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 entails users trying to determine if their DNA samples are similar to DNA sequences hosted in a larger remote repository. Nevertheless, end users may be reluctant to upload their DNA sequences, while the owners of remote genomics repositories are unwilling to openly share their database. To address this challenge, we propose two distinct approaches based on fully homomorphic encryption to preserve the privacy of the genomic data and enable queries directly on ciphertexts. The first is based on the ubiquitous MinHash algorithm and can determine if similar matches exist in the database, while the second involves a bespoke bloom filter construction for determining exact matches. We validate both approaches across various database sizes using both GPU and CPU-based cloud servers.

Keywords: Homomorphic encryption · Private genome association · MinHash · Bloom filters.

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 [38]. 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 [34]. Many different privacy-preserving solutions, such as secure multi-party computation (MPC) [20], utilize advanced cryptographic primitives to help secure the data while processing. However, in our assumed 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. To get around this, the data can be divided across multiple servers, but in this case, a critical assumption is that the servers are non-colluding. In some cases, a trusted third party [22] is used to carry out the correct computations while maintaining the privacy of the data.

With all this in mind, a promising solution to address all these challenges with a stronger threat model (in a semi-honest setting) is a cryptographic primitive called fully homomorphic encryption (FHE) [27], 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 resulting ciphertexts are *malleable* by design.

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 [16]. 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 [45, 32, 14] 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 (in the order of millions of characters), and very large databases with thousands of DNA sequences remain inefficient, and are impossible to implement in FHE due to existing algorithms having to make runtime decisions over encrypted data (which not feasible). This motivates the use of hashing-based techniques such as locality-sensitive hashing (LSH) [21] and approximate membership queries (AMQ) [26], which can help find similar data with high accuracy.

In this work, we employ MinHash, a heuristic LSH algorithm, which hashes the data to identify the amount of similarity with high probability based on comparable hashes [10]. Additionally, bloom filters, a type of approximate set membership data structure, can enhance the efficiency of database queries [6] by using hash functions to test for membership with a high probability. Both MinHash and 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:

- Investigating the use of MinHash in FHE to massively reduce the size of the DNA samples and find similarities accurately and efficiently.
- A batched bloom filter variation for FHE, which can handle multiple queries at the same time, resulting in fewer encrypted operations.
- Investigating parallelization and parameterization strategies to further optimize the runtime performance of the encrypted MinHash and batched bloom filter constructions.

Roadmap: The rest of the paper is organized as follows: Section 2 provides the necessary background on FHE and our probabilistic algorithms, namely MinHash and bloom filters, while Section 3 highlights challenges for implementing them efficiently in the encrypted domain, and investigates optimizations. Section 4 presents our proposed methodology and considerations of implementing both MinHash and bloom filter in the encrypted domain, as well as parameterizing them to increase the efficiency and accuracy of the probabilistic algorithms, while Section 5 discusses our experimental evaluation using DNA matching and genome analysis as the target application. Finally, Section 6 discusses relevant related work, and our concluding remarks are presented in Section 7.

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 [27], deriving its security from the Learning with Errors problem (LWE) [39]. A small amount of noise is added to each ciphertext to hinder cryptanalysis; unfortunately, as operations are conducted on the ciphertexts, this noise grows in magnitude. A bootstrapping mechanism can be utilized to homomorphically evaluate the decryption circuit and refresh the noise in the ciphertext to allow for 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 [9] and the CKKS [15] schemes, a single bootstrapping operation can take several seconds to minutes to execute on a CPU, depending on the cryptographic parameters. On the other hand, a boolean-based FHE scheme called FHEW [23] was introduced, with significantly

faster bootstrapping times (less than a second). Unlike CKKS and BGV, FHEW encrypts bits rather than integers. The CGGI cryptosystem [17], which builds upon FHEW, boasts even faster bootstrapping speeds of less than 10 milliseconds on a CPU; for this reason, we opt to use CGGI as the target FHE scheme 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 [33] and Boyer Moore [8] algorithms. Both have an average time complexity of $\mathcal{O}(n)$, where n is the length of S. In approximate string matching [35], 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 k. Edit distance [35] is a measurement of the similarity between two strings. The shorter the distance, the more similar the two strings are; examples of edit distances are hamming distance and Levenshtein distance.

2.3 Locality Sensitive Hashing (LSH)

This class of hashes consists of probabilistic algorithms, where similar items are hashed to similar buckets with high probability [31]. Different LSH algorithms are based on different similarity metrics; for example, some LSH algorithms are based on the cosine similarity metric [36], which measures the angle between vectors in a high-dimensional space. There is also the MinHash LSH heuristic algorithm [10] that approximates the Jaccard similarity metric between two sets [5]. The Jaccard similarity metric is given below:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}, \tag{1}$$

where A and B are sets of integers. While the Jaccard similarity requires linear time and space complexity to find the exact similarity between two sets, the MinHash algorithm can efficiently approximate the Jaccard metric, as shown in Algorithm 1. It consumes a set of positive integers and uses k different hash functions to find the minimum hash value, and stores each in the signature array M.

