HElix: Genome Similarity Detection in the Encrypted Domain

Rostin Shokri, Charles Gouert, and Nektarios Georgios Tsoutsos

University of Delaware
{rostinsh, cgouert, tsoutsos}@udel.edu

Abstract. As the field of genomics continues to expand and more sequencing data is gathered, genome analysis becomes increasingly relevant for many users. For example, a common scenario involves users trying to determine if collected DNA samples are similar to DNA sequences hosted in a remote repository, such as checking for disease mutations in patients with monogenic disorders. Nevertheless, end users may be reluctant to upload sensitive DNA sequences, which might reveal private biological information, while the owners of remote genomics repositories are unwilling to share their databases. To address this challenge, we propose a novel Bloom filter approach based on fully homomorphic encryption to preserve the privacy of the genomic data and enable queries directly on ciphertexts. We utilize the Circuit Bootstrapping and CMUX functionality in TFHE to implement the homomorphic indexing required in Bloom filters. Our EDA-inspired techniques can privately determine if a client's DNA sequence exists in a database. We validate our approach across various database sizes and compare directly with related works. We observe up to 4 orders of magnitude faster server evaluation times and many orders of magnitude smaller communication overhead for our methodology.

Keywords: Bloom filters, Homomorphic encryption, Private genome association, TFHE.

1 Introduction

Data privacy has become a great concern as cloud computing usage keeps growing, and both end users and cloud providers strive to preserve the confidentiality of their data. In many scenarios, both parties would like to have a computation performed across their joint inputs without leaking any information about their respective data. Doing so in the clear (on plaintexts) exposes at least one of the parties' data (i.e., the computing party can see all the data). A common scenario to consider is one where the end-user (client) wants to check if their data exists in a cloud provider's database (server); for example, in genome analysis applications, a doctor wants to verify if a patient's DNA sample is present in existing databases [31]. Additional use cases include image matching, where the

server has a list of illegal images and checks if a target user image is in that list [1], or plagiarism detection in the context of academic institutions [29].

Many different privacy-preserving solutions, such as multi-party computation (MPC) [18], utilize advanced cryptographic primitives to help secure data while processing. However, in our client-server model, MPC is not a well-suited solution because it requires the client to be involved in the computation process, and this is often infeasible for resource-constrained clients. In this case, the data can be divided across multiple servers with the critical assumption that the servers are non-colluding. In some cases, a trusted third party would perform the correct computations while maintaining data privacy.

With all this in mind, a promising solution to address these challenges with a stronger threat model (in a semi-honest setting) is a cryptographic primitive called fully homomorphic encryption (FHE) [22], which allows an unlimited amount of arbitrary computations directly on encrypted data. FHE guarantees that the computations on the encrypted data will not leak any information about the plaintext, and the results always remain decryptable with the secret key. Depending on the FHE scheme, the user can encrypt integers, bits, or floating point numbers, and the ciphertexts are *malleable*.

What makes FHE an excellent candidate is that all the computation required is offloaded to the cloud server, and it does not require any online communication and computation from the client. An important application of FHE and other privacy-preserving techniques is to find similar items or the amount of similarity between two items while maintaining the privacy of the data [13]. For example, in genome analysis, very large data samples, such as DNA and RNA, exist in public databases, and a lot of research has been conducted [39,26] towards finding similarities between samples so we can gain information and find patterns between different organisms. A simple example is ancestry, which aims to find the closest match to an input DNA sample from variations in DNA sequences (such as single-nucleotide polymorphisms).

With the large increase in genomic data and analysis in recent years, resource-restricted clients face significant challenges in downloading large databases and performing genome analysis due to the sheer size of DNA data. Many genome analysis applications have therefore been moved to the cloud, which introduces two primary challenges: the confidentiality of the public database and the privacy of the users uploading DNA samples. With public databases and untrusted third-party cloud providers, there have been several attacks on public DNA samples, including identity tracing, attribute disclosure, and completion attacks [3].

To address these challenges, FHE emerges as a robust solution to preserve the privacy of both the database and user data. To find similar DNA samples while encrypted, however, we must employ innovative approaches. Many efficient string-matching algorithms have been proposed and utilized in genome analysis to find and cluster similar DNA sequences. Even though some dynamic programming algorithms have linear time complexities, very long DNA sequences composed of millions of characters and large databases with thousands of DNA sequences remain inefficient and are impossible to implement in FHE due to

existing approaches requiring runtime decisions over encrypted data (which is not feasible).

Generic solutions like private information retrieval (PIR) [16] and private set intersection (PSI) [11] can be used to find similar DNA samples. However, most PIR solutions exhibit high communication overhead between the client and the server (where the client downloads a significant chunk of the database [25]), which results in poor scalability for very large database sizes. To employ PSI for genome similarity, exact matching algorithms must be used; these algorithms, as discussed in our background section, scale poorly (quadratically) as the DNA size increases. Therefore, there is a pressing need to identify efficient methods that scale linearly with the DNA sequences. This motivates the use of approximate membership queries (AMQ) data structures [21] such as Bloom filters, which can find similar data with high accuracy.

