MatcHEd: Privacy-Preserving Set Similarity based on MinHash

Rostin Shokri, Charles Gouert, and Nektarios Georgios Tsoutsos

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

Abstract. Fully homomorphic encryption (FHE) enables arbitrary computation on encrypted data, but certain applications remain prohibitively expensive in the encrypted domain. As a case in point, comparing two encrypted sets of data is extremely computationally expensive due to the large number of comparison operators required. In this work, we propose a novel methodology for encrypted set similarity inspired by the MinHash algorithm and the CGGI FHE scheme. Doing comparisons in FHE requires comparators and multiplexers or an expensive approximation, which further increases the latency, especially when the goal is to compare two sets of data. The MinHash algorithm can significantly reduce the number of comparisons required by employing a special Carter-Wegman (CW) hash function as a key building block. However, the modulus operation in the CW hash becomes another key bottleneck because the encrypted sub-circuits required to perform the modular reduction are very large and inefficient in an FHE setting. Towards that end, we introduce an efficient bitwise FHE-friendly digest function (FFD) to employ as the cornerstone of our proposed encrypted set-similarity. In a Boolean FHE scheme like CGGI, the bitwise operations can be implemented efficiently with Boolean gates, which allows for faster evaluation times relative to standard Carter-Wegman constructions. Overall, our approach drastically reduces the number of comparisons required relative to the baseline approach of directly computing the Jaccard similarity coefficients, and is inherently parallelizable, allowing for efficient encrypted computation on multi-CPU and GPU-based cloud servers. We validate our approach by performing a privacy-preserving plagiarism detection across encrypted documents.

Keywords: Secure computation · Fully homomorphic encryption · Privacy-preserving set similarity.

1 Introduction

With the growth of third-party cloud providers, the privacy of the outsourced data stored in these servers becomes a large concern for clients and must be promptly addressed. As a motivating example, a curious cloud provider can plausibly see the data stored on their servers to support targeted advertisement. Additionally, cloud servers have drawn the attention of attackers because sensitive information from several clients can reside on the same server. Existing threats such as cache-based side channel attacks [42, 43], RowHammer attacks, as well as other DRAM-based attacks [28, 29] can potentially affect and leak private data on multi-tenant machines, which include most cloud services today.

One possible way to address this challenge is to use symmetric encryption schemes such as AES [13], as these encryption algorithms can help us secure the sensitive data stored in a cloud provider's server [1]. Although symmetric encryption algorithms like AES provide strong security guarantees and relatively fast encryption and decryption overheads, they do not allow computation over encrypted data. Therefore, to apply meaningful operations on encrypted data that is stored on the cloud server, we need to retrieve the data from the cloud provider, decrypt the data, compute on the plaintext, and then re-encrypt it and upload it back to the cloud service, which is time-consuming and inefficient.

Fortunately, a more versatile form of cryptography, dubbed homomorphic encryption (HE), allows us to do computation directly on encrypted data. More concretely, HE allows a user to encrypt modular integers, bits, or floating point numbers and the resulting ciphertexts are *malleable* by design. Notably, modern HE schemes, called fully homomorphic encryption (FHE), allow for *any arbitrary computation* on encrypted data by supporting both addition and multiplication or a set of functionally complete Boolean operators.

Even though arbitrary encrypted computation is possible using this method, some algorithms that require knowledge of the underlying data, such as many sorting algorithms like QuickSort, are infeasible or at least very expensive due to the *termination problem* [20,33]. In particular, the termination problem states that the computing party (e.g., a cloud server) is unable to make a branching decision based on encrypted data, as any information related to the underlying plaintext can not be deduced without the client's decryption key (which is not shared). Likewise, algorithms that rely heavily on computing comparisons between encrypted values are largely impractical due to the computational overhead of approximating or exactly computing comparison operators in the encrypted domain. Unfortunately, comparing two encrypted values is non-trivial; for homomorphic encryption schemes over the integers, one can utilize a polynomial interpolation, such as the one proposed by Iliashenko and Zucca [25] for an encrypted less-than operation, which grows significantly more expensive as the plaintext modulus increases, rendering this approach infeasible for many applications that require high data precision. On the other hand, for FHE schemes where individual bits are encrypted as independent ciphertexts, one can utilize a Boolean comparator circuit to perform the equivalent computation. Regardless, the comparator circuits are quite large and exhibit high latency for Boolean FHE schemes.

Nevertheless, one of the major challenges in modern privacy-preserving methods like homomorphic encryption is the ability to compute similarities between sets of encrypted data without leaking any information about the underlying plaintext. In many privacy-aware applications, such as finding similar DNA sequences [35], finding similarities over proprietary or sensitive images [31], as well as detecting plagiarism across private documents [34], the confidentiality of the plaintext is a major goal.

A simple solution to finding similarities between two datasets is to compare each element of the first set with all the elements of the second set. This approach requires $\mathcal{O}(n * m)$ time complexity, where n and m are the sizes of the two sets. As a result, the computation time will massively increase when the sets scale to larger sizes. Conversely, locality sensitive hashing (LSH) offers a more efficient, heuristic-based approach for finding similarities between datasets [16, 30] and can result in computing altogether fewer comparison operations. In more detail, LSH is a hashing-based technique that can efficiently approximate the similarity between datasets based on some metric, such as the *Jaccard similarity index* [2]. LSH algorithms apply hash functions on the input data so that similar data points will hash to the same or nearby hash codes with high probability; this can significantly accelerate the computation costs compared to simpler, brute-force methods. Moreover, LSH algorithms offer high accuracy and high-performance if parameterized correctly.

In this paper, we propose a technique inspired by the MinHash LSH algorithm [6] to find similar datasets in a privacy preserving way using FHE. The MinHash set-similarity algorithm focuses on *estimating* the Jaccard similarity index between two sets, and the core operations required are computing hash digests and finding the minimum digest. The construction utilized to produce the digests is required to provide different mappings over the dataset, while the minimum operation finds the minimum digest (i.e., the signature) of a set. In this approach, we only need to compare the signatures generated for each set instead of comparing all elements with each other.

The function typically used for MinHash is a linear universal hash function such as the Carter-Wegman (CW) hash [9]. While this is one of the simplest forms of hashing, it is very fast compared to other hash functions and offers sufficiently-random mapping when used in an LSH algorithm. However, when generating a digest in the encrypted domain using Boolean-based FHE, this approach becomes prohibitively expensive because of the modular reductions in the CW construction, and the need for large multi-bit arithmetic circuits. Likewise, due to the nature of the MinHash algorithm, the modular reduction operation is used extensively. Therefore, to address this challenge and to introduce a MinHash variant specifically optimized for FHE, we introduce a judiciously designed (FFD) construction that is tailored for FHE performance. Our key observation is that arithmetic-based operations that include non-linear operations (such as the modulo operator) are not well-suited for FHE. On the other hand, bitwise operations can be directly translated to logical gates and yield relatively small Boolean circuits, which can be evaluated efficiently using FHE.

Overall, our contributions can be summarized as follows:

- Design of efficient and accurate digest generation techniques that are tailored for encrypted evaluation of LSH algorithms.

– A novel methodology for set-similarity in the encrypted domain using FHE.
– New strategies for efficient and parallelizable comparison operations with Boolean-based FHE on CPU and GPU targets.

Roadmap: The rest of the paper is organized as follows: Section 2 provides necessary background on FHE and the MinHash algorithm, while Section 3 highlights challenges for implementing MinHash in the encrypted domain. Section 4 presents our proposed methodology and considerations of implementing MinHash in the encrypted domain, as well as possible trade-offs for increasing the efficiency and accuracy of the encrypted set-similarity algorithm, while Section 5 discusses our experimental evaluation using plagiarism detection benchmarks 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 Primer

An encryption scheme with malleable ciphertexts that enable some form of computation directly on ciphertext data falls under the umbrella of homomorphic encryption. However, not all homomorphic cryptosystems exhibit the same properties; the HE schemes can be sub-divided into three distinct categories that indicate the computational power of the homomorphism: partial HE (PHE), leveled HE (LHE), and fully HE (FHE). PHE is the oldest form of HE but is limited in its computational abilities. In more detail, a PHE scheme allows for unbounded addition or multiplication, *but not both.* This makes it well-suited for specific applications like data aggregation but is not suitable for complex algorithms like the set-similarity techniques proposed in this paper.

