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 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. Towards that end, introduce an efficient bitwise Hash function to employ for encrypted set-similarity, which allows faster evaluation times relative to the 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.**

***Index Terms*—Encrypted MinHash, Fully homomorphic encryption, privacy-preserving set similarity.**

I. Introduction

With the growth of third-party cloud providers, the privacy of the outsourced data stored in these servers becomes a big concern and needs to 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 [1], [2], RowHammer attacks, as well as other DRAM-based attacks [3], [4] 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 [5], as these encryption algorithms can help us secure the sensitive data stored in a cloud provider's server [6]. 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. For example, some classical encryption schemes such as the RSA [7] and Paillier [8] cryptosystems have homomorphic properties: Paillier offers an additive property where by multiplying ciphertexts together, one can generate a valid encryption of the sum (i.e. $Enc(x) \times Enc(y) = Enc(x + y)$). Similarly, the RSA cryptosystem offers a multiplicative property where $Enc(x) \times Enc(y) = Enc(x \times y)$. Even though these properties can be useful in certain individual situations, non-trivial algorithms require both addition and multiplication. 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 the least very expensive due to the *termination problem* [9], [10]. 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.

Nevertheless, one of the major challenges in modern privacy-preserving methods like homomorphic encryption is the ability to compute similarities between encrypted data without leaking any information about the underlying plaintext. In many privacy-aware applications, such as finding similar DNA sequences [11], finding similarities over proprietary or sensitive images [12], as well as detecting plagiarism across private documents [13], 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 [14], [15].

In more details, LSH is a hashing-based technique that can efficiently approximate the similarity between datasets based on some metric, such as the *Jaccard similarity index* [16]. 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 conputation costs compared to simpler, brute-force methods. Moreover, LSH algorithms offer high accuracy and high-performance if parameterized correctly.

In this paper, we adopt the MinHash LSH algorithm [17] 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 hash functions and computing the minimum hash value. The hash functions are required to provide different mappings over the dataset, while the minimum operation finds the minimum hash value (i.e., the signature) of a set. In this approach, we only need to compare the hash signatures generated for each set instead of comparing all elements with each other.

The hash function typically used for MinHash is a linear universal hash function such as the Carter-Wegman (CW) hash [18]. 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 implementing a universal hash function 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 make MinHash practical in FHE, we introduce a judiciously designed add-rotate-xor (ARX) hash construction that is tailored for FHE performance. Our key observation is that bitwise operations can be directly translated to logical gates to be evaluated efficiently using FHE.

Overall, our contributions can be summarized as follows:

- Design of efficient and accurate hash functions 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 II provides necessary background on FHE and the MinHash algorithm, while Section III highlights challenges for implementing MinHash in the encrypted domain. Section IV 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 V discusses our experimental evaluation using plagiarism detection benchmarks as the target application. Finally, Section VI discusses relevant related work, and our concluding remarks are presented in Section VII.

II. Background

A. 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 [19] and CKKS [20], 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 [21] or its ring variant [22], 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. Luckily, a noise mitigation measure referred to as modulus switching can reduce the magnitude of the accumulated noise after multiplication by removing an underlying prime from the (composite) ciphertext modulus. This strategy also decreases the size of the ciphertexts as the coefficients become smaller, so the latency of the HE arithmetic decreases as well. However, modulus switching can only be done a finite number of times, as one will eventually run out of primes to remove from the composite modulus. The only way to support additional modulus switching is to choose a larger initial coefficient size for the ciphertext polynomials, which has a negative impact on security and must be balanced by increasing the polynomial degree. This approach, however, is not scalable for very deep circuits as the required parameters result in enormous ciphertexts and very expensive addition and multiplication operations.

FHE solves this scalability 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. This power comes from a powerful noise mitigation technique called *bootstrapping* that can be invoked an infinite number of times (unlike modulus switching) [23]. In particular, any LHE scheme can be transformed into an FHE scheme through the introduction of this mechanism. 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 [24]. The FHEW cryptosystem [25] was introduced to address the latency problems in bootstrapping and was the first Boolean FHE scheme. This class of FHE schemes encrypt individual bits, as opposed to integers or floating point numbers, and allow 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 [26] builds upon FHEW and is capable of achieving even faster bootstrapping speeds of less than 10 milliseconds per bootstrap on a CPU. Additionally, it incorporates an encrypted multiplexer gate that can obliviously choose between two encrypted bits based on an encrypted select bit; this functionality is incredibly useful for applications that require comparisons over encrypted data, like the encrypted MinHash algorithm proposed in this work. For these reasons, we opt to utilize the CGGI cryptosystem in this work.