In this work, we employ Bloom filters, a type of approximate set membership data structure, which can enhance the efficiency of database queries [5] by using hash functions to test for membership with a high probability of getting a correct response. Bloom filters have sublinear time complexities and can help us accurately and efficiently find similar DNA samples in the database while they are encrypted with FHE. Overall, our contributions are as follows:

- We propose a novel Bloom filter methodology in FHE to enable efficient server execution for large databases and achieve very high accuracy.
- We adopt techniques such as PRNG mask generation, modulus switching, and circuit bootstrapping (CBS) to massively reduce the communication overhead for the clients.
- We propose novel aggregation strategies to further reduce the communication overhead and optimize it for execution efficiency.
- We investigate parallelization and parameterization strategies to further optimize the runtime of encrypted Bloom filter constructions.

Roadmap. The rest of the paper is organized as follows: Section 2 reviews FHE and the probabilistic Bloom filter data structure, while Section 3 outlines challenges and optimizations for efficient implementation in the encrypted domain. In Section 4 we detail our methodology for deploying Bloom filters in the encrypted domain, while Section 5 presents our experiments using DNA matching and genome analysis, compared to related work. Finally, Section 6 discusses related works, and Section 7 provides our concluding remarks.

2 Background

2.1 Homomorphic encryption

In 2009, Gentry proposed the first fully homomorphic encryption scheme (FHE) that supports both addition and multiplication over ciphertexts based on the ideal coset problem (ICP) [22]. In 2011, Brakerski et al. improved upon Gentry's original construction by proposing the BGV scheme [8] that derives its

security from the learning with errors (LWE) problem [32] and its ring variant (RLWE). In this case, a small amount of noise is added to each ciphertext to hinder cryptanalysis; unfortunately, the noise grows as we operate on ciphertexts. A bootstrapping mechanism can be utilized to homomorphically evaluate the decryption circuit and refresh the noise, and allow arbitrary computation.

Although FHE gives us a lot of freedom in implementing different applications, the latency can be prohibitively high. Bootstrapping is a major bottleneck in modern FHE schemes; for example, in the BGV [8] and CKKS [12] schemes, a single bootstrapping operation can take seconds to minutes to execute on a CPU, depending on the parameters. On the other hand, a Boolean-based FHE scheme called FHEW [19] was introduced, with significantly faster bootstrapping times (less than a second). Unlike CKKS and BGV, FHEW encrypts bits instead of integers. The CGGI (i.e., TFHE) cryptosystem [14] supports a faster programmable bootstrapping (PBS) operation, which executes around 10 milliseconds on a CPU; for this reason, we use CGGI in this work.

2.2 String Matching

There are two types of string-matching algorithms: exact and approximate string-matching. For the former, the algorithm finds all occurrences of the pattern string P in the text string S. The most popular and efficient algorithms of this type are the KMP [27] and BM [7] algorithms. Both have an average time complexity of $\mathcal{O}(n)$, where n is the length of S. Another well-known exact matching algorithm is Bitap [4], which uses bitwise operations to efficiently match patterns in a text and is especially effective for short patterns on modern hardware due to its parallelism-friendly design.

In approximate string matching [30], also known as fuzzy string searching, the algorithm searches for all substrings in the text string S whose edit distance from the pattern string P is at most d. Edit distance [30] is a measurement of the similarity between two strings. The shorter the distance, the more similar the two strings are; examples of edit distances include the Hamming distance and Levenshtein distance.

A DNA sample consists of four nucleotide bases: adenine (A), cytosine (C), guanine (G), and thymine (T), and can be represented as a string, where the alphabet is $\Sigma = \{A, C, G, T\}$. Therefore, each character of a DNA string can be represented using two bits, yet the total length of a DNA string can be thousands of characters long. The sheer volume of DNA records stored in public databases presents a daunting obstacle for efficient genome sequencing and analysis.

2.3 Probabilistic Filters

To efficiently check if an element exists in a database, a Bloom filter [5] can be used. This space-efficient probabilistic data structure guarantees with high probability that an element exists in a database in $\mathcal{O}(1)$ time. Specifically, a Bloom filter is an array of size m, where each cell is initialized to the bit `0`; then, an input is hashed into h different indices in the array (using h hash functions).

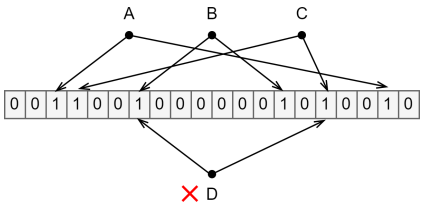


Fig. 1: Insertion of elements A, B, and C into a Bloom filter using two hash functions. Element D is a false positive because it hashes to two cells that were set to `1` by distinct elements.

When inserting an element to the array, we set the h cells generated by the hash functions to `1`. To query an element for membership, we hash the element using the same hash functions and if all h indices in the array are equal to `1`, the element exists in the database. Figure 1 shows the insertion process using two hash functions to insert elements A, B, and C and set the corresponding bits to `1`.

Due to hash collisions and the fixed size of the Bloom filter array, there is always a chance for false positives. In Figure 1, element D is queried to check if it exists in the filter, and the bits in the computed indices are set to `1`, so the filter will return a match. However, the two bits are set by two different elements B, and C, which means a false positive has occurred. Intuitively, increasing the size of the Bloom filter decreases the false positive rate since the chance of hash collisions decreases. Theoretical analysis of Bloom filters [5] can give us a good estimation of proper parameters to use for a desired false positive probability p; The optimal array size m can be derived as $m = -\frac{n \cdot \ln(p)}{\ln(2)^2}$.