The computation of the approximate similarity is presented in Algorithm 2, which compares signature arrays of the two sets pairwise. The number of equal pairs divided by the number of hash functions k is the similarity measurement, which is a number between 0 and 1. Generating the signature array requires $\mathcal{O}(kn)$ time complexity, where k is the number of hash functions, and n is the size of the input set. However, comparing the two signature arrays requires $\mathcal{O}(k)$ time complexity, which is close to constant time since k is much smaller than n in real applications. Therefore, after incurring a linear pre-computation overhead

Algorithm 1 MinHash Algorithm

Input: Set of positive integers S
Input: Number of hash functions k
Output: Signature Array M

```
1: procedure MinHash(S, k)
2:     Initialize array M of size k to +∞
3:     for i ← 1 to k do
4:         for each element x in S do
5:             hashValue ← hash_i(x)
6:             M[i] ← min(M[i], hashValue))
7:     return M
```

of generating the signature hashes, MinHash requires constant space to store them and constant time to compare them, making it much more efficient than computing the Jaccard similarity.

Algorithm 2 Similarity Measurement

Input: Signature arrays $M1$ and $M2$ of length k
Output: Similarity score Sim/k

```
1: procedure Similarity(M1, M2)
2:     Sim ← 0
3:     for i ← 1 to k do
4:         if M1[i] = M2[i] then
5:             Sim ← Sim + 1
6:     return Sim/k
```

2.4 Probabilistic Filters

To efficiently check if an element exists in a database, a bloom filter [6] 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 k different indices in the array (using k hash functions). When inserting an element into the array, we set the k 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 the k 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 in the array and set the bits in the corresponding indices 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

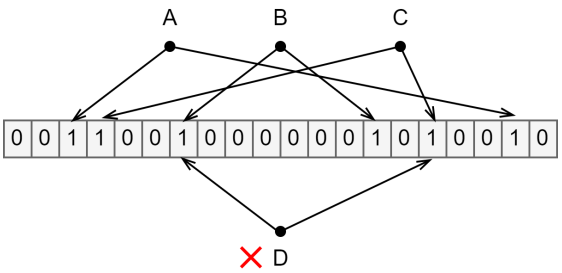

Fig. 1. Insertion of elements A, B, and C into the bloom filter with two hash functions. D represents a false positive, since it hashes to two different cells that are set to 1, but are set to 1 by two different elements.

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 [6] 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 k only depends on p and can be calculated as $k = -\frac{\ln(p)}{\ln(2)}$.

2.5 Threat Model

Our methodology utilizes the CGGI scheme [18] as an FHE backend. We assume an honest-but-curious cloud provider that will correctly evaluate the protocol but has an incentive to eavesdrop on the private data. We note that a malicious server might not carry out the intended operations and send incorrect results back to the user. In this scenario, the results are corrupted but no data can be learned about the plaintext corresponding to the encrypted data supplied by the client as inputs.

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]

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 to millions of characters long. Public databases used in genome analysis have millions of DNA records stored. The sheer amount of data makes DNA sequencing and analysis challenging to implement efficiently. 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 [40] 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 [11], these algorithms do not translate to FHE.

3.1 Challenges in FHE

The most optimal *exact* string matching algorithms have linear time and space complexity, such as the KMP and BM [8] algorithms, while the most optimal *approximate* string matching algorithms have quadratic time complexity, such as the Levenshtein distance algorithm and bitap algorithm. There are two main challenges when applying these algorithms using FHE: size expansion resulting due to encryption, and branching decisions on encrypted data.

Size expansion: As mentioned, each nucleotide in a DNA string can be encoded using 2 bits. Depending on the FHE scheme and its parameterization, a size expansion of at least three orders of magnitude will occur when encrypting a DNA sequence. As a result, a DNA sequence that has 1 million nucleotides (i.e., base pairs) will expand to at least 2 gigabits in size. One way to reduce the communication associated with uploading encrypted DNA sequences is transciphering [19]; briefly, transciphering involves encrypting the input data with a traditional cryptographic algorithm that results in little to no size expansion (such as AES). Then, when the server receives the data, it performs a homomorphic decryption (e.g., applies the AES decryption circuit homomorphically) to yield a purely FHE ciphertext. Nevertheless, the reduced communication overhead is traded for additional time complexity incurred for computing the decryption circuit to facilitate conversion to FHE ciphertexts. While transciphering can be a good solution in some applications such as medical diagnostics and private security log analysis [12, 7], the increase in execution time might hinder the practicality of some applications. For instance, a state-of-the-art transciphering framework

[1] We remark that the noiseless encoding is valid for *all users* under the assumption that the same cryptographic parameters are utilized, such as the default parameter set in TFHE-rs corresponding to 128 bits of security.

requires nearly 30 seconds to evaluate the AES decryption circuit of a single block on a cloud server [43].