Unlike PHE, LHE allows for both addition and multiplication and is therefore capable of performing arbitrary computation as these two operations form a functionally complete set. LHE ciphertexts in all popular schemes, such as BGV [5] and CKKS [11], take the form of tuples of high-degree polynomials where each coefficient is an integer modulo a large composite number (i.e., a product of primes) called the ciphertext modulus (typically several hundred bits in length). Notably, the security of LHE schemes typically relies on the LWE problem [36] or its ring variant [32], which entails adding a small amount of random noise to the coefficients of the ciphertext polynomials during encryption. An important consequence of the presence of noise in the ciphertexts is that the magnitude of the noise grows as the ciphertexts are computed upon; in simple terms, the noise increases slightly when adding ciphertexts, and significantly when multiplying ciphertexts. If the computation requires many subsequent multiplications to be computed over ciphertext data, the noise will start to corrupt the underlying message with high probability, and the final decrypted result will be non-deterministic.

FHE solves the noise growth problem and is the most powerful form of HE in terms of computational abilities: FHE schemes allow for unlimited operations on ciphertext data for a fixed parameter set. In particular, any LHE scheme can be transformed into an FHE scheme through the introduction of a *bootstrapping* mechanism [17]. Nevertheless, bootstrapping incurs a large computational overhead with respect to other HE operations and constitutes the core bottleneck of FHE evaluation. For instance, for the BGV and CKKS cryptosystems, a single bootstrapping operation can take several seconds to several minutes on a CPU, depending on the choice of parameters [21]. The FHEW cryptosystem [15] was introduced to address the latency problems in bootstrapping and was the first Boolean FHE scheme. This class of FHE schemes encrypts individual bits, as opposed to integers or floating point numbers, and allows for the evaluation of encrypted logic gate operations. The bootstrapping procedure in FHEW can be evaluated in less than a second on a CPU and serves a key computational role in the evaluation of each logic gate, so it must be invoked for each gate operation. Likewise, the CGGI cryptosystem [12] builds upon FHEW and is capable of achieving even faster bootstrapping speeds of less than 10 milliseconds per bootstrap on a CPU. For this reason, we opt to utilize the CGGI cryptosystem in this work.

2.2 Locality-sensitive Hashing

The premise of locality-sensitive hashing (LSH) is that similar items tend to have the same or nearby hash values with high probability. When the size of the data is large, the complexity of finding similar items using brute force methods (i.e., comparing all entities one by one) becomes impractical. LSH reduces the complexity by using special hash functions that map similar items to the same "bucket". Therefore, instead of comparing the items directly, one can compare the hash values and items that have similar hash values are considered similar. Typical LSH algorithms are able to compute the similarity of two sets using metrics, such as the Jaccard similarity of these sets. Specifically, the Jaccard similarity J between two sets A and B is defined as:

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

By definition, we note that $0 \leq J(A, B) \leq 1$ for any A and B, and in this paper, we define A and B to be the set of positive integers. An important consideration about Jaccard similarity is its poor scalability in terms of time and space required as the size and number of the datasets increase.

This challenge is addressed by MinHash, which is a probabilistic algorithm that relies on special hash functions to compute *signatures* for each set and then compares them to assess the level of similarity between the two sets. Notably, MinHash is a high-accuracy estimation of Jaccard similarity, so instead of using the Jaccard function directly to find similarity between each pair of data sets, MinHash generates hash signatures for each set and then compares the signatures pair-wise to find similar datasets. Therefore, even if the size and number of the datasets grow, MinHash becomes increasingly more efficient than the Jaccard similarity.

Minhash has two major components: the computation of primitive hash functions (used to generate mappings of the input set), and finding the minimum hash value of each hash function by comparing all generated hash values. We remark that in our implementation of the MinHash algorithm, instead of storing each hash digest and sorting a list of digests to find the minimum, we generate each digest on the fly and compare hash values as we go and always save the minimum value. The MinHash algorithm that computes hash signatures for each set is presented in Algorithm 1:

Algorithm 1 MinHash Algorithm

```
Input: List of sets $S$; Number of hash functions $k$
Output: List of MinHash signatures for each set
1: procedure MinHash($S, k$)
2:     Initialize matrix $M$ of size $k \times |S|$ to $\infty$
3:     for $i \leftarrow 1$ to $k$ do
4:         for each set $s$ in $S$ do
5:             for each element $x$ in $s$ do
6:                 $hashValue \leftarrow hash_i(x)$
7:                 $M[i, s] \leftarrow \min(M[i, s], hashValue)$
8:     return $M$
```

In sum, Algorithm 1 uses k different hash functions and a list of sets S. For each set $s \in S$, it computes a list of hash signatures of length k, stores them in matrix M, and returns it. Now, to estimate the similarity between two sets we compare the signatures; the number of equal signatures divided by the number of hashes constitutes the final similarity result. If the hash functions chosen have low collision probabilities and k is sufficiently large, then our output will be an accurate approximation of the exact Jaccard similarity of the two sets [8]. The time complexity of generating hash signatures for a set of size n and k different hash functions is $\mathcal{O}(k \cdot n)$. Then to compare the two sets we only compare the hash signatures which will be of complexity $\mathcal{O}(k)$. In almost all cases, the number of hash functions k will be small so that the complexity of the comparison of the MinHash signatures of two sets will be close to constant time. Conversely, the time complexity of the Jaccard similarity of two sets with sizes n and m when implemented efficiently is $\mathcal{O}(min(n, m))$. Thus, if we

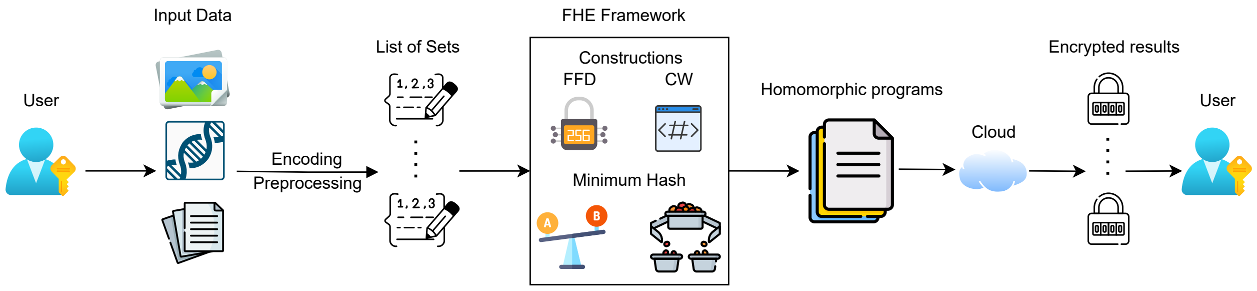

Fig. 1. Framework Overview: The user encodes data into sets, encrypts them, and then sends them to the cloud, which can run MinHash in the encrypted domain with either the CW or FFD construction. Finally, the encrypted set similarity measure is returned to the user for decryption.

only have one pair of datasets, the Jaccard similarity will likely be faster, but when scaling to thousands or millions of pairs, then MinHash has a significant performance advantage. This is because MinHash generates the hash signatures for each dataset *only once*, and reuses those signatures to find similarity between each pair of datasets in roughly constant time. Conversely, Jaccard similarity requires linear time to compute the similarity between a pair of sets, so as the number of pairs grows, Jaccard becomes very expensive.

2.3 Threat Model

In this work, we assume an honest but curious cloud service provider that is incentivized to view sensitive data uploaded by a client, but will not deviate in any way from the protocol (i.e., the prescribed MinHash algorithm). From a confidentiality perspective, the only information the computing party can gather is the size of the underlying plaintext since each ciphertext encrypts a single bit of information. Overall, our threat model is stronger than approaches involving more than one server (e.g., secure multiparty computation), where colluding servers could potentially leak sensitive data.

Note that our security guarantees come from the FHE scheme itself. Therefore, the MinHash algorithm and the digest generation functions utilized in the algorithm have no impact on data confidentiality. All digests and intermediate values are homomorphically encrypted and rely on the security of the LWE problem mentioned previously. Parameterizing MinHash and the constituent functions solely relates to the correctness of the plagiarism detection.