B. 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. One consideration about Jaccard similarity is its inability to reflect the frequency of elements in its similarity computation due to the nature of the sets. However, in most applications, like plagiarism detection demonstrated in this work, the frequency does not impact the results [27]; for example, even one sentence being similar should warrant flagging the document as plagiarized. For cases where the application needs to reflect the frequency of each element, the Cosine similarity is an alternative option [28], [29]. Morover, a second consideration about Jaccard similarity is its scalability in terms of time and space required as the size and number of the datasets increases.

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 compare the signatures pair-wise to find similar datasets. Therefore, even if the size and number of the datasets grows, 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 need to compare the signatures of the two sets; 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 [30]. 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 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.

C. 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). We remark that a malicious cloud would also be incapable of accessing the underlying plaintext data of the ciphertexts (due to FHE guarantees), but could return incorrect results to the client by deviating from the protocol, so this is not considered in our case. 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. Additionally, our underlying FHE toolchain uses the CGGI scheme [26] to securely evaluate Boolean circuits homomorphically; the security of the CGGI scheme is based on the learning with errors (LWE) problem [21], which is an NP-hard lattice problem. Overall, our threat model is stronger than approaches involving more than one servers (e.g., secure multiparty computation), where colluding servers could potentially leak sensitive data.

III. Privacy-Preserving Set Similarity with MinHash

Adopting MinHash in the encrypted domain comes with a set of challenges that need to be addressed. As mentioned earlier, the two core components of MinHash are hash functions and finding the minimum hash value for each function. Thus, to run MinHash homomorphically, we need to be able to compute the hash functions and find the minimum values in the encrypted domain. Another challenge is defining 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. Therefore, the size of each set must be initialized at compile time, resulting in multiple circuits for evaluating sets with different sizes.

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 [26]. As elaborated in the next subsections, we employ the add-rotate-xor (ARX) and the Carter-Wegman hash constructions homomorphically, along with efficient FHE comparison modules that process all encrypted hash values and return the minimum hash values without leaking any information about the underlying plaintext. Subsequently, our homomorphic programs (i.e., hashes, 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.

A. 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. In particular, if n and m are the sizes of the input datasets, the time complexity of computing the Jaccard similarity efficiently is $\mathcal{O}(min(n, m))$, which is further exaggerated when evaluated in the encrypted domain. Moreover, Jaccard similarity in FHE requires dynamic data structures, which cannot be implemented efficiently in FHE, while the corresponding Boolean circuits cannot be easily synthesized and optimized by automated logic tools. Even though a look-up operation is possible to implement in FHE, this incurs linear time complexity, so that the only way to compute the encrypted Jaccard similarity incurs the naive $\mathcal{O}(n \cdot m)$ complexity, where each element of the first set is compared with all the elements of the second set. Conversely, MinHash is significantly more efficient and decreases the computational effort in the encrypted domain.

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 [31] to generate a Verilog program. Next, the XLS Verilog code is optimized by the Yosys RTL synthesis suite [32] and serves as input to the HELM framework that generates an optimized fully homomorphic circuit [33]. 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.

B. Choice of Hash Functions

The hash functions used by MinHash are needed to generate different mappings of the input datasets to compute their unique signatures. In this work, we employ two universal hashes: the first is a Carter-Wegman (CW) construction, while