Branching and Early Termination: Another challenge with string-matching or database searches is that most algorithms include some sort of branching based on the input data or early termination. When applying these algorithms in FHE, the server cannot do any branching on the encrypted data as it does not have any knowledge about the underlying plaintext data. Most algorithms like the KMP algorithm leverage dynamic programming or a pre-computed lookup table to increase efficiency. However, operations such as dynamic indexing that are used in these dynamic programming approaches are not feasible in FHE: the server cannot access the right element of the array, since the index is encrypted. Therefore, there is no way for the server to continue with the execution without asking for hints from the client. The same logic applies to early termination; the server cannot terminate early when executing a search algorithm (e.g., binary search or trie) over the database. Enabling early termination will leak information about the underlying plaintext. For example, when searching for a similar string in a plaintext database, when a match is found with a similarity score that's higher than a certain threshold, the program exits. In FHE, accomplishing this functionality would leak information about the underlying plaintext in the encrypted input.

3.2 FHE-Friendly MinHash Construction

LSH constructions present a promising solution that can mitigate the challenges outlined above. The nature of LSH algorithms is hash-based, and to compute the similarity, we only need to compare the hashes instead of the input data directly. The key benefit of LSH over alternative solutions is the significantly lower memory and latency overheads. Indeed, working over hashes instead of raw DNA sequences results in a substantial size reduction as the size of the hash digests does not depend on the length of the DNA sequence. Therefore, even with DNA sequences with millions of nucleotides, we can generate a fixed number of hashes (the same number for each DNA sequence depending on the parameters used in the LSH algorithm). This translates well to FHE as the client no longer needs to encrypt a DNA sequence of very long length, instead they only need to encrypt a small number of hashes. In this case, the server generates the hashes of each DNA sequence in the database as a one-time preprocessing step. When working with hashes, the number of homomorphic operations needed to compute the similarity between two DNA sequences significantly decreases compared to other approaches that work over much larger DNA sequences.

A variation of MinHash used in Mash [38] is the k bottom-sketch approach, which uses only one hash function and saves the smallest k hashes to store them in a sorted order. Next, computing the similarity score between two different bottom sketches requires a merge sort to find the number of equal hashes between the two bottom sketches. Unfortunately, the algorithm requires dynamic indexing, which is not possible in FHE as discussed. Therefore, we use the classical

MinHash approach that requires k permutations using a uniform and independent hash family as mentioned in Algorithm 1. To apply this algorithm to DNA strings, we first convert each string into a list of all substrings of length s, also known as k-mers [37]. Thus, a DNA string of length n will have $n - s + 1$ different k-mers. A uniform hash function converts each k-mer into a positive 32-bit integer before we apply the MinHash algorithm to compute the signature hash array for each DNA sequence.

Next, the client encrypts their signature array and sends it to the server, which has an encoding of each signature array of its database and uses encrypted multiplexers and adders to compute the similarity between each signature array of the database and the client's signature array. The server returns the maximum similarity value by returning the integer Sim in Algorithm 2, which is the number of equal signature hashes. Since the client knows k, they can derive the approximate Jaccard similarity of the most similar entry of the database to their DNA sequence by dividing by k after decryption. Therefore, with a one-time cost of encoding the server's database, multiple clients with different FHE keys can query the database and get the closest match in constant time and space, compared to the size of the DNA sequences.

3.3 Efficient Encrypted Bloom Filter Evaluation

When querying a database, a straw-man approach is to check all the 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 in $\mathcal{O}(log(n))$ time complexity. However, these algorithms require branching on the input data, but in our case all the inputs are FHE encrypted. Therefore the computing party is unable to make a decision based on the underlying plaintext data of a ciphertext.

With the help of probabilistic data structures like bloom filters, we can scale the database in constant time and $\mathcal{O}(m)$ space with a probability p of false positives, where m is the size of the bloom filter. According to equations in Section 2.4, m scales linearly with the size of the database n. Bloom filters are very efficient in the plaintext domain; moreover, even more efficient probabilistic data structures, such as the Xor filter [28] and binary fuse filter [29], have been proposed. 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 which is not immediately possible in FHE. Therefore, to be able to query a server's bloom filter, we propose a query bloom filter (Qbf), using the same construction as the server bloom filter (Sbf). Both must be the same size m, and use the same k hash functions. The query bloom filter only has one element inserted. To see if that element exists in the Sbf we apply the following computations:

$$\mathbf{bf} = (\mathbf{Qbf} \wedge (\neg \mathbf{Sbf})) \tag{2}$$

$$\mathbf{bit} = \bigvee_{i=1}^{n} \mathbf{bf}[i] \tag{3}$$

Using Equation 2, we first negate all bits in Sbf, and then pairwise `AND` each bit with the Qbf. The resulting bloom filter (bf) should be all zeros if the element exists. If one cell in bf is 1, it means that the corresponding cell in the Qbf was 1 and the Sbf cell was 0 (indicating that the element does not exist). Now we use Equation 3 and `OR` all the bits of bf. If the resulting bit is 0, the element exists. The bitwise operators can be efficiently converted to boolean gates in our target CGGI FHE scheme.