3 Privacy-Preserving Set Similarity with MinHash

Adopting MinHash in the encrypted domain comes with a set of challenges that need to be addressed. To run MinHash homomorphically, we need to be able to compute the digests and find the minimum values in the encrypted domain. Another challenge is defining the size of the input datasets, which cannot change dynamically due to the nature of underlying Boolean FHE circuits, which are synthesized using logic synthesis frameworks to leverage their rigorous logic optimizations. In more detail, a Boolean 4-bit adder can not be used to directly compute the addition of 16-bit operands. Instead, the 4-bit adder can be utilized as a building block to construct the much larger circuit for a 16-bit adder. Therefore, the size of each set must be initialized at compile time, resulting in multiple circuits for evaluating sets with different sizes. As a consequence, one must re-run the logic synthesis process each time the structure and sizes of the input sets change, but we remark that this is a one-time cost for a specific set size configuration and the synthesized circuit can be re-used an arbitrary number of times in this case.

A general methodology for implementing MinHash in the encrypted domain is illustrated in Figure 1. As discussed earlier, the input can be any type of data that can be interpreted as sets, such as images, documents, and DNA samples. Since the MinHash algorithm operates on sets of integers as input, the data must be processed and encoded accordingly (our encoding process for documents is elaborated in Section 5). After the input pre-processing phase, the encoded sets are processed by our proposed framework that implements MinHash in FHE and generates equivalent homomorphic programs, which take the form of Boolean circuits due to our use of the CGGI cryptosystem [12]. As elaborated in the next subsections, we employ FFD and the Carter-Wegman hash homomorphically, along with efficient FHE comparison modules that process all encrypted digests and return the minimum digests without leaking any information about the underlying plaintext. Subsequently, our homomorphic programs (i.e., digest computation, comparators) and the encrypted user data are sent to the cloud for homomorphic evaluation and the encrypted results will be sent back to the user for final decryption.

3.1 LSH over Encrypted Data

As mentioned earlier, one key motivation for focusing on locality sensitive hashing algorithms, like MinHash, is the efficiency and scalability limitations of precise approaches like Jaccard similarity. Since the FHE programming model of CGGI expresses algorithms as Boolean circuits (with FHE logic gates), we can leverage electronic design automation (EDA) techniques to generate and optimize the gate netlists.

In our methodology, we construct the FHE-friendly MinHash algorithm in C++ and leverage the Google XLS compiler [14] to generate a Verilog program. Next, the XLS Verilog code is optimized by the Yosys RTL synthesis suite [38] and serves as input to the HELM framework that generates an optimized fully homomorphic circuit [19]. We remark that the generic synthesis flows will typically yield gates not directly compatible with the HELM FHE backend. To compensate for this, we perform the logic synthesis process and then insert an extra technology mapping step to force the cells to be standard 2:1 logic gates, instead of the larger compound gates that may be inserted during the standard flow. We further insert another optimization pass before finalizing the netlist. The encrypted programs (along with the FHE-protected datasets of the client) are uploaded to a cloud server that employs HELM's FHE evaluation engine so that all computations involving the sensitive datasets are end-to-end encrypted and the cloud service has no knowledge of the underlying plaintext. As soon as the encrypted result is computed and returned, the client can decrypt it using their secret key. All FHE computations occur completely offline, with the client only incurring the cost of key generation and the minor costs of encryption and decryption.

3.2 Choice of Primitive Constructions

The functions that compute digests are needed in MinHash to generate different mappings of the input datasets to compute their unique signatures. In this work, we employ two distinct constructions: the first is a Carter-Wegman (CW) hash, while the second is a bespoke FHE-Friendly Digest (FFD) construction. In more detail, the CW hash of input x is defined as:

$$h(x) = (ax + b) \mod p, \tag{2}$$

where p is a sufficiently large prime number, $a, b \in Z_p$ are parameters, and Z_p is a finite group based on prime p.

The standard CW construction offers a simple design with low collision probability that allows generating unique digests over the input x and can be implemented in FHE since all operations of Eq. 2 can be directly translated into Boolean circuits. Nevertheless, our analysis shows that generating a homomorphic program from the CW construction results in significantly oversized sub-circuits due to the inclusion of the modular reduction operator that grows quickly with the word size.

Therefore, we observe that the key to implementing a digest function that runs efficiently in FHE is to minimize the size of the generated Boolean circuits for homomorphic evaluation (and thus improve FHE run-time performance), necessitating a simple construction that can be configurable in order to generate different mappings with a different initialization parameter (to generate k unique digest mappings). To this end, we introduce an FHE-friendly digest (FFD) function based on bit-wise operators, where instead of modular reduction we mask the digests using the AND operation. Indeed, this masking behaves like a modular reduction by a power of 2^d, as long as the masking value is a Mersenne number 2^{d-1}, where d is the bitsize of the input. In the FHE domain, computing the bitwise AND is significantly cheaper than performing a modular reduction circuit by a factor of potentially hundreds or thousands of gates depending on the wordsize.

To successfully construct a suitable FFD function that works well with the MinHash algorithm, it needs to be injective and uniform to achieve optimal accuracy [18]. In this context, injective means that each digest maps to only one input, with no collisions for bounded input sizes. In addition, the uniformity requirement means that each input has an equal chance of being chosen as the minimum element. Therefore, to enable our performance optimizations, we choose an FFD function as described in Eq. 3, and show its uniformity and injective properties. In particular, to enable uniformity, we apply a fixed permutation on the input, which shuffles the input bits. Moreover, to support the ability to create a family of digests, we apply a bias term, similar to the constant term of the CW hash; in effect, this approach yields different permutations of the input. To mitigate any overflows caused by the bias, we mask the output using the Boolean AND (&) operator. Indeed, the permutation operation has almost negligible cost in CGGI, as it translates to simply copying the encrypted bits to the permuted order. Similarly, the addition and the AND (&) operations directly translate to a Boolean adder and encrypted AND gates. Our FFD-based digest is defined as follows:

$$h(x) = (\Pi(x) + b) \ \& \ p, \tag{3}$$

where $\Pi(\cdot)$ is a permutation mapping d bits to d bits, b is a constant parameter and p is our masking value, which is not necessarily a prime. Without loss of generality, we can instantiate $\Pi(\cdot)$ as a rotation operation $\lll$ by $d/2$ bits, where d is the number of bits in the input. Specifically, our FFD construction rotates the input to the left by half of the bit size (effectively swapping the left half bits of the input with the right half), then adds the rotated input to the constant parameter b, and finally applies the AND operator to limit the size of the output. Likewise, our choice for the masking value as $p = 2^d - 1$, where d is the number of bits in the output digest, ensures that the bitsize of the input of $h(\cdot)$ equals the bitsize of the output digest. Unlike the CW construction, the value of p does not have to be a prime, as any bits of p that are set to 0 can skew the output digests to a smaller range, which can create collisions; in this case, the form $p = 2^d - 1$ is an optimal choice.

As expected, the output of the FFD construction (Eq. 3) is a permutation (i.e., it is bijective) of the input x, and the different mappings depend on the value of parameter b, as visualized in the example of Figure 2. For a visual example of the family of permutations, we can assume $p = 15$, $d = 4$, $\Pi(x) = x \lll 2$, and an input set $x \in \{0, 1, 2, \ldots, 15\}$, and parameters $b = 1$ and b=12. For $b = 1$, After a left rotation by 2 and a bias of 1, our output set will look like $\{1, 5, 9, \ldots, 0\}$. The minimum digest 0 corresponds to the last element of the input set which is 15. However, if we choose $b = 12$, our output set will look like $\{12, 0, 4, \ldots, 11\}$. Here, the second element of the input set which is 1 corresponds to the minimum digest 0. Therefore, by varying the b parameter value, a different element will be mapped to the minimum digest. In the context of MinHash, if a different dataset includes the same element at that index, then their minimum digests can become equal. At the same time, we observe that the size of the b parameter value needs to be commensurate to the size of p, to have a substantial impact on the output digest. Therefore, without loss of generality, in this work we choose our b parameter to be in the range $[p/2, 3p/2]$.

In the general case, we can also show that our FFD construction is uniform. As each input gets mapped to a different value after the initial permutation, and then it is added to a constant, each digest will only have one pre-image. To demonstrate the uniformity of our function, it is sufficient to show that each digest appears with an equal probability. As shown in Eqs. 4-7, if we assume two inputs x and y, for the minimum to be equal, the outputs of the FFD function have to be equal. In this case, we can show that the corresponding permutations Π should also be equal, which is only possible if inputs x and y are equal. Therefore, the number of different digests equals the number of inputs, and for 2^d inputs, the frequency of each digest is $\frac{1}{2^d}$.