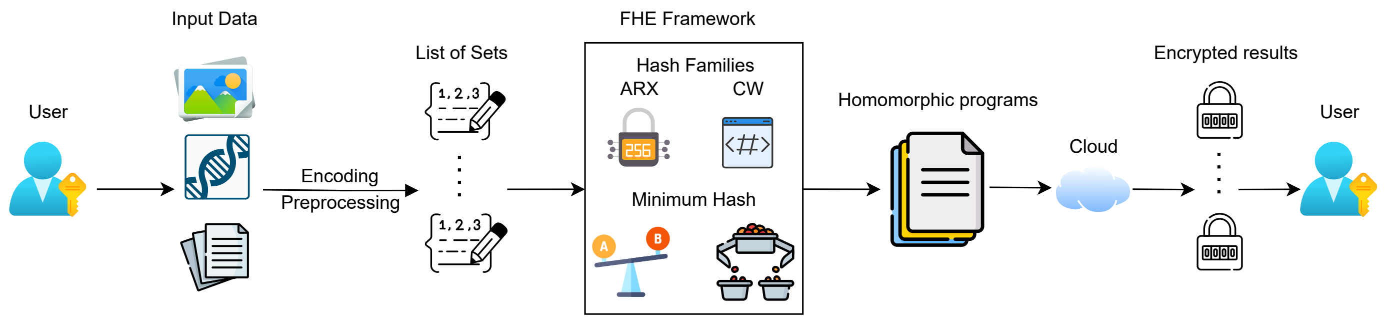

Fig. 1. **Framework Overview:** The user encodes data into sets, encrypts them, and then sends them to the cloud, which can run MinHash in the encrypted domain with a CW or ARX hash. Finally, the encrypted set similarity measure is returned to the user for decryption.

the second is an add-rotate-xor (ARX) construction. In more details, the CW universal 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, 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, to minimize the size of the generated Boolean circuits for homomorphic evaluation (and thus improve FHE run-time performance), we also introduce an ARX-based hash design, where modular reductions can be replaced with an `AND` operation. In this case, bitwise operators can be directly implemented as a Boolean circuit in the CGGI cryptosystem, which makes the construction very efficient. Therefore, our ARX-based universal hash is defined as follows:

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

where b is a constant parameter and p is our masking value, which is not necessarily a prime. In more details, our ARX construction adds the input to the constant parameter b then the `AND` operator approximates the functionality of the modular reduction in the CW hash, while limiting the size of the output. Next, we need to parameterize the ARX so that it generates a randomly-looking mapping for each input x, while maintaining efficiency. Towards that, our masking value follows the form $p = 2^d - 1$, where d is the number of bits in the output digest; this way, each bit of p is set to `1`. Note that p does not have to be a prime (unlike the CW hash), while any bits of p that are set to `0` will skew the output digests to a smaller range, which will increase the probability of collisions; thus the form $p = 2^d - 1$ is an optimal choice. Naturally, if the $(x + b)$ intermediate value exceeds p in Eq. 3, the $\&$ operator will ensure that output is at most d bits.

In effect, our proposed ARX construction is applying a rotation on the input x, and the length of this rotation depends on the value of parameter b, which is visualized in the example of Figure 4. In the simplest case, if we have an input set $\{1, 2, 3, 4\}$, $b = 1$ and $p = 15$, then our ARX hash will return $\{2, 3, 4, 5\}$; here, our minimum hash value is 2, which corresponds to the first element in our input set. However, if we set $b = 13$, then the resulting digests will be $\{14, 15, 0, 1\}$, where the minimum hash value is 0 corresponding to the third element in our input set. Therefore, based on our b parameter values, a random element will be mapped to the minimum hash value. which means that if a different dataset inputs the same element at that index, then their minimum hashes can become equal. At the same time, care must be taken on how the b parameters are chosen while creating a family of ARX hashes: if all b parameters are small integers and none of the $(x + b)$ values exceed p, then the minimum hash will always correspond to the minimum value in the input set. In this case, if only the minimum values are equal between two sets, the ARX-based MinHash will falsely indicate that the whole sets are equal. To avoid issue, we choose our b parameter to be in the range $[p/2, 3p/2]$, so that a broad range of inputs saturate $(x + b)$ values and the same minimum element will not be chosen multiple times. Choosing b parameter values larger than $3p/2$ is redundant because there is a maximum of p different digests for each input x.

C. *Verification of our ARX Construction*

The ARX universal hash introduced in Eq. 3 is a simple, yet surprisingly powerful construction that enables an FHE-friendly MinHash implementation. To further verify the effectiveness of the ARX 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 and we generate 100 different variants of the document for set similarity. Each document is then converted to a set of integers using *t-shingling* [34]. 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 to modulus 2^{32}. In our analysis, we also

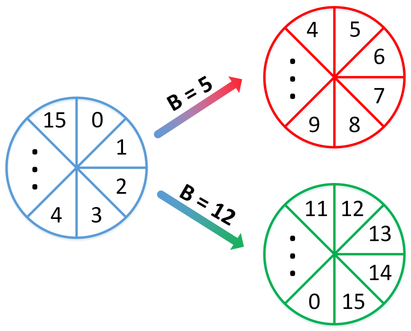

Fig. 2. Visual example of how the ARX hash maps inputs to digests. Different b parameters result in a different rotations over the input set, which means that a different element of the input set will correspond to the minimum hash value.

opt for a large modulus p for both the ARX 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 and ARX hashes (Eqs. 2 and 3). Specifically, the masking value for the ARX-MinHash is $p = 2^{32} - 1$, and the reduction value 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 ARX-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 ARX hash construction is a viable alternative to the CW hash, and closely approximates the Jaccard similarity.