To encode our DNA sequences to bloom filters, we generate a list of k-mers for each DNA sequence. For each DNA in the database, we create a separate Sbf, use k different uniform hash functions for each k-mer, and set the computed indices in the filter to 1. Next, to query a client's DNA sequence, we create a bloom filter for each k-mer in the client's DNA sequence. This way, we can know how many k-mers of the client's DNA exist in the database. The server does all the outlined computations for each Qbf and Sbf, aggregates the resulting bits, and returns an array that shows how many k-mers each DNA entry has in common with the client.

Batched query optimization The previous approach may become less efficient for large client DNA samples since we need to generate a Qbf for every k-mer of the client's DNA. Since every bit is encrypted, evaluating each Qbf separately becomes more time-consuming. To overcome this challenge, we introduce a new batching technique that stores multiple bits instead of single bits in the Qbf. We optimize our bloom filter methodology by storing multiple k-mers of batch size d in a single Qbf. For example, if our batch size is 32, we insert 32 k-mers into the Qbf. This would mean that every cell is a 32 bit integer, and the ith bit in each cell corresponds to the ith k-mer inserted. Equations 2 and 3 remain the same and apply the bitwise operators to integers of bit length d. The result is an integer instead of a bit and the number of 0 bits in the integer represents how many of the d k-mers exist in each Sbf. This optimization yields significantly faster execution times since the bitwise operations can be computed in parallel across the bits of integer ciphertexts, which is further elaborated in Section 4.

4 Implementation Details

Our methodology is implemented with TFHE-rs [47] as the FHE backend, which corresponds to the CGGI cryptosystem [18]. Notably, the CGGI cryptosystem is commonly used to encrypt single bits or very low-precision integers; however, TFHE-rs can encrypt inputs with arbitrary precision by treating large inputs as vectors of individual ciphertexts. In this case, each encryption encodes a digit of the plaintext value (where each digit is 2 bits in size for the default parameter set). We leverage this encoding to optimize our batched bloom filter approach

by computing bitwise operations across the digits of the input ciphertext vectors in parallel.

4.1 Client-side Operations

The client has a sample DNA sequence and wants to check if a similar sample exists in the database. First, the client needs to pre-process the DNA string, then apply either the MinHash or the batched bloom filter methodology, and then encrypt the resulting input using their secret homomorphic key to send to the server.

Pre-processing DNA Samples: For the first step, the client splits each DNA sequence into k-mers of length s. Choosing the value of s is important, as it directly affects the analysis of the similarity results [25]. For our MinHash approach, we opt for value $s = 16$, since it fits in 32 bits.

For the second step, we hash each k-mer using a popular hash function well-suited for hashing substrings into integers called MurmurHash [2]. This is a non-cryptographic hash function that generates a 32-bit, 64-bit, or 128-bit digest depending on the variant utilized. In our implementation, we use the 32-bit MurmurHash3 hash function to convert each k-mer to a 32-bit unsigned integer. This hash offers a good statistical distribution and therefore has a low hash collision probability [46]. After processing our input, we use the MinHash algorithm to generate the hash signatures, and the bloom filter approach to generate query bloom filters, then encrypt them homomorphically and send them to the server for evaluation.

MinHash Preprocessing: As mentioned previously, we use the k-minwise hashing approach of MinHash [10], where k is the number of hash functions used to generate the k different permutations of the input set. To generate different permutations, we employ the Carter-Wegman linear universal hash function [13], which is defined as $h(x) = a \cdot x + b \pmod{p}$, where x is a 32-bit unsigned integer of our input set, p is a prime number, and a and b are parameters that are in the field of $\mathbb{Z}_p$ (i.e., positive integers smaller than the prime). We choose the prime number $p = 2^{32} - 5$, so the hash digest fits in 32 bits. We also choose a and b to be random integers in the range of $[0, p-1]$. Note that the same k hash functions must be used on the server side for the algorithm to work properly. Next, we apply Algorithm 1 to generate a signature array of k hash digests, where each digest is 32 bits in size. We encrypt each 32-bit integer as an `FheUint32` (i.e., the TFHE-rs data type consisting of a vector of ciphertexts encoding a total of 32 bits of information) and send the encrypted signature array to the server.

Bloom Filter Preprocessing: In this case, we use k different 32-bit MurmurHash3 hash functions and seed them with different random numbers. Both the client and server must use the same seeded hash functions, and we apply

a modulus m to the result of each hash function so it fits in our bloom filters of size m. The client also needs to set a batch size d, to generate the batched query bloom filters (Qbf). In Section 5, we show that a batch size of $d = 128$ is the most efficient. Therefore, we batch each 128 k-mers of the client's DNA and generate a Qbf, encrypt each 128-bit integer as a `FheUint128`, and send it to the server to evaluation.