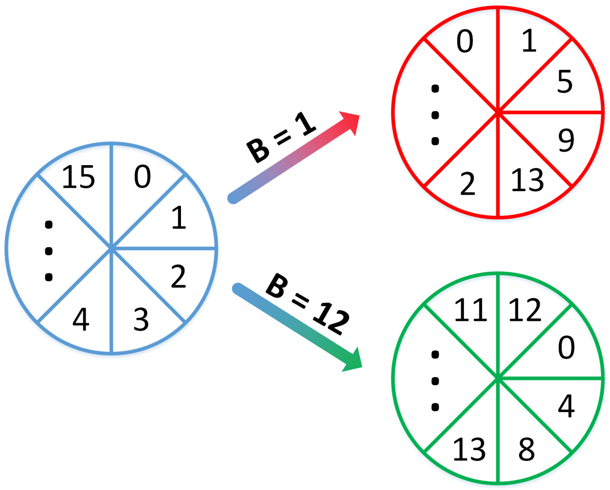

Fig. 2. Minimal example of how an FFD function maps inputs to digests, assuming $\Pi(x) = x \lll 2$. Different b parameters result in different mappings over the input set, which means that a different element of the input set will correspond to the minimum digest.

$$h(x) \equiv h(y) \qquad \pmod{2^d} \tag{4}$$
$$\Pi(x) + b \equiv \Pi(x) + b \pmod{2^d} \tag{5}$$
$$\Pi(x) = 2^d \cdot k + \Pi(y) \qquad (\text{for some } k \in \mathbb{N}) \tag{6}$$
$$\Pi(x) = \Pi(y) \qquad (\text{since } |\Pi(x)| < 2^d \text{ for any } x) \tag{7}$$

The FFD construction used in these equations uses the modulus 2^d as its reduction. However, in Eq. 3 the FFD construction uses an `AND` mask of 2^{d-1}. We remark these are equivalent, so the same conclusions hold. The modulus-based equation is more convenient for showing the uniformity of the construction, so it is preferred for Eqs. 4-7.

3.3 Verification of our FFD Construction

The FFD function introduced in Eq. 3 is a simple, yet surprisingly powerful construction that enables an FHE-friendly MinHash implementation. Indeed, more complex constructions result in larger Boolean circuits for FHE evaluation and therefore slower evaluation times. To further verify the effectiveness of the FFD construction for LSH, we empirically confirm that its impact on the accuracy of the set-similarity is minimal, compared to the CW-based MinHash and Jaccard-based LSH. Our empirical analysis is based on an input text document, which we generate 100 different variants of for set similarity. Each document is then converted to a set of integers using *t-shingling* [7]. In particular, our t-shingling technique converts every substring of size t of the input text to a 32-bit integer token by applying Python's built-in hash reduced by modulus 2^{32}. In our analysis, we also opt for a large modulus p for both the FFD and the CW construction, so that it becomes very unlikely for two 32-bit tokens to map to the same hash digest, and we compare MinHash (Alg. 1) based on the CW hash and FFD function (Eqs. 2 and 3). Specifically, the masking value for the FFD-MinHash is $p = 2^{32} - 1$, and the modulus for the CW-MinHash is prime $p = 2^{32} + 15$. We then compute the average MinHash accuracy across one hundred different input pairs and compare it with the

Jaccard similarity. Our findings indicate that the average accuracy of FFD-MinHash and CW-MinHash is very close (within 5% on average) to the result of the exact Jaccard similarity across different input sets. Overall, our empirical analysis shows that our FFD construction is a viable alternative to the CW hash, and closely approximates the Jaccard similarity.

3.4 Computing the Minimum of a Set of Digests

In MinHash, we need to return the smallest value in a set of digests as a signature, for each one of the k digest generation functions and for each input data set. Therefore, to find the minimum digest we need to compare all of them; however, as mentioned prior, comparisons in the encrypted domain are a challenge. In the CGGI cryptosystem, our observation is that this challenge can be addressed by translating the comparison operation directly into a Boolean circuit and using the encrypted output of the comparison as the select signals of a series of encrypted multiplexers (which are natively supported by CGGI). As this circuit is non-trivial and quite large, we apply rigorous circuit-level optimizations during an RTL synthesis step, to reduce the overall circuit size as we observe that this constitutes a key bottleneck of the entire MinHash evaluation. In the naive approach, set similarity requires $N \times M$ comparisons where N and M are the sizes of the two sets, while MinHash requires k comparisons where k is the number of digest generation functions employed (Alg. 1).

3.5 Input Size Considerations

With our chosen HLS framework, Google XLS, we remark that input size cannot be arbitrarily large, as it has an upper bound on the number of loop iterations that can be unrolled for synthesis. One potential way around this is to manually unroll loop iterations progressively until the total remaining iterations in the loop header is less than the strict threshold imposed by Google XLS. Likewise, performing logic optimizations with the Yosys synthesis suite is resource-intensive as the circuit size increases. It is possible to balance this somewhat by employing less rigorous synthesis and optimization flows for lower latency. In this case, however, the performance of the final FHE netlist is expected to be worse than a netlist generated with more optimization passes. Nevertheless, we remark that both HLS and RTL synthesis overheads only constitute a *one-time cost*; the resulting circuit can be evaluated an unlimited number of times on different input values.

4 Our MinHash Methodology for FHE

In this paper, we evaluate MinHash homomorphically by adopting an FHE-based framework (HELM), along with two EDA frameworks (Google-XLS and Yosys). The high-level steps outlining our methodology for running MinHash homomorphically are as follows:

- Expressing our HE-friendly variant of MinHash in C++,
- Translating the C++ program to a Verilog program using Google-XLS,
- Using the Yosys library to synthesize the Verilog program into an FHE-compatible netlist and perform rigorous logic optimizations,
- Processing the generated netlists with HELM to create and run the homomorphic program with CGGI.

4.1 Minhash in C++

The MinHash algorithm must be implemented in a way that can be converted to an equivalent synthesizable combinational Verilog program in order to render a Boolean circuit that can be executed with FHE. As discussed, the input given to the program must have static size; therefore, the MinHash algorithm of Alg. 1 cannot be directly utilized for homomorphic evaluation as the input to that function is dynamic. To address this challenge, we need to define all variables and inputs as static. As a result, the bitsize of variables S, k, M and $hashValue$ need to be fixed.

In more detail, we define a function called MinHash that takes as input two equal, constant-sized sets $S1$ and $S2$. We remark that the user doesn't necessarily need to choose two identically sized sets, but we opt for this approach without loss of generality. M is encoded as two signature arrays $sig1$ and $sig2$ of constant size k, which are initialized with zero, while the local variable k is a constant integer. The minimum value of each digest generation function for set $S1$ is stored in $sig1$ and for set $S2$ is stored in $sig2$. Next, the program compares the two signature arrays of constant size k, so that the number of equal signatures of the two sets is returned as the MinHash result. The program also stores the minimum digest for each set in the respective signature array, while the $hash_i$ operation is initialized with either the prime modulus (CW case) or the masking value (FFD case). Next, we discuss the implementations of the two digest generation families employed in our MinHash framework.

Carter-Wegman Hash The Carter-Wegman universal hash is the standard function used traditionally by MinHash for most applications. While this hash is a good candidate for MinHash over plaintext data, in the encrypted domain the CW hash becomes a major bottleneck. Specifically, a large part of the resulting Boolean circuit is allocated to the modular reduction to prime p, which creates big sub-circuits that make evaluating the overall CW hash in FHE very slow. Nevertheless, to adopt the CW hash for our encrypted MinHash, we store the different a values and b values (for each CW variant) in two local arrays of constant size k, while the prime number p is fixed. This way we can directly implement the hash with simple arithmetic operators in C++ based on Eq. 2.

FFD Construction Our key insight is that a digest derived from strictly Boolean operations is a more efficient alternative to the CW hash for implementing MinHash in CGGI. Indeed, our FFD construction can be evaluated much faster than the CW hash, as we report in our experiments. Like the CW case, we store the different b values in a local array and fix the masking value p based on the form $2^d - 1$ where d is the bitsize of the input. The rotation can be accomplished by circularly shifting the elements of the input (represented as a ciphertext vector) by $\frac{d}{2}$.