D. Computing the Minimum of a Set of Hash Digests

In MinHash, we need to return the smallest value in a set of digests as a signature, for each one of the k hash functions and for each input data set. Therefore, to find the minimum hash we essentially need to compare all hash values; however, comparisons in the encrypted domain are a challenge, since the computing party can not access any information about the underlying plaintext. Therefore, making a run-time decision based on the results of a comparison is impossible. Fortunately, it is certainly possible to multiplex two encrypted values with an encrypted select bit (in this case, the result of the comparison). Therefore, in the CGGI cryptosystem, our observation is that we address this challenge 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 relative to simple bitwise operations, it is important to apply rigorous circuit-level optimizations during an RTL synthesis step, to reduce the overall circuit size as this constitutes a key bottleneck of the entire MinHash evaluation. Indeed, a key motivation for adopting the MinHash algorithm in the first place is to reduce the total number of comparison operations to compute the set similarity. 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 hash functions employed (Alg. 1).

E. Input Size Considerations

Another major concern while evaluating encrypted data homomorphically is that the size of the input data should not be dynamic, to allow synthesizing an efficient combinational netlist that is compatible with FHE. This means that the homomorphic program should not create and evaluate an array of data at run-time, if the input size is unknown beforehand. Fortunately, in our FHE framework, we can generate multiple variants of the homomorphic program, for inputs of different sizes. At the same time, the input size cannot be arbitrarily large, as the Google-XLS framework that we employ for HLS has an upper bound on the number of loop iterations that can be unrolled for synthesis. Likewise, performing logic optimizations with the Yosys synthesis suite is resource-intensive as the circuit size increases. 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.

IV. 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 the MinHash algorithm 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,
- Processing the generated netlists with HELM to create and run the homomorphic program with CGGI.

A. 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 details, 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 hash 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 MihHash result. The program also stores the minimum value of the hash function 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 (ARX case). Next, we discuss the implementations of the two hash families employed in our MinHash framework.

1) Carter-Wegman Hash: The Carter-Wegman universal hash is the standard function used traditionally by MinHash for most applications. As mentioned earlier, the (unencrypted) CW hash is efficient and can provide randomly-looking mappings with a low probability of collisions. 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. Moverover, generating these circuits for each loop iteration of each set and hash function further exaggerates the problem. 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.

2) ARX Hash: An add-rotate-xor hash function is a more efficient alternative to the CW hash for implementing MinHash in CGGI, which natively supports Boolean operations over ciphertexts. As the operations used in the ARX hash directly translate to Boolean gates for encrypted evaluation, this hash can be evaluated many times 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 $2^d - 1$. Using the definition of Eq. 3, we implement the ARX hash using the addition and $\&$ operations of C++.

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 hash function parameters are set in a way that satisfies the hash constraints mentioned before. A further discussion about our setup is provided in our experimental evaluation.

B. 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 hash 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.

While there are multiple HLS tools that can accomplish this task, we opted for the Google-XLS [31] framework due to its popularity and the fact that its open-source. Additionally, it has seen use in the FHE literature already, as it serves as a plug-in for the Google Transpiler framework [35]. To use Google-XLS, 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.

C. 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 [32], 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. In conclusion, the netlists generated from Yosys will serve as a public input for FHE computation (i.e., the circuit to be evaluated).

D. Generating Homomorphic Programs Using HELM

To generate and run the final homomorphic programs, we use an open-source framework called HELM [33], 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. Note that we can generate multiple homomorphic programs for datasets of different sizes, and once we have our homomorphic circuit of a certain size, we can feed any two encrypted datasets of that size to it for encrypted evaluation and receive the encrypted result.

V. Experimental Evaluation

In this section, we provide a comprehensive evaluation of our proposed framework by comparing the performance of the ARX and the CW hash 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. We remark that our underlying runtime library (HELM) parallelizes the homomorphic circuit across all available CPU threads. 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.