4.2 Server-side Computations

Before the server receives the client's encrypted input, it must initialize the encoded database by pre-computing the signature hash array and bloom filter for each DNA entry. For the MinHash approach, we encode the signatures using FHE, but the bloom filters remain in plaintext. Then we apply the evaluation steps and send the encrypted result back to the user for decryption.

Initialization and Setup: The FHE parameters for both the client and server should be the same (i.e., the same ciphertext polynomial degree and coefficient size), so the server can do homomorphic evaluations between the client's FHE encrypted data and its encoded database. Note that the server does not need to actually encrypt the database; instead, the server generates *trivial LWE encodings* without injecting noise or using a secret key. We note that generating trivial/noiseless encodings results in negligible latencies with TFHE-rs and we further remark that the encoding of the database is a one-time cost, since any client using the same FHE parameters as the server can send their encrypted inputs for evaluation. We also emphasize that the result of a computation between a trivial encryption and a secure, client-encrypted ciphertext will result in a secure ciphertext encrypted under the same client key. Additionally, the server also needs the client's bootstrapping key to perform most HE operations.

Computing Encrypted MinHash Queries: Like the method discussed for the client side, the server generates the hash signatures of each DNA sequence in the database in the plaintext domain. However, the server only needs to generate LWE encodings of the signatures, resulting in an array of s LWE ciphertexts without noise, each representing a 32-bit unsigned integer. Now, to compute the best similarity score, the server compares the encrypted hash signatures of the client with each encoded hash signature of the database, computes the similarity score with each database entry, and saves the maximum result. To avoid information leakage, the similarity score is encrypted. If the similarity score was in the clear, the server would know what hashes of the client are equal to its own, allowing it to potentially recover the original k-mer of the client (given the MurmurHash3 used is non-cryptographic). Even if the hash function used was cryptographically secure, the server could still gain information about the client's input and might be able to brute force it.

To implement Algorithm 2 in FHE, we use encrypted comparators, multiplexers, and adders, and initialize an encrypted similarity value to 0. First, we

use an encrypted comparator to compare two 32-bit ciphertexts. The result is an encrypted bit whose plaintext value corresponds to an encrypted Boolean value depending on whether or not the inputs have the same underlying plaintext. This encrypted bit serves as the control bit of an encrypted multiplexer (`MUX`) that adds 1 to the encrypted similarity value if they're equal and adds 0 otherwise. Since the `MUX` is encrypted, the server will not know if the similarity value was added by 1 or by 0.

Batched Bloom Filter approach: The server bloom filters (Sbf) are generated for each DNA sample of the database, but it does not need to be encoded. When the server receives the query batched bloom filters (Qbf), we apply Equations 2,3 with some additional optimizations. First, the server can negate every Sbf before receiving the data from the client. If we assume the client uses a batch size of 128, the server receives multiple Qbf, each having m 128-bit FHE encrypted integers. Looking closely at Equation 2, when the bit in the negated Sbf is set to 1, the corresponding bit in the resulting bloom filter will be equal to the bit of the Qbf. And if it's 0, the resulting bit will be 0 regardless. Thus, instead of applying expensive bitwise & operation, we save the integers that the corresponding bit in the Sbf is set to 1, and `OR` all the integers together. In this case, we initialize a 128-bit encrypted integer to 0, then we check each bit of the Sbf and if it is set to 1, we `OR` the corresponding integer of the Qbf to our initialized variable. The number of 0s in the resulting integer represents how many k-mers of the 128 exist in that Sbf. Computing the number of 0s in an FHE integer would be very expensive since we need to shift every bit and apply an encrypted multiplexer. As a result, we send the resulting integers of each Qbf and Sbf back to the user, and the user can decrypt the integers and easily count the number of 0s of every integer in plaintext, then aggregate the results to find the number of common k-mers with every entry.

5 Experimental Evaluation

We perform a series of experiments to showcase the efficacy of both our MinHash and bloom filter approaches. All CPU-based experiments were performed on an `r5.12xlarge` AWS EC2 instance with 48 vCPUs and 384 GB of RAM. For GPU experiments, we utilize an NVIDIA GeForce RTX 4080. Both approaches are implemented using the high-level API provided by the TFHE-rs library [47], which is a Rust implementation of the TFHE (CGGI) scheme [18]. We use the default cryptographic parameters provided by the API, which corresponds to 128 bits of security.