An important consideration in optimizing the runtime of the MinHash algorithm in the encrypted domain is to use the smallest wordsize necessary to represent the input data and intermediate computations. For our experimental evaluation in FHE, we use a word size of 16 bits, which strikes a balance between latency and precision. Additionally, the digest generation function parameters are set in a way that satisfies the constraints mentioned before. A further discussion about our setup is provided in our experimental evaluation.

4.2 Converting C++ to Verilog Using Google-XLS

The next step in our methodology is to convert the high-level C++ code into a Verilog module so that the circuit can be run homomorphically with CGGI. Notably, for certain application types, writing Verilog by hand can be tedious and error-prone, particularly when working with relatively large arrays. As a result, we use High-Level Synthesis (HLS) to automatically generate synthesizable Verilog code based on the programmer's intent expressed as a C++ program. An automated process like this allows us to rapidly generate and compare multiple implementations of MinHash *for different digest generation functions and input sizes* in a matter of seconds. One potential limitation of HLS is that there are constraints on the supported high-level programming language operations and some of their constructs are not synthesizable; for example, pointers and dynamic memory allocation in C++ are not supported by HLS tools. Another possible limitation is that HLS might generate sub-optimal Verilog (compared to hand-optimized Verilog); we remark that the rigorous logic optimizations performed during the subsequent RTL synthesis step result in efficient circuits nonetheless.

To use our chosen HLS framework, Google-XLS [14], we annotate our MinHash function in C++ with simple pragmas that indicate what loops to unroll as well as which function is the top-level module. The Google-XLS first generates an intermediate representation (IR) of the C++ source code which is a data-flow oriented representation; a data structure used internally by the XLS compiler to do some optimizations so the generated circuitry is efficient. The IR has the static-single-assignment (SSA) property, which means each variable is assigned once and it must be defined before it is used. This property leads to the XLS not supporting some of the C++ primitives mentioned above. Then the Google-XLS can generate the equivalent combinational Verilog from the optimized IR using the codegen function. The IR itself is directly used in the Google Transpiler but in this paper, we only care about the generated Verilog.

4.3 Logic Synthesis Using Yosys

The Verilog code generated from the HLS tool cannot be directly used for homomorphic evaluation as it only describes the behavior of the circuit and does not indicate the particular logic gates needed to evaluate the algorithm. Therefore, structural Verilog with a gate-level abstraction is suitable for the Boolean-based programming model of CGGI. Towards that end, we employ the logic synthesis functionality of Yosys [38], which is an automated toolchain that converts high-level hardware description language code such as behavioral Verilog to gate-level netlists. Concurrently, Yosys also performs logic optimizations that aim to reduce the total number of gates in the circuit.

We utilize the following custom flow to generate optimized FHE circuits with the following operations:

- **proc**: Yosys synthesizes the blocks and converts sequential processes to multiplexers and flip flops.
- **flatten**: This collapses all modules in the design into a single module.
- **synth**: A generic synthesis script that performs several optimizations, such as wordsize reduction and redundant logic elimination, as well as generic technology mapping.
- **abc -g simple, -MUX**: This instructs the underlying ABC tool to perform a more specific technology mapping that results in cells that can be evaluated readily with the FHE backend. `simple` identifies common 2:1 logic gates, and the `-MUX` indicates the removal of the 2:1 MUX cell.
- **splitnets**: Changes wire naming conventions to align more closely with the format that the HELM parser expects.
- **write_verilog**: After the gates are generated, this operation writes them into an output Verilog netlist.

In conclusion, the netlists generated from Yosys will serve as a public input for FHE computation (i.e., the circuit to be evaluated).

4.4 Generating Homomorphic Programs Using HELM

To generate and run the final homomorphic programs, we use an open-source framework called HELM [19], which uses the CGGI scheme and serves as an execution environment to run netlists on parallel devices in the encrypted domain. HELM starts by finding all connections between each gate and cells in the netlists and maps the gates described in the netlist to FHE equivalent computations. It also incorporates a scheduler that identifies gates that can be executed concurrently, which are flagged for parallel execution.

After generating the homomorphic programs, we process our raw inputs (i.e., user data), encrypt them, and feed them to the input wires of the FHE circuits. The circuits can then be evaluated by a third-party cloud without gaining knowledge about the underlying data. Then, after the encrypted computation, the cloud will return the encrypted similarity result which is an integer between 0 and k. Lastly, the user can decrypt and see the result of the similarity between the provided sets.

5 Experimental Evaluation

In this section, we provide a comprehensive evaluation of our proposed framework by comparing the performance of the FFD and the CW families in the MinHash algorithm. For good measure, we also provide a naive method of set similarity with $\mathcal{O}(n^2)$ time complexity (assuming both sets are of size n) which compares each element of the first set with each element of the second set. We perform all CPU-based experiments on an `r5.12xlarge` AWS EC2 instance, which has 48 vCPUs and 384 GB of RAM. Additionally, we further utilize an NVIDIA A100 GPU with 80 GB of memory for GPU-based experiments. Lastly, we adopt the default cryptographic parameters for CGGI that are supported by HELM, which correspond to approximately 128 bits of security.

5.1 Methodology for Plagiarism Detection

For our privacy-preserving plagiarism detection, we assume that the client has an input document that they want to keep private and it is compared against a document owned by the third party cloud server (who also wants to preserve the privacy of their document). To ensure the privacy of both inputs, a privacy-enhancing technology must be utilized; for our case, HE protects the client input and the cloud server's document is implicitly protected as it never leaves the cloud server (and thus can be kept in plaintext form). By implementing MinHash in FHE, we can efficiently compute the similarity between input datasets that should remain private. For our experiments, we employ our encrypted MinHash to detect plagiarism between private text documents. The two main steps to achieve this are to encode the documents so they can serve as input to the homomorphic circuits and parameterize the homomorphic circuits for the specific document size.

Preprocessing/Encoding Input Data First, the documents must be encoded so that the contents of each one map to a set of integers. The size of the set and the range of each element in the set must be compatible with a homomorphic circuit of matching input size. We also note that there are many ways to encode text into a set of integer tokens, and in this work, we opt to use t-shingling [7]. Using this approach, we hash each substring of length $t = 9$ into an integer token, resulting in a set of hash digests. The size of the hash sets is directly correlated with the number of substrings of length t. The range of each integer token depends on the digest generation function utilized in the FHE circuit (i.e., FFD or CW). We note that each digest is reduced by the prime modulus p when using the CW hash function in Eq. 2, or the masking value p when using the FFD function in Eq. 3. Finally, the preprocessing phase is done in plaintext and when the documents are encoded to sets of integer tokens, we encrypt them and feed them to the homomorphic circuits for evaluation.

Parameterizing Homomorphic Circuits Next, we need to set the constant parameters for our homomorphic programs to run the encrypted sets correctly. Specifically, the parameters required for the MinHash circuits include the number of digest generation functions k (in this work we use $k = 5$ unless noted otherwise), the bitsize of our variables, and the parameters of our digest generation functions. As soon as the number of digest generation functions k is fixed, we can select the parameters for each to generate k unique digest generation functions. Regarding our C++ implementation, we use the 16-bit `unsigned short` type for the elements of each input set, as well as other intermediate variables. In the preprocessing stage, we set the digests to be $d = 14$ bits in size (we apply modulus 16384 to the digests) so the intermediate values can fit in 16 bits.

For CW hash functions, the required parameters are the coefficients a and b and the prime modulus p, as given in Eq. 2. The choice of p must strike a balance between collision probability and computational overhead; a large p will yield a lower probability of collisions in the reduced hash values but will result in exponentially larger computational overheads in the encrypted domain as the supported word size must be increased to accommodate values modulo p. We chose $p = 16381$ as our modulus, which allows for 16-bit word sizes and results in *worst-case* accuracy degradations of approximately 10% (relative to the exact Jaccard similarity). Regarding the a and the b parameters, we chose them in a way so that the term $ax + b$ exceeds the modulus p with high probability, but it is smaller than the value 65535, which is the maximum value supported by our 16-bit wordsize. This way, all intermediate values (i.e., $ax + b$) will very likely wrap around p and the digest generation functions can work as intended.