The size of the filter depends on the number of inputs to be inserted into the filter, n and the false positive probability p. Next, the optimal number of hash functions h only depends on p and can be calculated as $h = -\frac{\ln(p)}{\ln(2)}$.

2.4 Threat Model

Our methodology utilizes the CGGI scheme [15] as the FHE backend. In particular, we are inheriting the security guarantees of the CGGI cryptosystem without any modifications so that all ciphertexts and ciphertext computations are semantically secure by construction. We assume an honest-but-curious cloud provider that will correctly evaluate our protocol but has an incentive to eavesdrop on the private data. In this scenario, no data can be learned about the plaintext corresponding to the encrypted data supplied by the client as input. We also assume that the cloud has access to the plaintext database, but it remains private to the clients.

3 Overview of our Methodology

In our framework, we assume a client-server model where the server has access to the plaintext database, while the client has a DNA sample and wants to check

if a similar sample exists in the database. To accomplish this, the client encrypts the DNA sample using their public FHE key and sends it to the server. The server has a noiseless FHE encoding of the database precomputed, so when a client sends their encrypted input, it performs the necessary computations on the encrypted data and sends the encrypted result back to the user.[1] The result is returned as a homomorphic bit, which shows whether a similar DNA sample exists in the database.

A naive method in DNA similarity matching is to compare the query DNA string with each of the DNA records in a database. Then, an exact string-matching algorithm such as the KMP [33] or approximate matching like the Bitap algorithm [4] is employed to find similar samples. The main limitation of these algorithms is that, since the size of the DNA sequences is very large, the search is not scalable. While there are methods that leverage CPU caches and other heuristics to improve the efficiency of string-matching, these algorithms do not directly translate to FHE.

3.1 Challenges in FHE

Optimal *exact* string matching algorithms (e.g., KMP, BM) run in linear time and space, whereas *approximate* methods (e.g., Levenshtein distance, bitap) require quadratic time. When deploying these under FHE, two primary challenges arise: data size expansion due to encryption and the inability to branch on encrypted data.

Size Expansion. While each nucleotide requires only 2 bits, FHE encryption expands the data by at least three orders of magnitude; for example, a 1-million-nucleotide sequence may expand to 2 gigabits. Transciphering [17] addresses this by using a size-efficient traditional cipher (e.g., AES) to encrypt the data first, and then performing a homomorphic decryption on the server to yield an FHE ciphertext. Although this reduces communication overhead, it introduces additional time complexity. This trade-off may be acceptable in applications such as medical diagnostics and private security log analysis [9,6], but the added execution time could impact performance in other contexts.

Branching and Early Termination. Many string-matching and database search algorithms leverage data-dependent branching or early termination (e.g., dynamic programming, lookup tables, binary search, or tries) to improve efficiency. Under FHE, the server cannot make such conditional decisions on encrypted data without exposing information about the plaintext. As a result, the server must execute the entire algorithm without shortcuts, which can undermine efficiency and security.

[1] We remark that the noiseless encoding is valid for *all users* under the assumption that the same cryptographic parameters are utilized.

3.2 Efficient Encrypted Bloom Filter Evaluation

When querying a database, a straw-man approach is to check all elements of the database. This is done in $\mathcal{O}(n)$ time and will be very inefficient as the number of elements grows. More efficient algorithms and data structures, such as binary search and *tries*, can be leveraged to do this efficiently with $\mathcal{O}(log(n))$ time complexity. However, these algorithms require branching on the input data, while in our case, all the inputs are FHE-encrypted. Therefore, the computing party is unable to make a runtime decision based on the underlying plaintext data of a ciphertext [28,23].

With the help of probabilistic data structures like Bloom filters, we can query the database in constant time and $\mathcal{O}(1)$ space with a probability p of false positives and size of the Bloom filter m. According to equations in Section 2.3, m scales linearly with the size of the database. Bloom filters are very efficient in the plaintext domain, since they only require index lookups.

Nevertheless, translating these structures to FHE is not straightforward: as shown in Figure 1, Bloom filters require dynamic indexing to query an element in constant time. Essentially, it is a lookup operation, where the element in the corresponding index in the Bloom filter is returned. Since the PBS is only efficient up to 8 bits, only very small Bloom filters of size $m = 2^8 = 256$ can be indexed homomorphically using the PBS operation. To compute homomorphic indexing for larger sizes in the encrypted domain, we utilize the CMUX-tree method introduced in [15], which is also called `Vertical Packing`. Using this method, a series of homomorphic CMUX operations (which are a few orders of magnitude faster than the PBS) are done to compute the large lookup homomorphically. The CMUX operation is defined as $\text{CMUX}(S, R_a, R_b) = R_a + S \cdot (R_b - R_a)$.

The S ciphertext is a GGSW ciphertext holding a single encrypted bit of the input, and R_a and R_b are the sub-lookup tables encoded as GLWE ciphertexts. It is implemented using the homomorphic external product of a GGSW and an RLWE ciphertext; the output is a GLWE ciphertext, essentially selecting one of the two lookup tables homomorphically (for more details on the external product, see [15]). The output of the CMUX-tree is a single GLWE ciphertext, which at its first slot holds the result of the entire lookup operation. We extract the LWE ciphertext from the first slot using Sample-Extraction, which is a noiseless and essentially free operation. The LWE ciphertext in the bloom filter case will hold a single encrypted bit, which is the element corresponding to the input index in the filter.