A. Methodology for Plagiarism Detection

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.

1) 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 are 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 [34]. 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 are directly correlated with the number of substrings of length t. The range of each integer token depends on the hash utilized in the FHE circuit (i.e., ARX or CW). We note that each hash 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 ARX hash 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.

2) Parameterizing Homomorphic Circuits: Next we need to set the constant parameters for our homomorphic programs in order to run the encrypted sets correctly. Specifically, the parameters required for the MinHash circuits include the number of hashes k (in this work we use $k = 5$ unless noted otherwise), the bitsize of our variables and the parameters of our hash functions. As soon as the number of hash functions k is fixed, we can then select the parameters for each hash to generate k unique hash 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.

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 accomodate 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 hash functions can work as intended.

For the ARX hash functions, we use Eq. 3, where the required parameters are the coefficient b 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 $p = 4095$ as our reduction value. The b values are chosen to be close to the masking value, mostly in the range $[p/2, 3p/2]$. This way, just like the CW hash, most of the $(x + b)$ values will exceed the masking value so the hash function works properly.

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. 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.

B. Comparing Both MinHash Variants with the Naive Approach

For each hash 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 ARX-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-ARX circuits. This means that the MinHash algorithm with the ARX hash function is way more efficient than the standard CW-based MinHash using Boolean FHE, so that it is more scalable and can be used with larger sets as inputs and more hash functions.

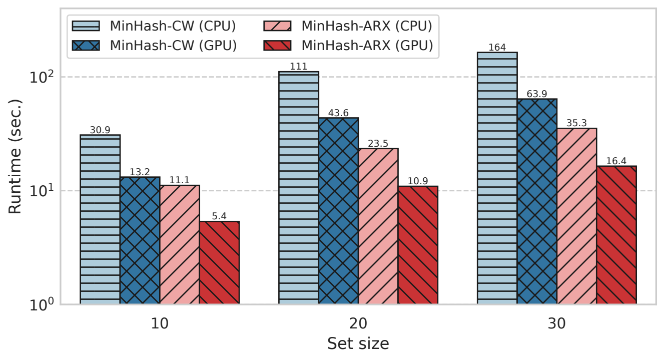

Fig. 3. Comparison between the ARX and CW hashes in MinHash.

Due to the graceful scalability of the MinHash-ARX circuits, we created additional circuits corresponding to set sizes of 50, 75, and 100 for the ARX variant. Next, we compare our MinHash-ARX 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.

For the naive method, we create circuits of size 10, 20, 30, 50, 75, and 100 to compare with the MinHash-ARX 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-ARX 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 shows that MinHash-ARX performs markedly better than the naive method for moderate and large set sizes.

C. Time-Accuracy Trade-off

All experiments above were conducted using $k = 5$ hash functions for the MinHash circuits. Although five hash 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-ARX circuits are the fastest and most scalable. Therefore, we use the MinHash-ARX circuit of input size 30 and implement it using both 25 and 50 hash functions. The circuit evaluations are shown in Figure 5, and we observe that as the number of hashes grows, the runtime increases linearly. Interestingly, the runtime of the MinHash-CW circuit of size 30 with five hash functions is around the same as the MinHash-ARX circuit of size 30 with 25 hash functions. This further solidifies MinHash-ARX as the superior alternative and even with an increase in hash functions, it is still viable and scalable.

D. CPU vs GPU Evaluation Comparison

All experiments were done with both a CPU-based cryptographic backend and a GPU-based backend, which differ depending on which type of device that is executing the underlying CGGI homomorphic operations. 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 $2\times$ 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.

VI. Related Works

Several works have focused on privacy-preserving operations between sets of data; in this section, we expound upon works that tackle both private set intersection (PSI) and private set similarity, which is closest to our work.

A. Private Set Intersection

The problem of private set intersection involves computing a subset consisting of elements present in two or more private sets. Kerschbaum [36] introduced a PSI protocol based on Bloom filters and the BGN homomorphic cryptosystem [37] and incorporates an additional computing party (independent from the client and server). Chen et al. [38] introduced a hash-based protocol tuned for computing the similarity between a large and small set. The protocol utilizes the BFV cryptosystem [19] 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. [39] that also incorporates verifiable computation and oblivious pseudorandom functions. Like the previous work, the authors target the BFV cryptosystem as the HE backend (particularly the RNS implementation in the Microsoft SEAL library [40]).

