameter $N(\varepsilon) = \lceil \log(\frac{n}{\varepsilon}) \rceil$, according to the proof in Section 3.3, all RS-OR operations with this parameter has the same correct probability which is $1 - \frac{\varepsilon}{n}$.

In the *Localization* step, we divide the original array into t intervals of equal length, so in this step, we calculate RS-OR for a total of t times. In our *Extraction* step, we performed RS-OR for each element in the target interval with length k, so we calculated a total of k times of RS-OR operations in this step. In addition, other parts of our algorithm did not use RS-OR operation, so our algorithm used $t + k$ times of RS-OR operation, if these operations do not occur any error, our algorithm will be correct.

Based on the conclusion in Section 5.2, we select the parameter $t = k = \sqrt{n}$ in this paper, so the probability of our algorithm being correct is:

$$Pr_{\mathsf{LEAF}} = \left(1 - \frac{\varepsilon}{n}\right)^{t+k} = \left(1 - \frac{\varepsilon}{n}\right)^{2\sqrt{n}}$$

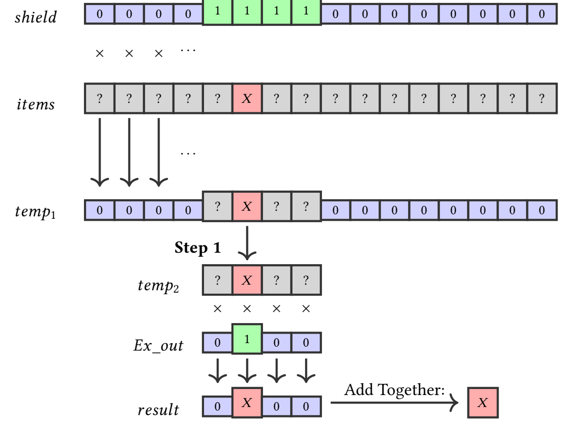


Figure 8: Our Retrieval Process, we don't apply Akavia's method since it will add 1 additional depth.

Lemma 4.1. $(1 - \frac{\varepsilon}{n})^n \geq 1 - \varepsilon$, $\forall n \in \mathbb{N}^+$, $\varepsilon < 1$, *the equal sign holds if and only if* $n = 1$.

Proof. We defer the concrete proof to Appendix C. □

For any $n > 4$, we have $2\sqrt{n} < n$, which means:

$$Pr_{\mathsf{LEAF}} = \left(1 - \frac{\varepsilon}{n}\right)^{2\sqrt{n}} > \left(1 - \frac{\varepsilon}{n}\right)^{n} > 1 - \varepsilon$$

Here, ε is the parameter that can be selected by the client, normally we will choose $\varepsilon = 2^{-80}$, the above equation illustrates that when $n > 4$, our algorithm's output is correct with overwhelming probability.

4.4 Retrieving Matches

Here we show how to retrieve the matched item given the encrypted *pluse array* without further interaction with clients. For the convenience of description, we describe the process of retrieving 1-bit content. For arbitrary long content, we can retrieve each bit in the same way.

Once we get the encrypted index of the target item, PIR would be a feasible solution to retrieve the desired item without further interactions at the cost of an extra degree of $O(\log n)$. Akavia et al. [2] proposed a depth-preserving method to retrieve matched item using $O(n)$ multiplications and increasing the depth by 1.

In this paper, by rearranging the multiplication order, we can complete the retrieval of matching items without increasing the calculation depth, and ensure that the required number of multiplication is the same as Akavia's algorithm required, both of which are $O(n)$, as shown in Figure 8.

It is worth pointing out that the **Step 1** in the figure is very similar to the **Step 3** we mentioned in our *Extraction* step, which is to extract non-zero intervals in an array by adding them, to be specific, define $temp_2 \in \{0,1\}^k$, $\forall j \in [k]$:

$$temp_2[j] \leftarrow temp_1[j] + temp_1[k+j] + \cdots + temp_1[(t-1) \cdot k + j]$$

Instead of manipulating the final output in the searching part, we utilize location information obtained from the intermediate process, which enables us to retrieve the target item without adding depth.