5.1 Encrypted MinHash Query Evaluation

The two main parameters that affect the execution times in our MinHash approach are the number of hash functions k, and the size of the database n. Each

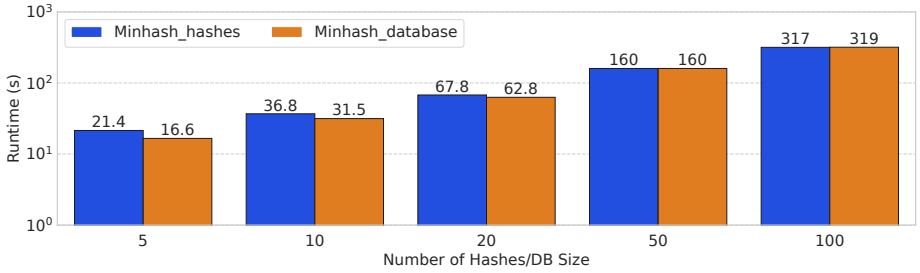

Fig. 2. CPU evaluation of our MinHash approach by varying the k parameter (left bars) and the n parameter (right bars).

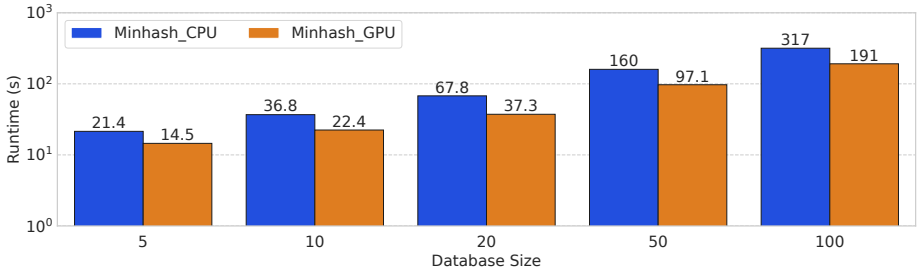

Fig. 3. CPU and GPU evaluations of our MinHash approach with $k = 100$.

database entry will have k integers, therefore every query needs to do k comparisons for each entry in the database. For all experiments, we use k-mers of size 16, and 32 bit encrypted integers. We expect linear growth in execution times as the size of the database or the number of hash functions grows. As shown in Figure 2, we observe a linear increase in latency when increasing the number of hash functions and the database size. Also, we can see very similar execution times for both evaluations, since $n = 100$ and $k = 100$ for the *MinHash_hashes* and *MinHash_database* evaluations respectively. The number of comparisons needed for both is $n \times k$, so their execution times should intuitively be very similar, as shown in Figure 2. Next, we compare the CPU implementation of MinHash with its GPU counterpart, which is implemented utilizing the CUDA-accelerated FHE operations implemented in TFHE-rs. As shown in Figure 3, we report that the GPU implementation outperforms the CPU by roughly 2×.

5.2 Bloom Filter Evaluation

The three parameters that affect execution times in our batched bloom filter approach are the batch size, the size of the bloom filter, and the size of the database. Note that the number of hash functions has no effect since the whole bloom filter is encrypted by the client and evaluated by the server. For all experiments, we use DNA sequences of length 143 and divide them to k-mers of

length 16, which results in 128 k-mers. According to the equations in Section 2.4, with a desired false positive probability of $p = 0.01$, the bloom filter size is $m = 1227$ and the number of hash functions is 7.

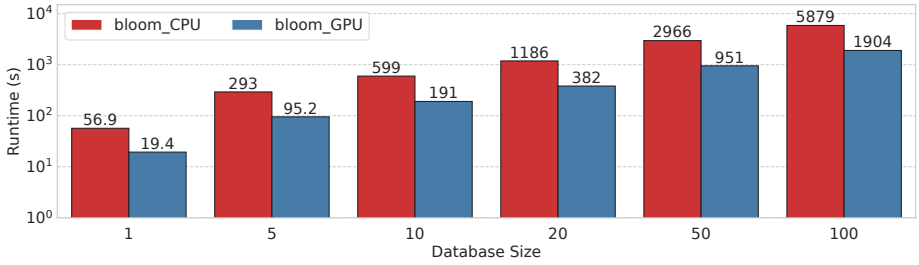

Fig. 4. CPU and GPU evaluations of our batched bloom filter approach with different database sizes.

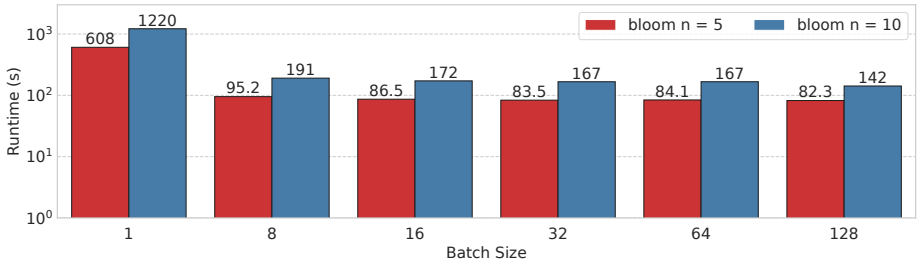

Fig. 5. GPU evaluation of our batched bloom filter approach with different batch sizes. Here n represents the size of the database.