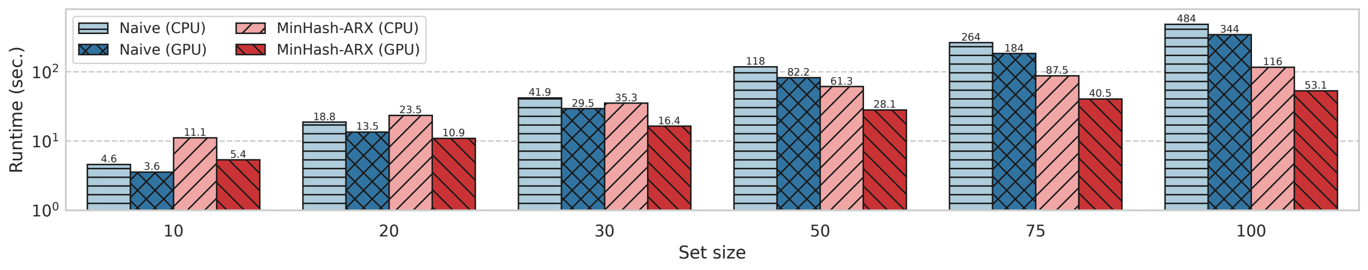

Fig. 4. Comparison between the MinHash-ARX circuits and the naive method with different set sizes. The naive method compares each hash digest of the first set with every hash digest of the second set.

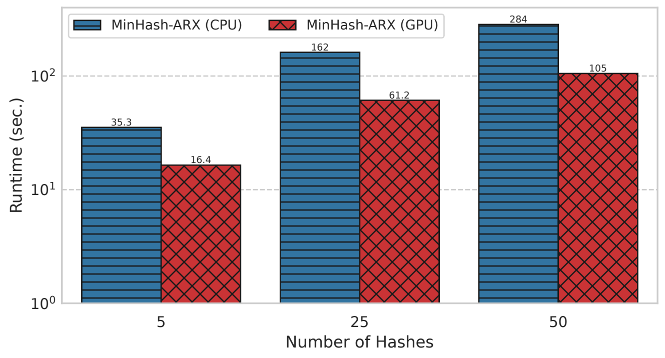

Fig. 5. Comparison between MinHash-ARX circuits of set size 30 with different numbers of hash functions.

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 [41]. Falk et al. propose another multi-party approach based on Bloom filters and is optimized for sets of unbalanced sizes [42].

Instead of returning the intersection of sets to clients, our work focuses on computing a similarity measurement between two sets. Nonetheless, many of these works employ similar types of mechanisms such as comparisons and hashing. 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.

B. Hash-Based Private Set Similarity Measurements

Other works focus on computing a similarity measure in a fashion similar to the proposed approach. Yan [43] estimates the Jaccard similarity using differential privacy, while Wong [44] 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 [45] computes a privacy-preserving MinHash variant to approximate the Jaccard similarity with differential privacy. However, this work requires a trusted third party that 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).

Lastly, the EsPRESSo [46] 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.

VII. Conclusion

In this paper, we present the MatcHEd framework for encrypted set-similarity based on MinHash. We introduce a new ARX-based hash 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 ARX hash family significantly outperforms the standard MinHash algorithm in the encrypted domain as well as a baseline approach that computes the exact set similarity.

References

[1] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-vm side channels and their use to extract private keys,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 305–316.

[2] ——, “Cross-tenant side-channel attacks in paas clouds,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 990–1003.

[3] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[4] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 695–711.

[5] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[6] R. Arora, A. Parashar, and C. C. I. Transforming, "Secure user data in cloud computing using encryption algorithms," *International journal of engineering research and applications*, vol. 3, no. 4, pp. 1922–1926, 2013.

[7] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[8] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[9] D. Mouris, N. G. Tsoutsos, and M. Maniatakos, "Terminator suite: Benchmarking privacy-preserving architectures," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 122–125, 2018.

[10] C. Gouert, D. Mouris, and N. G. Tsoutsos, "Sok: New insights into fully homomorphic encryption libraries via standardized benchmarks," *Cryptology ePrint Archive*, 2022.