5 EFFICIENCY ANALYSIS

5.1 Efficiency Estimate

Since multiplication takes much more time than addition, so in our analysis, we only consider multiplication. Assuming each multiplication operation costs T_{MUL_1}.

First, we use RS-OR t times to calculate the OR results. In Section 3.3, we know that each RS-OR is actually equivalent to doing normal OR operations on $\log(\frac{n}{\varepsilon})$ elements, so each operation costs $\log\frac{n}{\varepsilon}$ times multiplication, thus this step costs:

$$t \cdot \log(\frac{n}{\varepsilon}) \times T_{\mathsf{MUL}_1}$$

Then we calculate the inner product of v and *shield*, because v and *shield* have n elements, this step involves multiplying each pair of elements in the two arrays, so it contains n multiplication operations, this step costs:

$$n \times T_{\mathsf{MUL}_1}$$

Then we apply AGHL_Step algorithm to calculate the *stepped array* fin, it contains $k \cdot \log(\frac{n}{\varepsilon})$ times multiplication (AGHL_Step needs k times of RS-OR which requires $k \cdot \log(\frac{n}{\varepsilon})$ times of multiplication), so this step costs:

$$k \cdot \log(\frac{n}{\varepsilon}) \times T_{\mathsf{MUL}_1}$$

After obtaining two position information, we use *Reconstruction* method to reconstruct the output, it contains n times of multiplication, so this step costs:

$$n \times T_{\mathsf{MUL}_1}$$

And we have to add the time match procedure costs, assume it contains MOPN times of multiplication, so it costs:

$$\mathsf{MOPN} \times T_{\mathsf{MUL}_1}$$

so, the overall time costs is:

$$T_{\mathsf{LEAF}} = \left(t \cdot \log(\frac{n}{\varepsilon}) + 2n + k \cdot \log(\frac{n}{\varepsilon}) + \mathsf{MOPN}\right) \times T_{\mathsf{MUL}_1}$$

5.2 Concrete Parameters

To find the optimal parameters, we take the partial differential of the total time expression T_{LEAF} with respect to t, since MOPN, T and n is not a function of t, $k = \frac{n}{t}$, so the target expression is equal to:

$$\frac{\partial\left(t \cdot \log(\frac{n}{\varepsilon}) + \frac{n}{t} \cdot \log(\frac{n}{\varepsilon})\right)}{\partial t} = 0$$

we have:

$$t = \sqrt{n}$$

So we get that the number of multiplications required by the algorithm, it should be noted that since the matching algorithm needs to check every element in the database, the MOPN term is *linearly* dependent with the amount of data (n), that is, $\mathsf{MOPN} = \mu n = O(n)$, as stated in Section 3.6.1. Define OPN as total number of multiplications required by the protocol:

$$\begin{aligned}\mathsf{OPN} &= t \cdot \log(\frac{n}{\varepsilon}) + 2n + k \cdot \log(\frac{n}{\varepsilon}) + \mathsf{MOPN} \\ &= \sqrt{n} \cdot \log(\frac{n}{\varepsilon}) + 2n + \sqrt{n} \cdot \log(\frac{\sqrt{n}}{\varepsilon}) + \mathsf{MOPN} \\ &= O(n)\end{aligned}$$

The above formula indicates that the number of multiplications required by our algorithm has reached the optimal theoretical bound in the asymptotic sense, because even if we retrieve in plaintext case, at least $O(n)$ operations are still needed.