We compare the CPU and GPU implementations for our approach by using a batch size of 8 since it is highly parallelizable. Figure 4 shows a linear increase in execution time on both the CPU and GPU, with the GPU outperforming the CPU by roughly 3×. To determine the best batch size, we evaluate our approach with different batch sizes using the GPU implementation and compare it with the default Boolean bloom filter. Figure 5 shows that a batch size of 128 has the fastest speed, and is close to an order of magnitude faster than the baseline approach of batch size 1. The batch sizes have similar execution times since each FHE integer consists of a vector of ciphertexts that encrypts 2 bits of information and their operations are highly parallelized. Therefore, for larger FHE integers, there are more ciphertexts in the vector, which results in more parallelism for bitwise operations.

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. [41]

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. [44]; 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. The 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 operations.

There are multiple works on privacy-preserving bloom filters. Feng et al. [24] 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 [24], the authors utilize encryption parameters that correspond to negligible security, whereas our work corresponds to the current security standard of 128 bits of security. Chielle et al. [16] 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, it is the client who computes the hash of the input and then derives the positions of the ciphertexts to be returned, which requires no FHE computation on the server-side. We remark that this approach requires N copies of the database (in encrypted form) for N clients, while our approach only requires a single encoded database. Lastly, Stanciu et al. [42] propose using partial homomorphic encryption and bloom filters to do privacy-preserving crowd monitoring. Unlike our solution, their protocol requires the server to build a new bloom filter for each new query response from a client.

Finally, several works have employed leveled homomorphic encryption to do secure genome analysis. Ziegeldorf et al. [48] utilizes 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. While they exhibit low latency, they only use 80 bits of security, while 128 bits is the current standard. Gursoy et al. [30] 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

In this work, we propose two distinct approaches for privacy-preserving DNA matching designed specifically for efficient evaluation with fully homomorphic encryption. The first approach, based on MinHash, scales linearly with both the size of the database and the number of hashes utilized in the construction. With our GPU-accelerated implementation, we are able to perform an encrypted query

in approximately three minutes across a database consisting of one hundred DNA samples. Our second approach consists of a custom batched bloom filter algorithm, which significantly outperforms the baseline bloom filter approach for larger batch sizes.

Acknowledgments

This work has been supported by NSF Award #2239334.

References

1. Apple: Apple-Neuralhash for 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` (2008)
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**, 103815 (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. Bag, S., Kumar, S.K., Tiwari, M.K.: An efficient recommendation generation using relevant jaccard similarity. Information Sciences **483**, 53–64 (2019)
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
7. Boudguiga, A., Stan, O., Sedjelmaci, H., Carpov, S.: Homomorphic encryption at work for private analysis of security logs. In: ICISSP. pp. 515–523 (2020)
8. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM **20**(10), 762–772 (1977)
9. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
10. Broder, A.: On the resemblance and containment of documents. In: Proceedings of the Compression and Complexity of Sequences 1997. p. 21. SEQUENCES '97, IEEE Computer Society, USA (1997)
11. Cali, D.S., Kalsi, G.S., Bingöl, Z., Firtina, C., Subramanian, L., Kim, J.S., Ausavarungnirun, R., Alser, M., Gomez-Luna, J., Boroumand, A., et al.: Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 951–966. IEEE (2020)
12. Carpov, S., Nguyen, T.H., Sirdey, R., Constantino, G., Martinelli, F.: Practical privacy-preserving medical diagnosis using homomorphic encryption. In: 2016 ieee 9th international conference on cloud computing (cloud). pp. 593–599. IEEE (2016)
13. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. In: Proceedings of the ninth annual ACM symposium on Theory of computing. pp. 106–112 (1977)
14. Chen, L., Lu, S., Ram, J.: Compressed pattern matching in dna sequences. In: Proceedings. 2004 IEEE Computational Systems Bioinformatics Conference, 2004. CSB 2004. pp. 62–68. IEEE (2004)

15. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017. pp. 409–437. Springer International Publishing, Cham (2017)
16. Chielle, E., Gamil, H., Maniatakos, M.: Real-time private membership test using homomorphic encryption. In: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1282–1287. IEEE (2021)
17. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 3–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
18. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Tfhe: fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)
19. Cosseron, O., Hoffmann, C., Méaux, P., Standaert, F.X.: Towards case-optimized hybrid homomorphic encryption: Featuring the elisabeth stream cipher. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 32–67. Springer (2022)
20. Cramer, R., Damgård, I.B., et al.: Secure multiparty computation. Cambridge University Press (2015)
21. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the twentieth annual symposium on Computational geometry. pp. 253–262 (2004)
22. Du, W., Zhan, Z.: A practical approach to solve secure multi-party computation problems. In: Proceedings of the 2002 workshop on New security paradigms. pp. 127–135 (2002)
23. Ducas, L., Micciancio, D.: Fhew: Bootstrapping homomorphic encryption in less than a second. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. pp. 617–640. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
24. Feng, Y., Lu, Z., Cao, Q.: Secure sharing of private locations through homomorphic bloom filters. In: 2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing,(HPSC) and IEEE International Conference on Intelligent Data and Security (IDS). pp. 107–113. IEEE (2018)
25. Fofanov, Y., Luo, Y., Katili, C., Wang, J., Belosludtsev, Y., Powdrill, T., Belapurkar, C., Fofanov, V., Li, T.B., Chumakov, S., et al.: How independent are the appearances of n-mers in different genomes? Bioinformatics **20**(15), 2421–2428 (2004)
26. Geil, A., Farach-Colton, M., Owens, J.D.: Quotient filters: Approximate membership queries on the gpu. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 451–462. IEEE (2018)
27. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing. p. 169–178. STOC '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1536414.1536440, `https://doi.org/10.1145/1536414.1536440`
28. Graf, T.M., Lemire, D.: Xor filters: Faster and smaller than bloom and cuckoo filters. Journal of Experimental Algorithmics (JEA) **25**, 1–16 (2020)
29. Graf, T.M., Lemire, D.: Binary fuse filters: Fast and smaller than xor filters. Journal of Experimental Algorithmics (JEA) **27**(1), 1–15 (2022)
30. Gürsoy, G., Chielle, E., Brannon, C.M., Maniatakos, M., Gerstein, M.: Privacy-preserving genotype imputation with fully homomorphic encryption. Cell systems **13**(2), 173–182 (2022)

31. Jafari, O., Maurya, P., Nagarkar, P., Islam, K.M., Crushev, C.: A survey on locality sensitive hashing algorithms and their applications. arXiv preprint arXiv:2102.08942 (2021)
32. 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)
33. Knuth, D.E., Morris, Jr, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM journal on computing **6**(2), 323–350 (1977)
34. Murugesan, M., Jiang, W., Clifton, C., Si, L., Vaidya, J.: Efficient privacy-preserving similar document detection. The VLDB Journal **19**(4), 457–475 (2010)
35. Navarro, G.: A guided tour to approximate string matching. ACM computing surveys (CSUR) **33**(1), 31–88 (2001)
36. Nguyen, H.V., Bai, L.: Cosine similarity metric learning for face verification. In: Asian conference on computer vision. pp. 709–720. Springer (2010)
37. Nordström, K.J., Albani, M.C., James, G.V., Gutjahr, C., Hartwig, B., Turck, F., Paszkowski, U., Coupland, G., Schneeberger, K.: Mutation identification by direct comparison of whole-genome sequencing data from mutant and wild-type individuals using k-mers. Nature biotechnology **31**(4), 325–330 (2013)
38. Ondov, B.D., Treangen, T.J., Melsted, P., Mallonee, A.B., Bergman, N.H., Koren, S., Phillippy, A.M.: Mash: fast genome and metagenome distance estimation using minhash. Genome biology **17**(1), 1–14 (2016)
39. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM) **56**(6), 1–40 (2009)
40. Régnier, M.: Knuth-morris-pratt algorithm: an analysis. In: International Symposium on Mathematical Foundations of Computer Science. pp. 431–444. Springer (1989)
41. Shimizu, K., Nuida, K., Rätsch, G.: Efficient privacy-preserving string search and an application in genomics. Bioinformatics **32**(11), 1652–1661 (2016)
42. Stanciu, V.D., Steen, M.v., Dobre, C., Peter, A.: Privacy-preserving crowd-monitoring using bloom filters and homomorphic encryption. In: Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking. pp. 37–42 (2021)
43. Trama, D., Clet, P.E., Boudguiga, A., Sirdey, R.: A homomorphic aes evaluation in less than 30 seconds by means of tfhe. In: Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 79–90 (2023)
44. Wang, B., Song, W., Lou, W., Hou, Y.T.: Privacy-preserving pattern matching over encrypted genetic data in cloud computing. In: IEEE INFOCOM 2017-IEEE Conference on Computer Communications. pp. 1–9. IEEE (2017)
45. Weir, B.S.: Matching and partially-matching dna profiles. Journal of Forensic and Sciences **49**(5), JFS2003039–6 (2004)
46. Yamaguchi, F., Nishi, H.: Hardware-based hash functions for network applications. In: 2013 19th IEEE International Conference on Networks (ICON). pp. 1–6. IEEE (2013)
47. Zama: TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data (2022), `https://github.com/zama-ai/tfhe-rs`
48. Ziegeldorf, J.H., Pennekamp, J., Hellmanns, D., Schwinger, F., Kunze, I., Henze, M., Hiller, J., Matzutt, R., Wehrle, K.: Bloom: Bloom filter based oblivious outsourced matchings. BMC Medical Genomics **10**, 29–42 (2017)