For the FFD functions, we use Eq. 3, where the required parameters are the coefficient b, the number of bits, and the masking value p, which acts like the reduction modulus in the CW hash. The p value, unlike in the CW hash, should be of form $2^n - 1$, so we choose the value $n = 14$ and $p = 16383$ as our reduction value. The rotation when applied on the digests won't affect the bitsize and The b values are chosen to be close to the masking value, mostly in the range $[p/2, 3p/2]$.

Now that we have successfully encoded our raw input data and parameterized our homomorphic programs, we can encrypt the encoded data sets and correctly evaluate them. The encrypted result is then sent to the user which is an integer in the range $[0, k]$ where 0 means no similarity between the documents and k means equality. The user then decrypts the result with their private key.

Finally, we remark that the "naive" circuits discussed later in our experiments implement an exhaustive approach to search all pairs of similar elements, so they require no specific parameters.

5.2 Comparing MinHash Variants with the Naive Approach

For each digest generation family, we create 3 circuits of input sizes 10, 20, and 30. Due to the extreme size increase of the modular reduction circuits in the CW MinHash circuits, Yosys could not synthesize circuits corresponding to set sizes of more than 30. On the other hand, the FFD-MinHash circuits were way smaller than the CW counterparts and Yosys could synthesize them in a matter of seconds. Figure 3 shows that as the set size increases, the MinHash-CW circuits increase their runtime overhead at a faster rate than the MinHash-FFD circuits. This means that the MinHash algorithm with the FFD function is way more efficient than the standard CW-based MinHash using Boolean FHE so it is more scalable and can be used with larger sets as inputs and higher values of k.

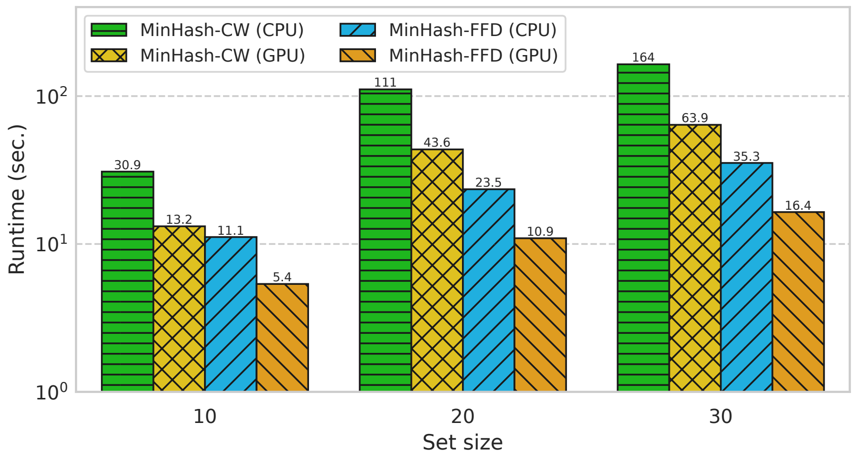

Fig. 3. Comparison between FFD and CW in MinHash.

Due to the graceful scalability of the MinHash-FFD circuits, we created additional circuits corresponding to set sizes of 50, 75, and 100 for the FFD variant. Next, we compare our MinHash-FFD circuits against a "naive" approach to further motivate the use of MinHash as a viable alternative for set similarity computation in the encrypted domain. Notably, we selected the naive method because implementing the Jaccard similarity from Eq. 1 remains inefficient in the encrypted domain, as it requires the use of *hashset* data structures. In particular, assuming both input datasets are of equal size, the time complexity of Jaccard similarity when implemented with hashsets is $O(n)$ where n is the size of the sets. However, dynamic data structures like hash sets where an operation such as a "lookup" depends on encrypted data cannot be implemented efficiently in FHE. That is why we select a naive $O(n^2)$ approach to assess the viability of our framework.

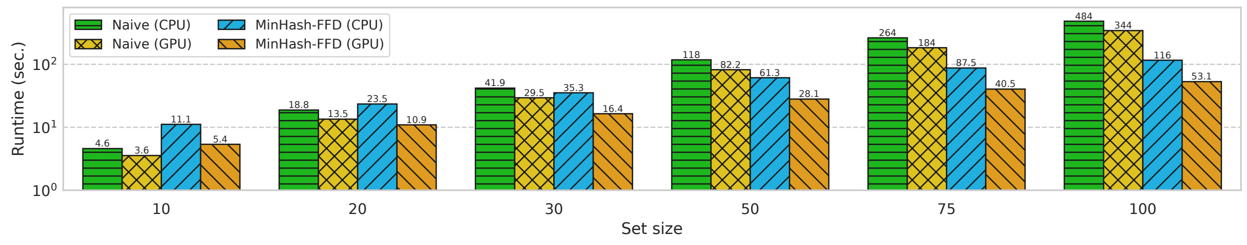

Fig. 4. Comparison between the MinHash-FFD circuits and the naive method with different set sizes. The naive method compares each digest of the first set with every digest of the second set.

For the naive method, we create circuits of size 10, 20, 30, 50, 75, and 100 to compare with the MinHash-FFD circuits of identical set sizes. As we can see in Figure 4, at smaller set sizes the naive method is faster, which is expected. But as the set size grows, the naive method scales significantly worse than the MinHash-FFD circuits and it becomes far less efficient. As mentioned previously, one of the major bottlenecks of implementing MinHash in FHE is the number of comparisons. This and the results of Figure 4 show that MinHash-FFD performs markedly better than the naive method for moderate and large set sizes.

Table 1 showcases the complete operational make-up of the FHE circuits for all three set similarity techniques. The overall gate composition of the circuit has an impact on computational overhead as `NOT`, `BUF`, and `CONST` gates exhibit very low latency, while `AND`, `OR`, and `XOR` gates are over an order of magnitude more costly. In this context, `BUF` gates constitute simple copy propagations from an input wire to an output wire, while the `CONST` gates load an “empty” wire (i.e., a wire that is not loaded with a valid encryption of `1` or `0`) with an encoding of a public bit. Additionally, the table demonstrates how the circuit size scales with larger set sizes; even though the naive technique exhibits the smallest circuit using set sizes of 10, both the FFD and CW techniques exhibit larger set sizes. In particular, the proposed FFD technique exhibits the best overall scalability.

Table 1. Characterization of Private Set Similarity Circuits

Circuit	AND	OR	NOT	XOR	BUF	CONST
FFD-10	7005	9132	1427	1874	320	27
FFD-20	15001	18991	3322	3734	640	27
FFD-30	22367	28562	5039	5689	960	27
CW-10	19150	28087	9361	10437	320	27
CW-20	77369	97935	27977	41111	640	27
CW-30	115053	145391	42092	61080	960	27
Naive-10	459	4115	312	1983	320	0
Naive-20	2188	17501	1575	8379	640	0
Naive-30	5078	40079	3756	19098	960	0

However, the total gate composition alone is not a good predictor of performance as it gives no intuition regarding the parallelization opportunities in the circuit as well as the critical path. Figure 5 shows the number of gates at each level of the Boolean circuits for the naive method. The circuit starts out extremely wide, with over 10000 gates in the earliest levels, and quickly levels off to tens of gates per level. Thus, the naive method exhibits limited parallelism as most of the levels in the circuit are quite shallow. Additionally, as the set size increases by 10 elements, the critical path, which is akin to the number of levels in the overall circuit, more than doubles.

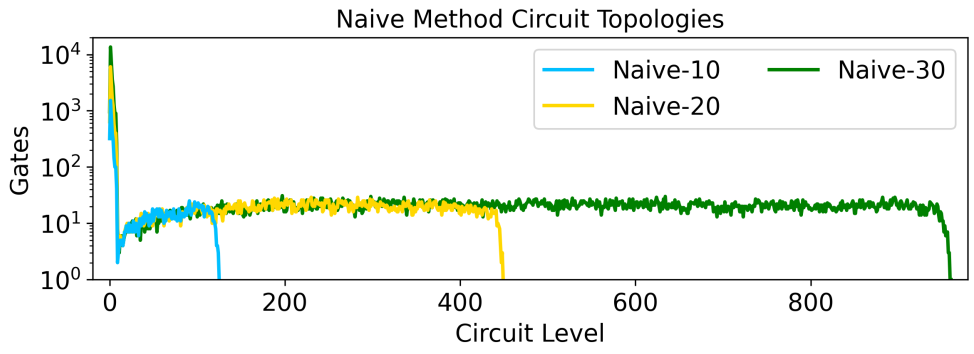

Fig. 5. Circuit characterization of the naive method for private set-similarity across different set sizes. The critical path more than doubles for each increase in set size.