If our input index is s bits, we need to construct s different GGSW ciphertexts, each encrypting a single bit. Since querying a Bloom filter requires h different hash functions, a single query would require $s \cdot h$ different GGSW ciphertexts. This would incur a very high communication overhead for the user, depending on the parameters, each GGSW ciphertext is around 100 KB.

Reducing communication overhead. To reduce the communication overhead even further, we propose utilizing the Circuit bootstrapping (CBS) technique introduced in [15], to convert LWE ciphertexts holding a single bit to their

equivalent GGSW ciphertext homomorphically. The CBS consists of two steps: the first step is the PBS operation to shift the encrypted bit to a desired position (for more details see [15]), and the second step is a series of homomorphic key switching operations to convert it to the output GGSW ciphertext.

Employing this technique, the user only needs to send their encrypted LWE ciphertexts, which are much smaller than GGSW ciphertexts. The server needs to compute the CBS operation for each LWE ciphertext as a one-time cost, so it can compute the CMUX-trees. To decrease the overhead even further, we can utilize a public PRNG such as a hash function to generate the mask of each LWE ciphertext, so we only need to send the body to the server, and the server can independently generate the mask elements with the PRNG in counter mode. However, a small seed must be communicated, which for AES-CTR that we use is 32 bytes, 16 bytes is the seed itself, and 16 bytes is the random IV. Therefore, for each bit of the input index, only the body of the LWE ciphertext is sent to the server, which is a 64-bit integer for the parameters we utilize in our experiments, where the ciphertext modulus is $q = 2^{64}$. We could utilize a smaller ciphertext modulus, such as 32 bits, to reduce communication overhead, but `tfhe-rs`, the FHE library we use for our experiments, is optimized efficiency-wise for a 64-bit modulus.

Since the output LWE ciphertexts from the server cannot be compressed using the PRNG technique, we use `Modulus-Switching` [15] to decrease the size of the mask and body of LWE ciphertexts. `Modulus-Switching` is used in the first step of PBS and reduces the ciphertext modulus from q to a smaller modulus N, where N is the ring dimension. N typically equals 1024 or 2048 for most parameters, yielding a $6\times$ reduction in communication overhead.

Aggregating the results. The output of each CMUX-tree is a single LWE ciphertext, encrypting one bit, which is the element corresponding to the input encrypted index in the filter. For h different hash functions we use, we would have h different LWE ciphertexts at the end. To aggregate the results into a single operation, we homomorphically compute the `AND` operations between all h LWE ciphertexts. The resulting LWE ciphertext will hold the encrypted bit 1 if all h ciphertexts are 1 (and 0 otherwise). We can do this by shifting and adding the two encrypted ciphertexts into a single ciphertext holding both bits, and computing a PBS that evaluates the `AND` between the two encrypted bits, and stores the result in the same ciphertext. We would need $h - 1$ PBS operations to aggregate the results.

To optimize this even further, we observe that the PBS has roughly the same execution time up until 4 bits of message. Therefore, instead of computing PBS on two bits, we can shift and add 4 ciphertexts at a time, and compute the homomorphic `AND` on all 4 bits. This requires roughly $\lfloor \frac{h}{3} \rceil$ PBS operations.

4 Technical Details

Our methodology is realized with TFHE-rs [41] as the FHE backend, which implements the CGGI cryptosystem [15]. Notably, the CGGI cryptosystem is commonly used to encrypt single bits or very low-precision integers, which is efficient for the PBS operation. However, when trying to compute larger lookup tables, like 16 or 20 bits or larger, we leverage the CBS and CMUX-tree to achieve an efficient Bloom filter solution in FHE. Additionally, the CBS operations are highly parallelizable, as each CBS is computed on each LWE bit, and the CMUX-tree operations can be parallelized across different index computations. Since the bloom filter approach requires efficient encrypted lookup operations, we adopt the CGGI scheme that can help us implement such a solution.

4.1 Protocol Description

Our protocol consists of three major steps outlined below:

- **Client preprocessing and encryption:** The client must hash the DNA sequence using h different hash functions, and encrypt each bit of each digest using the AES-CTR PRNG and send the ciphertext bodies and the PRNG seed to the server.
- **Server FHE evaluation:** The server reconstructs the ciphertext masks using the PRNG seed and generates a Bloom filter (or multiple Bloom filters, see Section 5) with the same h hash functions the client used. Then the server computes the Bloom filter query for each encrypted index, aggregates the results of the h ciphertexts, then modulus switches the output LWE ciphertext and sends it back to the client.
- **Client decryption:** The client decrypts the LWE ciphertext received from the server to get the result.

A major benefit of our approach is that it requires no post-processing from the client, nor any back-and-forth between the client and the server that might leak information. Figure 2 offers a general outline of our protocol. We provide details of each step of the protocol in the following paragraphs.

4.2 Client-side Operations

The client owns a sample DNA sequence and needs to check if a similar sample exists in a database. The client pre-processes its DNA sample, encrypts it, and sends it to the server for evaluation.