The depth of this algorithm is:

$$\begin{aligned}d_{\mathsf{LEAF}} &= \log d + \log\log(\frac{n}{\varepsilon}) + 1 + \log\log(\frac{n}{\varepsilon}) + 1 \\ &= O(\log\log n)\end{aligned}$$

The first term in the above formula is required by matching step, the second one is attributed to *Localization* step, and the third is due to *Extraction* step. The last two terms are the depths required to operate on the extracted interval. To be specific, the fifth term is corresponding to *Reconstruction*, while the fourth term is equivalent to replacing the n in the first term with t and removing the $\log d$, because the depth required by the matching algorithm is not required when operating on the target interval.

We recall the depth of AGHL algorithm for comparison:

$$d_{\mathsf{AGHL}} = \log d + \log\log(\frac{n}{\varepsilon}) = O(\log\log n)$$

It's shown that the algorithm does not increase the computational depth in the asymptotic sense, therefore, when n is large, the time required for the single multiplication of the two algorithms (T_{MUL_2} for AGHL and T_{MUL_1} for LEAF) becomes a constant ratio, according to the estimation formula in Gentry et al.'s work[17]:

$$\frac{T_{\mathsf{MUL}_2}}{T_{\mathsf{MUL}_1}} = (\frac{d_{\mathsf{AGHL}}}{d_{\mathsf{LEAF}}})^{\omega} = (\frac{\log\left(d\log(\frac{n}{\varepsilon})\right)}{\log\left(d\log(\frac{n}{\varepsilon})\right) + \log\log(\frac{n}{\varepsilon}) + 2})^{\omega}$$

observe that:

$$\begin{aligned}\lim_{n\to+\infty}\frac{T_{\mathsf{MUL}_2}}{T_{\mathsf{MUL}_1}} &= \lim_{n\to+\infty}(\frac{\log\left(d\log(\frac{n}{\varepsilon})\right)}{\log\left(d\log(\frac{n}{\varepsilon})\right) + \log\log(\frac{n}{\varepsilon}) + 2})^{\omega} \\ &= (\frac{1}{2})^{\omega} = O(1)\end{aligned}$$

where $\omega < 2.3727$ is the matrix multiplication exponent, therefore, by combining the multiplication number required by AGHL algorithm: $n \cdot \log(\frac{n}{\varepsilon})$, we can get the time required by AGHL algorithm:

$$T_{\mathsf{AGHL}} = n \cdot \log(\frac{n}{\varepsilon}) \times T_{\mathsf{MUL}_2} = O(n\log n) \times T_{\mathsf{MUL}_2}$$

thus we have:

$$\frac{T_{\mathsf{AGHL}}}{T_{\mathsf{LEAF}}} = \frac{O(n\log n)}{O(n)} \times \frac{T_{\mathsf{MUL}_2}}{T_{\mathsf{MUL}_1}} = O(\log n) \times O(1) = O(\log n)$$

It turns out that our algorithm LEAF reduced the AGHL algorithm by a log order of magnitude, mainly because we reduced the number of multiplications from $O(n\log n)$ to $O(n)$ while keeping the required computational depth asymptotically constant.

6 PROTOCOL WITH BOOTSTRAPPING LAZILY(LEAF^{+})

Compared with our algorithm LEAF, LEAF^{+} applies bootstrapping **only** to the elements in the extracted interval, so as to control the growth of the depth of computation, which leads to different effects:

- **Pros:** The bootstrapping step can control the computational depth required by the algorithm, and the optimization effect of the algorithm will be better when n is large;

- **Cons:** The introduction of bootstrapping step will bring about a large number of extra multiplication operations and computation depth, which will even make the efficiency of the algorithm lower than before when n is small.

After bootstrapping was introduced, although the computation depth of the algorithm did not change, the hidden constant in $O(\log\log n)$ became smaller, so when n is very large, the single multiplication time costs in this algorithm will be the same as in AGHL, which was different from LEAF algorithm.

At the same time, since we only need to do the bootstrapping step once for the extracted interval elements rather than for all the elements, this significantly reduces the extra time cost brought by the bootstrapping step.

With proper parameter analysis and selection, we could use the bootstrapping technology to bring benefits and lower down the consequent disadvantages.