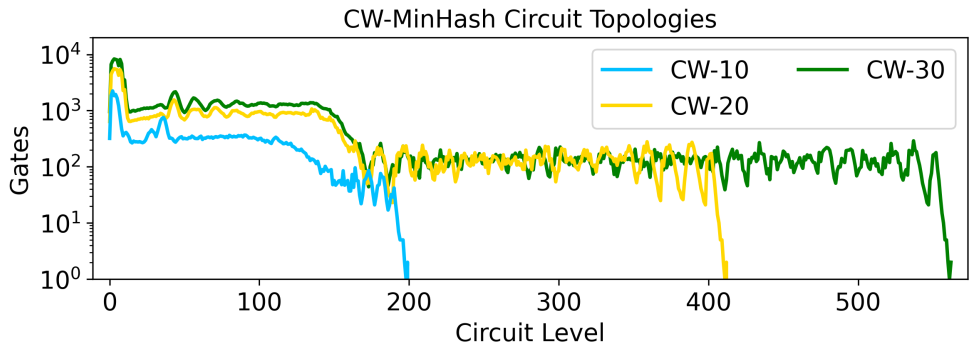

Fig. 6. Overall circuit topology of the MinHash approach utilizing the Carter-Wegman hash construction. The circuits are quite wide, with a long critical path. However, this approach exhibits better scalability in terms of the critical path length compared to the naive approach.

Figure 6 showcases the circuit topologies of privacy-preserving MinHash with the Carter-Wegman hash construction. These circuits are very wide compared to the naive approach, mostly due to the multi-bit arithmetic operations and modular reduction circuitry. For sets with 10 and 20 elements, the MinHash-CW circuits exhibit a longer critical path than the naive approach but have better scalability for larger set sizes (as evidenced by the critical path for set similarity between sets of size 30).

The MinHash-FFD approach is the optimal choice of the three techniques for increasingly large set sizes, as evidenced in Figure 4. From the topology graph shown in Figure 7, we observe that the critical path is shorter than the other variants and the circuit width remains moderately wide, which is conducive to efficient parallelism with the HELM backend.

5.3 Time-Accuracy Trade-off

All experiments above were conducted using $k = 5$ digest generation functions for the MinHash circuits. Although five digest generation functions still give acceptable accuracy in many scenarios, some applications may require higher accuracy guarantees in the set similarity measure. As the results of the previous experiments show, the MinHash-FFD circuits are the fastest and most scalable. Therefore, we use the MinHash-FFD circuit of input size 30 and implement it using both 25 and 50 digest generation functions. The circuit evaluations are shown in Figure 8, and we observe that as the number of digests grows, the runtime increases linearly. Interestingly, the runtime of the MinHash-CW circuit of size 30 with five digest generation functions

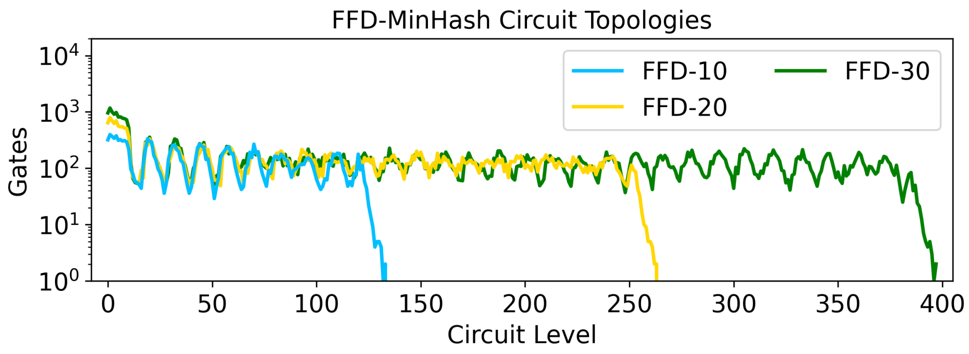

Fig. 7. Circuit overview of the MinHash approach utilizing the bespoke FFD methodology. The circuits exhibit moderate width, with shorter critical paths than the CW-MinHash variant.

is around the same as the MinHash-FFD circuit of size 30 with 25 digest generation functions. This further solidifies MinHash-FFD as the superior alternative and even with an increase in digest generation functions, it is still viable and scalable.

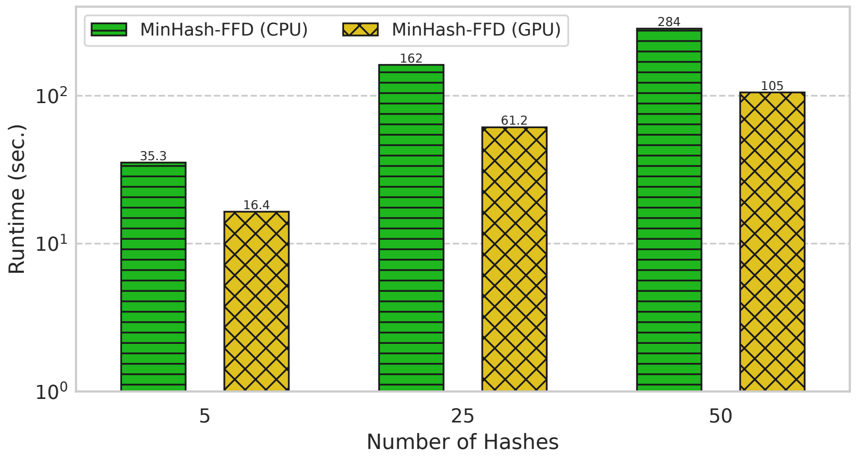

Fig. 8. Comparison between MinHash-FFD circuits of set size 30 with different numbers of digest generation functions.

5.4 CPU vs GPU Evaluation Comparison

All experiments were done with both a CPU-based cryptographic backend and a GPU-based backend. We observe that the GPU outperforms the multi-threaded CPU as most FHE operations are highly parallel in nature and can greatly benefit from the massive levels of parallelism that GPUs can provide. However, we observe that the speedup acquired from the GPU backend is highly dependent on the structure of the circuit being executed. As a case in point, the naive implementation depicted in Figure 4 is less than 2× faster when running on the GPU for all set sizes. This is primarily due to the fact that the circuit has a very long critical path (with up to several thousand levels for the largest set size). Additionally, most of the levels in the circuit are thin and consist of a limited number of gates, limiting the number of gates that can be evaluated concurrently. As a result, these circuits can only achieve modest utilization of the A100 GPU and result in limited speedups (i.e., $< 2\times$. However, circuits such as the MinHash-CW variants shown in

Figure 3 are very wide due to the large and wide Boolean subcircuits needed to evaluate operations such as modular reduction and multiplication. In these cases, where the levels of the circuit are very wide, we observe speedups of nearly $3\times$ for the GPU backend versus the CPU.

6 Related Works

In this section, we expound upon works that tackle both private set intersection (PSI) and private set similarity, which is closest to our work.

6.1 Private Set Intersection

The problem of private set intersection involves computing a subset consisting of elements present in two or more private sets. Kerschbaum [27] introduced a PSI protocol based on Bloom filters and the BGN homomorphic cryptosystem [4] and incorporates an additional computing party (independent from the client and server). Chen et al. [10] introduced a hash-based protocol tuned for computing the similarity between a large and small set. The protocol utilizes the BFV cryptosystem [5] and incorporates a technique to reduce communication overhead by making the ciphertexts smaller through modulus switching. A maliciously secure protocol based on HE was proposed by Jiang et al. [26] that also incorporates verifiable computation and oblivious pseudo-random functions, targeting BFV. Additionally, other works have solved this problem using secure multiparty computation. Hazay and Venkitasubramaniam proposed an approach that mimics a star topology where every party communicates with a designated party and avoids the need to perform broadcasts in favor of point-to-point communications [22]. Falk et al. propose another multi-party approach based on Bloom filters and is optimized for sets of unbalanced sizes [24].