In more detail, the client first selects the subsequence length, k, to form k-mers, since hashing an entire DNA sequence (e.g., one million bases) would almost never produce a match. The server must similarly construct its Bloom filter by inserting only k-mers of the same length.

Next, each k-mer is hashed using MurmurHash[2], a non-cryptographic hash function that can produce 32-, 64-, or 128-bit digests. For our application, we

Protocol Description (our Bloom Filter in FHE)

Input: Client DNA sequence A, server DNA database D, number of hash functions h, PRNG `seed` (seed + IV)

Client-Side:

- $\mathbf{a}_i = \texttt{AES_CTR}(\texttt{seed}) \quad \text{for } i = 0, \ldots, s \cdot h - 1$
- $\mathbf{H}_A = \{h_i(A) \mid i = 1, \ldots, h\}$
- For each H in $\mathbf{H}_A$:
  - represent $H = (b_0, \ldots, b_{s-1})$ and compute $ct_i = \texttt{LWE_Enc}(b_i, \mathbf{a}_i)$ for $i = 0, \ldots, s-1$
- Transmit to the server $\{ct_i\}_{i=0}^{s \cdot h - 1}$ and `seed`

Server-Side:

- Construct Bloom filter $\mathbf{B} \in \{0,1\}^m$ from DNA database D using hash functions $h_1, \ldots, h_h$
- Encode $\mathbf{B}$ as a lookup table: $\text{LUT}_{\text{bf}} = \texttt{GLWE_Enc}(\mathbf{B})$
- Define a secondary lookup table LUT_{AND} for 4-bit AND computation: $\text{LUT}_{\text{AND}} : \{0,1\}^4 \to \{0,1\}$
- Reconstruct PRNG mask vectors: $\mathbf{a}_i = \texttt{AES_CTR}(\texttt{seed})$ for $i = 0, \ldots, s \cdot h - 1$
- For each LWE ciphertext $ct_i = (\mathbf{a}_i, b_i)$:
  - Apply Circuit Bootstrapping: $ct_i' = \texttt{CBS}(ct_i)$, where ct_i' is a GGSW ciphertext
- For each index $j = 1, \ldots, h$:
  - Let $\{ct'_{j,i}\}_{i=0}^{s-1}$ be GGSW ciphertexts of bits of H_j
  - Evaluate CMUX tree: $ct_j^{\text{out}} = \texttt{CMUX-tree}(\{ct'_{j,i}\}_{i=0}^{s-1}, \text{LUT}_{\text{bf}})$
- Group each 4 ciphertexts $\{ct_j^{\text{out}}\}_{j=1}^{4}$:
  - Aggregate: $ct_{\text{agg}} = \sum_{j=1}^{4} \texttt{Shift_left}(ct_j^{\text{out}}, j-1)$
  - Apply PBS: $ct_{\text{final}} = \texttt{PBS}(ct_{\text{agg}}, \text{LUT}_{\text{AND}})$
- Apply modulus switching: $\tilde{ct} = \texttt{ModSwitch}(ct_{\text{final}}, q \to N)$
- Send $\tilde{ct}$ to the client

Client: Decrypt the returned LWE ciphertext to learn whether the queried element exists in the database.

Fig. 2: Protocol description for our novel Bloom filter methodologies. The output can be directly decrypted by the client.

use the 64-bit variant and reduce the digest modulo m (the Bloom filter size) to obtain well-distributed indices with low collision probability. After applying h distinct hash functions, the resulting ciphertexts (following the encryption strategy discussed previously) are sent to the server.

The choice of h depends on the desired false positive rate, as increasing h reduces the rate. For example, a false positive rate of 0.01 requires $h = 7$ hash functions.

4.3 Server-side Computations

Before receiving the client's input, the server generates an FHE-encoded Bloom filter from its database, performs the evaluation, and sends the result for decryption. Both parties must use identical FHE parameters (e.g., ciphertext polynomial degree and coefficient size) to ensure proper homomorphic operations. Trivial/noiseless encodings incur negligible latency with TFHE-rs, and since encoding the database is a one-time cost, any client with matching parameters can submit encrypted queries. Notably, combining a trivial encryption with a secure client-encrypted ciphertext (via the CMUX operation between the client's GGSW ciphertext and GLWE sub lookup tables) yields a ciphertext under the client key. Lastly, the server requires the client's bootstrapping key (which TFHE-rs compresses from gigabytes to megabytes) to conduct HE operations efficiently.

5 Experimental Evaluation

We perform a series of experiments to showcase the efficacy of our Bloom filter methodologies. All our experiments are single-threaded and were performed on an Ubuntu 22 server using an i9-12900K CPU and 128 GB of RAM. We use the `core-crypto` API provided by the TFHE-rs library [41], which implements the CGGI scheme in Rust [15]. We use custom cryptographic parameters corresponding to 128 bits of security, and a failure probability of $p_{fail} = 2^{-40}$ for the PBS operation. The DNA samples used in the experiments are derived from a public dataset [10].