6.1 Efficiency Estimate

Similar to the analysis in the previous section and consider the extra time bootstrapping takes, we could get the time that LEAF$^+$ algorithm takes:

$$T_{\mathsf{LEAF^+}} = \left(t \cdot \log(\frac{n}{\varepsilon}) + 2n + k \cdot \log(\frac{n}{\varepsilon}) + \mathsf{MOPN}\right) \cdot T_{\mathsf{MUL}_3} + t \cdot T_{\mathsf{BOO}}$$

where T_{MUL_3} represents the time taken for each homomorphic multiplication, T_{BOO} is the time required for single bootstrapping, define:

$$\alpha = \frac{T_{\mathsf{BOO}}}{T_{\mathsf{MUL}_3}}$$

which means the number of multiplication operations boostrapping procedure needs, we have:

$$T_{\mathsf{LEAF^+}} = \left((t \cdot \log(\frac{n}{\varepsilon}) + 2n + k \cdot \log(\frac{n}{\varepsilon}) + \mathsf{MOPN}) + \alpha t\right) \times T_{\mathsf{MUL}_3}$$

where $k = \frac{n}{t}$, T_{MUL_3} is not a function of t, according to Jung Hee Cheon et.al.'s work [8]: $\alpha = O(\log^2 \lambda)$, where λ is security parameter, independent of n, we take the partial with respect to t to get the optimal solution:

$$\frac{\partial\left((t \cdot \log(\frac{n}{\varepsilon}) + 2n + k \cdot \log(\frac{n}{\varepsilon}) + \mathsf{MOPN}) + \alpha t\right)}{\partial t} = 0$$

Since MOPN and n are not functions of t, we have:

$$\frac{\partial\left(t \cdot \log(\frac{n}{\varepsilon}) + \frac{n}{t} \cdot \log(\frac{n}{\varepsilon}) + \alpha t\right)}{\partial t} = 0$$

then we get the solution:

$$t = \sqrt{\frac{n}{1+\alpha}}$$

thus we have:

$$T_{\mathsf{LEAF^+}} = \left((2\sqrt{\frac{n}{1+\alpha}} \cdot \log(\frac{n}{\varepsilon}) + 2n + \mathsf{MOPN}) + \alpha\sqrt{n}\right) \times T_{\mathsf{MUL}_3} = O(n) \times T_{\mathsf{MUL}_3}$$

According to Chen et al.'s [6], bootstrapping requires depth $d_{\mathsf{BOO}} = \log(z) + \log(h)$ for BGV [4] and $d_{\mathsf{BOO}} = \log\log(z) + \log(h)$ for FV [15], where $h = ||s||_1$ is the *1-norm* of the secret key, and $z = p^r$ is the plaintext modulus. Thus, the depth of our protocol is :

$$d_{\mathsf{LEAF^+}} = \log d + \log\log(\frac{n}{\varepsilon}) + 1 + d_{\mathsf{BOO}} = O(\log\log n) \ ,$$

where the 1 term accounts for the depth of retrieval. Similarly,

$$\frac{T_{\mathsf{MUL}_2}}{T_{\mathsf{MUL}_3}} = (\frac{d_{\mathsf{AGHL}}}{d_{\mathsf{LEAF^+}}})^{\omega} = (\frac{\log\left(d\log(\frac{n}{\varepsilon})\right) + 1}{d_{BOO} + \log\left(d\log(\frac{n}{\varepsilon})\right) + 1})^{\omega} \ ,$$

where $\omega < 2.3727$ is the matrix multiplication exponent. After initializing the parameters, d_{BOO} does not change with n, therefore:

$$\lim_{n\to+\infty} \frac{T_{\mathsf{MUL}_2}}{T_{\mathsf{MUL}_3}} = \lim_{n\to+\infty} (\frac{\log\left(d\log(\frac{n}{\varepsilon})\right) + 1}{d_{\mathsf{BOO}} + \log\left(d\log(\frac{n}{\varepsilon})\right) + 1})^{\omega} = 1$$

that is, when n gets larger, the time multiplication needed of two protocol is approximately the same, which is why LEAF$^+$ algorithm performs better when n is large.