Instead of returning the intersection of sets to clients, our work focuses on computing a similarity measurement between two sets. Compared to the HE-based works, our solution utilizes FHE instead of LHE and can allow for arbitrary computation after the set similarity is computed or otherwise used as a building block of a more complex application. Additionally, the MPC solutions incur a higher communication overhead relative to our work in order to perform the computation and have a weaker threat model in the multi-party setting due to the threat of colluding parties. We note that this threat model can be strengthened in a two-party computation scenario between a single cloud server and the client. However, the client must actively participate in the computation, which is infeasible for resource-constrained devices. On the other hand, our methodology only involves a single computing party and the client can remain offline during the computation of the set similarity.

6.2 Hash-Based Private Set Similarity Measurements

Other works focus on computing a similarity measure in a fashion similar to the proposed approach. Yan [41] estimates the Jaccard similarity using differential privacy, while Wong [39] proposed a private protocol to compute Jaccard similarity using both differential privacy and homomorphic encryption. Compared to our work, both of these works are interactive and require the user to actively participate in the protocol during computation. Purely FHE-based solutions like ours, only require the client to only perform encryption, decryption, and key generation. PrivMin [40] computes a privacy-preserving MinHash variant to approximate the Jaccard similarity with differential privacy. However, this work requires a trusted third party which results in a weaker threat model. Our work assumes a single semi-honest computing party and the only assumption we make is that the computing party correctly executes the prescribed encrypted algorithm (which is a realistic assumption in the context of cloud computing).

He et al. [23] propose a differential privacy construction that incorporates locality sensitive hashing for computing record linkage between two databases and achieves low latency relative to previous works. However, in this context, both parties (the owners of each database) learn which specific records match;

this constitutes a different threat model than MatcHEd, where only the client/querying party learns any information about the similarity between his set and the cloud's set. Similarly, Wei and Kerschbaum [37] compute private record linkage and improve upon He et al. [23] by providing stronger security guarantees. Both approaches require joint computation between the database owners while MatcHEd performs a set similarity computation completely offline and requires no computation on the querying party's side other than the trivial overheads of encryption and decryption.

Lastly, the EsPRESSo [3] framework introduces two protocols, one for computing the exact Jaccard similarity and another that approximates it using MinHash. Both protocols are based on a custom MPC protocol with two computational parties. However, the security level of the instantiation of the scheme is 80 bits, which is lower than our approach that complies with the standard 128 bits of security.

7 Conclusion

In this paper, we present the MatcHEd framework for encrypted set-similarity based on MinHash. We introduce a new FFD function to use instead of the standard Carter and Wegman hash function used in the MinHash algorithm to improve evaluation speeds by a large margin in Boolean FHE. Further, we leverage EDA methodologies to synthesize and optimize our bespoke MinHash variants to allow for efficient execution in the encrypted domain. Lastly, we evaluate the first fully homomorphic plagiarism detection application using our proposed techniques and report that the MinHash variant based on our proposed FFD hash family significantly outperforms the standard MinHash algorithm in the encrypted domain as well as a baseline approach that computes the exact set similarity.

Acknowledgments

This work has been supported by NSF Award #2239334.

References

1. Rachna Arora, Anshu Parashar, and Cloud Computing Is Transforming. Secure user data in cloud computing using encryption algorithms. *International journal of engineering research and applications*, 3(4):1922–1926, 2013.
2. Sujoy Bag, Sri Krishna Kumar, and Manoj Kumar Tiwari. An efficient recommendation generation using relevant jaccard similarity. *Information Sciences*, 483:53–64, 2019.
3. Carlo Blundo, Emiliano De Cristofaro, and Paolo Gasti. Espresso: efficient privacy-preserving evaluation of sample set similarity. *Journal of Computer Security*, 22(3):355–381, 2014.
4. Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In Joe Kilian, editor, *Theory of Cryptography*, pages 325–341, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
5. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
6. A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, page 21, USA, 1997. IEEE Computer Society.
7. Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, COM '00, page 1–10, Berlin, Heidelberg, 2000. Springer-Verlag.
8. Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 327–336, New York, NY, USA, 1998. Association for Computing Machinery.
9. J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, page 106–112, New York, NY, USA, 1977. Association for Computing Machinery.
10. Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1243–1255, New York, NY, USA, 2017. Association for Computing Machinery.

11. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.
12. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
13. Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, 1999.
14. Wei Dai and Berk Sunar. XLS: Accelerated HW Synthesis. `https://github.com/google/xls`, 2020.
15. Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
16. Osman Durmaz and Hasan Sakir Bilge. Fast image similarity search by distributed locality sensitive hashing. *Pattern Recognition Letters*, 128:361–369, 2019.
17. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.
18. Sreenivas Gollapudi and Rina Panigrahy. Exploiting asymmetry in hierarchical topic extraction. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, page 475–482, New York, NY, USA, 2006. Association for Computing Machinery.
19. Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables. Cryptology ePrint Archive, Paper 2023/1382, 2023. `https://eprint.iacr.org/2023/1382`.
20. Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks. *Proceedings on Privacy Enhancing Technologies*, 2023(3):154–172, July 2023.
21. Shai Halevi and Victor Shoup. Bootstrapping for helib. *Journal of Cryptology*, 34(1):7, 2021.
22. Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In Serge Fehr, editor, *Public-Key Cryptography – PKC 2017*, pages 175–203, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
23. Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1389–1406, New York, NY, USA, 2017. Association for Computing Machinery.
24. Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Private set intersection with linear communication from general assumptions. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, WPES'19, page 14–25, New York, NY, USA, 2019. Association for Computing Machinery.
25. Ilia Iliashenko and Vincent Zucca. Faster homomorphic comparison operations for BGV and BFV. *Proceedings on Privacy Enhancing Technologies*, 2021(3):246–264, 2021.
26. Yuting Jiang, Jianghong Wei, and Jing Pan. Publicly verifiable private set intersection from homomorphic encryption. In *International Symposium on Security and Privacy in Social Networks and Big Data*, pages 117–137, Singapore, 2022. Springer Nature.
27. Florian Kerschbaum. Outsourced private set intersection using homomorphic encryption. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, page 85–86, New York, NY, USA, 2012. Association for Computing Machinery.
28. Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.
29. Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711, New York, NY, 2020. IEEE Computer Society and IEEE Reliability Society.
30. Kyung Mi Lee and Keon Myung Lee. Similar pair identification using locality-sensitive hashing technique. In *The 6th International Conference on Soft Computing and Intelligent Systems, and The 13th International Symposium on Advanced Intelligence Systems*, pages 2117–2119, New York, NY, 2012. IEEE.
31. Wenjun Lu, Avinash L Varna, and Min Wu. Confidentiality-preserving image search: A comparative study between homomorphic encryption and distance-preserving randomization. *IEEE Access*, 2:125–141, 2014.
32. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

33. Dimitris Mouris, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Terminator suite: Benchmarking privacy-preserving architectures. *IEEE Computer Architecture Letters*, 17(2):122–125, 2018.
34. Mummoorthy Murugesan, Wei Jiang, Chris Clifton, Luo Si, and Jaideep Vaidya. Efficient privacy-preserving similar document detection. *The VLDB Journal*, 19(4):457–475, 2010.
35. Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome biology*, 17(1):1–14, 2016.
36. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
37. Ruidi Wei and Florian Kerschbaum. Cryptographically secure private record linkage using locality-sensitive hashing. *Proc. VLDB Endow.*, 17(2):79–91, oct 2023.
38. Clifford Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`, 2013.
39. Kok-Seng Wong, Myung Ho Kim, et al. Preserving differential privacy for similarity measurement in smart environments. *The Scientific World Journal*, 2014(9):1–9, 2014.
40. Ziqi Yan, Jiqiang Liu, Gang Li, Zhen Han, and Shuo Qiu. Privmin: Differentially private minhash for jaccard similarity computation. *arXiv preprint arXiv:1705.07258*, 2017.
41. Ziqi Yan, Qiong Wu, Meng Ren, Jiqiang Liu, Shaowu Liu, and Shuo Qiu. Locally private jaccard similarity estimation. *Concurrency and Computation: Practice and Experience*, 31(24):e4889, 2019.
42. Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 305–316, New York, NY, USA, 2012. Association for Computing Machinery.
43. Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 990–1003, New York, NY, USA, 2014. Association for Computing Machinery.