5.1 Direct comparisons to related works

Many prior works have investigated using Bloom filters with FHE [20,42,13] in a single-party setting. The works in [20] and [42] use a naive solution to evaluate bloom filters in FHE: the client generates homomorphically encrypted Bloom filters for each query and send them to the server. Then, the server computes on the server-side filters and the encrypted query filter, aggregates the result, and returns it to the client. For very large filter sizes (e.g., in the millions), such techniques become quickly impractical, and the communication overhead grows to Gigabytes. The work in [13] takes a different approach. In a pre-processing step, the client sends many encrypted ciphertexts to the server (as large as the bloom filter). For each client, the server sorts its bloom filter homomorphically based on the ciphertexts given. Since all operations are in FHE, the server will not know which element holds what bit in the resulting bloom filter. Thus, the client simply queries in plaintext, and gets the corresponding ciphertexts to each index, and if all bits are set to 1, then the element exists. For similar database sizes and bloom filter parameters, we do a direct comparison with [13], and compare the communication overhead of client to server and server to client, as well as the execution time needed server-side.

Table 1: Comparison with [13] with respect to *communication* and *execution* overhead. Note that the communication of the FHE keys from client to server is not shown from either work, as for both works would be similar, and it is a pretty much unavoidable cost in all FHE applications.

DB Size	**C→S Comm.**		**Server Time (s)**		**S→C Comm.**	
(entries)	Ours	[13]	Ours	[13]	Ours	[13]
2^8	608 B	0.04 GB	5.19	3	1.4 KB	1.8 MB
2^{10}	704 B	0.04 GB	5.97	3	1.4 KB	1.8 MB
2^{12}	800 B	0.04 GB	6.81	3	1.4 KB	1.8 MB
2^{14}	896 B	0.04 GB	7.65	3	1.4 KB	1.8 MB
2^{16}	992 B	0.16 GB	8.54	50	1.4 KB	1.8 MB
2^{18}	1088 B	0.69 GB	9.72	800	1.4 KB	1.8 MB
2^{20}	1184 B	3 GB	11.68	10000	1.4 KB	1.8 MB
2^{22}	1280 B	12.85 GB	17.33	200000	1.4 KB	1.8 MB

In both experiments, the number of hash functions h is set to 6. For each database size, the bloom filter size m is chosen larger (see equations in Section 2) to accommodate the false positive probability.

The client-server communication overhead for our work is much smaller than the work in [13]. Their work requires the client to send many big ciphertexts in the setup phase, which grows with the size of the database after 2^{16}. The reason the overhead is the same for the first few Database sizes is that they use the BFV cryptosystem, and they use a batching of size 2^{14} for all their experiments. For a database size in the millions, the client in [13] is required to send Gigabytes of data, which is impractical and quickly becomes infeasible for even larger database sizes. On the other hand, our technique massively reduces the communication required by the client when they query the server. Using a PRNG to decrease the size of each LWE ciphertext, the client needs to only send $s \cdot h$ values b, where each b is 8 bytes and $s = \log_2(m)$, plus 32 bytes of the seed for the PRNG. For most database sizes shown in Table 1, the overhead is less than 1 kilobyte, and for the last few rows it is a little bit more than a kilobyte.

The server execution time for the work in [13] follows the same pattern as the communication overhead from client to server. For database sizes up to 2^{14}, they have the same execution time, since they fit in a batch of BFV ciphertext. For larger database sizes in the millions, the efficiency gains from the batching become negligible, and the execution time takes hours and grows at a very high rate, which makes it quickly impractical. Our solution at first is a bit slower, but for larger database sizes it becomes much faster than [13], specifically for a database size of 2^{22} it is 4 orders of magnitude faster. Up to 2^{16}, the execution time follows a linear growth, specifically 0.8 seconds, because up to this point the cost of the CMUX-tree is almost negligible to the cost of CBS, and the 0.8 seconds comes from the fact that we are computing 2 more CBS operations for the 2 added bits. For larger sizes, the CMUX-tree execution time grows at a higher rate.

The communication overhead from server to client for our work is around 3 orders of magnitude lower than in [13], which is because they return h BFV ciphertexts to the server, while our work also aggregates server-side and returns a single LWE ciphertext. It is also because LWE ciphertexts are much smaller in size than BFV ciphertexts.

5.2 DNA Size Considerations

When querying extensive DNA datasets consisting of billions of elements, the CMUX-tree becomes a bottleneck, requiring the server to allocate substantial memory (1 TB or higher) to process the tree. To mitigate this issue, we can partition large DNA samples into smaller segments, generate a separate Bloom filter for each, and apply the same evaluation techniques to every filter. This approach ensures that all CMUX-tree operations run smoothly. To aggregate the results from the different Bloom filters, we employ the methodologies introduced in Section 3; however, instead of computing the `AND` operation, we compute the `OR` across all Bloom filter LWE outputs, as a match in any one filter is sufficient.

5.3 Performance/Error tradeoff

There are two avenues of error that might lead to incorrect results in our methodology:

1. Probability of failure in the bootstrapping [15];
2. False positive probability of the Bloom filter.

The custom parameters that we use in TFHE-rs have a probability of failure $P_{fail} = 2^{-40}$, which corresponds to the likelihood of a programmable bootstrap (PBS) giving incorrect results. Many works in the FHE literature employ much lower probabilities of failure, such as 2^{-23} for AES transciphering [37] or 10^{-6} for encrypted neural network inference [36]. Reducing the probability of failure requires choosing larger parameters that will lead to larger execution times for the PBS. However, we optimize our parameters using the Concrete optimizer [40], where for almost all applications, including ours, the probability of failure we choose is sufficient.