[11] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy, "Mash: fast genome and metagenome distance estimation using minhash," *Genome biology*, vol. 17, no. 1, pp. 1–14, 2016.

[12] W. Lu, A. L. Varna, and M. Wu, "Confidentiality-preserving image search: A comparative study between homomorphic encryption and distance-preserving randomization," *IEEE Access*, vol. 2, pp. 125–141, 2014.

[13] M. Murugesan, W. Jiang, C. Clifton, L. Si, and J. Vaidya, "Efficient privacy-preserving similar document detection," *The VLDB Journal*, vol. 19, no. 4, pp. 457–475, 2010.

[14] K. M. Lee and K. M. 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*. IEEE, 2012, pp. 2117–2119.

[15] O. Durmaz and H. S. Bilge, "Fast image similarity search by distributed locality sensitive hashing," *Pattern Recognition Letters*, vol. 128, pp. 361–369, 2019.

[16] S. Bag, S. K. Kumar, and M. K. Tiwari, "An efficient recommendation generation using relevant jaccard similarity," *Information Sciences*, vol. 483, pp. 53–64, 2019.

[17] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.

[18] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1977, pp. 106–112.

[19] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[20] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.

[21] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[22] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer, 2010, pp. 1–23.

[23] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[24] S. Halevi and V. Shoup, "Bootstrapping for helib," *Journal of Cryptology*, vol. 34, no. 1, p. 7, 2021.

[25] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 617–640.

[26] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*. Springer, 2016, pp. 3–33.

[27] N. E. Diana and I. H. Ulfa, "Measuring performance of n-gram and jaccard-similarity metrics in document plagiarism application," in *Journal of Physics: Conference Series*, vol. 1196, no. 1. IOP Publishing, 2019, p. 012069.

[28] P. Xia, L. Zhang, and F. Li, "Learning similarity with cosine similarity ensemble," *Information sciences*, vol. 307, pp. 39–52, 2015.

[29] A. R. Lahitani, A. E. Permanasari, and N. A. Setiawan, "Cosine similarity to determine similarity measure: Study case in online essay assessment," in *2016 4th International Conference on Cyber and IT Service Management*. IEEE, 2016, pp. 1–6.

[30] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 327–336.

[31] W. Dai and B. Sunar, "XLS: Accelerated HW Synthesis," https://github.com/google/xls, 2020.

[32] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.

[33] C. Gouert, D. Mouris, and N. G. Tsoutsos, "Helm: Navigating homomorphic encryption through gates and lookup tables," *Cryptology ePrint Archive*, 2023.

[34] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Annual symposium on combinatorial pattern matching*. Springer, 2000, pp. 1–10.

[35] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak *et al.*, "A general purpose transpiler for fully homomorphic encryption," *arXiv preprint arXiv:2106.07893*, 2021.

[36] F. Kerschbaum, "Outsourced private set intersection using homomorphic encryption," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012, pp. 85–86.

[37] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2*. Springer, 2005, pp. 325–341.

[38] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1243–1255.

[39] Y. Jiang, J. Wei, and J. Pan, "Publicly verifiable private set intersection from homomorphic encryption," in *International Symposium on Security and Privacy in Social Networks and Big Data*. Springer, 2022, pp. 117–137.

[40] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[41] C. Hazay and M. Venkitasubramaniam, "Scalable multi-party private set-intersection," in *IACR international workshop on public key cryptography*. Springer, 2017, pp. 175–203.

[42] B. Hemenway Falk, D. Noble, and R. Ostrovsky, "Private set intersection with linear communication from general assumptions," in *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, 2019, pp. 14–25.

[43] Z. Yan, Q. Wu, M. Ren, J. Liu, S. Liu, and S. Qiu, "Locally private jaccard similarity estimation," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 24, p. e4889, 2019.

[44] K.-S. Wong, M. H. Kim *et al.*, "Preserving differential privacy for similarity measurement in smart environments," *The Scientific World Journal*, vol. 2014, 2014.

[45] Z. Yan, J. Liu, G. Li, Z. Han, and S. Qiu, "Privmin: Differentially private minhash for jaccard similarity computation," *arXiv preprint arXiv:1705.07258*, 2017.

[46] C. Blundo, E. De Cristofaro, and P. Gasti, "Espresso: efficient privacy-preserving evaluation of sample set similarity," *Journal of Computer Security*, vol. 22, no. 3, pp. 355–381, 2014.