7 EVALUATION

In this section, we empirically evaluate our algorithm.

Experimental Setup. We implement the secure search algorithm based on the open source homomorphic encryption library SEAL (version 3.4.5) [27]. SEAL is a widely used HE library written in C++ and it implements two HE schemes, i.e., BFV [15] and CKKS [9].

We use the BFV implementation since CKKS works better with fixed-point arithmetics while we only need Boolean operations for our search algorithm. We run our algorithms on a server equipped with Debian 10 (Buster) and Intel(R) Xeon(R) CPU E7-8867 v3 @ 2.50GHz, 1, 536 GigaByte of RAM. The state-of-the-art AGHL algorithm [2] is used as our baseline model. For fairness, no optimization technique is used for these two algorithms, like "batching".

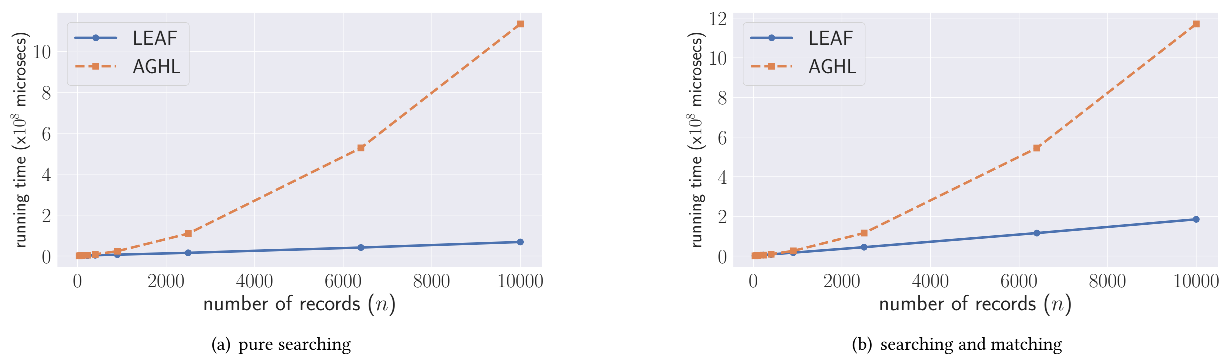
Results. Figure 9 shows the results. Since our algorithm mainly improves the searching part of the whole protocol, we conduct our experiments in two scenarios. First, we measure the time cost of the searching process alone, i.e., without matching, as shown in Figure 9(a), this could give us a direct impression of how our improvement works on the searching part. Second, we add the matching part back and see the total improvement, the result is shown in Figure 9(b). According to the experimental results, we conclude that improving the searching process indeed significantly accelerates the whole protocol. Note that in both scenarios, we set error rate $\varepsilon = 2^{-80}$, and we assume the keyword's length is $\mu = 16$-bits.

As shown by our formal analysis, our algorithm requires around double depth compared to AGHL, this makes our multiplications slower. According to our evaluation, the time cost per multiplication operation for our algorithm is around 4.17 times higher than AGHL. On the other hand, our algorithm requires a much less number of operations. Specifically, our algorithm only requires $2\sqrt{n}$ times RS-OR operation while AGHL requires n times RS-OR operation. This drives our algorithm much faster than AGHL in general.

The searching algorithm alone starts to perform better than AGHL when n is greater than 80, and the entire protocol outperforms the previous algorithm when $n \geq 400$. According to our formal analysis in Section 5.2, the advantage of our algorithm will increase further with the increase of the number of records n, not only for the searching part but also for the entire protocol.