The second error comes from the probabilistic nature of the Bloom filter algorithm. For the Bloom filter, for a given false positive probability, a minimum number of hash functions and a Bloom filter size is required. Lower false positive rates require more hash functions, which leads to an increase in execution time. Therefore, striking a balance between collision probability and performance highly depends on the application and can be easily tuned by the user. We show the trade-off between collision probability and performance for our Bloom filter methodology in Table 2. We conducted experiments on a single CPU core, with a database size of 2^{22}.

In Table 2, the execution time depends on both the Bloom filter size m and the number of hash functions h. When decreasing the false positive probability,

Table 2: Bloom Filter Timing Results

False Positive Probability (p)	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
Number of Hash Functions (h)	4	7	10	14	17	20
Time (s)	11.58	20.13	28.54	54.95	66.40	79.90

for a set database size, both h and m increase, as indicated in the equations in Section 2.3.

For the first three false positive probabilities, the execution time grows linearly. When jumping from $p = 10^{-3}$ to $p = 10^{-4}$, we see the execution time doubles, then the rest increases at a linear rate. This is because the index bits required for m increase, which doubles the execution time of the CMUX-tree. So to have a false positive probability of $p = 10^{-6}$, it's as if we are computing double the database size for a smaller p. Striking a balance between efficiency and accuracy is very dependent on the user's needs and should be carefully considered. The execution time can be further improved by parallelizing the CBS and CMUX-tree operations across different h index computations.

6 Related Works

A class of earlier works explores the problem of private string search in the context of DNA matching using related cryptographic techniques. Shimizu et al. [34] propose using searchable encryption (SE) in conjunction with additive homomorphic encryption and oblivious transfer (OT) to do string search and efficiently query a DNA database. Another cryptographic primitive, namely predicate encryption, is used by Wang et al. [38]; predicate encryption is a generalization of identity-based encryption where an attribute or a policy is attached to the ciphertext and the user can decrypt only when the policy is met. Authors utilize this property to apply private string matching to identify similar DNA samples and only allow decryption if the DNA samples are similar. Compared to these works, our proposed methodology with FHE provides strong security guarantees (128 bits of security under known lattice attack models) and only requires the client to perform inexpensive encryption and decryption.

There are multiple works on privacy-preserving Bloom filters. Feng et al. [20] employ Bloom filters with homomorphic encryption to find similar locations privately between two users. Specifically, both users need to generate their own Bloom filters, encrypt them, and then the computation is handled by a cloud server. Notably, for the bit-wise Bloom filter proposed in [20], the authors utilize encryption parameters that correspond to negligible security, whereas our work corresponds to the current security standard of 128 bits of security. Additionally, their work requires a third-party setting where an aggregator helps find the intersection of the bloom filters between two users. Conversely, our solution is a client-server approach, which supports multiple client queries.

Chielle et al. [13] uses FHE and Bloom filters to check if an element exists in a private database in $\mathcal{O}(1)$ time complexity. This is achieved through the

introduction of a random permutation of the server's Bloom filter that is only known to the client; during a query, the client computes the hash and derives the positions of the ciphertexts to be returned, which requires no FHE computation on the server side. However, the setup time for each client requires many hours to complete, and requires the client to send Gigabytes of information in the setup phase for large database sizes. On the other hand, our work, as shown in Table 2, has much lower overhead in both bandwidth and execution times. This approach requires N copies of the database (in encrypted form) for N clients, while our approach does not require them to be encrypted. Lastly, Stanciu et al. [35] propose using partial homomorphic encryption and Bloom filters to do privacy-preserving crowd monitoring. Unlike our solution, their protocol requires multiple parties to be trusted.

Finally, several works have employed leveled homomorphic encryption to do secure genome analysis. Ziegeldorf et al. [42] utilize the BGV scheme to encrypt and evaluate default Bloom filters to match single-nucleotide polymorphisms (SNP) of a query in a patient database for genetic disease testing. They only use 80 bits of security, while 128 bits is the current standard. Also, their approach requires Gigabytes of communication overhead and hours of server execution time for filter sizes in the millions. Conversely, our solution requires very small communication overhead, 1 KB or lower, and can evaluate filter sizes in the millions in seconds. Gursoy et al. [24] also use the BGV scheme, but their work focuses on genotype imputations to find missing genomic data using a p-impute algorithm similar to the k-nearest-neighbor approaches.

7 Conclusion

We propose a novel privacy-preserving DNA matching approach that leverages Bloom filters and fully homomorphic encryption. Our method uses Circuit Bootstrapping with CMUX-trees for efficient encrypted lookups and incorporates optimizations such as PRNG, modulus switching, and homomorphic aggregation to reduce client communication overhead. Single-threaded experiments demonstrate evaluation speeds up to four orders of magnitude faster with significantly lower communication costs compared to related work, which is significant for resource-constrained clients. By offloading all computations to the server, client operations are minimized.

Acknowledgments

This work has been supported by NSF Award #2239334.