8 FURTHER OPTIMIZATION

Single instruction multiple data (SIMD) is an optimization technique proposed by Smart-Vercauteren[28], this technique allows us



(a) pure searching

(b) searching and matching

Figure 9: Server's running time as a function of the number of records n. In the left figure, we show the time cost of the search process, and in the right one, we depict the time consumption of the whole secure search, i.e., searching and matching. The results show our work could achieve better efficiency even for a small number of records.

to pack many plaintext elements in a single ciphertext and apply operations to them at the same time. Plaintext values in a single ciphertext are referred to as "plaintext slots" of that ciphertext.

We can use SIMD technology as an optimization tool, it is worth noting that the selection of parameters will be a little different after using this method. Since SIMD technique operates on the elements in the slot simultaneously, we should choose the appropriate parameters so that the technique can be better combined with the size of the segments we are dividing.

Therefore, when using this optimization technique, further parameter analysis is very important. We need to carry out detailed analysis to weigh the multiple parallelism caused by more slots against the single operation inefficiencies caused by more slots.

9 RELATED WORK

Secure Pattern Matching (SPM): The task of SPM is to determine whether the plaintext corresponding to the two encrypted ciphertext meets certain conditions. Specifically, given an encrypted lookup value, it returns a vector of n ciphertexts $(c_1, \ldots, c_n)$, where c_i indicates whether the ith data element is a match to the lookup value (or sometimes returning only a YES/NO answer of whether a match exists). There are many works about SPM on FHE encrypted data, see [11, 12, 20, 21, 23, 31, 33] for details. The main drawback of these protocols is that the communication complexity and client's running time are proportional to the number of stored elements.

Private Information Retrieval (PIR): PIR is a useful protocol to retrieve at most a single record x_i in the encrypted array (as in SQL UNIQUE constraint), it provides a restricted search functionality, where the client's lookup value must be a *unique identier*. Low degree polynomials realizing secure data retrieval for these unique identifier settings have been shown in prior works (see [5, 14, 16]). We note that the server's run-time in a single server PIR (whether or not FHE based) is inherently linear in the size of dataset (n). This protocol provides only a restricted search functionality, which is incompatible in our setting.

Private Set Intersection (PSI): PSI refers to a setting where two parties each hold a set of private items and wish to learn the intersection of their sets without revealing any additional information except for the intersection itself. The most efficient works are shown in [7, 22, 24, 25], however, the protocol is inefficient in the sense of its communication complexity is at least linear dependence on the smaller database size (n).

Searchable Encryption (SE): SE enables highly efficient search over encrypted data. Specifically, SE focuses on achieving sublinear search time. There are two main primitives for searchable encryption: searchable symmetric encryption (SSE) [30] and public key encryption with keyword search (PEKS) [3]. However, the security is weakened to leak vital search information, like access pattern.

10 CONCLUSION

In this paper, we propose an efficient algorithm – LEAF (and its variants LEAF$^+$) – with low communication complexity for FHE-based secure search. Our scheme relies on three novel techniques including localization, extraction, and reconstruction. LEAF only requires the client to encrypt the lookup value and upload it to the server, and the server will return the encrypted coordinates corresponding to the matching items in the encrypted database. In the whole process, the client only needs to encrypt the lookup value and decrypt the output, which enables our algorithm to be deployed on weak-power devices and embedded systems.

The security of the protocol is guaranteed by the semantic security feature of FHE. The server could only access the encrypted data in the whole process, so the data privacy is enhanced. LEAF can be performed over $GF(2)$, which fits all current homomorphic encryption algorithm. Meanwhile, our algorithm can support unrestricted search function, which greatly expands its application scenarios.

ACKNOWLEDGEMENT

Yu Yu was supported by the National Key Research and Development Program of China (Grant No. 2018YFA0704701), National Natural Science Foundation of China (Grant No. 61872236 and 61971192), the National Cryptography Development Fund (Grant No. MMJJ20170209), and the Major Program of Guangdong Basic and Applied Research (Grant No. 2019B030302008).

REFERENCES

[1] Adi Akavia, Dan Feldman, and Hayim Shaul. 2018. Secure Search on Encrypted Data via Multi-Ring Sketch. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 985–1001.

[2] Adi Akavia, Craig Gentry, Shai Halevi, and Max Leibovich. 2019. Setup-Free Secure Search on Encrypted Data: Faster and Post-Processing Free. *Symposium on Privacy Enhancing Technologies Symposium* (2019).

[3] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. 2004. Public Key Encryption with Keyword Search. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, 506–522.

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory* (2014).

[5] Zvika Brakerski and Vinod Vaikuntanathan. 2014. Efficient Fully Homomorphic Encryption from (Standard) LWE. *SIAM Journal on Computing* (2014).

[6] Hao Chen and Kyoohyung Han. 2018. Homomorphic Lower Digits Removal and Improved FHE Bootstrapping. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, 315–337.

[7] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 1243–1255.

[8] Jung Hee Cheon, Kyoohyung Han, and Duhyeong Kim. 2017. Faster Bootstrapping of FHE over the Integers. *Cryptology ePrint Archive, Report 2017/079* (2017).

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, 409–437.

[10] Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, and Keewoo Lee. 2019. Numerical Method for Comparison on Homomorphically Encrypted Numbers. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, 415–445.

[11] Jung Hee Cheon, Miran Kim, and Myungsun Kim. 2016. Optimized Search-and-Compute Circuits and Their Application to Query Evaluation on Encrypted Data. *IEEE Transactions on Information Forensics and Security* (2016).

[12] Jung Hee Cheon, Miran Kim, and Kristin Lauter. 2015. Homomorphic Computation of Edit Distance. In *International Conference on Financial Cryptography and Data Security (FC)*. Springer, 194–212.

[13] Jack L. H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. 2018. Doing Real Work with FHE: The Case of Logistic Regression. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*. ACM, 1–12.

[14] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. 2014. Bandwidth Efficient PIR from NTRU. In *International Conference on Financial Cryptography and Data Security (FC)*. Springer, 195–207.

[15] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *Cryptology ePrint Archive, Report 2012/144* (2012).

[16] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 169–178.

[17] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Annual International Cryptology Conference (CRYPTO)*. Springer, 75–92.

[18] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play Any Mental Game. In *Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 218–229.

[19] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. 2019. Efficient and Secure Multiparty Computation from Fixed-Key Block Ciphers. *Cryptology ePrint Archive, Report 2019/074* (2019).

[20] Myungsun Kim, Hyung Tae Lee, San Ling, Shu Qin Ren, Benjamin Hong Meng Tan, and Huaxiong Wang. 2016. Better Security for Queries on Encrypted Databases. *Cryptology ePrint Archive, Report 2016/470* (2016).

[21] Myungsun Kim, Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang. 2017. Private Compound Wildcard Queries Using Fully Homomorphic Encryption. *IEEE Transactions on Dependable and Secure Computing* (2017).

[22] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 818–829.

[23] Kristin Lauter, Adriana López-Alt, and Michael Naehrig. 2014. Private Computation on Encrypted Genomic Data. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*. Springer, 3–27.

[24] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2019. SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension. In *Annual International Cryptology Conference (CRYPTO)*. Springer, 401–431.

[25] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2018. Scalable Private Set Intersection Based on OT Extension. *ACM Transactions on Privacy and Security* (2018).

[26] A. A. Razborov. 1987. Lower Bounds on the Size of Bounded Depth Circuits over a Complete Basis with Logical Addition. *Mathematical notes of the Academy of Sciences of the USSR* (1987).

[27] SEAL 2020. Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[28] Nigel P Smart and Frederik Vercauteren. 2014. Fully Homomorphic SIMD Operations. *Designs, Codes and Cryptography* (2014).

[29] Roman Smolensky. 1987. Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity. In *Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 77–82.

[30] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 44–55.