References

1. Apple: CSAM detection. `https://www.apple.com/child-safety/pdf/CSAM_Detection_Technical_Summary.pdf` (2020)

2. Appleby, A.: Murmurhash. `https://sites.google.com/site/murmurhash`
3. Arshad, S., Arshad, J., Khan, M.M., Parkinson, S.: Analysis of security and privacy challenges for dna-genomics applications and databases. Journal of Biomedical Informatics **119** (2021)
4. Baeza-Yates, R., Navarro, G.: A faster algorithm for approximate string matching. In: Annual Symposium on Combinatorial Pattern Matching. pp. 1–23. Springer (1996)
5. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
6. Boudguiga, A., Stan, O., Sedjelmaci, H., Carpov, S.: Homomorphic encryption at work for private analysis of security logs. In: ICISSP. pp. 515–523 (2020)
7. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM **20**(10), 762–772 (1977)
8. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
9. Carpov, S., et al.: Practical privacy-preserving medical diagnosis using homomorphic encryption. In: IEEE CLOUD. pp. 593–599 (2016)
10. Chauhan, N.S.: DNA dataset. `https://www.kaggle.com/datasets/nageshsingh/dna-sequence-dataset` (2020)
11. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: ACM CCS. pp. 1243–1255 (2017)
12. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: ASIACRYPT. pp. 409–437. Springer (2017)
13. Chielle, E., Gamil, H., Maniatakos, M.: Real-time private membership test using homomorphic encryption. In: DATE. pp. 1282–1287. IEEE (2021)
14. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: ASIACRYPT. pp. 3–33. Springer (2016)
15. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)
16. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. Journal of the ACM **45**(6), 965–981 (1998)
17. Cosseron, O., Hoffmann, C., Méaux, P., Standaert, F.X.: Towards case-optimized hybrid homomorphic encryption: Featuring the elisabeth stream cipher. In: Asiacrypt. pp. 32–67. Springer (2022)
18. Cramer, R., Damgård, I.B., et al.: Secure multiparty computation. Cambridge University Press (2015)
19. Ducas, L., Micciancio, D.: FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In: EUROCRYPT. pp. 617–640. Springer (2015)
20. Feng, Y., Lu, Z., Cao, Q.: Secure sharing of private locations through homomorphic bloom filters. In: IEEE BigDataSecurity, HPSC and IDS. pp. 107–113 (2018)
21. Geil, A., Farach-Colton, M., Owens, J.D.: Quotient filters: Approximate membership queries on the GPU. In: IPDPS. pp. 451–462. IEEE (2018)
22. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC. p. 169–178. ACM, NY, USA (2009)
23. Gouert, C., Mouris, D., Tsoutsos, N.G.: SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks. Proceedings on Privacy Enhancing Technologies **2023**(3), 154–172
24. Gürsoy, G., et al.: Privacy-preserving genotype imputation with fully homomorphic encryption. Cell systems pp. 173–182 (2022)

25. Henzinger, A., et al.: One server for the price of two: Simple and fast single-server private information retrieval. In: USENIX Security. pp. 3889–3905 (2023)
26. Kim, J.W., Kim, E., Park, K.: Fast matching method for dna sequences. In: International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies. pp. 271–281. Springer (2007)
27. Knuth, D.E., Morris, Jr, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM journal on computing **6**(2), 323–350 (1977)
28. Mouris, D., Tsoutsos, N.G., Maniatakos, M.: Terminator suite: Benchmarking privacy-preserving architectures. CAL **17**(2), 122–125 (2018)
29. Murugesan, M., et al.: Efficient privacy-preserving similar document detection. The VLDB Journal **19**(4), 457–475 (2010)
30. Navarro, G.: A guided tour to approximate string matching. ACM CSUR **33**(1), 31–88 (2001)
31. Ondov, B.D., et al.: Mash: fast genome and metagenome distance estimation using minhash. Genome biology **17**, 1–14 (2016)
32. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM) **56**(6), 1–40 (2009)
33. Régnier, M.: Knuth-morris-pratt algorithm: an analysis. In: MFCS. pp. 431–444. Springer (1989)
34. Shimizu, K., Nuida, K., Rätsch, G.: Efficient privacy-preserving string search and an application in genomics. Bioinformatics **32**(11), 1652–1661 (2016)
35. Stanciu, V.D., Steen, M.v., Dobre, C., Peter, A.: Privacy-preserving crowd-monitoring using bloom filters and homomorphic encryption. In: EdgeSyS. pp. 37–42 (2021)
36. Stoian, A., et al.: Deep neural networks for encrypted inference with TFHE. In: CSCML. pp. 493–500. Springer (2023)
37. Trama, D., Clet, P.E., Boudguiga, A., Sirdey, R.: A homomorphic AES evaluation in less than 30 seconds by means of TFHE. In: WAHC. pp. 79–90 (2023)
38. Wang, B., Song, W., Lou, W., Hou, Y.T.: Privacy-preserving pattern matching over encrypted genetic data in cloud computing. In: INFOCOM. pp. 1–9. IEEE (2017)
39. Weir, B.S.: Matching and partially-matching DNA profiles. Journal of Forensic and Sciences **49**(5) (2004)
40. Zama: Concrete: TFHE Compiler that converts python programs into FHE equivalent (2022), `https://github.com/zama-ai/concrete`
41. Zama: TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data (2022)
42. Ziegeldorf, J.H., et al.: BLOOM: BLoom filter based oblivious outsourced matchings. BMC Medical Genomics **10**, 29–42 (2017)