[31] Haixu Tang, Xiaoqian Jiang, Xiaofeng Wang, Shuang Wang, Heidi Sofia, Dov Fox, Kristin Lauter, Bradley Malin, Amalio Telenti, Li Xiong, et al. 2016. Protecting genomic data analytics in the cloud: state of the art and opportunities. *BMC medical genomics* (2016).

[32] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 162–167.

[33] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. 2013. Secure Pattern Matching Using Somewhat Homomorphic Encryption. In *ACM Cloud Computing Security Workshop (CCSW)*. ACM, 65–76.

Appendices

A SEQUENTIAL RETRIEVAL

In the main body, we provide a faster secure search algorithm that takes as input an encrypted array, consisting of encrypted 0s and 1s, and produces as output the first 1's index and its corresponding item. For completeness, we recall from Akavia et al. [2] how to extend secure search functionality to return the rest matching elements.

Assume that we have obtained the first match's index, say i, and want to retrieve the second matching. The intuition is that once we set the first non-zero term to 0, the second non-zero term now becomes the first in the original encrypted array, so one just applies the secure search algorithm and repeats the above process.

It's not very hard to set the first non-zero term in the original array to 0, since we obtain this encrypted binary array by applying the matching method on each item in the database, we can change the original matching criteria by adding additional requirement, i.e., output 1 if and only if the item in the database meets the following two conditions simultaneously:

- The item satisfies the original matching condition;
- The item's index is greater than i (the first matching's index);

Then, apply this new matching method to the database to obtain a new encrypted binary array, which differs only in that the first non-zero term in the original array is set to 0. More details are found in Akavia et al.'s work [2], which takes the same time complexity ($O(n)$) as a normal exact matching in the asymptotic sense.

To our best knowledge, retrieval following matching needs further interaction as we have to send the encrypted index i to the server and need to rerun the matching and searching algorithms, which is time-consuming. However, at the same time it makes adversary infeasible to find out the number of matchings in the database, since the client can choose whether to continue fetching the next matching or not, which enhances the privacy.

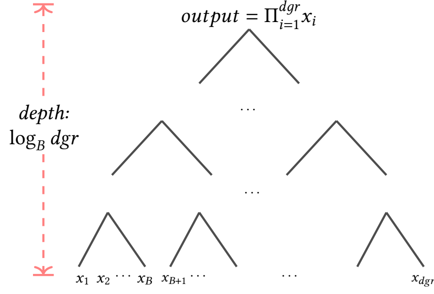


Figure 10: An example that a polynomial of degree *dgr*, e.g., $\Pi_{i=1}^{dgr} x_i$ can be implemented by a balanced B-ary tree for minimized depth $\approx \log_B$ *dgr*.

B MULTIPLICATION STRUCTURE

C PROOF OF LEMMA 4.1

Lemma. *$(1 - \frac{\varepsilon}{n})^n \geq 1 - \varepsilon, \forall n \in \mathbb{N}^+, \varepsilon < 1$, the equal sign holds if and only if $n = 1$.*

Proof. Define

$$f(x) = (1 - \frac{\varepsilon}{x})^x$$

Take the derivative with respect to x, define the derivative function as $g(x)$:

$$g(x) = \frac{df(x)}{dx} = (1 - \frac{\varepsilon}{x})^x (\frac{\varepsilon}{x(1 - \frac{\varepsilon}{x})} + \ln(1 - \frac{\varepsilon}{x}))$$

Take the derivative with respect to x:

$$\frac{dg(x)}{dx} = -\frac{\varepsilon^2}{x^3(1 - \frac{\varepsilon}{x})^2} < 0$$

which means that $g(x)$ is a monotonically decreasing function, so we have:

$$g(x) > g(+\infty) = 0$$

which means that $f(x)$ is a monotonically increasing function, so for $n \geq 1$, we have:

$$f(n) \geq f(1) = 1 - \varepsilon$$

that is, if $n \geq 1$, we have:

$$(1 - \frac{\varepsilon}{n})^n \geq 1 - \varepsilon$$

□